Social Network Project

COMPGC06:

Database and Information Management Systems

Team 21:

Ayrand Cruz

Brian Ho

Ping Ren

Gun-Woo Nam

Contents

Project Overview

This report covers team 21's COMPGC06 project entitled: ***mybebofacespacebook***. This is a social networking website that can be likened to websites such as Facebook, Bebo and Myspace.

The live hosted version of our website can be viewed at: gc06team21.azurewebsites.net

Main Features

- User registration and authentication
- Uploading photos
- Maintaining a blog
- Searching for and adding friends
- Friend recommendations according to a user's interests (collaborative filtering)
- Creating friend groups (circles)
- Messaging between friends
- Deleting friend groups (circles)
- Notifications for likes, friend requests and circles

YouTube link

A demonstration of our website's functionality can be viewed at:

https://youtu.be/eOTdxQvzF7M

Conceptual and logical design

Conceptual Design

Design methodology is the structured approach in facilitating the process of design. It involves 3 phases of design: Conceptual, Logical and Physical. Each stage has their own rules and steps that define them. For our project we demonstrate our progress through Conceptual and Logical design.

The rule set that is defined in Connolly and Begg for building a conceptual data model is as follows:

1. Identify entity types
2. Identify relationship types
3. Identify and associate attributes with entity or relationship types
4. Determine attribute domains
5. Determine candidate, primary, and alternate key attributes
6. Consider use of enhanced modelling concepts (optional step)
7. Check model for redundancy
8. Validate conceptual model against user transactions
9. Review conceptual data model with user

We followed the rule set to produce the conceptual data model in Figure 1. We identified the entity types by studying the requirements and identifying the nouns and the objects mentioned. We were also able to define the relationships between them using the verbs used in the requirements.

The next step was assigning choosing the primary keys from the candidate keys. We made sure to the guidelines and selected the candidate keys that had:

- The candidate key with the minimal set of attributes;
- The candidate key that is least likely to have its values changed;
- The candidate key with fewest characters (for those with textual attribute(s));
- The candidate key with smallest maximum value (for those with numerical attribute(s));
- The candidate key that is easiest to use from the users' point of view.

Following this, the redundancy between the entities were checked which showed we had several redundant relationships between entities. This was the relations between photo and comment entity as well as the user and comment entity. We removed the relation between user and comments as the comments could be accessed through the photos. There was also redundancy with user and messages entities where the messages could be accessed through circles. The redundancy was removed by removing the relation between user and messages.

Lastly, we made sure to validate the model so that it all the requirements of the project were could be met before moving on to logical design.

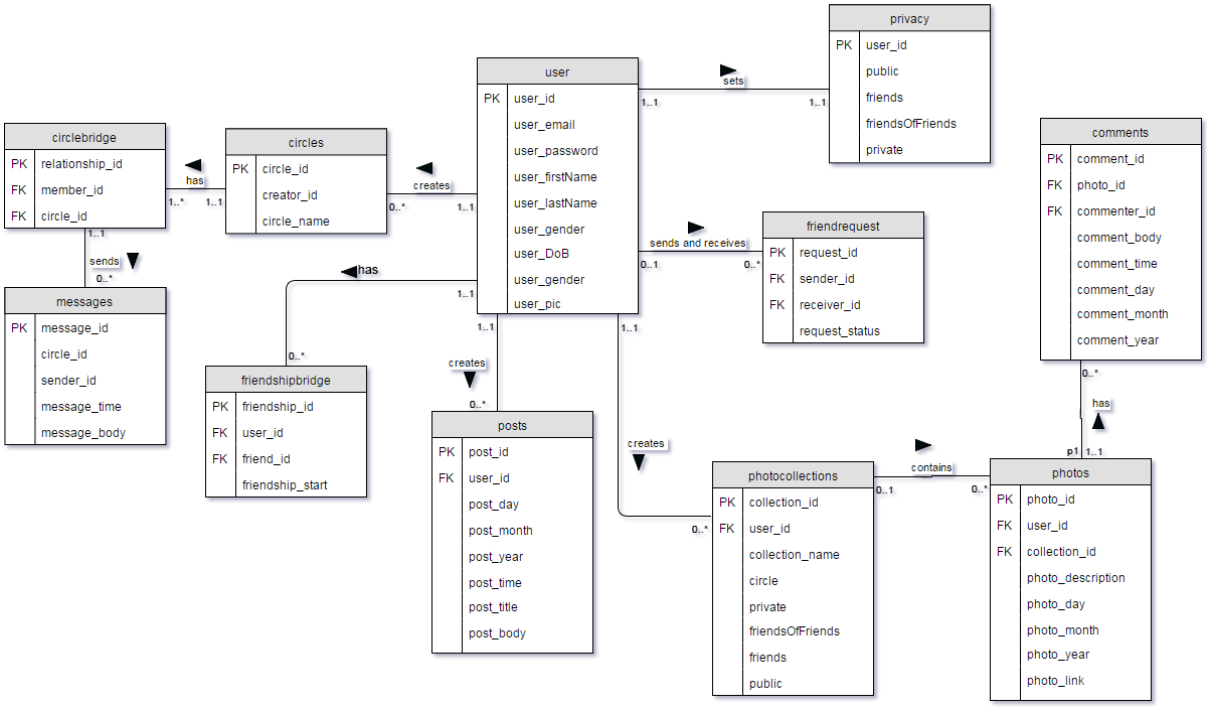

Figure 1: An entity-relationship diagram showing our conceptual data model

Logical Design

The resulting and final entity relationship diagram derived from the logical design rule set is displayed in Figure 2.

The main differences from the conceptual model design is the additions of new entities such as likes and notifications. They were added to the database to support the functionality of the website's extra features.

We derived the relationships which allowed us to closely examine our Primary Key and Foreign Key attribute placement. This identified the parent and child entities involved in the relationships. We examined the relationship between the user and privacy entities as this was a 1:1 relationship. We determined that it is an optional relationship as the user does not have to change the privacy settings as they have a default. This meant that we could keep privacy table separate from the users as it was not mandatory.

We also examined the entities and ensured that any weak entities and strong were dealt with properly. The normalisation analysis is an important step in logical design and this was described in the next section.

After the normalisation we made sure to check that the required transactions were still able to be done with the normalized database. =The referential integrity of the database is shown in our messages implementation with the circles table. When the primary key, circle ID, is deleted we cascade and delete the corresponding Foreign Key in circlebridge and messages table to maintain the integrity of the database.

Finally, logical model of the database was examined for future growth. We were able to see that we could add more features relatively easily if we an extended period of time to continue the project. Therefore, logical model would be able to evolve with minimal effect on the user.

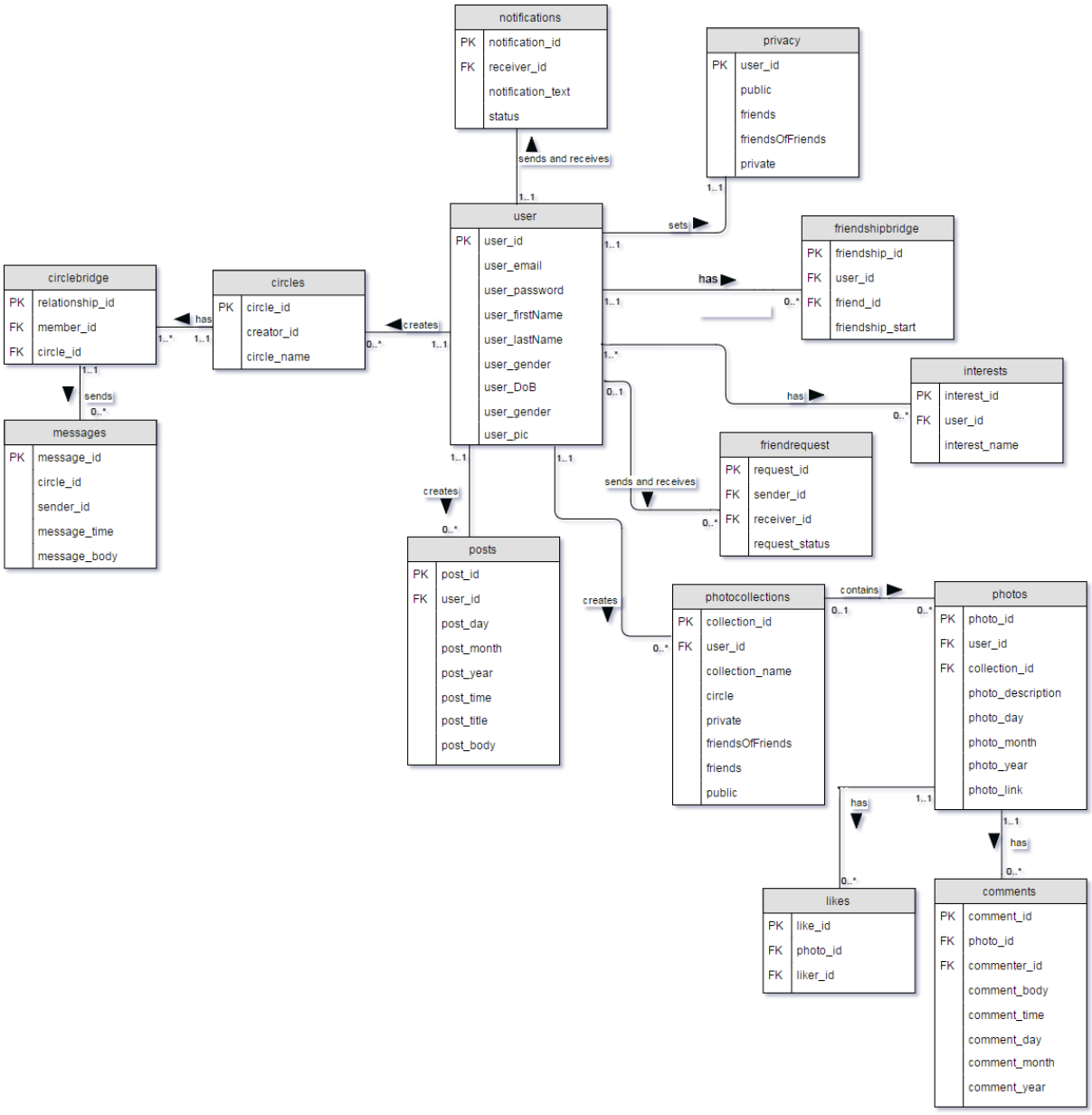

Figure 2: An entity relationship diagram of our Social Network

Database Schema

Our social network consists of a total of 14 tables. These are:

- circlebridge
- circles
- comments
- friendrequests
- friendshipbridge
- interests
- likes
- messages
- notifications
- photocollections
- photos
- posts
- privacy
- user

In the following sub-sections, we provide the structure of each table. (Note: Primary Key = PK, Foreign Key = FK, Auto Increment = AI)

- circlebridge
  - **`relationship_id`** int(11) PK AI
  - **`member_id`** int(11) FK
  - **`circle_id`** int(11) FK

- circles
  - **`circle_id`** int(11) PK AI
  - **`creator_id`** int(11)
  - **`circle_name`** varchar(100)

- comments
  - **`comment_id`** int(11) PK AI
  - **`photo_id`** int(11) FK
  - **`commenter_id`** int(11) FK
  - **`comment_body`** varchar(100)
  - **`comment_time`** time
  - **`comment_day`** int(2)
  - **`comment_month`** int(2)
  - **`comment_year`** year(4)

- friendrequests
  - **`friendship_id`** int(11) PK AI
  - **`user_id`** int(11) FK
  - **`friend_id`** int(11) FK
  - **`friendship_start`** timestamp

- interests
  - **`interest_id`** int(11) PK AI
  - **`user_id`** int(11) FK
  - **`interest`** text

- likes
  - **`like_id`** int(11) PK AI
  - **`photo_id`** int(11) FK
  - **`liker_id`** int(11) FK

- messages
  - **`message_id`** int(11) PK AI
  - **`circle_id`** int(11) FK
  - **`sender_id`** int(11) FK
  - **`message_time`** timestamp
  - **`message_body`** text

- notifications
  - **`notification_id`** int(11) PK AI
  - **`notification_text`** text
  - **`status`** int(1)
  - **`receiver_id`** int(11) FK

- photocollections
  - **`collection_id`** int(11) PK AI
  - **`user_id`** int(11) FK
  - **`collection_name`** varchar(50)
  - **`public`** tinyint(1)
  - **`friends`** tinyint(1)
  - **`friendsOfFriends`** tinyint(1)
  - **`private`** tinyint(1)
  - **`circle`** tinyint(1)

- photos
  - **`photo_id`** int(11) PK AI
  - **`user_id`** int(11) FK
  - **`collection_id`** int(11) FK
  - **`photo_description`** varchar(100)
  - **`photo_day`** int(2)
  - **`photo_month`** int(2)
  - **`photo_year`** year(4)
  - **`photo_link`** varchar(100)

- posts
  - **`post_id`** int(11) PK AI
  - **`user_id`** int(100) FK
  - **`post_time`** timestamp
  - **`post_title`** text
  - **`post_body`** text

- privacy
  - **`user_id`** int(11) PK AI
  - **`public`** tinyint(1)
  - **`friends`** tinyint(1)
  - **`friendsOfFriends`** tinyint(1)
  - **`private`** tinyint(1)

- user
  - **`user_id`** int(100)
  - **`user_email`** varchar(100)
  - **`user_password`** varchar(100)
  - **`user_firstName`** text
  - **`user_lastName`** text
  - **`user_DoB`** date
  - **`user_gender`** text
  - **`user_pic`** varchar(100)

Normalisation Analysis

Normalisation process allows us to produce a set of relations with desirable properties, with a set of data requirements. It is beneficial as it will be much easier for the user to maintain and access the data. It also requires minimal storage space which reduces the cost. This optimises the data integrity of the tables by minimising the data redundancy and prevents update anomalies such as insertion, deletion and modification.

To carry this out we examined the functional dependencies of the attributes within our tables. The process is carried out in a series of stages that meet a specific normal form that has its specific properties. As the process continues through the different levels, the data integrity becomes higher but the cost of the retrieval time.

The most commonly used normal forms are: 1st (1NF), 2nd (2NF) and 3rd (3NF). To fulfil, 3rd normal form, the database needs to have:

1. Tables with a defined primary key, wherein the intersections of each row and column contain one and only one value
2. Tables where every-non primary key attribute is fully dependent on any candidate key.
3. Tables which have no non-primary-key attribute transitively dependent on any candidate key

First Normal Form

Our database meets 1NF as all 14 of our tables have a defined primary key with each row and column intersection only containing 1 value.

Second Normal Form

Our database meets 2NF as we do not use any composite keys as primary keys. Therefore, we do not have any partial dependencies as all our non-key attributes are fully dependent on the primary key. Any table that is in 1st normal form and contains only a single key as the primary key is automatically in 2nd normal form.

Third Normal Form

As our database is in 1NF and 2NF, the other condition to satisfy 3NF is that there should be no transitive dependencies in our tables. For example, A is functionally dependant on B and B is functionally dependent on C, which means C is transitively dependent on A through B.

Our tables are in 3NF as we do not have any transitive dependencies, all non-key attributes are fully dependent on the primary key of the table.

Table Name	1st Normal Form	2nd Normal Form	3rd Normal Form
user	yes	yes	yes
friendshipbridge	yes	yes	yes
friendrequest	yes	yes	yes
posts	yes	yes	yes
photocollection	yes	yes	yes
photos	yes	yes	yes
likes	yes	yes	yes
comments	yes	yes	yes
interests	yes	yes	yes
privacy	yes	yes	yes
notifications	yes	yes	yes
circles	yes	yes	yes
circlebridge	yes	yes	yes
messages	yes	yes	yes

SQL Query Explanations

Below, we have included most of the key queries that are used in our project along with a short explanation for each one describing what the aim of each query is and how each one goes about bringing about the desired result.

Managing Profiles

user_insert.php

```
INSERT INTO user (user_email, user_password, user_firstName, user_lastName, user_DoB, user_gender, user_pic)
 VALUES ('$email', '$hash_pw', '$firstName', '$lastName', '$birthday', '$gender', 'user_default.png')";
```

Aim: Register new user.

This query is set inside an if statement checking to see if a user with the email address that has been typed into the form already exists in the database. Only if it does not already exist, we insert the input email, password, first name, last name, date of birth and gender into the database's "user" table. The image is set by default to "user_default.png".

```
INSERT INTO privacy (user_id) VALUES (".$rowNewID[0].")
```

Aim: Establish privacy settings for a user.

Once inserted into the "user" table, we also add this new user ID into the privacy table. The values for the different columns in this table are left to their default values.

profile.php

```
SELECT * FROM user WHERE user_email = '$logged_email'
```

Aim: Identify and retrieve the session's user's details.

$logged_email is the variable that holds the email that has been saved to the session. We need to get the user_id for the logged in user so that we can query for it later when we need to find the user's details.

```
SELECT * FROM user WHERE user_id = '$userID'
```

Aim: Retrieve details of user.

We use these to display these details on their profile.

```
SELECT interest FROM interests WHERE user_id = $userID
```

Aim: Find all interests for a user.

This query searches and returns all of the interests for a given user. $userID here is the user ID for the profile currently on view.

checkPrivacy.php

```
SELECT * FROM privacy WHERE user_id = '$userID'
```

Aim: Identify privacy settings for a user.

This returns all of the privacy option selections for a given userID from the privacy table.

addNewInterest.php

```
SELECT * FROM interests WHERE (user_id = $sessionUserID AND interest = '$newInterest')
```

Aim: Find a user's interests.

This returns all of the interests for the logged in user that matches the new interest that the user wishes to add. This is to ensure there are no duplicates added to their interests.

```
INSERT INTO interests (user_id, interest) VALUES ('$sessionUserID', '$newInterest')
```

Aim: Add new interest for a user.

This inserts a new interest into the interests table with the correct user ID.

Friends System

Searchresult.php

```
SELECT * FROM user WHERE CONCAT(user_firstName, ' ', user_lastName) LIKE '%".$query."%'
```

Aim: Identify and return users with a name similar to a search query.

This select from the user table searches for any first and last names that are “like” the query. The returned values are user id, first name and last name for any users who fall under the query.

The search query is stored in a $query variable. This must be a minimum length of 3 characters in length and is modified before being used in the query. Modifications include htmlspecialchars($query) which changes characters to their html names and the mysqli_real_escape_string($con, $query) which helps protect against SQL injection attacks.

Depending on the result user id, different buttons are revealed. If the result is yourself, no buttons come up. If the result is in your friends list, a tick to show they are your friend appears and a blog button is displayed to take you to their blog.

There may also be a pending or add option displayed if they have sent you a friend request or if you have sent them a friend request.

recommendedFriends.php

```
"SELECT user_id, COUNT(*) rank
 FROM friendshipBridge
 WHERE friendshipBridge.user_id <> '$sessionUserID' AND friendshipBridge.friend_id <> '$sessionUserID'
 AND (friendshipBridge.user_id IN (SELECT user.user_id
                                   FROM friendshipBridge
                                   JOIN user ON friendshipBridge.user_id = user.user_id
                                   WHERE friendshipBridge.friend_id = '$sessionUserID'
                                   UNION ALL
                                   SELECT user.user_id
                                   FROM friendshipBridge
                                   JOIN user ON friendshipBridge.friend_id = user.user_id
                                   WHERE friendshipBridge.user_id = '$sessionUserID'))
 AND (friendshipBridge.friend_id IN (SELECT user.user_id
                                     FROM friendshipBridge
                                     JOIN user ON friendshipBridge.user_id = user.user_id
                                     WHERE friendshipBridge.friend_id = '$sessionUserID'
                                     UNION ALL
                                     SELECT user.user_id
                                     FROM friendshipBridge
                                     JOIN user ON friendshipBridge.friend_id = user.user_id
                                     WHERE friendshipBridge.user_id = '$sessionUserID'))
 GROUP BY friendshipBridge.user_id
 ORDER BY rank DESC";
```

Aim: Rank friends according to number of mutual friends.

This large query is made up of a number of nested SELECT queries. Here we wish to return the user_id and a count which we refer to as “rank” from the friendshipBridge table. We filter the results by saying that we do not want the user_id or the friend_id to == the logged in user’s userID. We also want both the user_id and friend_id fields to be present in the friends list for the given session user ID. The results are grouped by user_id and ranked according to the count(*). The overall result is a ranking according to how many mutual friends any friend of the session user has with the session user (when their ID is the one in the friendshipBridge.user_id column i.e. they are the one to have sent the friend request/are more active).

```
"SELECT * FROM user WHERE user.user_id IN (SELECT user.user_id
                                           FROM friendshipBridge
                                           JOIN user ON friendshipBridge.user_id = user.user_id
                                           WHERE friendshipBridge.friend_id = '$topFriendUserID'
                                           UNION ALL
                                           SELECT user.user_id
                                           FROM friendshipBridge
                                           JOIN user ON friendshipBridge.friend_id = user.user_id
                                           WHERE friendshipBridge.user_id = '$topFriendUserID')
                                           AND user.user_id NOT IN (SELECT user.user_id
                                                                    FROM friendshipBridge
                                                                    JOIN user ON friendshipBridge.user_id = user.user_id
                                                                    WHERE friendshipBridge.friend_id = '$sessionUserID'
                                                                    UNION ALL
                                                                    SELECT user.user_id
                                                                    FROM friendshipBridge
                                                                    JOIN user ON friendshipBridge.friend_id = user.user_id
                                                                    WHERE friendshipBridge.user_id = '$sessionUserID')
                                           AND user.user_id <> '$sessionUserID'";
```

Aim: Recommends all users in the friends list of the top friend of the session user (has the most mutual friends shared with the session user) and are not already friends with the session user.

Here we are returning all of the details from the user table for a user whose id is in the friend’s list of the $topFriendUserID (variable holding the ID of the friend with the most mutual friends as the session user) and is not in the friends list of the session user and is not (<>) the session user.

similarInterestsFiltering.php

```
SELECT interests.user_id, COUNT(*) rank
 FROM interests JOIN (SELECT * FROM interests WHERE user_id = '$sessionUserID') myInterests
 ON interests.interest = myInterests.interest
 WHERE interests.user_id <> '$sessionUserID'
 AND (interests.user_id IN (SELECT user.user_id
                            FROM friendshipBridge
                            JOIN user ON friendshipBridge.user_id = user.user_id
                            WHERE friendshipBridge.friend_id = '$sessionUserID'
                            UNION ALL
                            SELECT user.user_id
                            FROM friendshipBridge
                            JOIN user ON friendshipBridge.friend_id = user.user_id
                            WHERE friendshipBridge.user_id = '$sessionUserID'))
 GROUP BY interests.user_id
 ORDER BY rank DESC";
```

Aim: Apply method of collaborative filtering based on mutual interests. Find the number of mutual interests between session user and friends.

This large query above is the key to our collaborative filtering function. It returns a ranking of everybody on the logged in user's friends list (as specified via the nested SELECT query of the IN clause) according to how many interests they have in common with the logged in user. This is done by first filtering out only the session user's interests and then joining this resulting table back on to the table of interests of the user's friends on every matching interest and then grouping by the user ID of each user in the interests table.

```
SELECT user.user_firstName, user.user_lastName, interests.user_id, COUNT(*) rank
 FROM user JOIN interests ON user.user_id = interests.user_id
 JOIN (SELECT * FROM interests WHERE user_id = '$sessionUserID') myInterests
 ON interests.interest = myInterests.interest
 WHERE interests.user_id <> '$sessionUserID'
 AND (interests.user_id NOT IN (SELECT user.user_id
                                FROM friendshipBridge
                                JOIN user ON friendshipBridge.user_id = user.user_id
                                WHERE friendshipBridge.friend_id = '$sessionUserID'
                                UNION ALL
                                SELECT user.user_id
                                FROM friendshipBridge
                                JOIN user ON friendshipBridge.friend_id = user.user_id
                                WHERE friendshipBridge.user_id = '$sessionUserID'))
 GROUP BY interests.user_id
 ORDER BY rank DESC";
```

Aim: Apply method of collaborative filtering based on mutual interests. Find users not currently friends with the session user and has above a minimum number of mutual interests with the session user.

This query picks up from where the previous query left off. From the results of the previous query, we identify the average number of similar interests that the logged in user has between them and all of their friends. We then run the query as displayed above which identifies and returns the names and IDs of all users that are not friends with the logged in user and are not the logged in user themselves. We rank these in order of number of similar interests and for each result, we only display them in the recommendations if they have a greater number of similar interests than the average number of mutual interests that we calculated earlier between the session user and their friends.

Accept_requestTEST.php

```
DELETE FROM friendrequests WHERE (sender_id='$thisFriend' AND receiver_id='$sessionUserID')
```

Aim: Accept a friend request that has been received.

We delete from friendrequest where the sender_id and the receiver_id combination is the same in the friendrequest table. Here sender_id = the user_id of the person who sent the request whilst receiver_id = the user_id of the user in session.

```
INSERT INTO friendshipbridge (user_id, friend_id) VALUES ('$sessionUserID','$thisFriend')
```

Aim: Record friendship in friendshipBridge.

To fully add someone as a friend we also insert their details into the friendshipbridge table. We insert the user_ids of the receiver and the sender of the friend request into the user_id and the friend_id of the friendshipbridge table in the database.

```
"INSERT INTO notifications(notification_text, status, receiver_id )
  VALUES ('Friend Request accepted from $FirstName $LastName','0', '$thisFriend' )";
```

Aim: Notification created for friend request acceptance.

Creates a notification entry for the sender of the friend request – notifying them that the friend request has been accepted

Add_friends.php

```
INSERT INTO friendrequests(sender_id, receiver_id, request_status) VALUES ('$sessionUserID','$thisFriend', '1')
```

Aim: Create friend request with sender and receiver.

This query inserts the user_ids of the user in session and the user_id of the person they want to add to the friendship request table as sender_id and receiver_id, respectively. This will allow the recipient to view this in the friendrequest received and the sender to view the request they have sent.

```
"INSERT INTO notifications(notification_text, status, receiver_id )
 VALUES ('Friend Request from $FirstName $LastName','0', '$thisFriend' )";
```

Aim: Notification created for friend request received.

This creates an entry in the notification table for the recipient of the friend request – describing the name of the sender.

Delete_friends.php

```
"DELETE FROM friendshipbridge WHERE (user_id = '$thisFriend' AND friend_id = '$sessionUserID')
 OR ( user_ID = '$sessionUserID' AND friend_id = '$thisFriend')";
```

Aim: Remove users from friends list.

It does this by taking the columns in the table user_id and friend_id and finding the correct pair to delete. The user_id can be the user id of the person you want to delete and the friend_id can be the user id of the user in session OR vice versa as in our friendshibridge table we just enter the pair of user ids only once and use our queries to get the relationship.

So the pair of user ids that is needed could be entered in 1 of 2 ways so this query accounts for that

Reject_request.php

```
DELETE FROM friendrequests WHERE (sender_id='$thisFriend' AND receiver_id='$sessionUserID')
```

Aim: Reject a friend request that has been received.

The request is present in the friendrequest table in our database. To reject it we find the pair of user ids and delete them from the table - these are the sender_id (which would be the user_id of the sender) and the receiver_id (which would be the user_id of the user logged in)

friendsList.php

//Friend request received section

```
SELECT user.user_firstName, user.user_lastName, user.user_id, user.user_pic FROM friendrequests
 JOIN user ON friendrequests.sender_id = user.user_id WHERE friendrequests.receiver_id = '$sessionUserID'
```

Aim: Find friend requests that the session user has received.

This query gets the details of the users that have sent friend requests to the user that is logged in. This returns the user id, firstname and lastname of those users.

To do this it joins the friendrequest table with the user table where the sender_id from friendrequest matches the user_id in the user table where the friendrequest receiver_id is the user_id of the user that is logged in to the session.

```
SELECT friendrequests.request_status FROM friendrequests
 WHERE (friendrequests.sender_id = '$sessionUserID' AND friendrequests.receiver_id = '$thisFriendID'  )
```

Aim: Find friend request status.

This query is within the while loop running the query above. It gets the status from the friendrequest table of the specified user id pair (sender_id and receiver_id).

A request status of 1 = friend request pending.

//Friend requests that have been sent by the session user

```
SELECT user.user_firstName, user.user_lastName, user.user_id, user.user_pic FROM friendrequests
 JOIN user ON friendrequests.receiver_id = user.user_id WHERE friendrequests.sender_id = '$sessionUserID'
```

Aim: Aim: Find friend requests that the session user has sent.

This query gets the details of all of the users that the logged in user has sent friend requests to. This returns the user id, firstname and lastname of those users.

To do this it joins the friendrequest table with the user table where the receiver_id from friendrequest matches the user_id in the user table where the friendrequest sender_id is the user_id of the user that is logged in.

//Display user's friends

```
"SELECT user.user_firstName, user.user_lastName, user.user_id, user.user_pic from friendshipBridge
  JOIN user ON friendshipBridge.user_id = user.user_id
  WHERE friendshipBridge.friend_id = '$sessionUserID'
  UNION ALL
  SELECT user.user_firstName, user.user_lastName, user.user_id, user.user_pic FROM friendshipBridge
  JOIN user ON friendshipBridge.friend_id = user.user_id
  WHERE friendshipBridge.user_id = '$sessionUserID'";
```

Aim: Find the names of all the friends for the user whose profile is currently on view.

This query allows us to get the friends of the user that is logged in. There are 2 parts to this query in the form of two similar queries that are given as a single result via the UNION ALL function. The first part gets the user details of each friend from a query that joins friendshipbridge to the user table on

friendshipBridge.user_id = user.user_id

WHERE friendshipBridge.friend_id = '$sessionUserID'

This returns the friends of the user in cases where the logged in user's user_id is in the friend_id column of the friendship bridge.

The second part of the query is different in where we apply the join in our friendshipBridge and user tables.

ON friendshipBridge.friend_id = user.user_id

WHERE friendshipBridge.user_id = '$sessionUserID'";

The join here is on fields where the friend_id is the same as the user_id of the user table and the user_id of the person logged in is in the user_id column of the friendship bridge table.

These 2 parts are required as each friendship exists as one single entry in the friendshipBridge table and so the user id of the person logged in could appear in either of the user_id or the friend_id columns (it would depend on who sent the friend request).

Circles

Circles Page

```
$get_myCircles = "SELECT circles.circle_name, circles.circle_id, circles.creator_id FROM
circleBridge JOIN circles ON circleBridge.circle_id = circles.circle_id WHERE circleBridge.member_id = '$userID'";
```

Aim: To populate a panel with the circle groups the user has created or is part of.

The query joins circleBridge and circles tables together and selects the circle name, id and creator id where the member id matches the current user.

```
$get_myFriends5 = "SELECT user.user_firstName, user.user_lastName, user.user_id, user.user_pic from friendshipBridge
                   JOIN user ON friendshipBridge.user_id = user.user_id
                   WHERE friendshipBridge.friend_id = '$userID'
                   UNION ALL
                   SELECT user.user_firstName, user.user_lastName, user.user_id, user.user_pic FROM friendshipBridge
                   JOIN user ON friendshipBridge.friend_id = user.user_id
                   WHERE friendshipBridge.user_id = '$userID'";
```

Aim: To populate a panel with the current user's friends to create a circle group with.

The query uses union between the current user and the friend's list from the friendshipBridge table where the id is the current session user's id. It displays the friend's name, user id and user photo.

new_circle.php

```
$insertCircle = "INSERT INTO circles (creator_id, circle_name) VALUES ('$userID', '$circleName')";
```

Aim: To insert new circle into circle table.

Insert the user id of the current user into creator_id, and the name of the new circle into circle_name from the circles table.

```
$get_CircleID = "SELECT * FROM circles WHERE circle_name = '$circleName'
                 AND creator_id = '$userID'";
```

Aim: To find the id of the new circle just created from the $insertCircle query.

Select all from the circles table where the circle name and creator id matches the same php variables from the $insertCircle query.

```
$addFriend2Circle = "INSERT INTO circleBridge (member_id, circle_id) VALUES ($optionArray[$i],$circleID)";
```

Aim: To add the friends invited to the circleBridge table.

Insert the array of the friends invited into circleBridge as unique member id's to the circle.

Circle_group page

```
$get_circleName = "SELECT * FROM circles WHERE circle_id = '$get_circleID'";
```

Aim: To populate the circle page title with the correct circle name.

To select the circle name from the circles table where the id matches the circle id.

```
$get_messages = "SELECT user.user_firstName, user.user_lastName, user.user_pic, messages.message_body, messages.sender_id, messages.message_time
                 FROM messages JOIN user ON messages.sender_id = user.user_id WHERE messages.circle_id = '$get_circleID' ORDER BY message_id DESC";
```

Aim: To populate the circle chat window with messages.

Join the messages with the users where the messages sender id matches the current user's id from the correct circle id. On the chat window, display the user's name, photo, message and time of the message.

```
$checkCircleID = "SELECT * FROM circles WHERE circles.circle_id = $get_circleID";
```

Aim: A check to counter the back button once a circle has been deleted

If a user hits the back browser button after deleting a circle group, the circle group id is lost, so this check is to direct the user straight to circles if they press the back button.

```
$get_circleFriends = "SELECT user.user_firstName, user.user_lastName, user.user_pic
                FROM circleBridge
                JOIN circles
                ON circleBridge.circle_id = $get_circleID AND circleBridge.circle_id = circles.circle_id
                JOIN user
                WHERE circleBridge.member_id = user.user_id AND circleBridge.member_id != $userID";
```

Aim: To populate the circle group friends list with the correct users.

To match the circle id between circles and circleBridge table. From circleBridge, take the member id's and find their name and photo to populate the circle friend's list.

New_message.php

```
$insertMessage = "INSERT INTO messages (circle_id, sender_id, message_time, message_body) VALUES ($passed_circleID, $userID, now(), '$circle_message')";
```

Aim: To add a new message to the messages table.

Insert the circle id, sender id, message body and time of the message sent to the correct fields of the messages table. The time of the message is determined using the php 'now()' function i.e. the current time.

Delete circle function

```
$get_userID = "SELECT * FROM circles WHERE circles.circle_id = $circleID";
```

Aim: To identify whether the user id matches the creator id for the option to delete the circle.

Select all from the circles table where the circle id matches the variable $circleID.

```
$delete_circle1 = "DELETE FROM circles WHERE circles.circle_id = $circleID";
$delete_circle2 = "DELETE FROM circleBridge WHERE circleBridge.circle_id = $circleID";
$delete_messages = "DELETE FROM messages WHERE messages.circle_id = $circleID";
```

Aim: To delete a circle group

To delete the circle from the circles and circleBridge tables where the id matches the circle id you're trying to delete. Delete_messages takes the same circle id and deletes all the messages associated with that circle

Photos

upload_photo.php

```
INSERT INTO photos (user_id,collection_id,photo_description,photo_day, photo_month, photo_year, photo_link)
 VALUES ('$sessionUserID','$collectionArray[0]','$photoDescription','$photoDay', '$photoMonth', '$photoYear', '$dest')
```

Aim: Add a photo to a collection.

Inserts a new row into the table of photos. Details include the session user's ID, the ID number of the collection, a description of the photo as typed in by the user, the date of upload and the link to the photo's location in storage. Photos are stored in the upload folder.

```
UPDATE user SET user_pic = '$dest2' WHERE user_id = '$sessionUserID'
```

Aim: Upload a new profile picture.

Here we update the user_pic field with the new location of the newly uploaded profile picture. We also ensure it only occurs for the session user via their user ID. Photos are stored in the user_image folder.

delete_photo.php

```
DELETE  FROM photos WHERE photo_id = '$photo_id'
```

Aim: Delete a photo.

Here we pass the photo ID as a variable when we click on the delete button for any given photo. We then query for the photo with this photo ID and delete it from the table.

```
DELETE  FROM likes WHERE photo_id = '$photo_id'
```

Aim: Delete the record of likes on a photo that is being deleted.

Here we also ensure that we are deleting all of the likes that are associated with the photo that we are deleting, minimising the unnecessary load on the database.

new_collection.php

```
INSERT INTO photoCollections (collection_name,public,friends,friendsOfFriends,private,circle, user_id)
 VALUES ('$collectionName','$collectionPublic','$collectionFriends','$collectionFOF','$collectionPrivate',
   '$collectionCircle', $sessionUserID)";
```

Aim: Create a new collection for photos to be uploaded to.

We insert into the photoCollections table a new collection using the following details: the name of the collection the binary value of 1 or 0 for each type of privacy setting (there should only ever be a single privacy column with the value 1 and the remaining 3 should hold a value of 0).

new_photo_comment

```
"INSERT INTO comments (photo_id,commenter_id,comment_body,comment_day,comment_month,comment_year,comment_time)
 VALUES ('$pre_photoID','$userID','$photo_comment_content','$dateDay','$dateMonth ','$dateYear','$dateTime')";
```

Aim: Comment on a photo.

We use this query to insert a new entry into the comments table. We need to ensure that we include the photo ID (identified according to which photo we click on to add the new comment), the user ID of the user leaving the comment, the text of the body itself (as typed in by the user) and the date and time of comment submission.

like_photo.php

```
"INSERT INTO likes (photo_id,liker_id)
  VALUES ('$likePhotoID','$sessionUserID')";
```

Aim: Like a photo.

This query adds information about new likes on photos into the likes table. We only need 2 pieces of information here: the photo ID of the photo on which the like has been added and the user ID of the user who has liked the photo (i.e. the user currently in session).

photo collection.php

```
SELECT collection_id,collection_name from photocollections WHERE user_id = $sessionUserID
```

Aim: Display all collections for a user.

This is a simple query to return all of the collection IDs and names of collections for the logged in user. We can then display these all in via a while loop.

```
SELECT * FROM photos WHERE user_id = '$userID' ORDER BY photo_id DESC
```

Aim: Display all photos for a user.

This query returns all of the information for every photo that exists in the photos table for any given user ($userID) and orders them highest to lowest by photo ID i.e. by newest upload first (photo ID is auto incremented for each upload).

```
SELECT COUNT(*) FROM likes WHERE (photo_id = $thisPhotoID
```

Aim: Display the number of likes for a photo.

This query returns the number of likes for any given photo ($thisPhotoID).

comment_photo.php

```
SELECT * FROM photos WHERE photo_id =$get_photo_id
```

Aim: Display the selected photo.

A simple query returning the details for the photo matching the selected photo's photo ID ($get_photo_id).

```
SELECT user.user_firstName, user.user_lastName, user.user_pic, comments.photo_id,
 comments.comment_body, comments.comment_day, comments.comment_month,comments.comment_year,comments.comment_time
 FROM comments INNER JOIN user ON comments.commenter_id = user.user_id WHERE comments.photo_id = '$get_photo_id'
```

Aim: Display all comments and details for each comment for the selected photo.

This query returns information regarding every comment on a selected photo ($get_photo_id) including the first and last names of the user that posted each comment and their profile picture. This is done by joining the comments table and the user table on every entry where commenter_id matches the user_id.

Blogs

delete_post.php

```
DELETE FROM posts WHERE post_id = '$post_id'
```

Aim: Delete an existing post.

$post_id is the id of a post that has been passed as a PHP variable on the click of a post. This query simply deletes this post from the database table "posts".

new_post.php

```
"INSERT INTO posts (user_id, post_day, post_month, post_year, post_time, post_title, post_body)
  VALUES ('$sessionUserID','$dateDay', '$dateMonth', '$dateYear', '$dateTime', '$postTitle', '$postBody')"
```

Aim: Publish a new post.

Here, we insert a new post typed into an input text field by the user. We insert the day, month, year, time, title, text body and author user ID to the table "posts".

retrieve_posts.php

```
SELECT * FROM posts WHERE user_id = '$userID'
```

Aim: Display all posts for a given user on their blog.

Here, we retrieve all of the posts for a given user ID.

Extras

Fetch.php

```
UPDATE notifications SET status=1 WHERE status= '0' AND receiver_id = '$sessionUserID'
```

Aim: Update notifications that have been seen/read.

This updates the notifications that have been read (when the notifications button has been clicked). It updates the entries that have the specified user_id and a status of 0 (unread) to being 1 (read).

```
SELECT * FROM notifications WHERE receiver_id = '$sessionUserID' ORDER BY notification_id DESC LIMIT 5
```

Aim: Display the latest 5 notifications.

This gets all the notifications from the notifications table for the user. It does this only selecting entries in the notifications table for the specified user_id.

Home_feed.php

```
$get_myFriends5 = "SELECT user.user_firstName, user.user_lastName, user.user_id, user.user_pic, posts.post_title, posts.post_body, posts.post_time, posts.post_id
                   FROM user JOIN posts ON posts.user_id = user.user_id JOIN friendshipBridge ON friendshipBridge.user_id = user.user_id
                   WHERE friendshipBridge.friend_id = '$userID'
                  UNION ALL
                   SELECT user.user_firstName, user.user_lastName, user.user_id, user.user_pic, posts.post_title, posts.post_body, posts.post_time, posts.post_id
                   FROM user JOIN posts ON posts.user_id = user.user_id JOIN friendshipBridge ON friendshipBridge.friend_id = user.user_id
                   WHERE friendshipBridge.user_id = '$userID' ORDER BY post_id DESC";
```

Aim: To get the friends blog posts to populate the blog feed

This is a variation on the find friends query where the friends' user post details are also selected (added JOIN to posts where posts.user_id = user.user_id).

Acknowledgements

1. https://www.w3schools.com/ - a great website for learning web based languages such as HTML, CSS, Javascript, Jquery
2. http://getbootstrap.com/ - this is the template we used for the website. It is a great open source front-end framework.
3. http://bootboxjs.com/ - a small JavaScript library that allowed you to create simple Bootstrap modals