



## **Database Systems Project**

CS - 353 - Section 1

Project link: <https://github.com/theGuiltyMan/CS-353>

## **Social Gaming Marketplace-Mist Project Design Report**

**Alptuğ Albayrak - 21501970**

**Alperen Erkek - 21501265**

**Barış Eymür - 21502879**

**Deniz Şen - 21502997**

# TABLE OF CONTENTS

<b>1 Revised E/R Model</b>	<b>3</b>
<b>2 Relation Schemas and Normalizations</b>	<b>7</b>
2.1 Relational Schemas	7
2.1.1 Users	7
2.1.2 Games	8
2.1.3 Buy	9
2.1.4 In_game	10
2.1.5 Library	11
2.1.6 Plays	12
2.1.7 Genres	13
2.1.8 Game Genres	14
2.1.9 Friend Request	15
2.1.10 Friends	16
2.1.11 Messages	17
2.1.12 Send_invitation	18
2.1.13 Discussions	19
2.1.14 Moderates	20
2.1.15 Banned_users	21
2.1.16 Comments	22
2.1.17 Posts	23
2.1.18 Replies	24
2.2 Normalization	25
<b>3 Functional Components</b>	<b>26</b>
3.1 Use Cases / Scenarios	26
3.1.1 User	28
3.1.2 Admin	29
3.1.3 Game Developer	29
3.2 Algorithms	31
3.2.1 Game-Related Algorithms	31
3.2.2 User-Interaction Algorithms	31
3.2.3 Logical Requirements	32
3.3 Data Types	33
<b>4 User Interface Design &amp; Corresponding SQL Statements</b>	<b>34</b>
4.1 Login	34
4.2 Library	36

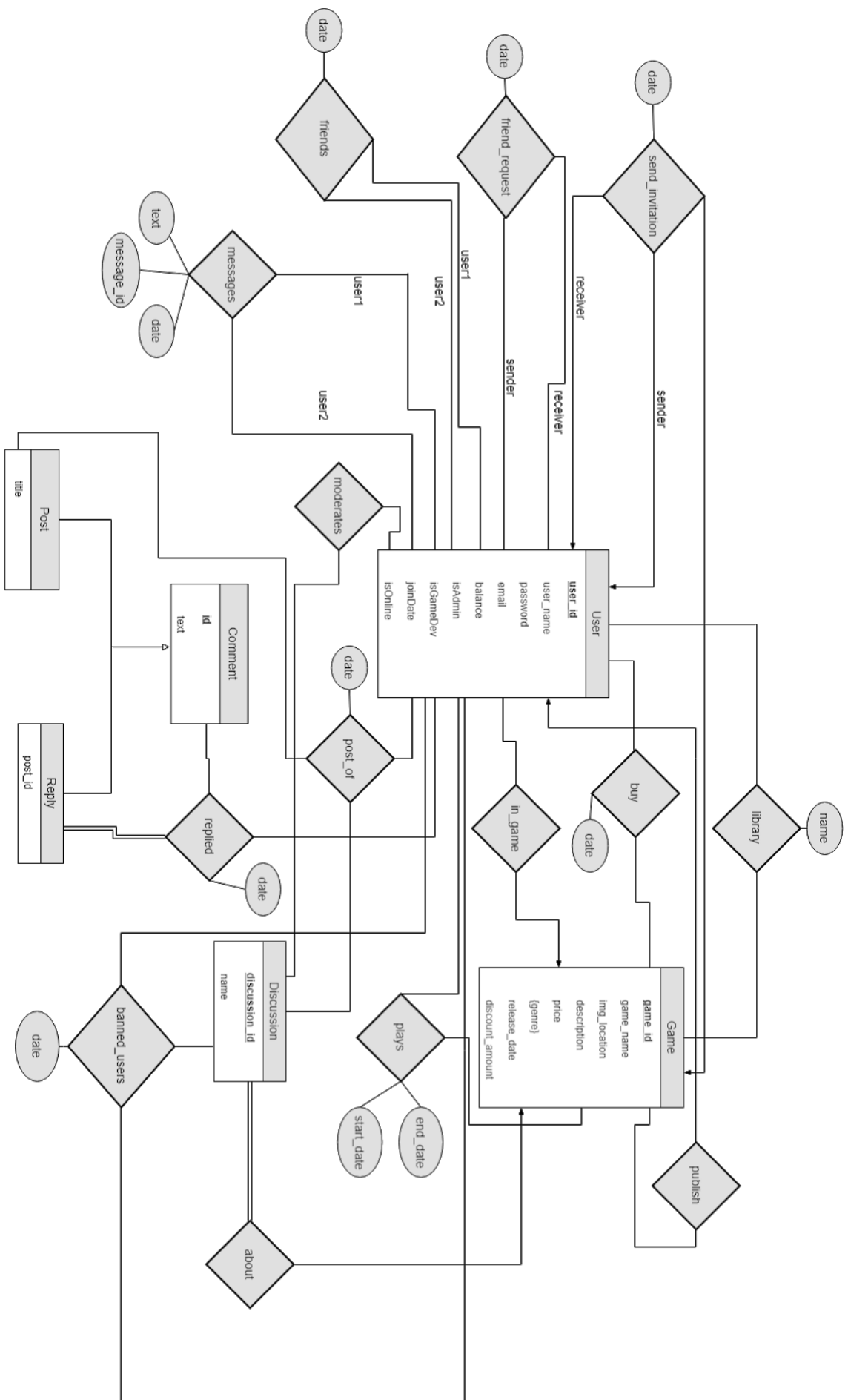
4.3 Store	40
4.4 Discussion	45
4.5 Friends	48
4.6 Activity	50
<b>5 Advanced Database Components</b>	<b>52</b>
5.1 Views	52
5.1.1 Games In User Library View	52
5.1.3 Posts In A Discussion View	52
5.1.4 Replies of Posts View	53
5.1.5 Trending Games View	53
5.1.6 Games by Genre	53
5.2 Reports	54
5.2.1 Total Number of Users	54
5.2.2 Total Number of Games	54
5.2.3 Total Number of Purchased Games	54
5.2.4 Top 10 Purchased Games	54
5.2.5 Number of Users in Game	55
5.2.6 Number of Users That Are Playing a Specific Game	55
5.2.7 Total Online Users	55
5.2.8 Total Money Spent	55
5.2.9 Number of Posts In a Discussion	56
5.2.10 Number of Replies in a Post	56
5.2.11 Total Time Spent In a Specific Genre	56
5.2.11 Total Number of Comments a User Posted	56
5.2.12 Total Number of Banned Users Per Genre	57
5.3 Triggers	57
5.4 Constraints	57
5.5 Stored Procedures	58
5.5.1 Releasing a Discount	58
5.5.2 Restoring the Price of a Discounted Game	58
5.5.3 Addition of Posts and Replies	59
<b>6 Implementation Plan</b>	<b>60</b>
<b>7 Website</b>	<b>60</b>

# 1 Revised E/R Model

We have made the following changes in our E/R model, according to the feedback of our TA and the details we noticed which were missing in our initial diagram. Our revised E/R model reflects the structure of our database system better:

- We have removed the Game Developer and Admin entities, which were specializations of the User entity. Instead, we have added 2 attributes to the User entity, which are isAdmin and isGameDev. These new attributes serve as Boolean values. Also, we added a joinDate attribute to the User entity, which holds the date the user registered to the system and an isOnline attribute which denotes whether the user is online or not.
- We have removed the “place\_order” relationship and we have added the “buy” relationship between User and Game. This is a many-to-many relationship and it has a date attribute.
- We removed the Activity entity, its specializations and its relationships. We will display the activities of a User by creating views.
- We removed the Library weak entity.
- We added 3 new relationships between User and Game, which are “library”, “plays” and “in\_game”. “library” relationship is many to many and it has a name attribute. This way, users will be able to place their games in different libraries they want. “plays” relationship is also many to many, and it has 2 attributes; start\_date and end\_date. “in\_game” relationship is many to one.
- We used generalization and added a new entity Comment which is extended by the older entities Post and Reply. Comment has 2 attributes, id and text. We also modified the Post and Reply entities; Post has a title attribute and Reply has a depth attribute. We added a “replied” ternary relationship between User, Comment and Reply and this relationship has a date attribute. Reply entities totally participate in this relationship. We also added a “post\_of” ternary relationship between User, Post and Discussion, which has a date attribute.
- We noticed that some players can be banned from some discussions due to their inappropriate comments. We realized this situation by adding a ternary relationship “banned\_users” between an “admin” User, a “banned” User and Discussion. This relationship has a date attribute.

- We modified the “moderates” relationship. It is now between a User and a Discussion.
- We modified the “send\_invitation” relationship. It is now a ternary relationship between a “sender” User, a “receiver” User and a Game. This relationship has a date attribute.
- We removed the Friend List entity. Instead of it, we added a “friend” many to many relationship between two Users, user1 and user2. This relationship has a date attribute. We thought that the queries for getting friendships between users will be more efficient this way.
- We added a “message” many to many relationship between two Users, user1 and user2. This relationship has 2 attributes; text and date.
- Discussion was a weak entity, but we changed it to a strong entity.
- We added new attributes to Game. These attributes are; img\_location which denotes the location of the image of the game, release\_date and discount\_amount.



## 2 Relation Schemas and Normalizations

### 2.1 Relational Schemas

#### 2.1.1 Users

##### **Relational Model:**

users(user\_id, user\_name, password, email, balance, joinDate, isAdmin, isGameDev, isOnline)

##### **Functional Dependencies:**

user\_id → user\_name, password, email, balance, joinDate, isAdmin, isGameDev

email → user\_id, user\_name, password, balance, joinDate, isAdmin, isGameDev

user\_name → user\_id, password, email, balance, joinDate, isAdmin, isGameDev

##### **Candidate Key:**

{(user\_id) (email) (user\_name)}

##### **Normal Form:**

BCNF

##### **Table Definition:**

```
CREATE TABLE users (  
    user_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    user_name VARCHAR(20) NOT NULL UNIQUE,  
    password VARCHAR(45) NOT NULL,  
    email VARCHAR(255) NOT NULL UNIQUE,  
    balance DECIMAL(6,2) NOT NULL DEFAULT 0,  
    joinDate DATETIME DEFAULT NOW(),  
    isAdmin BOOLEAN NOT NULL DEFAULT FALSE,  
    isGameDev BOOLEAN NOT NULL DEFAULT FALSE,  
    isOnline BOOLEAN NOT NULL DEFAULT FALSE  
);
```

## 2.1.2 Games

### Relational Model:

games(game\_id, game\_name, price, img\_location, description, release\_date, discount\_amount)

### Functional Dependencies:

game\_id -> game\_name, price, img\_location, description, release\_date, discount\_amount

game\_name -> game\_id, price, img\_location, description, release\_date discount\_amount

### Candidate Key:

{(game\_id) (game\_name)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE games (  
    game_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    game_name VARCHAR(255) NOT NULL UNIQUE,  
    price DECIMAL(5,2) NOT NULL,  
    img_location VARCHAR(255),  
    release_date DATETIME DEFAULT NOW(),  
    description LONGTEXT,  
    publisher_id INT,  
    discount_amount DECIMAL (2,2) DEFAULT 0,  
    FOREIGN KEY (publisher_id) REFERENCES users(user_id) ON DELETE  
CASCADE  
);
```



### 2.1.3 Buy

**Relational Model:**

buy(user\_id, game\_id, date)

**Functional Dependencies:**

user\_id, game\_id -> date

**Candidate Key:**

{{user\_id, game\_id}}

**Normal Form:**

BCNF

**Table Definition:**

```
CREATE TABLE buy(  
    user_id INT NOT NULL,  
    game_id INT NOT NULL,  
    date DATETIME NOT NULL DEFAULT NOW(),  
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,  
    FOREIGN KEY (game_id) REFERENCES games(game_id) ON DELETE CASCADE,  
    PRIMARY KEY (user_id, game_id)  
);
```

## 2.1.4 In\_game

### Relational Model:

in\_game(user\_id, game\_id)

### Functional Dependencies:

None

### Candidate Key:

{{user\_id, game\_id}}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE in_game (  
    user_id INT NOT NULL,  
    game_id INT NOT NULL,  
    FOREIGN KEY (user_id) REFERENCES users (user_id) ON DELETE CASCADE,  
    FOREIGN KEY (game_id) REFERENCES games (game_id) ON DELETE CASCADE,  
    PRIMARY KEY (user_id, game_id)  
);
```

## 2.1.5 Library

### Relational Model:

library(library\_name, user\_id, game\_id)

### Functional Dependencies:

library\_name, user\_id -> game\_id

### Candidate Key:

{(library\_name, user\_id)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE library(  
    library_name VARCHAR(255) NOT NULL DEFAULT "My Games",  
    user_id INT NOT NULL,  
    game_id INT,  
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,  
    FOREIGN KEY (game_id) REFERENCES games(game_id) ON DELETE CASCADE,  
    PRIMARY KEY (user_id, library_name)  
);
```

## 2.1.6 Plays

### Relational Model:

plays(user\_id, game\_id, start\_date, end\_date)

### Functional Dependencies:

user\_id, game\_id → start\_date, end\_date

user\_id, start\_date → game\_id, end\_date

### Candidate Key:

{{(user\_id, game\_id)(user\_id, start\_date)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE plays(  
    user_id INT NOT NULL,  
    game_id INT NOT NULL,  
    start_date DATETIME DEFAULT NOW(),  
    end_date DATETIME NULL,  
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,  
    FOREIGN KEY (game_id) REFERENCES games(game_id) ON DELETE CASCADE,  
    UNIQUE(user_id, start_date)  
);
```

## 2.1.7 Genres

### Relational Model:

genres(genre\_name)

### Functional Dependencies:

None

### Candidate Key:

Candidate keys: {(genre\_name)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE genres (  
    genre_name VARCHAR(30) NOT NULL PRIMARY KEY  
);
```

## 2.1.8 Game Genres

### Relational Model:

game\_genres(game\_id, genre\_name)

### Functional Dependencies:

None

### Candidate Key:

{(game\_id, genre\_name)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE game_genres (  
    game_id INT NOT NULL,  
    genre_name INT NOT NULL,  
    FOREIGN KEY (genre_name) REFERENCES genres (genre_name) ON DELETE  
CASCADE,  
    FOREIGN KEY (game_id) REFERENCES games (game_id) ON DELETE CASCADE,  
    PRIMARY KEY (game_id, genre_name)  
);
```

## 2.1.9 Friend Request

### Relational Model:

friend\_request(sender, receiver, date)

### Functional Dependencies:

sender, receiver -> date

### Candidate Key:

Candidate keys: {(sender, receiver)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE friend_request(  
    sender_id INT NOT NULL,  
    reciever_id INT NOT NULL,  
    date DATETIME DEFAULT NOW(),  
    FOREIGN KEY (sender_id) REFERENCES users(user_id) ON DELETE  
CASCADE,  
    FOREIGN KEY (reciever_id) REFERENCES users(user_id) ON DELETE  
CASCADE,  
    PRIMARY KEY (sender_id, reciever_id)  
);
```

## 2.1.10 Friends

### Relational Model:

friends(user\_id1, user\_id2, date)

### Functional Dependencies:

user\_id1, user\_id2 → date

### Candidate Key:

Candidate keys: {(user\_id1, user\_id2)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE friends(  
    user_id1 INT NOT NULL,  
    user_id2 INT NOT NULL,  
    date DATETIME DEFAULT NOW(),  
    FOREIGN KEY (user_id1) REFERENCES users(user_id) ON DELETE CASCADE,  
    FOREIGN KEY (user_id2) REFERENCES users(user_id) ON DELETE CASCADE,  
    PRIMARY KEY (user_id1, user_id2)  
);
```



## 2.1.11 Messages

### Relational Model:

messages(message\_id, sender\_id, receiver\_id, date, text)

### Functional Dependencies:

message\_id -> sender\_id, receiver\_id, date, text

sender\_id, receiver\_id, date -> message\_id, text

### Candidate Key:

{{message\_id} (sender\_id, receiver\_id, date)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE messages (  
    message_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    sender_id INT NOT NULL,  
    reciever_id INT NOT NULL,  
    date DATETIME DEFAULT NOW(),  
    text LONGTEXT,  
    FOREIGN KEY (sender_id) REFERENCES users(user_id) ON DELETE  
CASCADE,  
    FOREIGN KEY (reciever_id) REFERENCES users(user_id) ON DELETE  
CASCADE,  
    UNIQUE(sender_id, reciever_id, date)  
);
```

## 2.1.12 Send\_invitation

### Relational Model:

send\_invitation(sender\_id, receiver\_id, game\_id, date)

### Functional Dependencies:

sender\_id, receiver\_id, game\_id -> date

### Candidate Key:

{{sender\_id, receiver\_id, game\_id}}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE send_invitation(  
    sender_id INT NOT NULL,  
    reciever_id INT NOT NULL,  
    game_id INT NOT NULL,  
    date DATETIME DEFAULT NOW(),  
    FOREIGN KEY (sender_id) REFERENCES users(user_id) ON DELETE  
CASCADE,  
    FOREIGN KEY (reciever_id) REFERENCES users(user_id) ON DELETE  
CASCADE,  
    FOREIGN KEY (game_id) REFERENCES games(game_id) ON DELETE CASCADE,  
    PRIMARY KEY (game_id, sender_id, reciever_id)  
);
```

## 2.1.13 Discussions

### Relational Model:

discussions(discussion\_id, discussion\_name, game\_id)

### Functional Dependencies:

{{discussion\_id} (discussion\_name, game\_id)}

### Candidate Key:

Discussion\_id -> discussion\_name, game\_id

Discussion\_name, game\_id -> discussion\_id

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE discussions(  
    discussion_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    discussion_name VARCHAR(30) NOT NULL,  
    game_id INT NOT NULL,  
    FOREIGN KEY (game_id) REFERENCES games(game_id) ON DELETE CASCADE,  
    UNIQUE(game_id, discussion_name)  
);
```

## 2.1.14 Moderates

### Relational Model:

moderates(user\_id, discussion\_id)

### Functional Dependencies:

None

### Candidate Key:

{{user\_id, discussion\_id}}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE moderates(  
    user_id INT NOT NULL,  
    discussion_id INT NOT NULL,  
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,  
    FOREIGN KEY (discussion_id) REFERENCES discussions(discussion_id)  
ON DELETE CASCADE,  
    PRIMARY KEY (user_id, discussion_id)  
);
```

## 2.1.15 Banned\_users

### Relational Model:

banned\_users(banned\_user\_id, moderator\_id, discussion\_id)

### Functional Dependencies:

None

### Candidate Key:

{(banned\_user\_id, moderator\_id, discussion\_id)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE banned_users (  
    banned_user_id INT NOT NULL,  
    moderator_id INT NOT NULL,  
    discussion_id INT NOT NULL,  
    date DATETIME DEFAULT NOW(),  
    FOREIGN KEY (banned_user_id) REFERENCES users(user_id) ON DELETE  
CASCADE,  
    FOREIGN KEY (moderator_id) REFERENCES users(user_id) ON DELETE  
CASCADE,  
    FOREIGN KEY (discussion_id) REFERENCES discussions(discussion_id)  
ON DELETE CASCADE,  
    PRIMARY KEY (banned_user_id, moderator_id, discussion_id)  
);
```

## 2.1.16 Comments

### Relational Model:

comments(comment\_id, text)

### Functional Dependencies:

comment\_id → text

### Candidate Key:

{(comment\_id)}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE comments(  
    user_id INT NOT NULL,  
    comment_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    date DATETIME DEFAULT NOW(),  
    text LONGTEXT NOT NULL  
);
```

## 2.1.17 Posts

### Relational Model:

posts(comment\_id, title)

### Functional Dependencies:

comment\_id -> title

### Candidate Key:

{{comment\_id}}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE posts(  
    title varchar(255) NOT NULL,  
    comment_id INT NOT NULL PRIMARY KEY,  
    discussion_id INT NOT NULL,  
    FOREIGN KEY (comment_id) REFERENCES comments(comment_id) ON DELETE  
    CASCADE  
);
```

## 2.1.18 Replies

### Relational Model:

replies(comment\_id, post\_id)

### Functional Dependencies:

comment\_id -> post\_id

### Candidate Key:

{{comment\_id}}

### Normal Form:

BCNF

### Table Definition:

```
CREATE TABLE replies(  
    comment_id INT NOT NULL PRIMARY KEY,  
    parent_id INT NOT NULL,  
    post_id INT NOT NULL,  
    FOREIGN KEY (comment_id) REFERENCES comments(comment_id) ON DELETE  
CASCADE,  
    FOREIGN KEY (post_id) REFERENCES comments(comment_id) ON DELETE  
CASCADE  
);
```



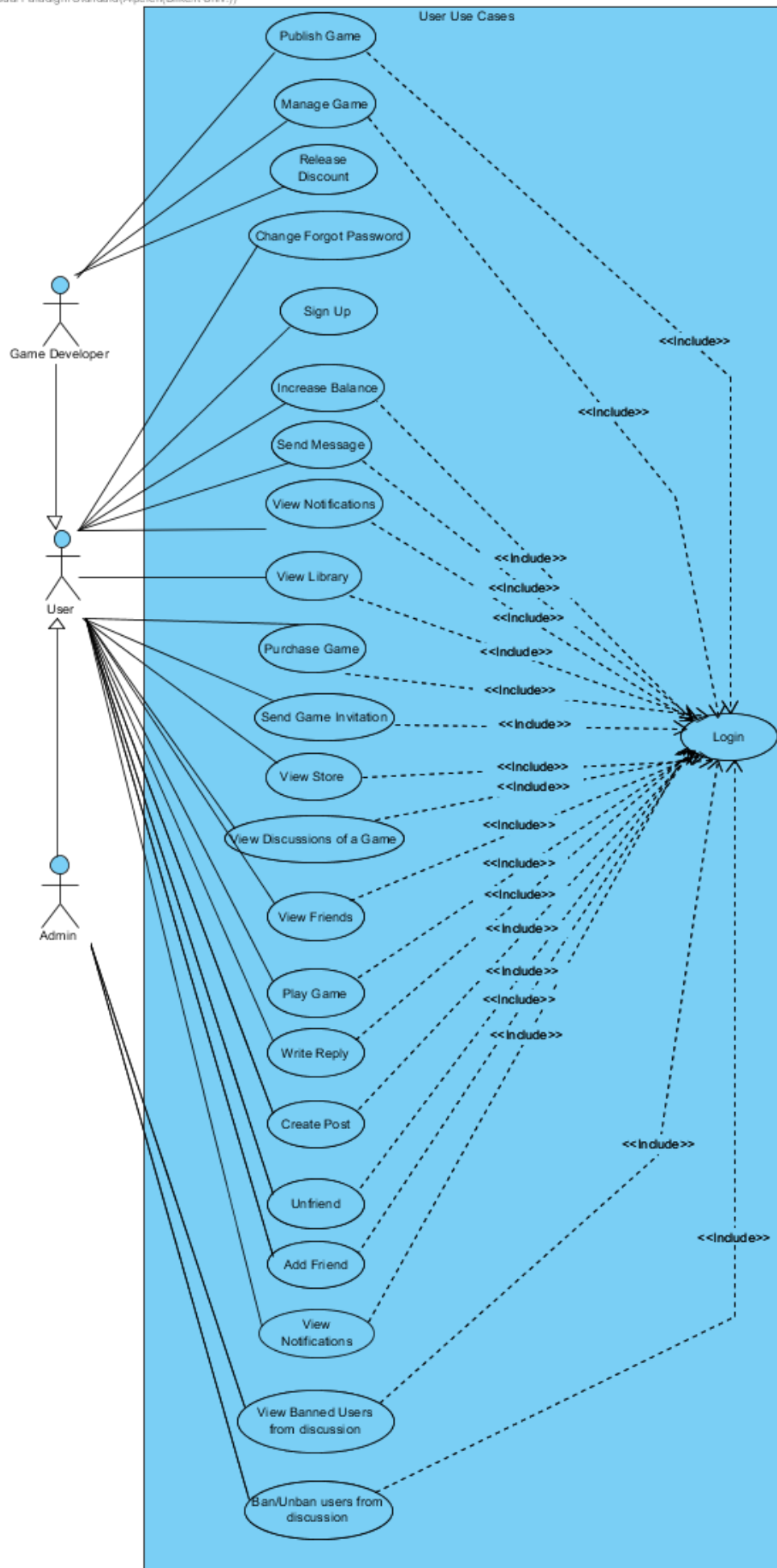
## 2.2 Normalization

Our relational schemas were already shown above and they were already prepared in Boyce-Codd Normal Form. Hence, there is no need for further explanations about normalizations of the tables.

## 3 Functional Components

### 3.1 Use Cases / Scenarios

In our program, MIST, there are 3 types of users: User, admin, game developer. In order to use the system, all users have to log-in. There are similarities and differences between each user type. Admins and game developers are able to do what every user can. But they have more privilege. We did not write common cases in Admin and Game Developer for simplicity however details of these privileges will be explained in each part.



### 3.1.1 User

1. **Sign Up:** Users can create a new account by giving a unique nickname, email address, along with a password.
2. **Login:** Users can login to the system by entering either their nickname or email address, along with their password.
3. **Change Forgotten Password:** Users can change their password in case they forget their password by clicking on the “Forgot Password” button. By so, the system will ask their email address and a password retrieval email will be sent to the indicated email address.
4. **View Library:** Users will see their libraries as soon as they log in to the system.
5. **Purchase Game:** Users can buy games from store and add them to their desired library. It is performed by selecting a specific game and clicking buy button. If the user has enough balance, then the game is purchased.
6. **View Discussions of a Game:** Users can open each game’s discussion by clicking on the game and pressing “Discussion” button. Then a pop-up will appear with a list of discussions-that is: “Technical Issues” , “Gameplay Issues”, “Guides & Tips”, “General”. Users can then select a discussion.
7. **Create Post:** Users can create a post message by clicking “New Post” button or opening a post by clicking the name of the post in their selected discussion.
8. **Write Reply:** Users can write a reply to a post by clicking “reply” button under the desired post.
9. **Play Game:** Users can open their game, while they are in library window, by selecting a game and clicking “play” button.
10. **View Friends:** Users can see their friend list by clicking on the “friends” tab. Friends who are online will be seen in green font and friends who are offline will be seen in red font. Inside the friends tab, a user can click on a particular friend’s nickname, which will display 4 buttons. These buttons are “invite to game” button, “direct message” button, the “friend’s libraries” and the “unfriend” button. If user is not in a game, the “game invitation” button will be colorless and unclickable.
11. **Add Friend:** A user can send a friend request to another user by clicking on the “add friend” button. The requested user can accept or decline the invitation.
12. **Send Game Invitation:** A user can send game invitation to his friend by clicking on the “game invitation” button. Then, a pop-up will appear which says “Invitation sent”.

13. **Send Message:** When “send message” button is clicked, a new pop-up will open in which the user can write the message and then send it by clicking the “send” button.
14. **Unfriend:** When user clicks the unfriend button a confirmation pop-up will appear to the user.
15. **View Store:** Users can click on the “store” tab, which will open the store. Inside the store, users can buy games as long as their balance is enough for the purchase.
16. **Increase Balance:** Users can increase their balance by first clicking on “Increase Balance” button which will allow them to proceed on increasing balance by entering their credit card information.
17. **View Notifications:** Each user will have a notification center which will show any new messages, game invitations and replies to his/her posts. A user can see how many new notifications he/she has with the help of the notification button which can be seen in each tab. When the user clicks this button, he/she will see each notification in a new tab.

### 3.1.2 Admin

1. **Ban/Unban Users From a Discussion:** Admins can ban/unban users from the discussions they moderate. They can see a special button under each post which can be used to ban users who have posted under this specific post.
2. **View Banned Users:** Admins can see the list of users who are banned from the discussions they moderate. They can see the list of banned users by clicking a “banned users” button which is located in discussion page. This button can only be seen by admins of that specific discussion.

### 3.1.3 Game Developer

1. **Manage Published Games:** Game developers can access their published games in a page called “Manage Games”, to which they can access by clicking a button in the store page that is only visible to the game developers. In the “Manage Games” page, the game developers can see their published games as a list and each list item will show a game. Then, game developers can select a game and click “manage” button to change specifications of that game.

2. **Publish a Game:** Game developers can publish their games in “Manage Games” page. On this page, game developers can click “Publish new game” button to publish a new game. Publishing a game will consist of entering a title for the game, a URL for the photo of the game, a description text and a price, which are all obligatory for the publishment of a game.
3. **Release Discount:** Game developer can release a discount for one or every game that was released by him/her. The amount will be set as a percentage for the current price.

## 3.2 Algorithms

### 3.2.1 Game-Related Algorithms

In order to be able to play a game, a user has to buy the game first. After the user buys a game, some amount of money which is equal to the price of the game will be subtracted from the balance of the user. The operations of putting the game to the main library of the user and subtracting the price of the game from user's balance will be performed atomically, so that if an error occurs in the system during the buying process of a game, the system will make sure that either both or none of these operations are performed. After buying the game, the user will be able to play the game whenever he/she wants and he/she will be able to place this game in other libraries.

### 3.2.2 User-Interaction Algorithms

Different users which use MIST will be in interaction with each other. There are different kinds of interactions such as sending friend requests, sending game invitations, sending messages and posting comments on discussions.

When a user sends a friend request to another user, a notification will appear on the screen of the receiving user. The receiving user will be able to accept or reject the friend request. If he/she accepts, these users will be added as friends to the friends table. If he/she rejects, the specific request will be removed from the friend\_request table.

If two users have the same game in their libraries, then a user can send an invitation to play the game together to the other one. This game invitation will be inserted to the send\_invitation table and it will appear on the homepage of the receiving user as a notification. If the receiving user accepts this invitation, then these users will be able to play the game together.

Users can send text messages to each other in MIST. A user does not have to be friends with another user to send a message. These messages will be displayed in the inbox of the receiving user.

There will be several discussions about a game. These discussions will be moderated by an admin. Users will be able to share their comments in these discussions and other users will be able to reply to them. These comments can be deleted by the user himself/herself who posted or the admin of the discussion. There will also be replies of replies. If a user is banned from a discussion, then he/she will not be able to make a comment on that discussion again and a row about that particular user and discussion will be added to the banned\_users table.

### 3.2.3 Logical Requirements

Logical errors can occur in database systems and they are often caused by attributes which are responsible to hold dates. Our database system should prevent logical errors in order to have accurate information. This can be done by checking boundary dates of events carefully.

The start\_date and end\_date attributes of the plays table show the start and end times of a user's playing time for a game. Therefore, the end\_date should be later than the start\_date. If the start\_date is not empty but the end\_date is empty, then it means that the user is still playing that game.

Moreover, date of a reply should be later than the comment which the reply is for. Otherwise, the reply would be irrelevant.

Several tables of the system have date attributes. These tables are friends, friend\_request, send\_invitation, messages, comments, banned\_users and buy. The date values of these tables should have proper values. For example, a user cannot send a game\_invitation to another user before being friends with him/her. Therefore, date of the friends table should be earlier than the date of send\_invitation table. Another example is that, date of the friend table should be later than date of the friend\_request table for 2 specific users, since one user should add the other one first and then they will become friends when the receiving user accepts the friend invitation. Similar situations should be managed properly in order to keep accurate and reasonable data in the database system.



### 3.3 Data Types

For the attribute domains we use Numeric type, Data-Time type and String type data types of mySQL.

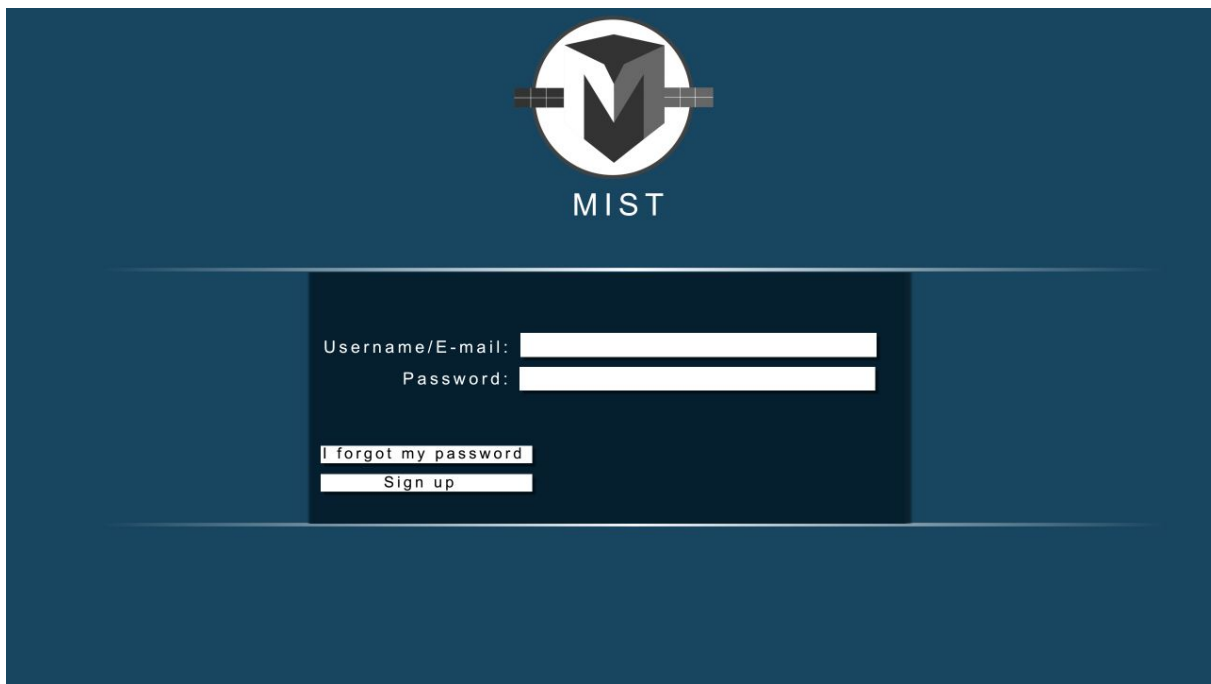
We used numeric types to store numeric data such as id numbers, prices and balances. We used **INT**, **DECIMAL** and **BOOLEAN** by considering necessary ranges for attributes and adjusting the storage spaces accordingly. For instance, we used **BOOLEAN** values for true/false conditions and we used **DECIMAL** for discount percentages of games.

We used string types to store attributes which are consisted of characters. We used **VARCHAR** and **CHAR** for storage of strings.

To store dates and times of events, we used **DATETIME**.

## 4 User Interface Design & Corresponding SQL Statements

### 4.1 Login

The image shows a login interface for a system named MIST. At the top center is a logo consisting of a stylized 'M' inside a circle, with the word 'MIST' written below it. Below the logo is a dark blue rectangular box containing the login form. The form has two input fields: 'Username/E-mail:' and 'Password:'. Below these fields are two buttons: 'I forgot my password' and 'Sign up'.

**Input:** @email\_username, @password, @user\_name, @email

**Process:** Once user opens the MIST, he/she will be welcomed with the login screen. If he/she forgot what his/her password was, it will be always available to retrieve/change it through “I forgot my password” button. Similarly user can sign-up for the system after pressing “Sign up” button. After logging in, the user will be forwarded to her/his default library called “My games”.

## SQL Statements:

### Login User:

```
/*
    Login Screen
    Inputs: @email_username, @password
*/

SELECT user_id
FROM users
WHERE email = @email_username OR user_name = @email_username AND
password = @password
;
```

### Sign-Up New User:

```
/*
    Sign-up Screen
    Inputs: @user_name, @password, @email
*/

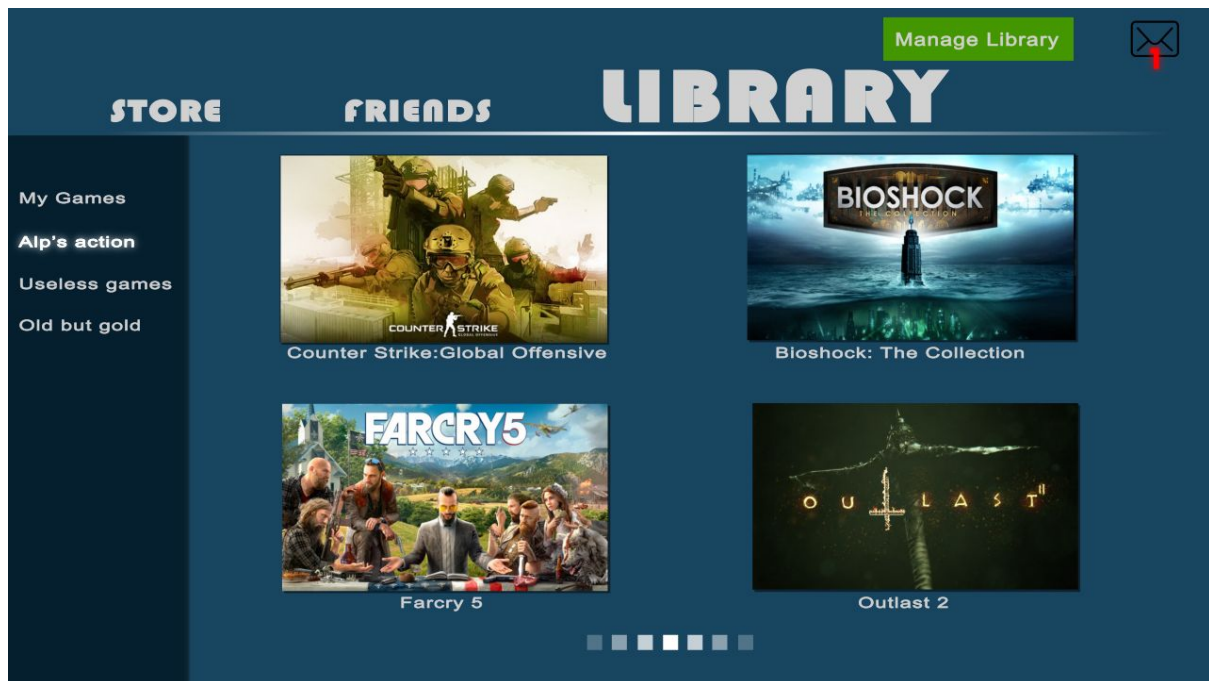
INSERT INTO users (user_name, password, email) VALUES (
    @user_name,
    @password,
    @email
);
```

### Changing Password:

```
/*
    Changing Password
    Inputs: @users_id, @password
*/

UPDATE users
SET password = @password
WHERE user_id = @user_id
;
```

## 4.2 Library



**Inputs:** @user\_id, @library\_name

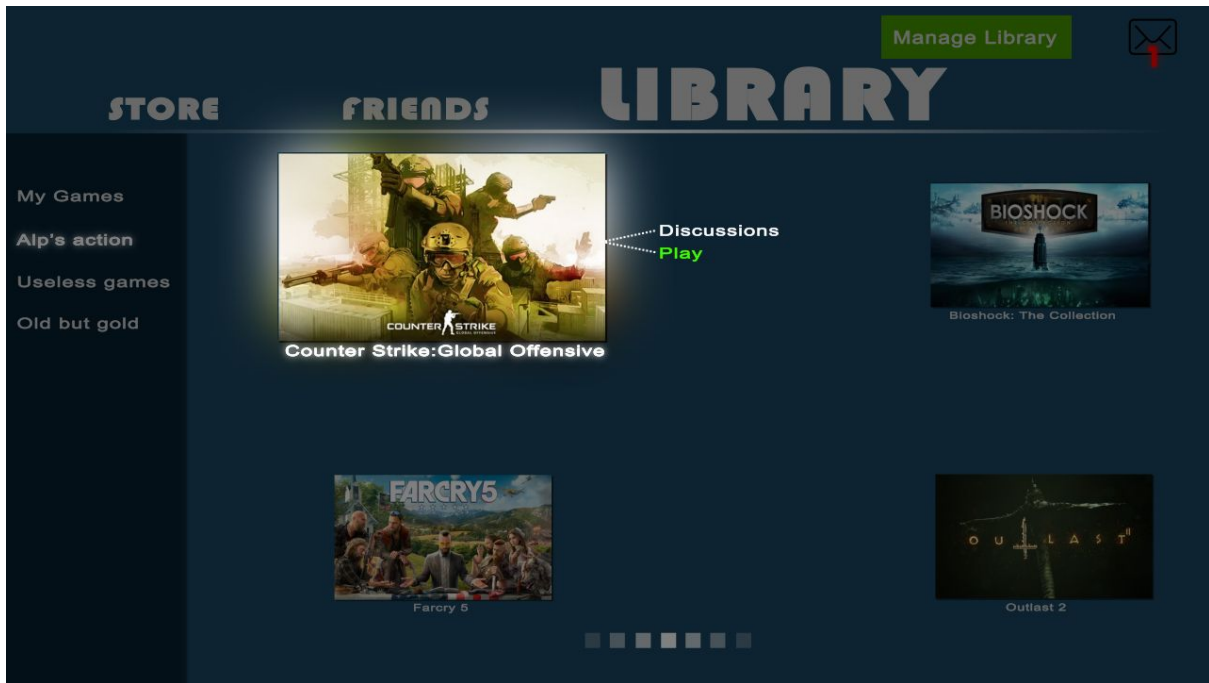
**Process:** In the library tab, a user can select his/her different libraries and see what games are in them. Once user clicks on his/her favorite game, different buttons will appear.

**SQL Statement:**

**Viewing games:**

```
/*
    Viewin games in library
    Inputs: @user_id, @library_name
*/

SELECT game_name, img_location
FROM games_in_library
WHERE user_id = @user_id AND library_name = @library_name
ORDER BY game_name
```



**Input:** @user\_id, @game\_id

**Process:** Once the user clicks on his/her favorite game, two additional button will appear. If the user wants to open discussions about the game, he/she can click on “Discussions” button. If he/she wishes to play the game, “Play” button will start the game.

**SQL Statements:**

**Starting the Game:**

```

/*
    Starting to Play a Game
    Inputs: @user_id, @game_id
*/

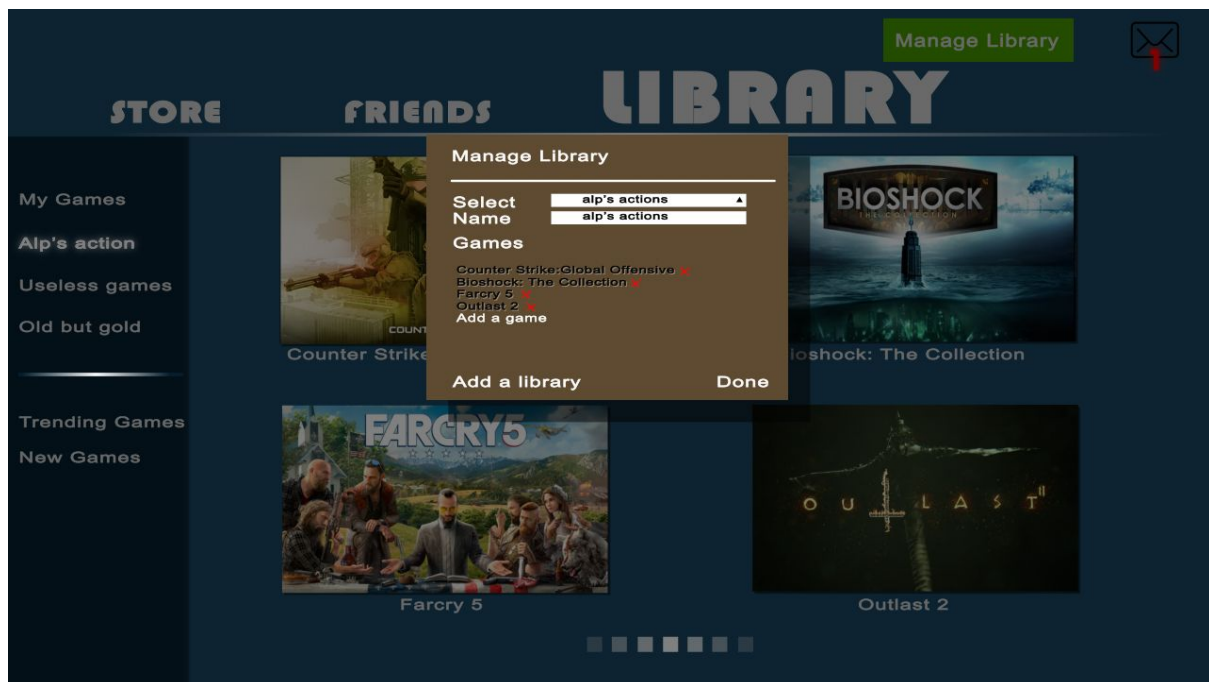
INSERT INTO plays (user_id, game_id) VALUES (
    @user_id,
    @game_id
);

```

Exiting the Game:

```
/*  
    Exiting From Game  
    Inputs: @user_id, @game_id  
*/
```

```
UPDATE plays  
SET end_date = NOW()  
WHERE user_id = @user_id AND  
      game_id = @game_id AND  
      end_date IS NULL  
;
```



**Input:** @library\_name, @game\_id, @user\_id

**Processes:** If the user clicks on “Manage Library” button, a pop-up screen will appear in which the user will be able to manage current games or add an additional library and game or deleting an existing library except than the default library “My Games”.

## SQL Statements:

### Adding a New Library:

```
/*
    Adding a New Library
    Inputs: @library_name, @user_id
*/
INSERT INTO library (user_id, library_name, game_id) VALUES (
    @user_id,
    @library_name
    NULL
);
```

### Deleting a Library:

```
/*
    Deleting a Library
    Inputs: @library_name, @user_id
*/
DELETE FROM library
WHERE user_id = @user_id, library_name = @library_name
```

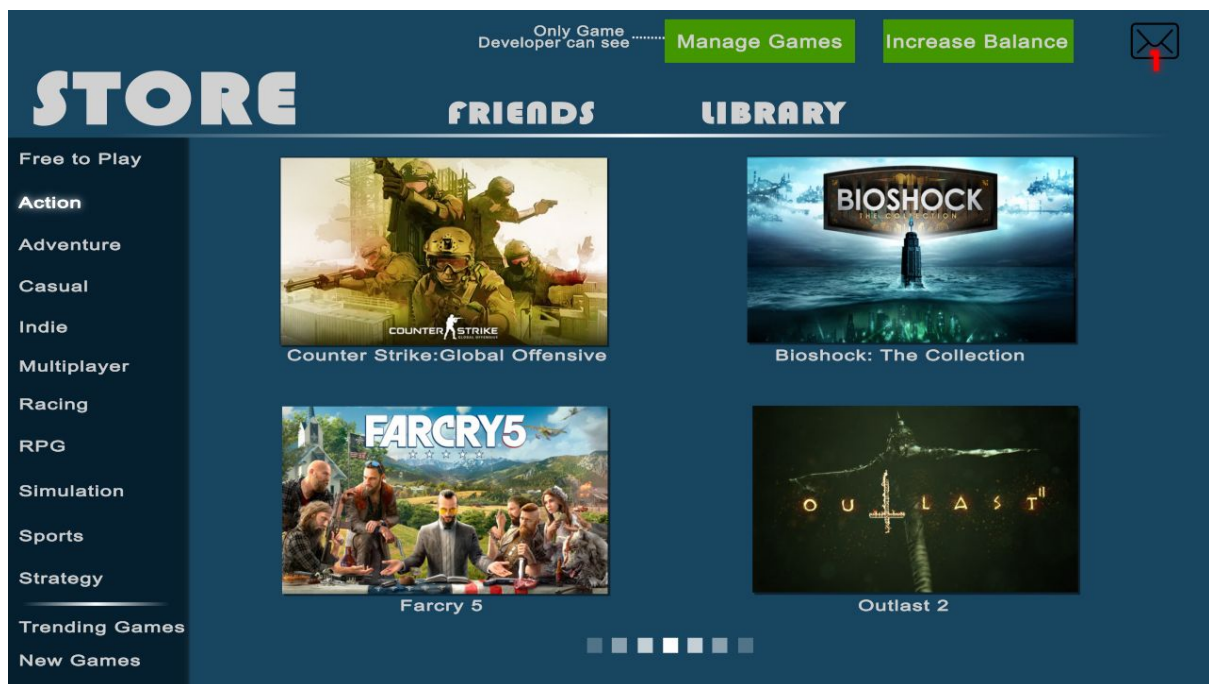
### Adding a Game to Existing Library

```
/*
    Adding a Game to Library
    Inputs: @library_name, @user_id, @game_id
*/
INSERT INTO library (user_id, library_name, game_id) VALUES (
    @user_id,
    @library_name
    @game_id
);
```

## Deleting a Game to Existing Library

```
/*
    Deleting a Game to Library
    Inputs: @library_name, @user_id, @game_id
*/
DELETE FROM library
WHERE library_name = @library_name AND user_id = @user_id AND game_id =
@game_id
```

## 4.3 Store



**Input:** @genre\_name

**Process:** Users can view different games belonging to different genres by clicking genre list on the left side. After selection of a genre, user can click on bottom boxes to navigate through games. Additionally user can see “Trending Games” and “New games” by clicking relative buttons.



## SQL Statements:

### Games by Genre:

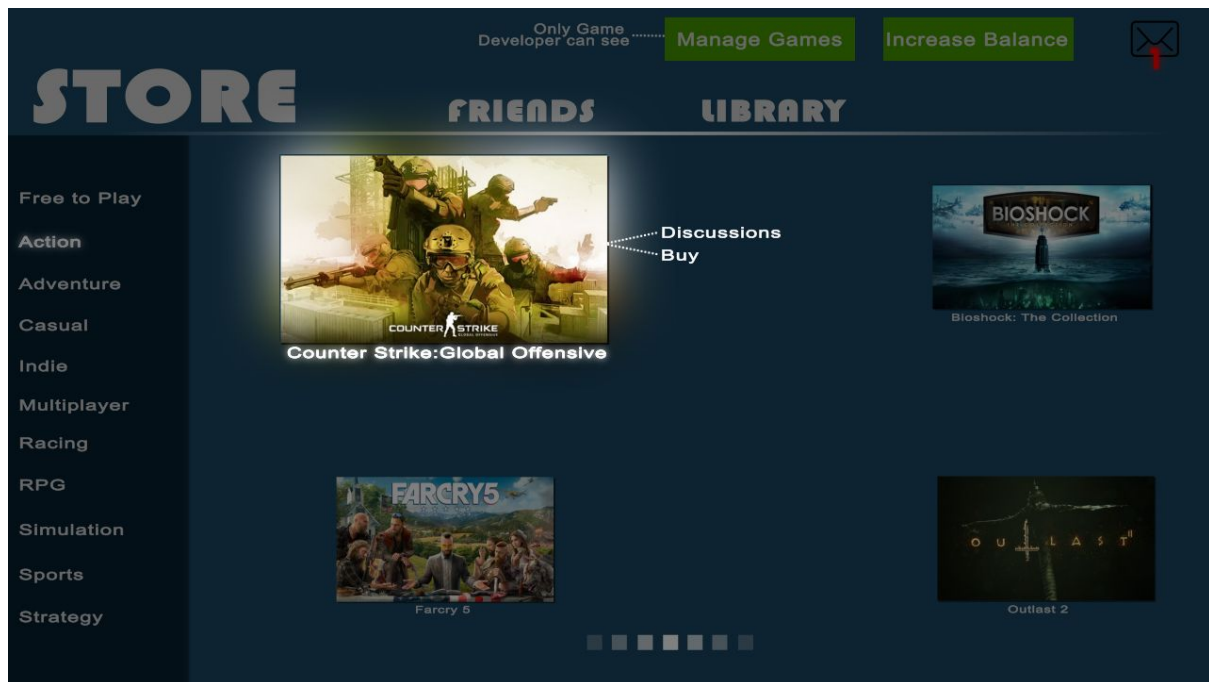
```
/*  
    Games by Genre  
    Inputs: @genre_name  
*/  
  
SELECT game_name, img_location  
FROM games_genre  
WHERE genre_name = @genre_name  
ORDER BY game_name;
```

### New Games:

```
/*  
    New Games  
    Inputs: NONE  
*/  
  
SELECT game_name, img_location  
FROM games  
ORDER BY release_date DESC
```

### Trending Games:

```
/*  
    Trending Games  
    Inputs: NONE  
*/  
  
SELECT *  
FROM trending
```



**Input:** @user\_id, @game\_id, @amount

### Process:

When the user selects a game, he/she will see 2 choices. Either user can open a game's discussion by clicking on it or user can buy the game. Once user clicks on "Buy" button, a pop-up will appear and he/she will see his/her balance. If it's enough, game will be available to buy. However, if the balance is not enough, the user can increase his/her account balance by clicking on "Increase Balance" button.

### SQL Statements:

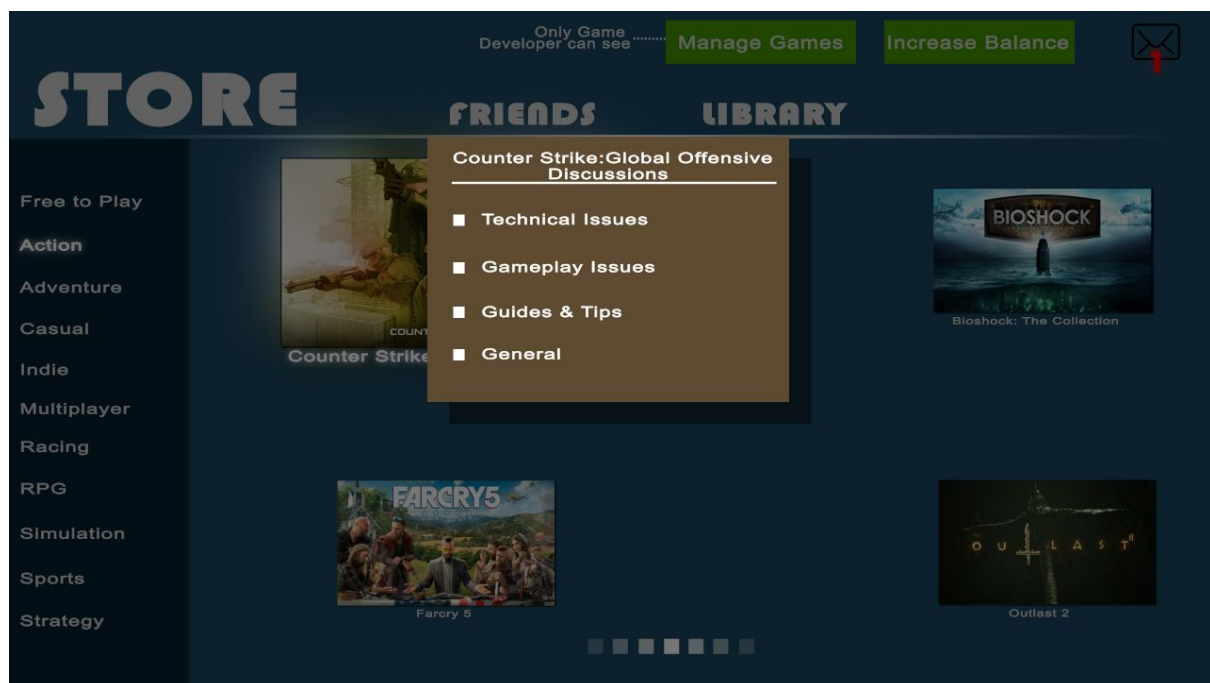
Buying Game:

```
/*
    Buy Game
    Inputs: @user_id, @game_id
*/

INSERT INTO buy (user_id,game_id) VALUES (
    @user_id,
    @game_id
);
```

Increasing the Balance:

```
/*  
    Increase balance  
    Inputs: @user_id, @amount  
*/  
  
UPDATE users  
SET balance = balance + @amount  
WHERE user_id = @user_id  
;
```



**Input:** @game\_name, @price, @img\_location, @publisher\_id, @game\_id, @genre\_name, @description, @genre\_name

**Process:** If user clicks on “Discussions” button either in Store or Library, a pop-up will appear with a list of discussions. Once he/she chooses one, he/she will be forwarded to that discussion page. Game Developers will see an additional button called “Manage Games” which will open the game management page. In this page, there will be another button called “Publish a New Game”. This button will open up a page where the developer has to enter a name, description, an image URL and a price tag for the new game.

## SQL Statements:

### Publish a Game:

```
/*
    Publish a Game
    Inputs: @game_name, @price, @img_location, @publisher_id,
    @description
*/

INSERT INTO games (game_name, price, img_location, description,
publisher_id) VALUES (
    @game_name,
    @price,
    @img_location,
    @description,
    @publisher_id
);
```

### Adding a Genre to a Game:

```
/*
    Add Genres to Published Game
    Inputs: @game_id, @genre_name
*/

INSERT INTO game_genres (game_id, genre_name) VALUES (
    @game_id,
    @genre_name
);
```

## 4.4 Discussion



**Input:** @user\_id, @input\_title, @input\_text, @type, @discussion\_id, @parent\_id, @post\_id

**Process:** In discussion page, the user will able to make a new comment or click the ones that are already posted. Admins will able to see additional "Remove Post" button on top of each post.

Viewing Discussion of a Game:

```
/*
    Viewing a discussion of the game
    Inputs: @discussion_id
*/
SELECT title, user_name, date
FROM posts_in_discussion
WHERE discussion_id = @discussion_id
```



**Input:** @banned\_id, @moderator\_id, @discussion\_id, @post\_id

**Proccess:** In a post, user can reply to many posts by clicking the “Reply” button. For admins, there will be 2 additional buttons that will be called “Ban User”, “Show Banned Users” in which they will be able to ban an user or unban him/her. When a user is replied to, he/she will get notification on top of the right side of the screen indicating the reply.

### SQL Statements:

Viewing Replies of a Post:

```
/*
    Viewing a Replies of a post
    Inputs: @post_id
*/
SELECT title, user_name, date
FROM replies_of_post
WHERE post_id = @post_id
```

### Banning Users:

```
/*  
    Ban Users  
    Inputs: @banned_id, @moderator_id, @discussion_id  
*/  
  
INSERT INTO banned_users (banned_user_id, moderator_id, discussion_id)  
VALUES (  
    @banned_id,  
    @moderator_id,  
    @discussion_id  
);
```

### UnBanning Users:

```
/*  
    UnBan Users  
    Inputs: @banned_id, @discussion_id  
*/  
  
DELETE FROM banned_users  
WHERE discussion_id = @discussion_id, banned_id = @banned_id  
;
```

## 4.5 Friends



**Input:** @user\_id1, @user\_id2, @sender\_id, @reciever\_id, @game\_id, @text

**Process:** In friends tab, users will be able to see whether their friends are online, offline or in a game with the name of the game. If they click on a friend's name, they'll be able to see 3 buttons called "send invitation", "message" and "unfriend". Additionally, if they click "add a friend" button, by typing the nickname of the user, they will be able to add a new friend. Once a user is sent friend request, he/she will get a notification on top of the right side of the screen in which they will be able to decline or accept.

### SQL Statements:

Adding a Friend:

```
/*
    Add Friends
    Input: @user_id1, @user_id2
*/
INSERT INTO friends (user_id1,user_id2) VALUES (
    user_id1,
    user_id2
);
```



### Sending Game Invitation:

```
/*  
    Send Game Invitation  
    Inputs: @sender_id, @reciever_id, @game_id  
*/  
  
INSERT INTO send_invitation (sender_id, reciever_id, game_id) VALUES (  
    @sender_id,  
    @reciever_id,  
    @game_id  
);
```

### Sending Message to Another User:

```
/*  
    Send Message  
    Inputs: @sender_id, @reciever_id, @text  
*/  
  
INSERT INTO messages (sender_id, reciever_id, text) VALUES (  
    @sender_id,  
    @reciever_id,  
    @text  
);
```

### Unfriending a Friend:

```
/*  
    Unfriend  
    Inputs: @user_id1, @user_id2  
*/  
  
DELETE FROM friends  
WHERE (user_id1 = @user_id1 AND user_id2 = @user_id2) OR  
      (user_id1 = @user_id2 AND user_id2 = @user_id1)  
;
```

## 4.6 Activity



**Inputs:** @user\_id

**Process:** Once user clicks on “Activity” tab, he/she will able to see social or gaming activities for past 7 days.

## SQL Statements:

### Viewing Social Activity:

```
/*
    Viewing Social Activity
    Inputs: @user_id
*/
WITH friends_ids AS ((SELECT user_id2 as friend_id
    FROM friends
    WHERE user_id1 = @user_id
    ) UNION
    (SELECT user_id1 as friend_id
    FROM friends
    WHERE user_id2 = @user_id
    ))

SELECT user_id1 as friend, user_id2 as friends_friend
FROM friends f, friends_ids ff
WHERE (f.user_id1 = friend_id OR f.user_id2 = friend_id) AND
    date >= (DATE(NOW()) - INTERVAL 7 DAY)
```

;

### Viewing Game Activity:

```
/*
    Viewing Game Activity
    Inputs: @user_id
*/
WITH friends_ids AS ((SELECT user_id2 as friend_id
    FROM friends
    WHERE user_id1 = @user_id
    ) UNION
    (SELECT user_id1 as friend_id
    FROM friends
    WHERE user_id2 = @user_id
    ))

SELECT game_id, f.friend_id, start_date, end_date
FROM plays p NATURAL JOIN friends_ids f on p.user_id = f.user_id
WHERE end_date >= (DATE(NOW()) - INTERVAL 7 DAY);
```

# 5 Advanced Database Components

## 5.1 Views

### 5.1.1 Games In User Library View

This view contains all the existing libraries with their games in system.

```
CREATE VIEW games_in_library AS
    SELECT games.name, library.img_location
    FROM library l natural join games g on g.games_id =
l.games_id
```

### 5.1.3 Posts In A Discussion View

This view stores all posts from discussions for future use.

```
CREATE VIEW posts_in_discussion AS
    SELECT posts.title as title, comments.text as text,
discussion_id, user_name, c.date as date
    FROM comments c natural join posts p on p.comment_id =
c.comment_id NATURAL JOIN users u on u.user_id = c. user_id
    ORDER BY comments.date DESC
```

### 5.1.4 Replies of Posts View

This view stores all replies of posts for future use.

```
CREATE VIEW replies_of_post AS
    SELECT comment_id, text, user_name, c.date as date, post_id
    FROM replies r natural join comments c on r.comment_id =
c.comment_id NATURAL JOIN users u on u.user_id = c.user_id
    ORDER BY comments.date DESC
```

### 5.1.5 Trending Games View

When a user wants to display popular games among the users of MIST, a list of most sold games of past week will be shown.

```
CREATE VIEW most_sold AS
    SELECT game_id, COUNT(user_id) as units_sold
    FROM buy
    WHERE date >= (DATE(NOW()) - INTERVAL 7 DAY)
    ORDER BY units_sold DESC
;
```

```
CREATE VIEW trending AS
    SELECT game_name, img_location
    FROM most_sold m NATURAL JOIN games g
;
```

### 5.1.6 Games by Genre

This view stores games with their genres.

```
CREATE VIEW games_genre AS
SELECT game_name, img_location, genre_name
FROM games g NATURAL JOIN genres gg on g.game_id = gg.game_id
ORDER BY genre_name
;
```

## 5.2 Reports

### 5.2.1 Total Number of Users

For further implementations, it is good to create a query to generate the current number of users. This report is written for this kind of situation in advance.

```
SELECT count(user_id) AS no_user
FROM users
```

### 5.2.2 Total Number of Games

The number of games that are shared in the system will be computed with this report, for further possible computations.

```
SELECT count(game_id) AS no_games
FROM games
```

### 5.2.3 Total Number of Purchased Games

The number or purchase records can be computed via this report. It may as well be helpful for statistical purposes.

```
SELECT count(game_id) AS no_purchase
FROM buy
```

### 5.2.4 Top 10 Purchased Games

The trending games page will use this report in order to display the most purchased games.

```
SELECT *
FROM game_purchase_count
LIMIT 10;
```

### 5.2.5 Number of Users in Game

The number of currently in-game users can be computed using this report which can also be used for statistical purposes.

```
SELECT count(user_id) AS count_in_game
FROM in_game
```

### 5.2.6 Number of Users That Are Playing a Specific Game

This report will compute each game's currently in-game user number. This data will be useful to find the currently most anticipated games.

```
SELECT game_id, count(user_id) AS number_of_players
FROM in_game I
WHERE I.game_id = @game_id
```

### 5.2.7 Total Online Users

This report will show the number of users that are currently online in our system, which can be constantly displayed similar to other marketplace systems.

```
SELECT count(user_id) AS online_no
FROM users U
WHERE U.is_online = TRUE
```

### 5.2.8 Total Money Spent

This report will compute the total money made by the system, which can also be used for statistical purposes.

```
SELECT count(G.price) AS total_spent
FROM buy B NATURAL JOIN Game G
```

### 5.2.9 Number of Posts In a Discussion

Number of posts in a discussion can be shown before the user enters in a specific discussion, which can be seen at many discussion pages in similar systems.

```
SELECT count(*) AS post_count
FROM posts_of_discussion
WHERE comment_id = @discussion_comment_id
```

### 5.2.10 Number of Replies in a Post

Because of the same reason as the previous section (5.2.9), the number of replies in a specific post is computed.

```
SELECT count(*) AS reply_count
FROM replies_of_post
WHERE comment_id = @post_comment_id
```

### 5.2.11 Total Time Spent In a Specific Genre

```
SELECT genre_name, sum(p.end_date - p.start_date)
FROM games_genre
GROUP BY genre
```

### 5.2.11 Total Number of Comments a User Posted

```
SELECT user_id, sum(comment_id)
FROM comments
GROUP BY user_id
```



### 5.2.12 Total Number of Banned Users Per Genre

```
SELECT genre, sum(user_id)
FROM discussions s NATURAL JOIN game g ON s.game_id = g.game_id NATURAL
JOIN banned_users b ON s.discussion_id = b.discussion_id
GROUP BY g.genre
```

## 5.3 Triggers

- When a user buys a game, that game will be added to default library of the user called “My games”.
- When a user starts a game, in\_game and plays relations will be updated according to the start date.
- When user exits a game, the related tuple’s end date in plays relation will be updated. Also, the related tuple will be deleted from in game relation.
- When a friend request is accepted, it will be deleted from friend\_request relation and the sending and receiving users will be added to friends relation. If rejected, it will only be deleted from friend\_request relation.
- User’s balance will be reduced after a successful purchase.

## 5.4 Constraints

- The system cannot be used without logging in.
- Each user’s library names must be different from each other.
- Users can not buy a game if they don’t have enough balance.
- Plays relation’s end\_date cannot be earlier than start\_date
- Banned users cannot create a new post or reply to a post.
- Each game’s name must be unique.
- A user can only send game invitations to the their friends.
- Library names cannot be blank.
- Post or reply messages cannot be blank.
- A game developer can only edit the games that he/she published.
- A user cannot delete a game from default library “My games”.
- A user cannot add a game twice into same library.

## 5.5 Stored Procedures

### 5.5.1 Releasing a Discount

We will use a stored procedure to discount a given game. This procedure takes two inputs. “game\_id” for determining the game to apply discount and “amount” the amount of discount (a value between 0 and 1) to be applied. By this way, discounts will be performed more easily.

#### SQL Code:

```
DELIMITER $$
CREATE PROCEDURE activate_discount (
    IN game_id INT,
    IN amount DECIMAL(2,2)
)
BEGIN
    UPDATE games
    SET price = price - (price * amount), discount_amount = amount
    WHERE game_id = input_game_id

END $$
DELIMITER ;
```

### 5.5.2 Restoring the Price of a Discounted Game

A stored procedure will be used in order to easily restore the discounted game's price.

#### SQL Code:

```
DELIMITER $$
CREATE PROCEDURE restore_price (
    IN game_id INT
)
BEGIN
    UPDATE games
    SET price = price / (1 - discount_amount), discount_amount = 0;
END $$
DELIMITER ;
```

### 5.5.3 Addition of Posts and Replies

A stored procedure will be used in order to add posts and replies to a discussion. The type of comment is chosen by input “type”; if type is ‘p’ then this comment is a post and input arguments: input\_title and discussion\_id must also be provided correctly. If the type is not p then this comment is a reply to another post or reply, input arguments: parent\_id and post\_id must be provided correctly.

#### SQL Code:

```
DELIMITER $$

CREATE PROCEDURE insert_comment (
    IN user_id INT,
    IN input_title VARCHAR(255),
    IN input_text LONGTEXT,
    IN type VARCHAR(1),
    IN discussion_id INT,
    IN parent_id INT,
    IN post_id INT
)
BEGIN
    DECLARE id INT;

    INSERT INTO comments(user_id, text) VALUES (user_id,
input_text);

    SET id = LAST_INSERT_ID();

    IF (type = 'P' OR type = 'p') THEN
        INSERT INTO posts(comment_id,title,discussion_id)
VALUES (id,input_title,discussion_id);
    ELSE INSERT INTO replies(comment_id,parent_id,post_id)
VALUES (id,parent_id,post_id);
    END IF;

END $$

DELIMITER ;
```

## 6 Implementation Plan

We are going to use MySQL Server for our project's database management system. For front-end, application logic and user interface; PHP, JavaScript, CSS, HTML and Bootstrap will be used. For JavaScript object manipulation, we will mostly use JQuery.

## 7 Website

<https://github.com/theGuiltyMan/CS-353>