Table of Contents

Relations Breakdown

To prevent any data anomalies, all relations have been normalised to Boyce-Codd Normal Form (BCNF):

- There are no partial functional dependencies (FDs).
- There are no transitive FDs.
- Every determinant is a unique candidate key.

User Information Relations

`UserPublic`

- Holds information about users which can be viewed by other users.
- To maintain referential integrity, any relation with the `UserName` attribute references this relation's **`UserName`** attribute as a foreign key.
- `UserName` is the primary key; functional dependencies are as follows:

```
UserName     Avatar, CreationDate, AccountStatus, LastLogin,
UserStatus
```

`UserPrivate`

- Holds all information about users which cannot be viewed by other users.
- `UserName` is the primary key (with the foreign key constraint); functional dependencies are as follows:

```
UserName     Password, FirstName, LastName
```

`Email`

- Holds all user email addresses.
- Email addresses are unique (making them candidate keys). They cannot be assigned to multiple accounts.
- UserName is the primary key; functional dependencies are as follows:

```
UserName     Email
```

Friendship Information Relations

`Friends`

- Holds a list of all user friendships.

`FriendRequest`

- Holds any friend request information.

See the Friendships section for further information on these relations.

Game Information Relations

`Game`

- Holds information about games.
- To maintain referential integrity, any relation with the `GameID` attribute references this relation's

`GameID` attribute as a foreign key.
- `GameID` is the primary key (automatically assigned); FDs are as follows:

```
GameID    AgeRating, DefaultImage, Name, AverageRating, NoOfRatings,
Publisher, ScoreFormat, SortOrder, ReleaseDate, TextDescription,
URL, Version, MaxScore, MinScore
```

Genre

- Holds all the game genre information.
- `GenreID` is the primary key; FDs are as follows:

```
GenreID     Name
```

GameImage

- Holds information about images to be linked with games.
- `GameImageID` is the primary key; FDs are as follows:

```
GameImageID     GameID, FilePath, DefaultImage
```

Linking Relations for Users, Games and Genres

These relations are used for one-to-many and many-to-many relationships.

UserToGame

- This relation is used to link Users to Games holding all information about each user's separate instance of a game.
- Holds foreign key references to both `UserName` and `GameID.`
- `ID` is the primary key; FDs are as follows:

```
ID     UserName, GameID, GameInProgress, InMatch, HighestScore,
LastScore, LastPlayDate, UserRating, AgeRating, Comments
```

GameToGenre

- This relations is used to link Games to Genres.
- Holds foreign key references to both `GameID` and `GenreID.`
- The primary key is multi-valued (both `GameID` and `GenreID`).

Achievements

Friendships

All user friendships are stored in the **Friends** relation:

AccHolder	**Friend**
UserName *of the Account Holder*	UserName *of their Friend*

The primary key for this relation is multi-valued (AccHolder, Friend) to ensure that each account holder pairing is not duplicated. Note that all friendships are bidirectional. This means that a matching reverse friendship must exist for all friendship pairs. For example:

AccHolder	**Friend**
AlexParrott	ScarlettJo
ScarlettJo	AlexParrott

This design decision was made primarily to make queries simpler:

- All friendships are stored in a single relation.
- The same simple query can be used to get a complete friend list for all users
  `SELECT * Friends WHERE AccHolder = 'ScarlettJo';`

A matching friendship is automatically generated or deleted by the ProcessRequest() procedure (see below). INSERT and UPDATE triggers in MySQL cannot insert new data into a table that is already being amended. Therefore, if a trigger was used to create matching reverse friendships, a second Friends relation would be needed. Also, triggers have the additional drawbacks of being global and vulnerable to hidden consequences.

Creating and Deleting Friendships

All changes to friendships (creation or deletion) are done via the **FriendRequest** relation:

RequestID	**Requester**	**Requestee**	**Email**	**Response**
Unique Friend Request ID	UserName *of the requester*	UserName *of the requested user*	Email *address of the requested user*	*Flag for status of the request:* - *Pending* - *Accepted* - *Declined* - *Completed*

When a FriendRequest is created it is assigned a unique RequestID which can be used to create a new friendship (when Response='Accepted') or delete a friendship (Response='Declined') by calling ProcessRequest().

How to create new friendships

Step 1: Create a request

- This is done by calling **CreateRequest()**. Potential friends can be looked up using either their UserName or their Email address. Ensure the delete flag parameter is set to False.

Step 2: Respond to the request

- If a user wishes to accept a friend request then the Response attribute in the relevant FriendRequest must be updated to 'Accepted'. If they do not want to accept the request, the Response attribute can be updated to 'Declined'. In this case the request will be marked as complete and deleted when **ProcessRequest()** is next called.

Step 3: Process the request

- This is done by calling **`ProcessRequest()`**. If the response is set to `'Accepted'` then the friendship will be inserted into the `Friends` relation.

How to delete friendships

Step 1: Create a request

- This is done by calling **`CreateRequest()`**. Friends can be looked up using either their `UserName` or their `Email` address. Ensure the delete flag parameter is set to `True`.

Step 2: Process the request

- This is done by calling **`ProcessRequest()`**. This will delete the friendship from the `Friends` relation. The request will then be automatically deleted.

`CreateRequest()`

Description:
This procedure creates a new request in the `FriendRequest` relation using a provided `UserName` to look up requested user to create or delete a friendship.

Parameters:

1. The `UserName` of the of the user making the request.
2. The `UserName` or `Email` of the user to request/delete friendship.
3. Delete flag: `TRUE` = request new friend; `FALSE` = request friendship deletion
4. Email flag: `TRUE` = lookup user with `Email` attribute; `FALSE` = lookup user with `UserName` attribute

Example Usage:
Request new friendship with `UserName` lookup:
```
CALL CreateRequest('AlexParrott','WillWoodhead',FALSE,FALSE);
```

Request friendship deletion with `UserName` lookup:
```
CALL CreateRequest('AlexParrott','WillWoodhead',TRUE,FALSE);
```

Request new friendship with `Email` lookup:
```
CALL CreateRequest('AlexParrott','Will@Woodhead.com',FALSE,TRUE);
```

Request friendship deletion with `Email` lookup:
```
CALL CreateRequest('AlexParrott','Will@Woodhead.com',TRUE,TRUE);
```

`ProcessRequest()`

Description:
This procedure processes a specified request in the `FriendRequest` relation according to the request response status:

• `Response = 'Accepted'`	: Creates a new friendship pair and matching reverse friendship in the `Friends` relation.
• `Response = 'Declined'`	: Deletes the friendship pair and the matching reverse friendship in the `Friends` relation.
• `Response = 'Pending'`	: No action.
• `Response = 'Completed'`	: Deletes entry from `FriendRequest` relation.

Parameter:

- The `RequestID` (INT) of the `FriendRequest` to action.

Example Usage:
```
CALL ProcessRequest(14);
```

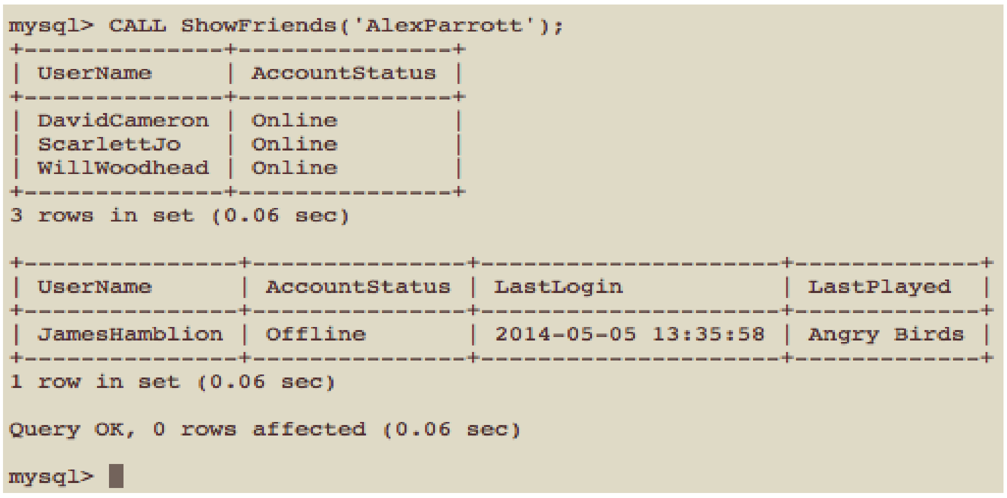
Friendship Features

ShowFriends()

The `ShowFriends()` procedure lists all of a specified user's friends. All online friends are shown in one table and then all offline friends are shown including their last logon time and the name of the last game they were playing. The user to lookup is specified by passing the `UserName` as a parameter when calling this procedure:

```
CALL ShowFriends('AlexParrott');
```

Example output:



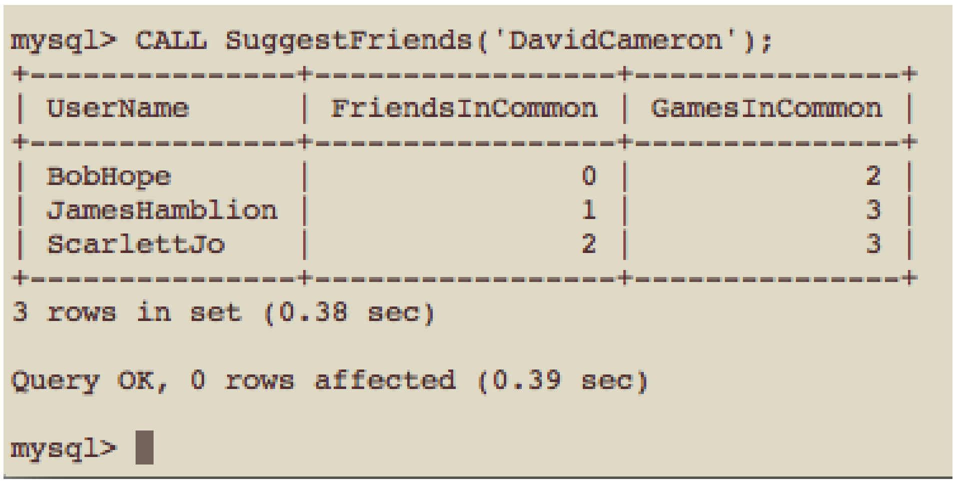
```
mysql> CALL ShowFriends('AlexParrott');
+--------------+---------------+
| UserName     | AccountStatus |
+--------------+---------------+
| DavidCameron | Online        |
| ScarlettJo   | Online        |
| WillWoodhead | Online        |
+--------------+---------------+
3 rows in set (0.06 sec)

+---------------+---------------+---------------------+-------------+
| UserName      | AccountStatus | LastLogin           | LastPlayed  |
+---------------+---------------+---------------------+-------------+
| JamesHamblion | Offline       | 2014-05-05 13:35:58 | Angry Birds |
+---------------+---------------+---------------------+-------------+
1 row in set (0.06 sec)

Query OK, 0 rows affected (0.06 sec)

mysql>
```

SuggestFriends()

The `SuggestFriends()` procedure creates a list of suggested friends for a specified user. This works by compiling a list of any users who are not already friends with the user and have 2 or more friends *or* owned games in common with the user. The number of friends and games in common are also displayed in the final list. The user to lookup is specified by passing the `UserName` as a parameter when calling this procedure:

```
CALL SuggestFriends('DavidCameron');
```

Example output:



```
mysql> CALL SuggestFriends('DavidCameron');
+---------------+-----------------+---------------+
| UserName      | FriendsInCommon | GamesInCommon |
+---------------+-----------------+---------------+
| BobHope       |               0 |             2 |
| JamesHamblion |               1 |             3 |
| ScarlettJo    |               2 |             3 |
+---------------+-----------------+---------------+
3 rows in set (0.38 sec)

Query OK, 0 rows affected (0.39 sec)

mysql>
```

Games

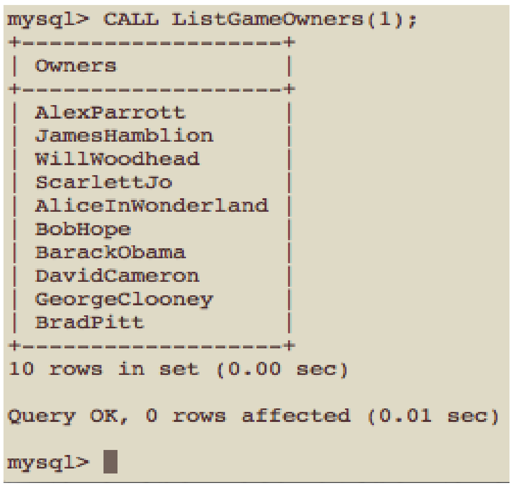
Games Features

`ListGameOwners()`

The `ListGameOwners()` procedure creates a list of all the users who own a specified game. The game to lookup is specified by passing the `GameID` as a parameter when calling this procedure:

```
CALL ListGameOwners(1);
```

Example output:



```
mysql> CALL ListGameOwners(1);
+--------------------+
| Owners             |
+--------------------+
| AlexParrott        |
| JamesHamblion      |
| WillWoodhead       |
| ScarlettJo         |
| AliceInWonderland  |
| BobHope            |
| BarackObama        |
| DavidCameron       |
| GeorgeClooney      |
| BradPitt           |
+--------------------+
10 rows in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

mysql> 
```

`UpdateAverage()`

The `UpdateAverage()` procedure updates the average user rating of a specified game (stored in the `UserToGame` relation as the `AverageRating` attribute). Average ratings only apply when a game has 10 or more user ratings. If a game has less than 10 ratings, the average is set to `NULL`.

Note, this procedure is called automatically by the following triggers:

`AfterInsertUserToGame`	: Triggers after any insert to the `UserToGame` relation.
`AfterUpdateUserToGame`	: Triggers after any update to the `UserToGame` relation.
`AfterDeleteUserToGame`	: Triggers after any delete from the `UserToGame` relation.

Therefore, **anytime a user rates a game the average rating is automatically updated in the database**.

`CatchCheaters()`

The `CatchCheaters()` function is setup to prevent cheaters who fix their scores. A maximum score (`MaxScore`) and minimum score (`MinScore`) attributes are stored in the `Game` relation. If these are not set to `NULL` then this function checks a provided score against the played game's max and min scores. If the score provided is legal then it is returned to the function caller unchanged. However, if the provided score is an illegal value then the minimum score for than game is returned.

Note, this function is called automatically by the following triggers:

`BeforeInsertUserToGame`	: Triggers before any insert to the `UserToGame` relation.
`BeforeUpdateUserToGame`	: Triggers before any update to the `UserToGame` relation.

Both of these triggers set the the `LastScore` in the `UserToGame` relation. Therefore, **anytime a user's last score is added or updated, it is automatically checked against the legal values.**