COMS20700 Databases: Coursework 03

Report

May 18, 2014

Liban Abdulkadir
la12808@my.bristol.ac.uk
Ana Dumitraş
ad12461@my.bristol.ac.uk
Andra Irimia
ai12821@my.bristol.ac.uk
Ioan Troană
it12754@my.bristol.ac.uk

Abstract

This report represents an outline of the Databases third coursework. The aim of this coursework is to design and implement, in a group of four, a database for an online multiplayer social gaming network similar to the Game Centre on iOS. **Section 1** includes a brief summary of our approach. In **Section 2** we mention what decisions and assumptions regarding the specification we have made. We then continue with detailing what the system we implemented can do (in **Section 3**), by going through the SQL statements requested by the client. **Section 4** discusses project management, while **Section 5** approaches some future improvements. **Sections 6 and 7** include References and Appendices.

1 Introduction

The goal of this coursework was to design and implement a database for a multiplayer social gaming network similar to the Game Centre on iOS. You can connect using a unique username and there will be displayed games that you can play with other users. Some games may feature achievements, where for completing a certain task, the player is rewarded points. Depending on the game, a leaderboard may be present where a player can compare his or her score with friends or with other players from around the world. The main features, however, were proposed by the client in the specification.

In order to accomplish this, we have decided to use `Postgre SQL` as a relational database management system. It runs on all major operating systems and it is easy to set up. Moreover, it is fully ACID compliant, having full support for custom types, arrays, foreign keys, joins, views, triggers and stored procedures. [1]

2 Design and Implementation

Schema

By closely following the specification, we have created a database schema that would match the requirements as well as ease our work when retrieving data. For this purpose we have split the three main tables (Users, Games and Achievements) into 9 separate, smaller ones according to the normalisation rules. This way, we made sure that no many-to-many relations are present. We accomplished this by using a third junction table (or cross-reference table), say, AB with two one-to-many relationships A $\rightarrow$ AB and B $\rightarrow$ AB (for instance, table `ach` with two one-to-many relationships: `gameOwnAch` $\rightarrow$ `ach` and `gameAch` $\rightarrow$ `ach`).

The need to have a list of games each user owns required us to introduce a new table, `gameOwn`, that would have its own primary key and a foreign key to connect with the `user` table. Also, it will include the required fields *rating, comment*, etc., as mentioned in Section A.2. of the specification.

In order to store which achievements the user has gained in a specific game, we have used a new table, `gameOwnAch`, that is linked with foreign keys to `gameOwn` and `ach`. A required attribute is *dateAchieved*, which specifies when (date and time) the achievement was actually unlocked.

The `friend` table acts as a friends list with links between users and it can be used as follows: *userId1* represents the id of the user that sent the request, while *userId2* is the id of the user the request was sent to. An attribute named *status* specifies if the status of the friend request and it is represented as a custom type that can take the values 'accepted', 'rejected' or 'awaiting'. The actual functionality is implemented in functions `send_friend_request(username1, username2, email)` and `friend_request_action(userame1, username2, action)`, that can be found in functions.sql. The first functions allows you (*username1*) to send a request to another user that is identified by username (*username2*) or e-mail address (*email*). The action is then performed in the second function.

In order to provide one or more categories for each game we have created the `category` table, with fields name and rating (which specifies the age rating). Also, we decided to use a different table that would allow to have a ranking system within the genre/category. Therefore, we created table `gameCat` for this purpose, table that has a rank field, a primary key for identification and two foreign keys that link it with tables `category` and `game`. The table also allowed us to avoid a many-to-many relation.

The client specification mentions that one important feature of our system should be a leaderboard for each game, which shows the top 10 scores for that specific game. We have approached this by creating a function,

`leaderboard(gameName)`, that, given a specific game, returns a view containing the usernames and the scores of the top 10 scores achieved for that game.

As mentioned in the specification, a maximum of 100 points per achievement is allowed. We have dealt with this by using a *CHECK constraint* to limit the range that can be placed in column *value* of table `gameAch`, $0 \leq value \leq 100$.

Two additional constraints were required to check the following:

- a maximum of 100 achievements per game,
- and a total maximum of 1000 points for all the achievements belonging to any particular game.

However, we have done this by creating a trigger `check_gameAch()` that is automatically invoked when an *INSERT event* occurs in table `GameAch`.

As we thought that users' *security* is important in a system like this, we decided to encrypt the users' passwords by storing the MD5 hash of the initial string (in hexadecimal). We implemented a trigger which checks each time a new user is created and also updates an existing user's password if it has been changed. This offers minimal security, since if someone would break into the database, the users' passwords cannot be easily decrypted and accounts will not be compromised.

See ***Appendix A*** for the **ER Diagram** of our final database describing the various tables and their keys and showing how the entities/attributes are connected. In terms of data types of the columns used when creating the tables, we have used the `Postgre SQL` data types [2] as their proper use implies format validation of data and rejection of data outside the scope. A detailed list of the types used for each attribute can be found in ***Appendix B***.

Database dump

In order to test if the queries, functions and triggers we have written return the correct output, we have generated random data for each attribute of the relations. This has been done by generating fake data in `Ruby` using the *Faker* [3] library. As we progressed through the questions, we realised there is a need of relations between data and that the ID randomness, for instance, prevented us from checking the accurateness of the queries' output. Changes also occurred when we had to introduce new attributes.
An example (the `category` table) can be seen below:

```
require 'faker'
require 'digest/sha1'
require 'rubystats'
salt = 'saltysalt'

n = 30

puts 'INSERT INTO "category"
(name, rating) VALUES '
n.times do |i|
    name = Faker::Company.name.gsub("'","\\'\\'")
    rating = [Rubystats::NormalDistribution.new(3.5,1).rng.round,5].min

    puts "('#{name}','#{rating}')" +
      (if i < (n-1) then ', ' else '' end)
end
```

3 Results and Evaluation

After designing the system, creating tables with the triggers and functions mentioned above and populating them with dummy data, we were able to approach the queries and, this way, to add additional functionality to our system. A detailed explanation for each task can be found in this section. It is worth mentioning that the '...' in the resulting tables suggest that only part of the data was included in the report, for visualisation purposes.

Question 1

Given a game, list all the users who own that game.

For solving this question it is needed to select from the `gameOwn` table the IDs of all users who own the given game. Then from `user` are selected all usernames whose IDs are in the list resulted from the previous query.

```
SELECT q1('1');
```

q1
gilda
mercedes_hudson
arden
abelardo
gaston
kira.mohr
annabelle.reilly
liana
linda
haylee
ignacio
donald_mohr
godfrey.conn
murl
clair

Question 2

Automatically update a game's average rating whenever a user adds or updates their rating for that game.

We solve this question by implementing a *trigger*. Whenever an entry is updated or inserted into `gameOwn`, the average rating for the given game is recalculated. This is done by averaging all entries from `gameOwn` whose *gameId* equals the ID of the game whose information was updated or inserted.

Question 3

Change the database so that a game's average rating only appears if it has been rated by 10 or more users.

Every time an entry is deleted or inserted into `gameOwn`, the table that links users with the games they own, it is needed to check if a game has been rated by at least 10 users. We assume that a user can rate a game only after he or she owns it.

Question 4

Given a user and a game, display the user's score, rank on that game's leaderboard (even if they are outside the top 10) and an indication of where they appear relative to the average e.g. "85000 points - 1788 (Top 20%)" or "13000 points - 16364 (Bottom 40%)".

The user's highscore, rank and relative position in a particular game can be found using my_game_status *function*, which takes as parameter the user's username and game name and returns a status string.

```
SELECT * FROM my_game_status('murl', 'Goodwin, Windler and Morar');
```

my_game_status
110 points(int) - 4 (Top 27%)

Question 5

Create a list of the top 10 rated games in each genre/category.

For this question, it is needed to compare the id of the `category` table with the id of the `gameCat` table to make sure they are a match, and also the *gameId* from `gameCat` to the actual *gameId*. It is then returned the category name, game name and rank for the top 10 games in the respective categories.

```
SELECT category.name, game.name, gameCat.rank FROM category, gameCat, game WHERE
   category.id = gameCat.catId AND
   gameCat.gameId = game.id AND
   gameCat.rank < 11
ORDER BY category.name, gameCat.rank;
```

category_name	game_name	rank
Baumbach, Dietrich and Kerluke	Hagenes, Koepp and Mohr	1
Baumbach, Dietrich and Kerluke	Gutkowski Group	2
Boyer-Bogisich	Hagenes, Koepp and Mohr	1
Boyer-Bogisich	Gutkowski Group	2
Cruickshank, Veum and Fisher	Goodwin, Windler and Morar	1
Cruickshank, Veum and Fisher	Gutkowski Group	2
Dach Inc	Gutkowski Group	1
Dach Inc	McKenzie LLC	2
Hane, OConner and RoweConner and RoweConner and Rowe	McKenzie LLC	1
Hane, OConner and RoweConner and RoweConner and Rowe	Mitchell-Sauer	2
Hartmann, Walker and Spencer	Goodwin, Windler and Morar	1
Hartmann, Walker and Spencer	Hagenes, Koepp and Mohr	2
...	...	...

Question 6

To help prevent cheating, add an optional maximum and minimum score value to each game and check that user's scores are within this range.

To solve this question we have added two new attributes in the `game` table: *minimum* and *maximum*. The two new columns store for each game the minimum and maximum value of the score that a user can achieve. Every time a high score is updated we check if it is in the range: $minimum \leq highScore \leq maximum$. In case the score is too high or too low, an *exception* message will be displayed.

Question 7

Add daily and weekly leaderboards for each game showing the best scores achieved this day or week.

In order to complete this question, we wrote a *function* which takes as a parameter a string. If the string is 'weekly', then weekly leaderboards are returned; if it is 'daily', then the daily leaderboards are returned. A new field was added to the `gameOwn` table as it was needed the date the high score was achieved. The query

checks the dates of these high scores and if they match the criteria, then they get displayed in a descending order.

```
SELECT * FROM q7('weekly') -- for weekly leaderbords
```

f_game_name	f_user_name	f_highscore
Hagenes, Koepp and Mohr	gaston	159
Beatty-Swift	donald_mohr	156
Heaney-Rice	gilda	156
Lynch-Thompson	mary_shanahan	148
Waters, Strosin and Wunsch	faye_emmerich	146
Bernier Group	maverick	143
Keebler, Bruen and Bartoletti	queenie.schneider	142
Williamson-Ortiz	maverick	140
Williamson-Ortiz	godfrey.conn	140
Osinski-Herzog	hildegard.walsh	138
...	...	...

```
SELECT * FROM q7('daily') -- for daily leaderbords
```

f_game_name	f_user_name	f_highscore
Bernier Group	maverick	143
Howe-Eichmann	stephanie.batz	136
Mitchell-Sauer	arch	131
Kilback LLC	hildegard.walsh	130
Herman, Nikolaus and Ziemann	dameon	120
Ondricka, Kreiger and Trantow	matilda_dare	117
Sanford and Sons	jed.ko	112
Torp and Sons	hugh.towne	107
Williamson-Ortiz	jed.ko	102
Lynch-Thompson	al	100
...	...	...

Question 8

Check new usernames against a list of obscene or offensive terms (if possible check for substrings as well as the whole name). If such a username exists, lock the account.

When a user creates a new account, its username is automatically checked against a list of obscene or offensive words. If the username matches any of the words, the account will be deleted. We assumed locking the account means deleting the entry from the `user` table.

Question 9

The client wants to add a 'Hot List' which shows the 10 games, which have been played most often in the past week add the necessary fields, tables, queries, triggers and/or procedures to achieve this.

In order to find the most played games, we required a table that would hold the game play times (when a game starts) for each game. Therefore, we created `gameTime`, which has the*gameOwnId* and the *playedOn* timestamp as fields. We then calculated the count of gameplay dates in the past week and this way, were able to make a 'Hot List' which shows the most played games in that specific week.

```
SELECT game.name, COUNT(gameTime.playedOn::date) FROM game, gameOwn, gameTime WHERE
   game.id = gameOwn.gameId AND
   gameTime.gameOwnId = gameOwn.id AND
   gameTime.playedOn::date >= (current_date - integer '7') AND
   gameTime.playedOn::date <= current_date
   GROUP BY game.name
   ORDER BY COUNT(gameTime.playedOn) DESC
LIMIT 10;
```

name	count
Kilback LLC	175
Lynch-Thompson	166
Price Inc	160
Moore Group	156
Heaney-Rice	150
Hagenes, Koepp and Mohr	143
Wiza, Daugherty and Rowe	141
Howe-Eichmann	130
Dietrich, Altenwerth and Bins	127
Kuhlman-Kemmer	127

Question 10

The client wants to allow users to send friend requests to other users using their username or email address. What additional fields/tables would you need to add to allow for this functionality? You will need to keep track of the request, the response (if any) and some way of keeping track of user friends lists. If possible, extend this system to allow users to send an invite to a friend to play a particular game.

In order to support friend requests, we have created a new table `friend` that stores the IDs of the two users and their friendship status. We have split the task into two functions: one that sends the friend request to the second user (`send_friend_request`) and another that lets the second user respond to the friend request (`friend_request_action`). The first function checks if there already exists a friend request between the two users and if the usernames are valid (the users are in the database). If no error was encountered so far, then the function inserts the two users IDs into the `friend` table. The friend status is set by default to 'awaiting'. The second function updates the friend status in the `friend` table. It takes as input the usernames of the two users and an action of type friendStatus.

```
DELETE FROM friend;
SELECT send_friend_request('dameon', 'sherwood', '');
SELECT send_friend_request('dameon', 'gilda', '');
SELECT send_friend_request('sasha.hartmann', 'dameon', '');
SELECT send_friend_request('yvonne', 'dameon', '');
SELECT friend_request_action('sherwood', 'dameon', 'accepted');
SELECT friend_request_action('dameon', 'yvonne', 'accepted');
SELECT friend_request_action('gilda', 'dameon', 'rejected');

SELECT unnest(my_friends('dameon')) AS friends;
```

friends
sherwood
yvonne

Question 11

Given a user and a game, show a leaderboard which lists just the user and their friends.

For this question we have used the `send_friend_reques` function mentioned in the previous question in order to make sure that we actually have enough entries to display. Afterwards, we user the function `friend_leaderboard`, which is similar to the function that displays the leaderboard for a given user, in this case giving as parameters both the user and the game we are interested in.

```
SELECT send_friend_request('gilda', 'billy', '');
SELECT send_friend_request('gilda', 'murl', '');
SELECT send_friend_request('gilda', 'linda', '');
SELECT send_friend_request('gilda', 'liana', '');
SELECT send_friend_request('gilda', 'arden', '');
UPDATE friend SET status = 'accepted'::friendStatus;

SELECT * FROM friend_leaderboard('gilda', 'Goodwin, Windler and Morar');
```

rank	highscore	currency	username
1	82	int	liana
2	85	int	linda
3	106	int	gilda
4	110	int	murl
5	122	int	arden

```
SELECT * FROM friend_leaderboard('liana', 'Goodwin, Windler and Morar');
```

rank	highscore	currency	username
1	82	int	liana
3	106	int	gilda

Question 12

List all of a user's friends and show whether they are currently logged on. If they are not logged on, show when they were last logged on and what they were playing.

The function first returns a list with all the user's friends. For each friend it is then checked if it is logged on. If this is not the case, then it returns the last time they were logged on by getting the value from the table and the last game they played. In order to retrieve the last game that was played by a given user, we wrote a helper function `last_played`, that checks all the games a given user owns and returns the one that was played most recently.

```
SELECT * FROM friend_games('gilda');
```

fusername	floggedin	flastlogin	flastplayed
billy	t		
arden	f	2012-08-18 01:13:53+01	Kilback LLC
liana	f	2013-07-20 12:48:40+01	Kilback LLC
linda	f	2012-11-25 07:55:08+00	Kilback LLC
murl	t		

Question 13

Given a user and a game, show how many achievements they've got (out of how many in total the game

has) and how many points worth of achievements they have for that game e.g. "16 of 80 achievements (95 points)".

We wrote a function that, given a user and a game id, returns how many achievements the user has unlocked out of how many the game has and also how many points those achievements are worth. The game id is used instead of the game name because in the database there might be more than one version of the same game. In order to return the required message, it is first needed to check how many achievements the given game has and then count the number of achievements the user has unlocked for that particular game. In order to obtain the number of points those achievements are worth, we count the value of all achievements the user has unlocked.

```
SELECT show_achievements('gilda', 1);
"1 of 1 achievements (18 points)"
```

Question 14

Show a status screen for the player showing their username, status line, number of games owned, total number of achievement points and total number of friends.

To solve this task, we first wrote a *query* to find the username, status line and the number of games owned. Afterwards, we wrote a similar query but for the number of achievement points and the number of friends. Following this, we merged the 2 tables into one, this way displaying all the required values. This was aquired by using an inner join on the username. In the end, we moved the queries into a *function* in order to allow the user to specify the user id he/she requires information for.

```
SELECT * FROM q14(11);
```

n_username	n_status	n_games	n_ach_points	n_friends
yvonne	ut maiores quibusdam ducimus labore illum ...	2	7692	6

Question 15

Given a user and a game, list the achievements for that game (apart from ones which have the hidden flag set and the user hasn't earned). They should be listed with ones the player has earned listed first. For each achievement, list the title, points, description (which will vary depending on whether the player has earned that title or not) and when the achievement was earned.

For this question we used *functions* in order to allow the user to specify the exact username and game he requires information for. We select from the database the achievement title, value, description (which depends on whether the achievement was earned or not) and the date the achievement was earned. In order to return the correct result, it was needed to perform a number of checks on the achievements: if the achievement can be shown or not, if the achievement belongs to that game, that user and so on. At the end, the results were ordered by the date they were achieved.

```
SELECT * FROM q15(1, 13);
```

f_ach_title	f_ach_value	f_description	f_date_achieved
Regional Usability Supervisor	26	["Sit est esse nihil."]	2012-01-25 00:00:00+00
Internal Solutions Administrator	12	["Labore officia ..."]	2012-07-05 00:00:00+01
Principal Accountability Executive	63	["Earum consec ..."]	2013-02-21 00:00:00+00
Global Implementation Representative	51	["Molestiae vero ..."]	2013-07-02 00:00:00+01
Corporate Security Planner	21	["Nemo magni ..."]	2013-10-27 00:00:00+01

Question 16

Given a user and a named friend, for each game they both own, list the name of the game and the number of achievement points each of the users has for that game. If possible, at the end of that list add all the games that are owned by only one of the two users, showing how many points they have and a blank for the other user.

To solve this question we first find the games owned by each user and store them in separate tables. For each of the two users we have a table that has the IDs of the games they own and the total number of achievements they unlocked for each game. In order to find the common games for the two users, we apply an inner join on the two tables mentioned above. The resulted table only has the IDs of the games owned by both users and the number of achievements each user has unlocked. At the end of the table are appended the games only owned by the first user by selecting from the table that stores the games owned by the first user the ones that are not in the common table. We do the same for the games owned only by the second user. We update the resulted table with the name of each game based on their ID. The table now stores the ID of each game, its name, and the total achievements unlocked by each of the two users. The *function* will only return the last 3 attributes of the resulted table. If a user does not own the game, then the achievement column will have value 0; same if the user has not unlocked any achievements for a game they own.

```
SELECT * FROM common_games('jarod_ledner', 'yvonne');
```

game_name	no_achievements1	no_achievements2
Wisoky and Sons	298	276
Gutkowski Group	436	0
Wolff Group	406	0
Wiza, Daugherty and Rowe	297	0
Keebler, Bruen and Bartoletti	367	0
Kuhlman-Kemmer	193	0
Hagenes, Koepp and Mohr	431	0
Howe-Eichmann	0	284

Question 17

The client has realised that for some games (e.g. golf, driving games) lower scores are better. They want to add a "sort order" flag to the game to indicate this. They've also realised that many games have a format other than simply points when displaying scores. Add a "score format" field (e.g. int, time, money) to the Game table. The sort order and score format should be taken into account when displaying leaderboards.

For this question we implemented two new types for `game` table: the sort order and score format. Sort order can be ascending or descending, depending on the game and score format can be int, time, money or any other currency relevant to the game. Both types are now taken into account when displaying new leaderboards.

Question 18

Suggest another user as a friend, if they have a number of friends and/or games in common with the user. You should show the name of the potential friend and how many games/friends they have in common with the user.

To solve this question we wrote two helper functions. The first one returns a table with all the users that own the same games as the given user. The table is in descending order so that the user with the most games in common is on the first row of the table. The second function returns a table with the users that have friends in common with the given user. Again, the table is in descending order of number of friends they have in common. The main function calls both of these helper functions and does a join on the table

such that the user with most games and most friends in common would be the first entry in the resulted table. The username, number of games in common and number of friends in common of that user is thus returned.

```
SELECT send_friend_request('gilda','billy','');
SELECT send_friend_request('gilda','murl','');
SELECT send_friend_request('gilda','linda','');
SELECT send_friend_request('gilda','liana','');
SELECT send_friend_request('gilda','arden','');
SELECT send_friend_request('haylee','liana','');
SELECT send_friend_request('haylee','arden','');
UPDATE friend SET status = 'accepted'::friendStatus;

SELECT * FROM suggest_friend('gilda');
```

username	common_firends_no	common_games_no
haylee	2	4

Question 19

Automatically suggest a game, which a user might be interested in. Games should only be suggested if the user doesn't already own them. Suggestions could be made on the basis that other users who make similar ratings also rate this item highly (though you may use another recommendation metric if you prefer).

We decided to suggest a new game based on the popularity of the game among the given user's friends. The function checks the list of games owned by all the given user's friends against the list of games owned by the user and returns the first game that the user does not already own. In case the user has no friends or they all own the same games, an error message will be displayed.

```
SELECT suggest_game('gilda');
```

suggest_game
Dietrich, Altenwerth and Bins

```
DELETE FROM friend;
SELECT suggest_game('gilda');
ERROR: Not enough friends or they all own the same games
```

Question 20

Write another query/trigger/procedure of your choice, which really shows off what your database can do.

For this part we have decided to change the way a user's highscore is updated depending on their in-game rank. This was achieved by splitting all players for each game into 4 different groups of equal size when ordered by their rank. This means that the top 25% percent are in the first group and so on. Whenever the highscore is increased, a multiplier is applied to the difference. The multiplier is defined as follows:

Group	Multiplier
1	0.4
2	0.8
3	1.2
4	1.4

This ensures that players at the top of the leaderboard will find it harder to increase their score even further.

– clair has id = 99 and rank 1 with score 123 in game id 1

```
SELECT highScore FROM gameOwn WHERE userId = 99 AND gameId = 1;
```

highScore
123

```
UPDATE gameOwn SET highScore = highScore + 10 WHERE userId = 99 AND gameId = 1;
SELECT highScore FROM gameOwn WHERE userId = 99 AND gameId = 1;
```

highScore
127

We can therefore see that th score actually increased by 4 (40% of 10).

4 Project Management

This project was divided, even from the beginning, into the following stages:

1. Database design and E/R Diagram
2. Scripts for dummy data
3. Tables creation
4. Queries, Triggers and Functions for the given tasks
5. Report

We worked all together when designing the database in order to make sure that everyone is aware and agrees with the decisions we took. One of the team members then easily designed the *E/R Diagram* using the `Gliffy` online tool.

After that, we split the tasks among us, starting with the scripts for dummy data and then spending the most time on concurrently implementing the required features of our system. During development, we encountered questions which required us to slightly change the database, having to update the *E/R Diagram* and the scripts. For this stage we have used `Trello`, an online kanban board, which allowed us to know the state of the project at any point and to make sure that we are not working on the same task, at the same time.

For version control we have decided to use git due to the team members' familiarity with the tool and a free private git repository on `GitHub`. Each of us worked on a separate branch, sending pull requests when done with a task. This way we made sure that the master branch only contained code that was reviewed and tested by at least one member of the team.

`Google Docs` allowed us, as a team, to contribute in real time to the report. This constituted a draft, the main report being written in LaTeX.

5 Future improvements

In terms future improvements, it is worth mentioning that here are a number of attributes in our database that are not being used at the moment. For instance, the `category` table stores the age rating. This can be used in restricting access to certain users if their age is below a given value. In order to do this, however, we will also need to know the user's age, so we will need to add another column to the `user` table that will store the age or date of birth. Although we are not using some of the attributes, we decided to keep them in the database so it will be easier to add extra features to the system.

6 References

In addition to the lecture slides, we have used the following resources in order to fully understand the concepts used in this assignment and to be able to easily work with PostgreSQL:

[1] `http://www.postgresql.org/about`.

[2] `http://www.tutorialspoint.com/postgresql/postgresql_data_types.htm`.

[3] `https://github.com/stympy/faker`.

[4] `http://en.wikipedia.org/wiki/Game_Center`.

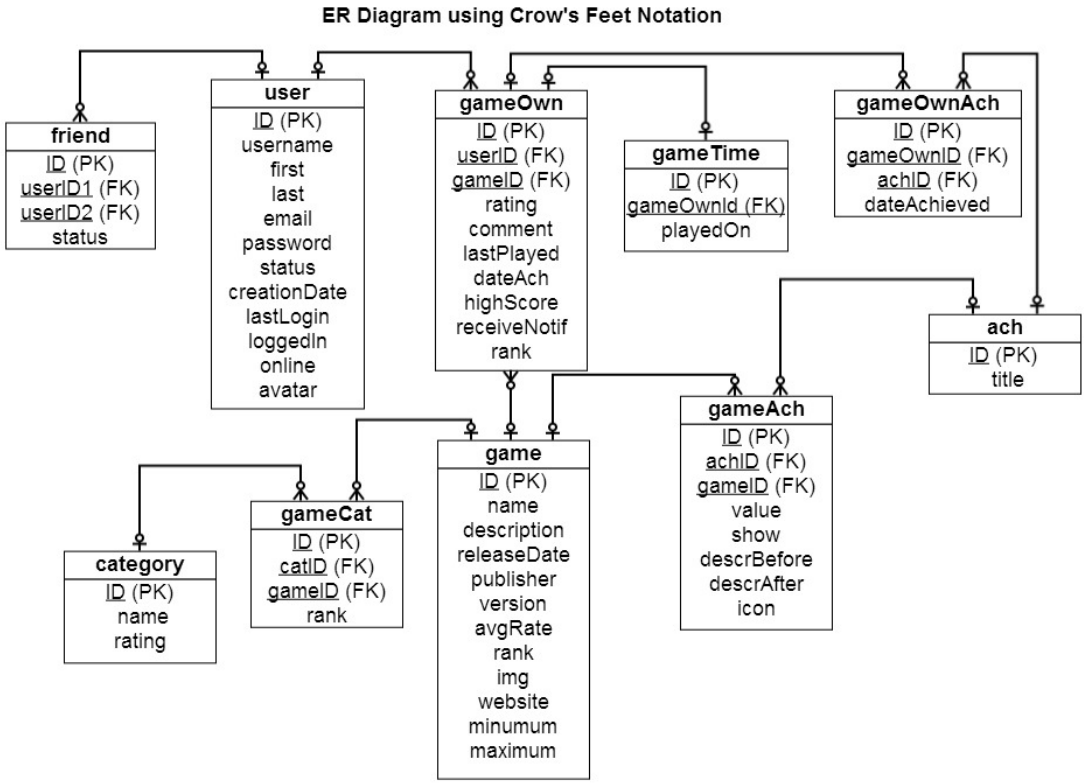
[5] `http://www.postgresql.org/docs`.

[6] `http://www.tutorialspoint.com/postgresql`.

[7] Carolyn E. Begg Thomas M. Connolly. Database systems: A practical approach to design, implementation and management, 2005.

7 Appendices

Appendix A: ER Diagram



Appendix B: Data Types

Attribute name	Data type	Length	Description
friend			
id	serial		
userId1	integer		not null
userId2	integer		not null
status	friendStatus		not null, default 'awaiting'
user			
id	serial		
username	varchar	50	not null
first	varchar	25	not null
last	varchar	25	not null
email	varchar	50	not null
password	varchar	50	not null
status	text		
creationDate	date		not null
lastLogin	timestamp with time zone		
loggedIn	boolean		not null
online	boolean		not null
avatar	varchar	255	
gameOwn			
id	serial		
userId	integer		not null
gameId	integer		not null
rating	integer		0 ≤ rating ≤ 5
comment	text		
lastPlayed	timestamp with time zone		
dateAch	timestamp with time zone		
highScore	double precision		not null, default 0
receiveNotif	boolean		not null, default true
rank	integer		
gameOwnAch			
id	serial		
gameOwnId	integer		not null
achId	integer		not null
dateAchieved	timestamp with time zone		
gameTime			
id	serial		
gameOwnId	integer		not null
playedOn	timestamp with time zone		

Attribute name	Data type	Length	Description
category			
id	serial		
name	varchar	255	not null
rating	integer		0 ≤ rating ≤ 5
gameCat			
id	serial		
catId	integer		not null
gameId	integer		not null
rank	integer		
game			
id	serial		
name	varchar	255	
description	text		
releaseDate	date		not null
publisher	varchar	255	not null
version	integer		
avgRate	integer		
rank	integer		
img	varchar	255	
website	varchar	255	
minimum	integer		default 0
maximum	integer		
sorting	ordering		not null, default 'asc'
value_type	game_currency		not null, default 'int'
gameAch			
id	serial		
achId	integer		not null
gameId	integer		not null
value	integer		not null, 0 ≤ value ≤ 100
show	boolean		not null
descrBefore	text		
descrAfter	text		
icon	varchar	255	
ach			
id	serial		
title	varchar	50	not null

* serial = autoincrementing integers

Appendix C: Mark Allocation

Name	Allocated Mark
Liban Abdulkadir	0.25
Ana Dumitraş	0.25
Andra Irimia	0.25
Ioan Troană	0.25