COMS20700 Databases: Coursework #3

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 conclusions as well as future improvements, while **Sections 5 and 6** 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`.

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. An example (for the `user` 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
```

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 the next section.

3 Results and Evaluation

Question 1
Given a game, list all the users who own that game.

Question 2
Automatically update a game's average rating whenever a user adds or updates their rating for that game.

Question 3
Change the database so that a game's average rating only appears if it has been rated by 10 or more users.

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%)".

Question 5
Create a list of the top 10 rated games in each genre/category.

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.

Question 7
Add daily and weekly leaderboards for each game showing the best scores achieved this day or week.

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.

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.

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.

Question 11
Given a user and a game, show a leaderboard which lists just the user and their friends.

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.

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)".

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.

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.

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.

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.

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.

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).

Question 20
Write another query/trigger/procedure of your choice, which really shows off what your database can do.

4 Conclusions

5 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`.

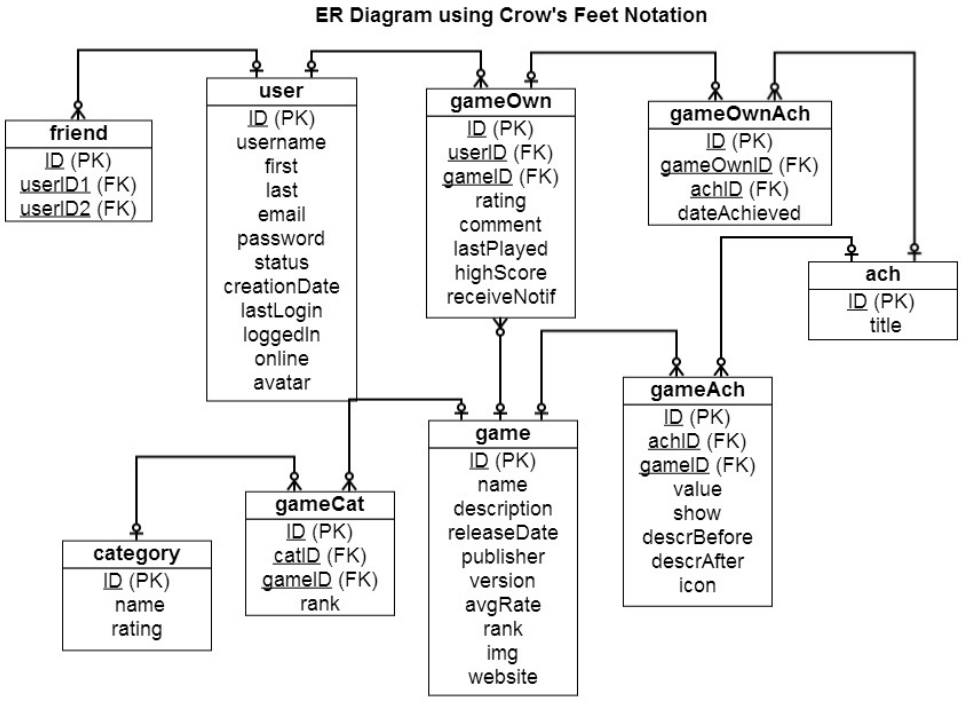
[5] `http://www.postgresql.org/docs`.

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

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

6 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
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		
gameId	integer		
rating	integer		
comment	text		
lastPlayed	timestamp with time zone		
highScore	double precision		not null, default 0
receiveNotif	boolean		not null, default true
gameOwnAch			
id	serial		
gameOwnId	integer		not null
achId	integer		not null
dateAchieved	timestamp with time zone		

* serial = autoincrementing integers

Attribute name	Data type	Length	Description
category			
id	serial		
name	varchar	255	not null
rating	integer		
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	
gameAch			
id	serial		
achId	integer		not null
gameId	integer		not null
value	integer		not null
show	boolean		not null
descrBefore	text		
descrAfter	text		
icon	varchar	255	
ach			
id	serial		
title	varchar	50	not null

Appendix C: Mark Allocation

Name	**Allocated Mark**	**Signature**
Liban Abdulkadir	0.25	L.A
Ana Dumitraş	0.25	A.D
Andra Irimia	0.25	A.I
Ioan Troană	0.25	I.T