Social Video Game Database System

Final Report

CS 4318: Database Management Systems

By Robert Jackson

12/4/2016

**Abstract**

This report describes the implementation of a database that allows users to create catalogues of games, rate them, comment on them, and view other catalogues. It also allows users to search the database of video games, developers and consoles for informational purposes and to obtain statistical information such as average ratings, average age of users, etc.. An ER model and relational data were used to design the database, use cases were used to define database functionalities. Normalization to BCNF is used in logical design phase to help fine tune the database design. The database itself was implemented using Microsoft Server 2016, the Microsoft Server Management System, and SQL. No user interface was specified or created for the database.

**Introduction**

The goal is to create a database that supports a theoretical social gaming site. In this site a user can create a catalogue of games, marking them as whether they are games they have played, want to play, own or want to own, or the opposite. They can also rate the games and make comments on them. Then they can view other users’ catalogues, view their opinions on the games, and view any other data relevant to the game. This data includes more information on the game, the developers who made the game, and information on the consoles the game is played on. The database will also allow developers to have accounts created for them so they can modify any games in the database they have made, add games they have made, and keep an eye on stats to find out what games the community enjoy and what they don’t. Moderators will be allowed to help the database administrator correct video game information and delete entries in catalogues with inappropriate comments. The Database Administrator will have the power to modify, add, and delete information from any of the databases. Since this is a database for a theoretical site, no web or user interface was created as the design and functionality was what was sought after.

**Product Requirements**

The requirements for this database will be defined by the actors and the use cases. The next couple of sections will define the actors, and then explain what the actors will be expected to do.

**Actors Definitions:**

* Video Game Enthusiasts: Social users who will create their own game catalogues, review and rate them.
* Developers: Game creators who will be able to view ratings and comments left regarding their games and view basic demographical information (age, gender) of the video game enthusiasts.
* Moderators: Those who help maintain the web site and will monitor games being added and comments for vulgarity.
* Data Base Administrator: User who will maintain and update database as needed and create developer accounts on developer request and confirmation of said developer.

**Use Cases by Actor:**

Video Game Enthusiast:

* Add a game to database.
* Add an entry to personal catalogue
* Modify personal account.
* Modify personal catalogue.
* Search other Video Game Enthusiasts by: (will only return user name, age, gender)
  + User Name
  + Age or range of ages
  + Gender
  + Gender and Age or range of ages.
  + By Video Game Name from Catalogue
* Find the Number of users in database:
  + By number of user names.
  + By age or range of ages.
  + By gender.
* Find the average age of users:
  + By total users.
  + By age range.
  + By gender.
* Search Catalogues by:
  + User Name
  + Age or range of ages.
  + Gender
  + Video Game Name
  + Rating or range of ratings
  + Console
  + Developer
  + Any combination of play attributes in the catalogue.
* Find number of:
  + Entries in overall catalogue
  + Number of play attributes by user name (has played, wants to play, owns, want to own)
  + Number of opposite attributes by user name.
* Find average rating of a game.
* Find average rating of a game based on users who have played the game.
* Search for video games by:
  + Video game name
  + Developer
  + Console
  + Genre
  + Maturity rating
  + User Names
* Find the total sum of:
  + Video games in catalogue
  + Games by developer, console, genre, maturity rating
* Find the number of developers.
* Find the Number of consoles.
* Find the Number of consoles created by a specific company.

Developer:

Everything that a Video Game enthusiast can do with the following exceptions:

* Cannot create a video game catalogue.
* Can add games in database that they have created.
* Can only modify games in database that they have created.

Moderator:

Everything the above can do, and in addition:

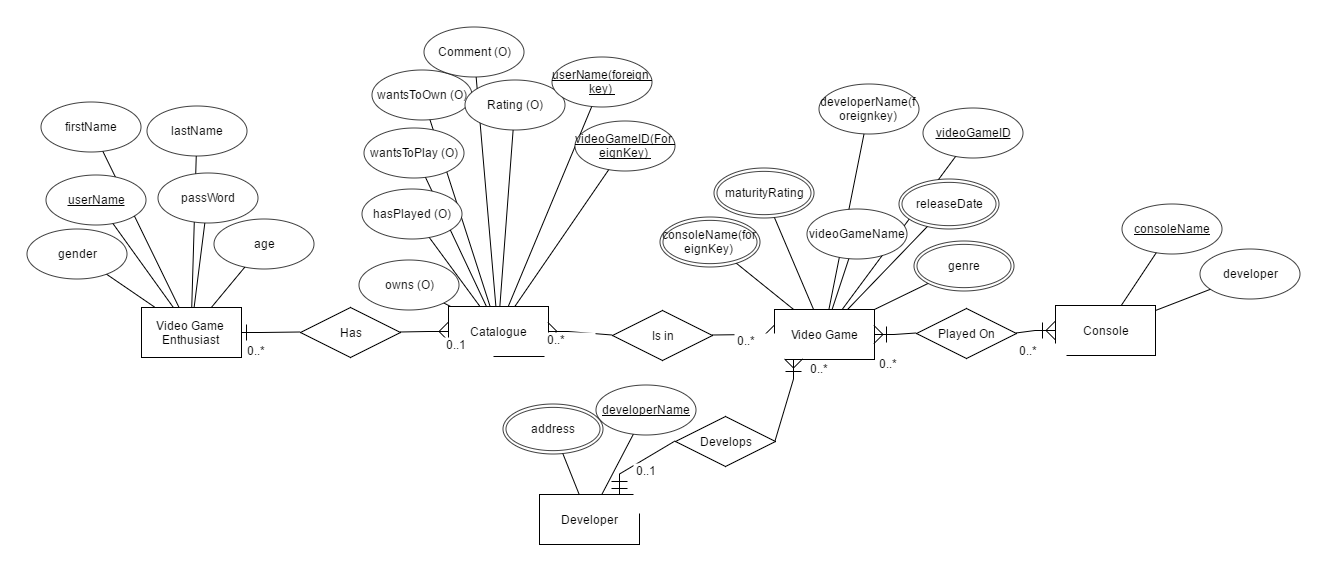
* Delete entries from Catalogue for inappropriate commentary.
* Modify all video games per database administrator.

Database Administrator:

Everything the above can do, plus the following:

* Add or delete an entry from any of the tables.
* Modify any entry from any of the tables.

**Design Part 1:**

**ER Diagram**

The underlined portions of the ER diagram are the primary keys for that table, and foreign keys are labelled.

**Relational Model**

First we will describe the entities, attributes, primary and foreign keys. Then, using that information, the relational models for the various searches will be described.

Entities:

Video Game Enthusiast:

Attributes: firstName, lastName, age, gender, userName, password

Primary Key: userName

Foreign Key: userName

Catalogue:

Attributes: userName, videoGameID, comment, rating, owns, wantsToOwn, hasPlayed, wantsToPlay.

Primary Key: userName, videoGameID

Forign Key: userName, videoGameID

Video Game:

Attributes: videoGameID, videoGameName, consoleName, developerName, genre, maturityRating, releaseDate

Primary Key: videoGameID

Foreign Key: console, developer

Console:

Attributes: consoleName, consoleDeveloper

Primary Key: consoleName

Foreign Key: consoleName

Developer:

Attributes: developerName, developerAddress

Primary Key: developeName

Foreign Key: developerName

Relationships:

VideoGameEnthusiast Has Catalogue:

This is a one to many relationship on both end. A Video Game Enthusiast can have 0 to multiple entries in the Catalogue, and a Catalogue can have multiple users accessing it. The way they interact with each other is through the userName, which is a foreign key and primary key for the Catalogue but a primary key for VideoGameEnthusiast.

Video Game Is In Catalogue:

This is also a one to many relationship, as a catalogue can be made up of multiple video games, and the video game can be a part of multiple users catalogues. They interact with each other through the videoGameID, which is a primary key for Video Game and a foreign key for Catalogue.

Developer develops Video Game:

The developer has a 0 to many relationship with Video Game, as a developer can develop multiple video games. But the Video game has a 0 to 1 relationship with developer, as a video game can only be developed with one developer. They interact via developerName, which is a foreign key for Video Game and a primary key for developer.

Video Game is played on Console:

This is also a 0 to many relationship on both ends. A video game can be played on 0 to many consoles, and a console can have 0 to many video games played on it. They interact via consoleName, which is the foreign key for Video Game and primary key for Console.

Use Case Realization:

Here I will describe how the various use cases will be realized given the relational model.

Add a Game to Database:

*Input:* videoGameID, videoGameName, developer, consoleName, maturityRating(optional), genre(optional), releaseDate(optional).

*Steps:* ensure videoGameID is unique. If it is not an error message will appear when trying to create the table. If it is, table will be created.

*Relations Accessed:* Video game

*Relations Updated:*  Video game

Add an entry to personal catalogue:

*Input:*  userName, videoGameID, hasPlayed, wantsToPlay owns, wantsToOwn, rating(optional), comment(optional).

*Steps:* Will check to ensure that the userName and videoGameID combo is unique. If it is, then it will be added to the database, if not, an error will appear.

*Relations Accessed:* Catalogue

*Relations Updated:* Catalogue

Modify Video Game Enthusiast:

*Input:* userName and attributes to be modified.

*Steps:*  Will check to see if userName exists. If it does, it will update the modified attributes.

*Relations Accessed:* Video Game Enthusiast

*Relations Updated:* Video Game Enthusiast.

Modify Catalogue:

*Input:* userName, videoGameID, attributes to be modified

*Steps:*  Ensure that userName and videoGameID are in Catalogue. If so, input attributes will be modified. If not, an error message will ensue.

*Relations Accessed:* Catalogue

*Relations Updated:* Catalogue

Search other Video Game Enthusiasts within Video Game Enthusiast table:

*Input:* User Name OR age OR an upper and lower age range OR a gender OR a gender and age OR a gender and upper and lower age range.

*Steps:* The procedure will search for and display rows that match input search, or will display nothing if there is no match.

*Relations Accessed:* Video Game Enthusiast

*Relations Updated:* None

Search other Video Game Enthusiasts using Video Game ID:

*Input:* Video Game ID OR Video Game Name

*Steps:* Will check Video Game ID and display all users that have that ID in their catalogue. Or will check Video Game Name and use that to find all ID’s tied to that name, then display all users who have that ID in the catalogue.

*Relations accessed:* Catalogue. If Video game name used, Catalogue and Video Game.

VideoGameID will be found via videoGameName, then the ID will be used to reference Catalogue, which will then find userName that have videoGameID in their catalogue. Then Video Game Enthusiast will display rows where userName from Catalogue match userName in Video Game Enthusiast.

*Relations Updated:* None.

Find the number of users in database:

*Input:* Choose output desired, by userName, age or gender. Can also choose to input upper and lower range of ages.

*Output:* A table displaying the sum of us users by displaying sum of all users (count userName), sum of a particular age(cout input age), sum of chosen gender(count input gender) or sum of users within age range(count between input ranges).

*Relations Accessed:* Video Game Enthusiast

*Relations Updated:* None.

Find the average age of users:

*Input*: Choose average age of entire table, or input upper and lower range of ages, or input gender.

*Steps*: Depending on input, procedure will be called to AVG either all of the ages in the table, the ages within input range, or the ages of a gender in the table, and display it in a custom table as Average Age.

*Relations Accessed:* Video Game Enthusiast

*Relations Updated:* None.

Search Catalogue by using attributes within Catalogue by:

*Input:* User name, Video Game ID, rating or range of ratings, any combination of the play attributes(hasPlayed, wantsToPlay, owns, wantsToOwn, opposites).

*Steps:* A procedure will search the table and display all rows that match the input search parameter.

*Relations accessed:* Catalogue

*Relations Updated:* None

Search Catalogue by using attributes in Video Game Enthusiast:

*Input:* Age or range of ages, gender.

*Steps:* Procedure will find all users in the Video Game Enthusiast table that match the input parameters. It will then use the user name as a means to join with the catalogue and display the rows of Catalogue where the userName matches.

*Relations accessed:* Video Game Enthusiast, Catalogue, referenced via userName

*Relations modified:* None.

Search Catalogue by using attributes from Video Game:

*Input:* videoGameName, console, developer.

*Steps:* Procedure will take input parameter and find all videoGameID’s that match the parameter in the Video Game table. Then it will use a join using the videoGameID with the Catalogue and display all rows in the catalogue that have a matching videoGameID, or no rows if there is no match.

*Relations Accessed:* Catalogue and Video Game Table referenced via videoGameID

*Relations Updated:* None.

Find Counts in the Catalogue:

*Input:* Choose the sum of all entries in Catalogue, or input a userName and either normal or opposite attributes.

*Steps:* Procedure will either display a Count of all of the entries in the table. Or it will take a userName and produce a table with a column with the userName, hasPlayed, wantsToPlay, owns, wantsToOwn and the sum of every game that falls under the shown attribute. Or they can choose the opposite, in which userName, hasNotPlayed, doesNotWantToPlay, doesNotOwn, doesNotWantToOwn will be displayed and the sums that fall under those attributes.

*Relations Accessed:* Catalogue

*Relations Updated:* None.

Find average rating of a game:

*Input:* Input a game and whether you want overall average rating or average rating of games users have actually played.

*Steps:* One of two procedures will run. Both will take videoGameID input or videoGameName input and, after correlating between Video Game and Catalogue tables if necessary, will display the videoGameName and the average rating from all users who have rated the game in the Catalogue. The second will do the same as above, except it will take the rating by filtering the Catalogue to only include users who have marked they have actually played the game.

*Relations Accessed:* Catalogue and Video Game referenced using videoGameID

*Relations Updated:* None.

Search for video games in Video Game catalogue by the various attributes in the Catalogue:

*Input*: Input videoGameName, developer, console, genre or maturityRating.

*Steps:* Procedure will search through table and display all rows with matching input parameters, or nothing if there are no rows.

*Relations Accessed:* Video Game

*Relations Updated:* None

Search for Video Games using userName from Catalogue:

*Input:* Input a userName.

*Steps:* Procedure will take a userName input, and assuming it exists, find all of the matching userName in the catalogue. It will then take those rows videoGameID’s and reference to the Video Game by the videoGameID’s. From Video Game, it will display all rows that have the matching videoGameID.

*Relations accessed:* Catalogue and Video Game, reference each other via videoGameID.

*Relations Updated:* None

Find sums within the Video Game table:

*Input:* Any of the attributes in the Video Game Table, or run procedure to get total sum of Video Game Table.

*Steps:* If attribute input, a table will display the total sum of games that have the matching attribute (i.e: all games in the First Person Shooter genre). If a total sum was sought, it will display the total sum of all rows in the Video Game Table.

*Relations accessed:* Video Game.

*Relations Updated:* None

Search Developers by videoGameName:

*Input:* Input a videoGameName.

*Steps:* Procedure will find all entries with input videoGameName. Then it will do a join operation with the developer table to find the developer. Table will display videoGameName, developerName, developerAddress. If There is multiple developers tied to one videoGameName, then it will show that many rows and developers.

*Relations Accessed:* Video Game, Developer

*Relations Updated:* None

Find sum of Developers:

*Input:* None, simply run the procedure.

*Steps:* Procedure will display a table with the total sum of all rows in the Developer table.

*Relations Accessed:* Developer

*Relations Updated:* None

Search Consoles by videoGameName:

*Input:* Input a videoGameName

*Steps:* Procedure will find all matching videoGameName in Video Game Table. It will then do a join with Console table using consoleName and display table with videoGameName, consoleName and consoleDeveloper.

*Relations Accessed:* Video Game, Console, referenced via consoleName

*Relations Updated:* None

Search Consoles by console Developer:

*Input:* input a consoleDeveloper.

*Steps:* Procedure will find all matching rows with consoleDeveloper and display the rows.

*Relations Accessed:* Console

*Relations Updated:* None

Find sum of Consoles:

*Input:* Either run procedure for total sum, or input consoleDeveloper to find a sum of all consoles made by said developer.

*Steps:* If total sum run, a table will appear displaying the total number of rows in table. If a consoleDeveloper is sought, procedure will look add all rows that have matching consoleDeveloper and display that sum.

*Relations Accessed:* Console

*Relations Updated:* None

Add a Developer:

*Input:* Input a developer’s name and address.

*Steps:* Error will appear if developer is already in table. If not it will be added with input address.

*Relations Accessed:* Developer

*Relations Updated:* Developer

Modify a Developer:

*Input:* developer’s name. If change is to developer’s name, leave address blank. Otherwise fill in address as well.

*Steps:* If developer’s name was changed, change will occur in Developer and Video Game tables. If just the address was changed, then change will occur in address. If the developer’s name does not exist, then an error will display.

*Relation Accessed:* Developer, possible Video Game

*Relations Updated:* Developer, possibly Video Game

Add a Console:

*Input:* Input a console’s name and its developer.

*Steps:* Error will appear if developer is already in table. If not it will be added with input developer.

*Relations Accessed:* Console

*Relations Updated:* Console

Modify a Console:

*Input:* consoleName. If change is to console’s name, leave consoleDeveloper blank. Otherwise fill it in as well.

*Steps:* If console’s name was changed, change will occur in Console and Video Game tables. If just the consoleDeveloper was changed, then change will occur only in Console. If the console’s name does not exist, then an error will display.

*Relation Accessed:* Console, possible Video Game

*Relations Updated:* Console, possibly Video Game

Add a Video Game:

*Input:* videoGameName, developer, console, genre(optional), releaseDate(optional), maturityRating(optional).

*Steps:* procedure will see if the combination of videoGameName, developer and console already exist. If they do, then an error will occur. If not, a videoGameID will be generated and the row will be added to the table.

*Relations Accessed:* Video Game

*Relations Modified:* Video Game

Modify a Video Game:

*Input:* videoGameID, then all attributes to be modified.

*Steps:* if the videoGameID exists, the input attributes will be modified.

*Relations Accessed:* Video Game

*Relations Modified:* Video Game.

Delete a Video Game Enthusiast:

*Input:* userName.

*Steps:* if userName does not exist, an error will return. Otherwise, the associated row will be deleted from Video Game Enthusiast, and subsequently so will all the rows in the Catalogue that match that userName.

*Relations Accessed:* Video Game, Catalogue

*Relations Modified:* Video Game, Catalogue

Delete a Catalogue:

*Input:* userName

*Steps:* if userName does not exist, an error will be displayed. Otherwise the row will be deleted from the Catalogue.

*Relations Accessed:* Catalogue

*Relations Modified:* Catalogue

Delete a Video Game:

*Input:* videoGameID

*Steps:* If videoGameID does not exist, an error will return. If it does the row will be deleted from the table. Subsequently all rows in the Catalogue that share the videoGameID will also be deleted.

*Relations Accessed:* Catalogue, Video Game

*Relations Modified:* Catalogue, Video Game

Delete a Developer:

*Input:* developerName

*Steps:* if developerName does not exist, an error message will display. Otherwise developer will be deleted from developer table, and subsequently all entries in Video Game table made by said developer will be deleted, as such all entries in Catalogue whose video game ID’s match those just deleted will also be deleted.

*Relations Accessed:* Catalogue, Video Game, Developer

*Relations Modified:* Catalogue, Video Game, Developer

Delete a console:

*Input:* consoleName

*Steps:* if consleName does not exist, an error message will display. Otherwise, if it does, the row will be deleted from console. In addition, all video game titles in Video Game that have that console attribute will be deleted. Also, all Catalogue entries that have previously deleted video game id’s will also be removed.

*Relations Accessed:* Catalogue, Video Game, Console

*Relations Modified:* Catalogue, Video Game, Console

**Design Part 2**

**BCNF Verification:**

Video Game Enthusiast:

userName -> age

userName -> gender

userName -> fName

userName -> lName

userName -> password

The Video Game Enthusiast table is automatically 2nf because the primary key, userName, is not dependent on age, gender, fName or lName, or password. This is so because multiple users can have the same age, the same gender, the same fName, lName and password and can be logged in and have multiple catalogues, but it’s all tied back to the userName. As such it is also in 3NF, and since the non primary key attributes do not in any way determine the primary key userName, it is in BCNF.

Catalogue:

userName -> videoGameID

userName -> rating

userName -> comment

userName -> hasPlayed

userName -> wantsToPlay

userName -> owns

userName -> wantsToOwn

The Catalogue table is automatically 2nf because the primary key, userName, is not dependent on any of the other attributes. In fact, since other users can have the same videoGameID’s in their catalogue, similar ratings and comments and similar play attributes, it is also 3NF. And, since none of the non-primary key attributes have no bearing on the userName, the Catalogue relation is in BCNF.

Developer:

developerName -> developerAddress

The catalogue is in 2NF because the primary key, developerName, is not depended on the address. In fact multiple developers can be at the same address but still be their own independent developers. So it is also 3NF. And since the address has no bearing on developerName, this relation is in BCNF.

Console:

consoleName -> consoleDeveloper

The catalogue is in 2NF because primary key, consoleName, is not depended on consoleDeveloper. In fact, many consoles can have the same developer but still be their own console (i.e: playstation 4 made by sony, but so is playstation 3 and playstation 2). So it is 3NF. Also, since the developer has no bearing on the console, relation is in BCNF.

Video Game:

videoGameID -> videoGameName

videoGameID -> consoleName

videoGameID -> developerName

videoGameID -> maturityRating

videoGameID -> genre

videoGameID -> releaseDate

The catalogue is in 2NF because the primary key, videoGameID, is not dependent on any of the other attributes. It is possible to have several video games of the same name and have a different ID, also have several games play the same consoles without being tied to an ID. The same goes with developerName, as multiple games can be made by the same developer, and all games share some kind of maturityRating, genre or releaseDate, but none are exclusive. Because of such, it is also 3NF and BCNF.

**Stored Procedures:**

Since there are many stored procedures for this database, they will be included in a separate file.

**Testing**

**Test Plans**

The plan is to test every stored procedure that supports the use case function. This will be done as follows:

* Test search procedures
* Test aggregate operations
* Test modify procedures
* Test add procedures
* Test delete procedures.

**Test Results**

Most tests went without issue. Or that is to say since there were so many similar procedures most tests were fixed by fixing any issues found in the first couple of tests. So those will be listed, along with their fixes.

Search Procedures: The searches were successful in returning correct data. The issue found was the order in which the columns were displayed. It was hardly aesthetic or practical.

Fix: Ensure tested and any further search procedures orders the rows in an organized and pleasing manner. For example: Video Game Enthusiast searches would return userName, age and gender in that order each time to be consistent.

Aggregate Operations: All tests for sums or averages were successful. Only issues were properly naming the columns that would store the answers.

Fix: Rename the columns

Modify, add and delete procedures: While these procedures were successful,they were impractical as they required too much information to identify which tuple to change in the Video Game and the Catalogue tables.

Fix: Normalization. Instead of depending on videoGameName, developerName and consoleName to uniquely identify a video game, a videoGameID was created using those criteria and then became a foreign key for the Catalogue table and the primary key for the Video Game Table.

**Conclusion**

A social video game database was successfully created that caters to the pre-described use cases. It has five entities, all of which interact with at least one other entity, and adequate aggregate operations. It returns values based on numerous search operations and is easy to add, delete, and modify data in the tables. Full functionality of having username and passwords that are enforced, as well as an interface with which to use them was not created. But the underlying database and its procedures was and works as intended.

**References**

Connolly, Thomas M., and Carolyn E. Begg. Database systems: a practical approach to design, implementation, and management. Pearson Education, 2005.