Video Game Sales Database

Tae Yong Namkoong, Nate Sackett, Hyuk Joon Yang

CS 564, Summer 2020 : Professor Nguyen

**1. Introduction**

**1.1 Motivation**

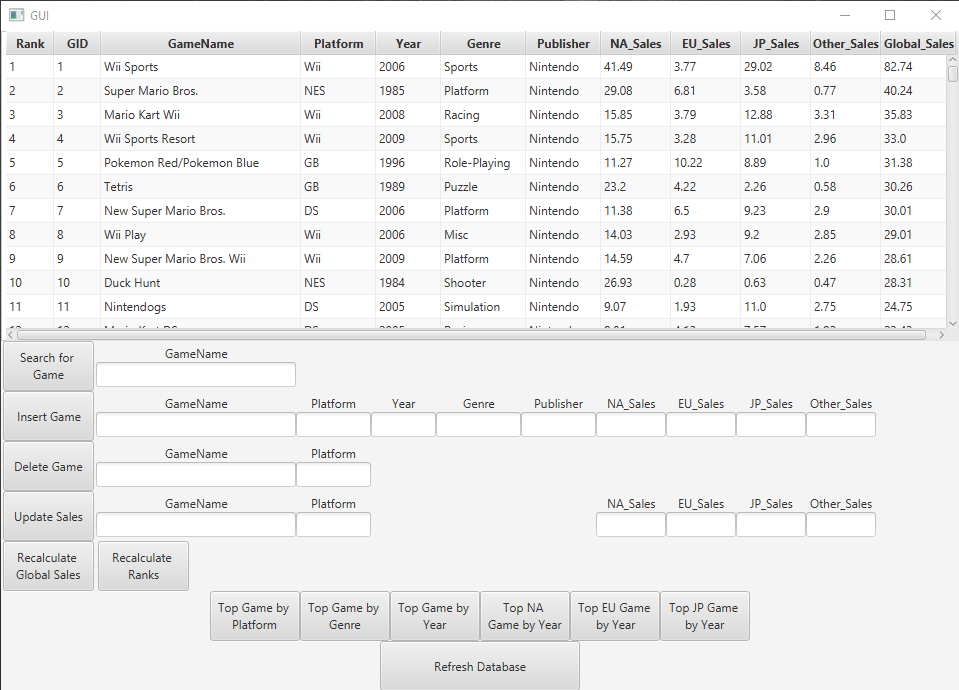
We had several motivations for choosing our domain; all our group members were all interested in gaming. Our primary motivation was to analyze and predict sales trends in the gaming industry. By applying our knowledge that we have learned in DBMS, we thought that this would provide meaningful demographic data in the gaming industry which can provide game designers with consumer demand, buying behavior, and consumption patterns in the gaming market.

There is a great need for DBMS because millions of games are sold every year and the video gaming industry is rapidly growing and expanding. In order to track the success of a particular game, we have to view it in relation/context with various categories such as platform, genre, company, regional sales, and in relation to competitors. By integrating relations between categorical information, we believe that this will provide meaningful information to users through an intuitive and easy-to-use GUI that we have implemented.

**1.2 Application Description**

Our application is standalone. Our back-end uses stored procedures and queries in MySQL which interfaces with our JavaFX GUI in the front-end. Our application enables users to view the games in the database, sort them by year, genre, etc., and to add, delete, or search for any game. We chose to host the database locally; we had initially planned to host the database online, but changed to make implementation easier.

The front-end is written in Java, with the GUI implemented using JavaFX and SceneBuilder, and the MySQL interface handled using JDBC. The original data is parsed from a csv file and then stored in MySQL. The data can be refreshed at any time by deleting the current tables and refilling them from the CSV file.



The application allows the user to view all 16,000+ games at once. The user can see the game ranking (in terms of overall sales), the ID of the game, its name, the platform it appears on, the year it was released, the genre, publisher, sales data in millions of units sold by region for North America, Europe, Japan, Other, and Global regions (with Global being the sum of all other regions). The user can search for any game using exact or partial match to its name, and all matching games are returned to the current table. The user is also allowed to insert a new game, the Game ID and Rank will automatically be calculated, and the table is automatically refreshed. The user can also Delete a game or update the Sales data for the game using the provided text boxes and the delete or update button. Finally, the user is also given several shortcuts to view the top game by platform, genre, or year, or the top game by year in the NA, EU, or JP regions.

**1.3 Task Assignment**

* Task assignment for each team memberA screenshot of a cell phone

  Description automatically generated

**1.4 Final Report Table of Contents**

Section 1 : Introduction

1.1 Motivation

1.2 Application Description

1.3 Task Assignments

1.4 Final Report Table of Contents

Section 2 : Implementation

2.1 System Architecture

Diagrams

Explanation of Frontend/Backend

2.2 Description of the dataset

2.3 ER diagram

2.4 Relational model

2.5 Implementation: Description of the prototype

2.6 Evaluation

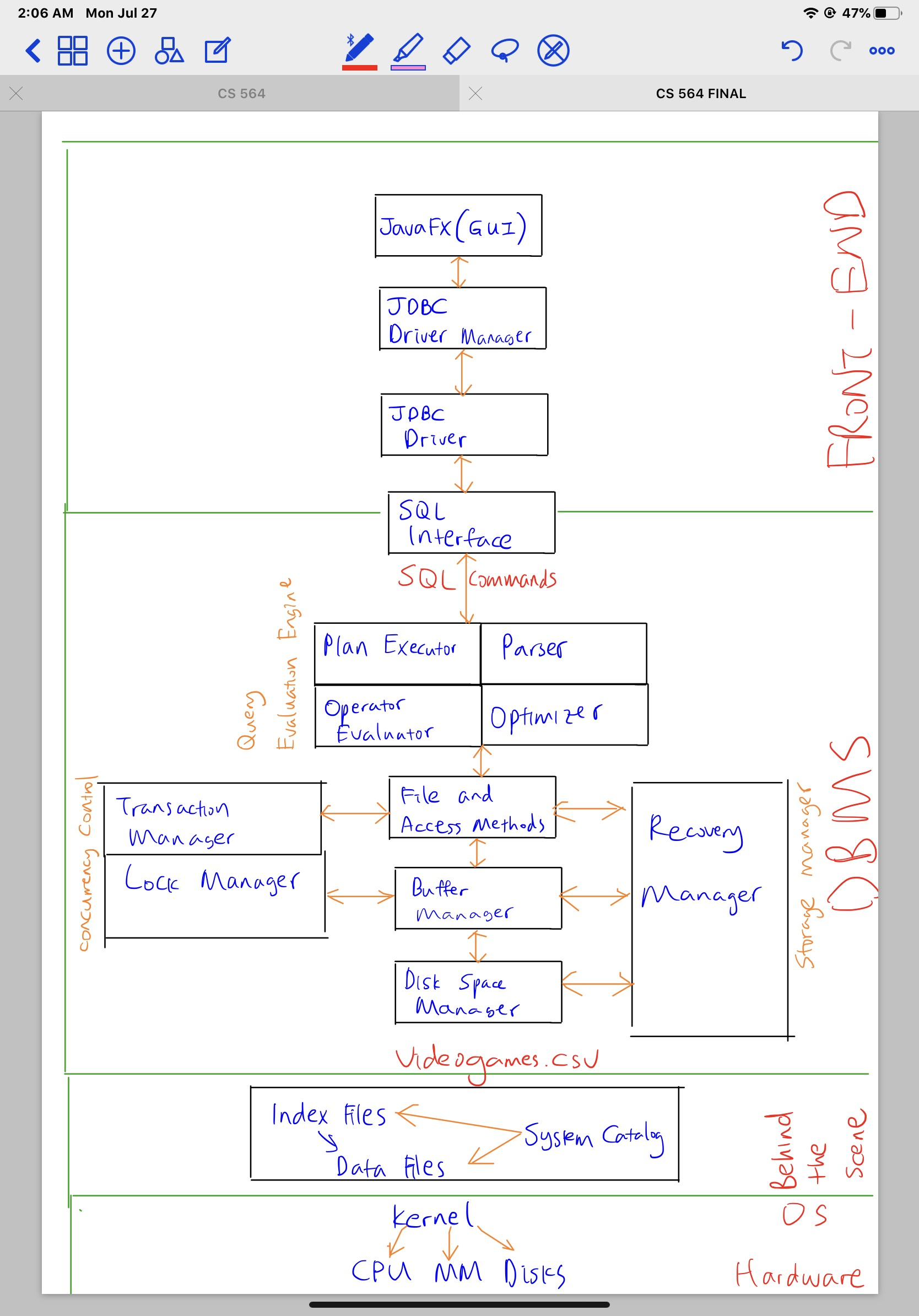
Section 3 : Conclusion

3.1 Lessons Learned/Knowledge Gained

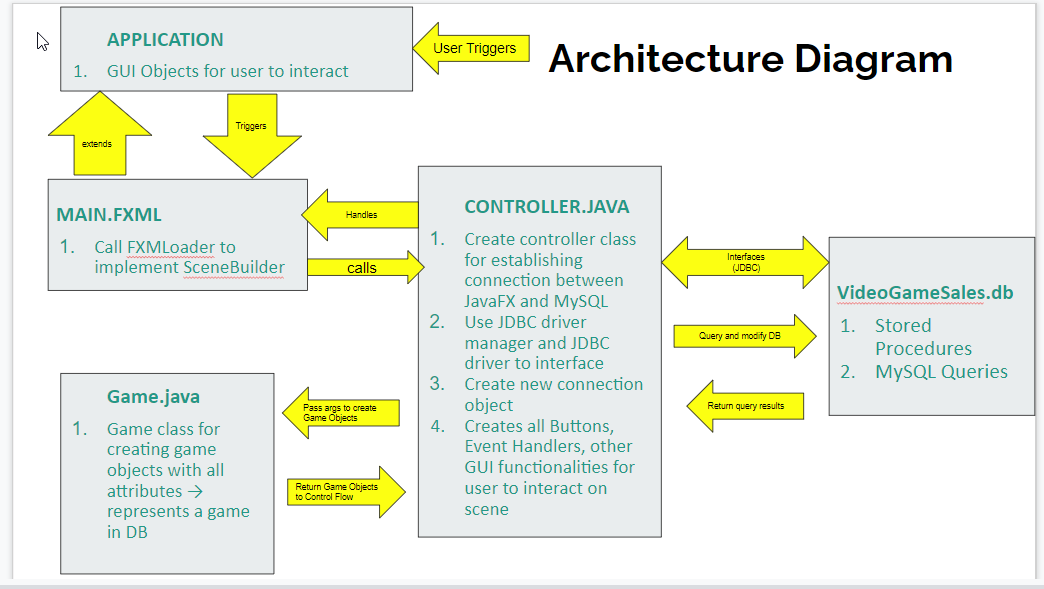
3.2 DB related issues

**2. Our Implementation**

**2.1 Description of the system architecture**

Our conceptual design:

Our final design:



Backend:

The backend consists of 4 MySQL tables, each describing the games, publishers, rankings, and sales data. We have stored procedures to update the global sales based on the other sales data, the rank based on the global sales, various sortings for the data, and one to view the current state of the database. We have additional stored procedures for the user to add, delete, update sales, or search for a game.

Frontend:

The frontend is written in Java; it consists of four classes and an fxml file. The Main class launches the JavaFX GUI and nothing else. Main.fxml is the xml description of the GUI, its elements, and the functions implemented when a button is clicked; it was automatically generated using SceneBuilder, a 3rd party extension for JavaFX. SceneBuilder allowed for easy development and rapid changes to the GUI layout. The Game class is the Java representation of a tuple (consisting of the joined values of the games, publishers, rankings, and sales tables) from the MySQL database. The SQLConnection class uses JDBC to establish a connection between the MySQL database and this Java project. That connection is then exported to our final class for use in displaying the game data and implementing the various functions allowed by our GUI.

The final Java class is the controller class, and this is where the bulk of the workload is handled. The Controller class initializes the fields in the GUI, the main components being the table used to display the game data, the TextFields that allow the user to input/alter the database, and the buttons used to implement the various stored procedures and queries in MySQL. The Controller class implements a function for each button in the GUI, translating the SQL query into a Statement for MySQL to execute, then handling the returned ResultSet, displaying the results in the GUI’s table. This was by far the most time-intensive portion of this project, taking 40-50 hours to implement. (However, much of that time was spent learning JavaFX, SceneBuilder, JDBC, and attempting to implement various failed projects like dynamically renaming and allocating table columns.)

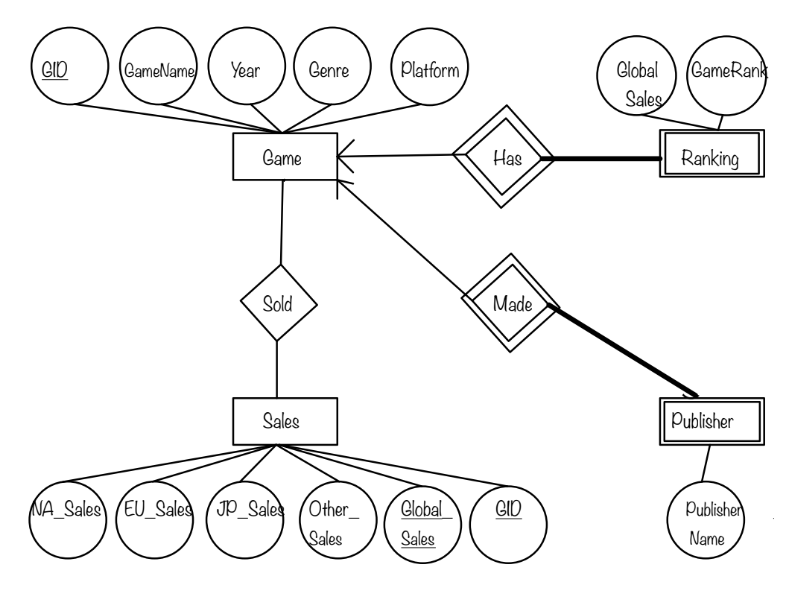
**2.2 Description of the dataset**

The dataset was obtained from Kaggle.com and is titled videogames.csv. It contains the names of video games, the platform on which the games were released, the year of publication, their genre, publisher, and the sales data for North American, European, Japanese, Other, and Global regions. Our dataset has 16,591 rows and 11 columns, which met the minimum requirement of at least 10 columns and 10,000 columns. We chose this dataset because it was relatively clean and came from a single-source, which required no multiple source integration and minimal cleaning.

We had a few challenges handling the dataset. The global sales attribute determines the rank for each game, but several ranks were missing, so we knew that we needed to recalculate the game ranks. Next, we realized that global sales was not consistent with regional sales; the dataset had rounding errors or wrong global sales data for some games, so we knew that that needed to be recalculated based on the regional sales data. Lastly, we found that there was no way for the Games Table to be in either BCNF or 3NF without adding something to the table. GameName was a non-unique, games with the same name were released for multiple systems or in multiple years or even by multiple publishers. A candidate key for the Game table would have required a composite key consisting of the name, platform, year, and publisher, which is not in 3NF with the multiple functional dependencies derived from this. The only unique attribute was the game rank, but we knew that the game rank was going to change as the sales data updated.

Solution : Normalization:

To resolve these issues, we created a Game Identification Number, or GID. We had a unique quantity, the game rank, but that value was not static, so we created the GID based on the original, unique game rank. We then made every table 3NF by making all columns functionally dependent on the GID. By letting GID be the primary key, we were able to achieve 3NF and ensure a lossless and dependency preserving decomposition. For the sales table, since global sales is the sum of all regional sales, we had to make both GID and global sales as the primary key. ER diagram (final version from the previous checkpoint copied here)

**2.3 ER Diagram**

**2.4 Relational Model**

i. Game Table

Game (GID: BIGINT, GameName: String, Platform: String, Year: String, Genre: String)

* GID - BIGINT;
* GameName - String; ≤ 135 Characters
* Platform - String; ≤ 15 Characters
* Year - String ≤ 5 characters (Some games has no data for year)
* Genre - String; (Action, Adventure, Fighting, Misc, Platform, Puzzle, Racing, Role-Playing, Shooter, Simulation, Sports, Strategy)
* Primary Key : GID

ii. Sales Table

Sales(GID: int, NA\_Sales: double, EU\_Sales: double, JP\_Sales: double, Other\_Sales: double, Global\_Sales: double)

* GID – BIGINT
* NA\_Sales – Positive double; 0.0 ≤ Sales ≤ 100,000
* EU\_Sales – Positive double; 0.0 ≤ Sales ≤ 100,000
* JP\_Sales – Positive double; 0.0 ≤ Sales ≤ 100,000
* Other\_Sales – Positive double; 0.0 ≤ Sales ≤ 100,000
* Global\_Sales – Positive double; 0.0 ≤ Sales ≤ 100,000 (Primary)
* Primary Key : GID, Global\_Sales
* Foreign Key : (GID) references Game(GID)

iii. Publisher Table

Publisher (GID: int, PublisherName: String)

* GID – BIGINT; ≤ # of rows in table (Primary)
* PublisherName - String; ≤ 50 Characters
* Primary Key : GID
* Foreign Key : (GID) references Game(GID)

iv. Ranking Table

Ranking (GID: int, GameRank: int, Global\_Sales: double)

* GID – Unique Ranking; ≤ # of rows in table (primary)
* GameRank - Unique Ranking; ≤ # of rows in table (Primary)
* Global\_Sales – Positive double; 0.0 ≤ Sales ≤ 100,000
* Primary Key : GID
* Foreign Key : (GID, Global\_Sales) references Sales(GID, Global\_Sales)

Entity Sets Explained:

1. Game entity has attributes GID (Unique id for each game), GameName (Name of each game), Year (Year the game is released), Genre (Genre of the game), and Platform (Platform that the game is based on, such as PC, Nintendo.
2. Sales entity set contains NA\_Sales (Sales in North America), EU\_Sales (Sales in Europe), JP\_Sales (Sales in Japan), Other\_Sales (Sales in other regions), Global\_Sales (Total sales), and GID (Unique id for game). We can use GID as the primary key but we need the Global\_Sales from Sales in the Ranking. As we cannot define foreign key constraint, we set (GID, Global\_Sales) as primary key
3. Publisher entity set is a weak entity set associated with Game entity set. It has PublisherName (Name of publisher). All entities in the Publisher entity set participate in the relationship with the Game entity by Many to One relationship constraint. Also, some games do not have publishers so we decide to define the Publisher entity set as a weak entity set.
4. Ranking entity set contains Global\_Sales (Total Sales) and GameRank (Rank of the game). All entities in the set participate in the relationship with the Game and all entities in the Ranking entity set are associated with one entity in the Game entity set. Furthermore, some ranks can have the same global sales while the rank should be constantly updated when there is a change in the global sales. Therefore, we define Ranking entity set as a weak entity set.

We used an ER approach to translate ER diagrams into a Relational model. Therefore, every weak entity set is translated into relations including key attributes of its superclass entity set and all attributes of its entity set.

Non-Trivial Functional Dependencies

1. *Non-Trivial FDs in relation Game*

GID → GameName, Platform, Year, Genre

GameName, Platform → GID

1. *Non-Trivial FDs in relation Sales*

GID → EU\_Sales, NA\_Sales, Other\_Sales, JP\_Sales, Global\_Sales

1. *Non-Trivial FDs in relation Ranking*

GID → Ranking

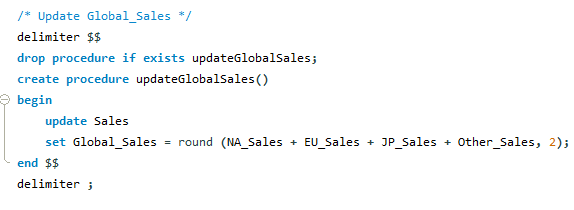
Note : This is initially a trivial FD but becomes a non-trivial FD as we update rank.

1. *Non-Trivial FDs in relation Publisher*

GID → Publisher\_Name

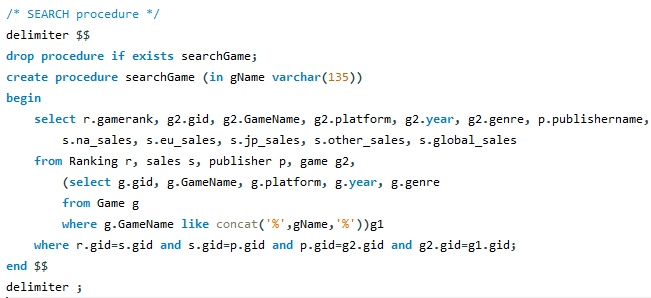
**2.5 Implementation: Description of the prototype**

Stored procedures and queries in DBMS

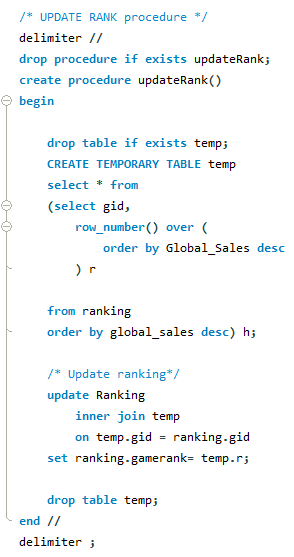
1. Stored procedures

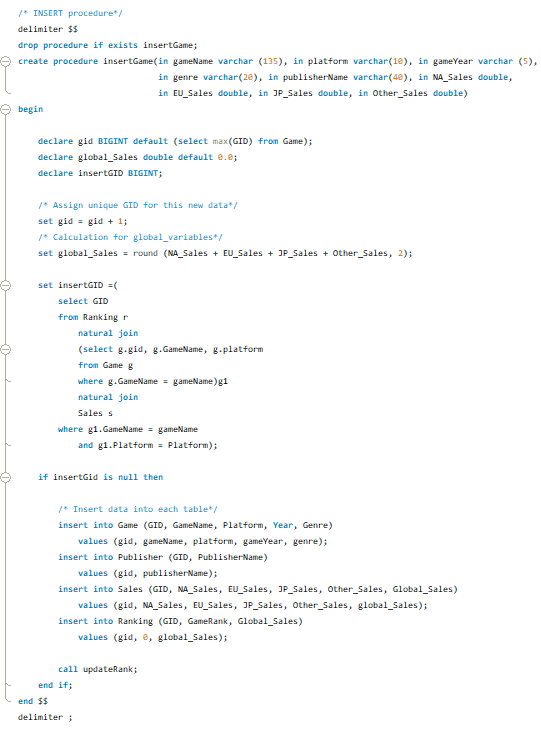
i. updateGlobalSales

ii. This procedure helps the DBMS to correctly update the Global\_Sales of each entity in the Sales entity set according to the sales in North America, Europe, Japan and Other regions. This procedure is called whenever there is a change in the sales of any region.

ii.searchGame

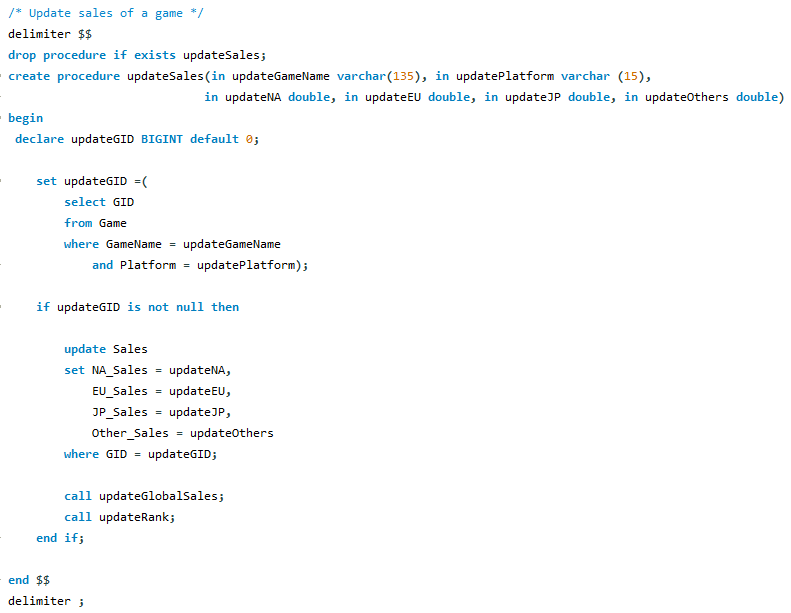
This procedure reads user input from Java to return all data of the game the user wants to search. If a user types a certain name of a game, this procedure searches all games that have the user input in their names. If there are several entities that have the string value in their game names, it returns a list of those entities. This procedure helps the user to see the sales of the game containing the name the user wants to search.

iii. updateRank  
  
This procedure helps the DBMS to keep the rank of the game updated according to the changes in the global sales of the game. It creates a temporary table that copies all instances in the Sales relation and sorts them by the Global\_Sales. Then the GameRank of Sales relation is updated by the row number where the GID of the instance is placed in the temporary table. 



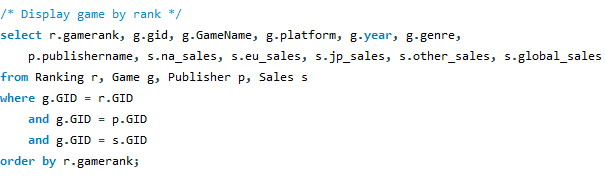
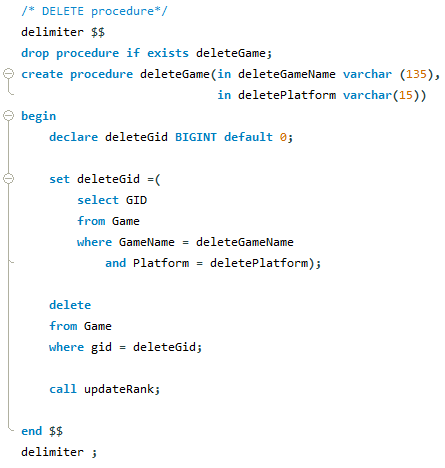
iv. insertGame  
  
This stored procedure inserts a new instance(Game) from user input into the database. It automatically calculates the available GID for the new instance. Then it searches if there is an existing game with the same name and platform in the database. If the game does not exist in the database, the instance is inserted into each relation with the values from the parameters. If the game already exists, the game is not inserted into the database.

v. deleteGame  
  
This procedure is called when the user wants to delete a certain instance from the database. The procedure uses the name and the platform of a game as parameters because the attributes are super key in the relation Game. If this procedure finds the GID of a game to be deleted, it deletes the instance from the Game relation. Then all instances from other relations will be deleted by CASCADE operation. If there is no such game, no deletion of an instance is performed.

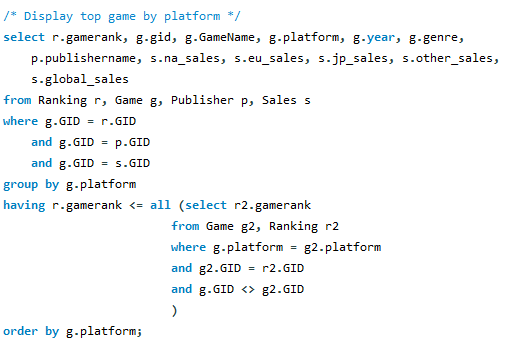


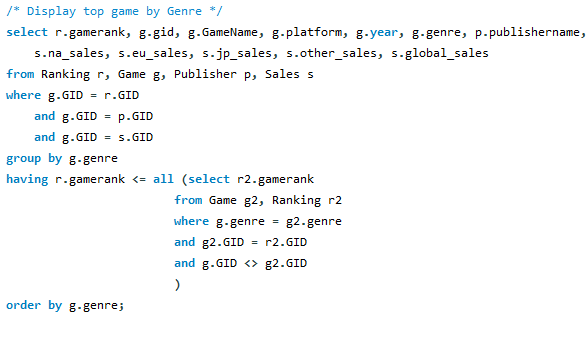
vi. updateSales  
  
This stored procedure helps the user to update changes in the sales of a game stored in the database. It first finds the GID of the game user wants to update sales. If there is no such game in the database, no update is performed. If the game is found, then it updates the sales in North America, Europe, Japan and Other regions by the parameters. Then it calls updateGlobalSales to update the Global\_Sales of each entity set and updateRank to update the rank of each game.

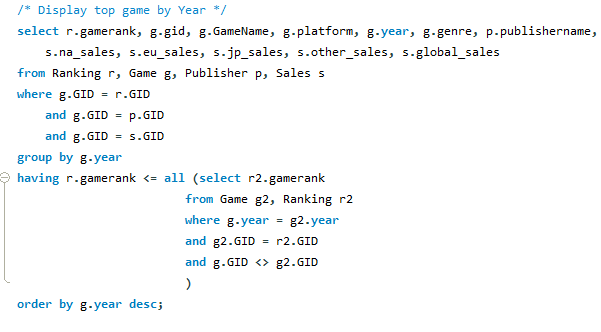
1. Queries

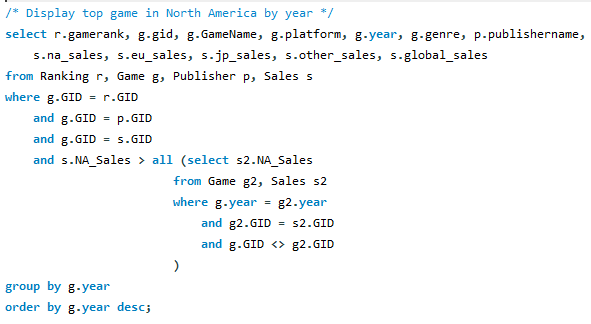


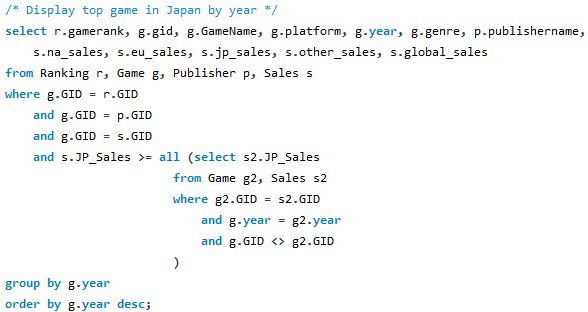
i.Display game sorted by rank  
  
This query displays all games stored in the database sorted by the rank. This query will be called to display all data in our interface.

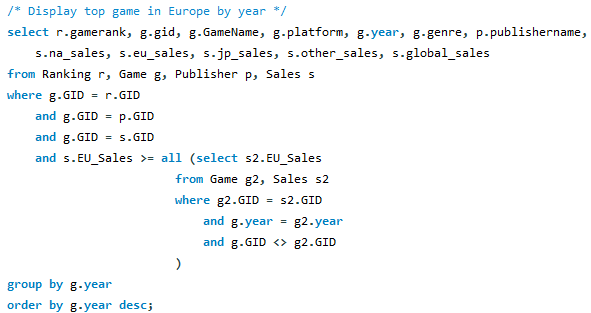
ii. Display top game by each platform  
  
This query is called to display the top game for each platform in the database. This helps the user see the top game by each platform

iii. Display top game by genre  
  
This query is called to display the top game by genre.

iv. Display top game per year  
  
This query is called to return the best-selling game in each year. From this data, the user can understand the change of consumer’s preferences over years.

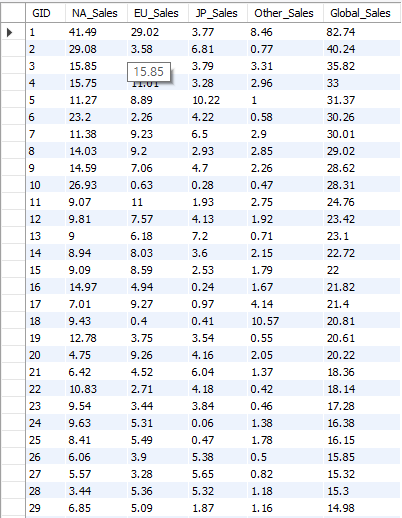
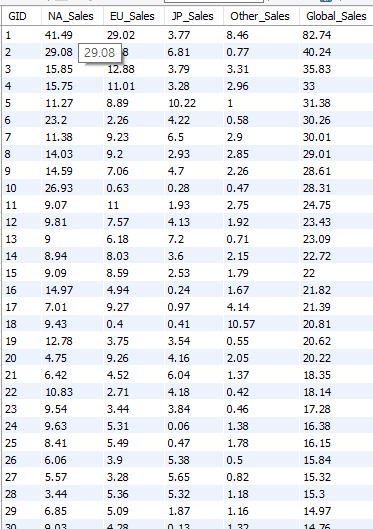
v. Display top game in North America by year  
  
This query returns a table that contains the top game sold in North America by each year. The user can understand changes in the sales trend of the game market in North America from this query.

vi. Display top game in Japan by year  
This query returns a table that contains the best selling game in Japan each year. The user can refer to this query to understand the changes in the sales trend of the game market in Japan.

vii. Display top game in Europe by year  
This query returns a table that contains the best selling game in Europe each year. The user can refer to this query to understand the changes in the sales trend of the game market in Europe.

**2.6 Evaluation:**   
We evaluate our project by checking to see if our application can successfully and correctly add, remove, modify, and search games based on user input and event triggers, and checking to see if attributes such as rank and global sales for each game are updated consistently.   
  
Below, we have tested each stored procedure that reads user input and updates the rank in our database. We test our stored procedure by comparing the expected result based on the given CSV file with the actual result returned after calling each stored procedure.

1. updateGlobal Sales

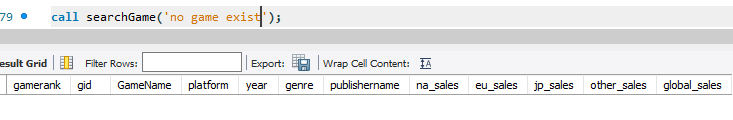
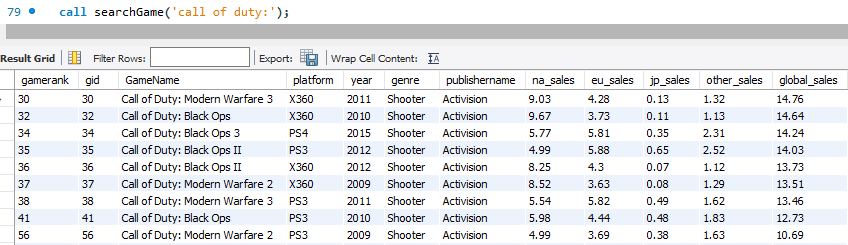
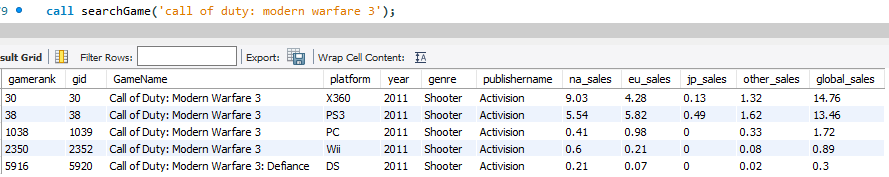
→  
Left table displays the initial values of Global\_sales for each instance in the dataset. In the initial table, we found there are miscalculations of global\_sales in many games. For example, GID: 3, 5, 8, 9, 11, 12, 13, 14 had different results from our calculations. So we implemented this procedure to check if the procedure correctly updates the values of Global\_sales for each instance by comparing with our calculations. In fact, the procedure updates all the values exactly the same as our calculations. Since the calculation method in the procedure is the same as our calculation method while the tests are successful, we can evaluate this procedure to be successful.

1. updateRank()

→

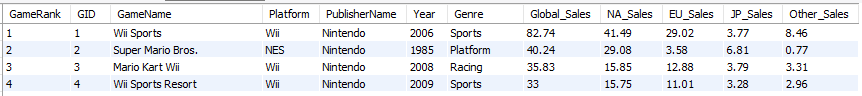
Left table is the initial table when the dataset is first imported into the database. We can find the game rank is not consistent because the next rank of 1531 is 1533. 1532 is missing in the rank. Furthermore, we imported 16591 instances so we expect the last rank to be 16591 but the last rank was bigger than our expected rank. After implementation of the stored procedure, we confirmed that the rank was successfully updated because the rank became consistent while the last rank became 16591 as we expected.

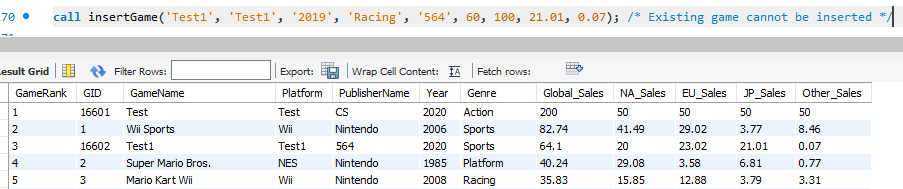
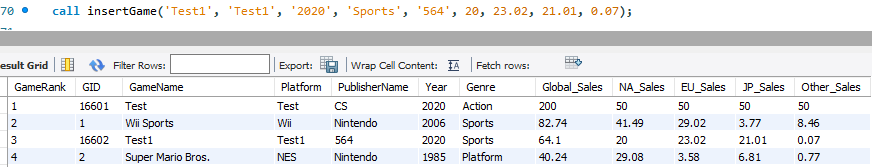
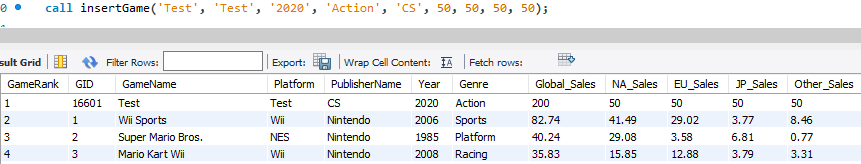
1. SearchGame()



First scenario is to search for a game that has only one instance that has the same game name. We found there are 5 different platforms for ‘Call of Duty: Modern Warfare 3’ in the dataset so we tested the procedure if it returns 5 instances. As shown in the top most table, the procedure returns the instances we expected.  
Second scenario is to search for a game that has the string in its name. We found there are 48 games containing ’Call of Duty:’ in their names so we tested the procedure if it returns 48 instances when we call the procedure with ‘Call of Duty’ as input. The procedure returns all 48 instances that we expected to be returned.   
Last scenario is to search for a game that does not exist. We expect no instance to be returned from the procedure, and the procedure returned no instance.  
Since 3 different scenarios confirmed the procedure returns expected instances, the procedures are tested to be correct.

1. insertGame()

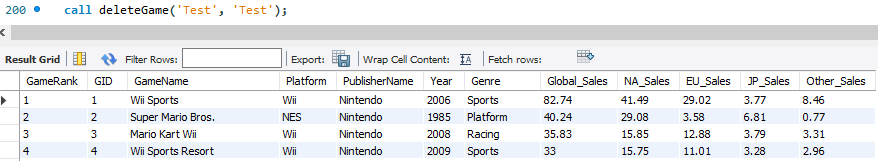
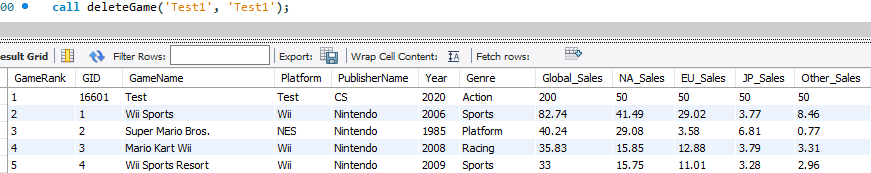
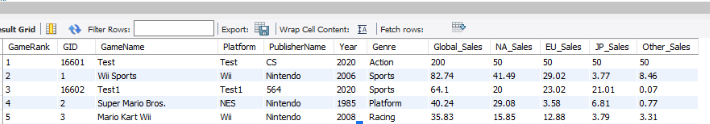




The first table is the initial data. We first tested if the procedure inserts correct values of a game with the updates of rank.We expected the game ‘Test’ to be ranked as first place by giving largest values for all sales in regions. After implementing the procedure, we found the game is ranked as first with the correct value for Global\_Sales.

Then we tested to insert a new game ‘Test1’ that should be ranked as 3rd. This scenario is to test if the procedure can insert a new game that is in the middle of ranks in the database. The procedure is confirmed to insert the game ‘Test1’ successfully.  
Last scenario is to test inserting a game that already exists in the database. We expected there should be no change in the database. As the last table shows, there was no change in the database.  
Since the procedure passes 3 different scenarios, the test performs successfully.

5. removeGame()

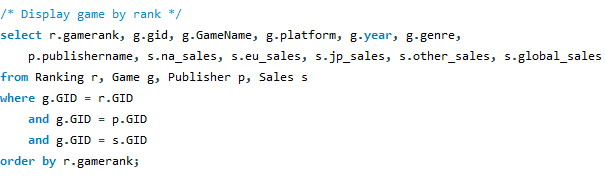


The first table is the initial table before removeGame is implemented. We tested if the rank of a given game is correctly updated after deleting a game, and if the game is successfully deleted. Deleting a game in the middle of rank and deleting a game in top rank returns expected instances. Then we test deleting a game that does not exist in the database and the procedure returns all instances with no change, which we expected to be returned. Therefore, the procedure performs correctly.

Evaluation of queries

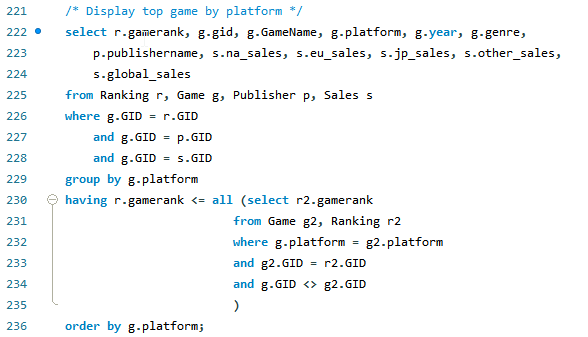
For queries, we tested queries by checking if the instances of the result table are correctly provided by the purpose of the queries.

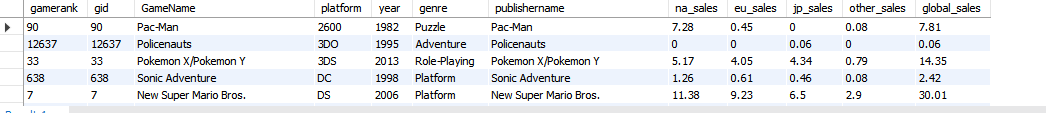
1. Display Game by Rank



This query displays all games sorted in descending order of their rank. We tested by sorting the games by their global\_sales since a game with the highest global\_sales should also be ranked at the top of our list and then checking if the games ranked higher do have higher global\_sales than games ranked below the list. The image below shows the output when the SQL query is run. Clearly, the output matches our expected output since the games with higher global\_sales are ranked higher than the games with lower global\_sales for all games.

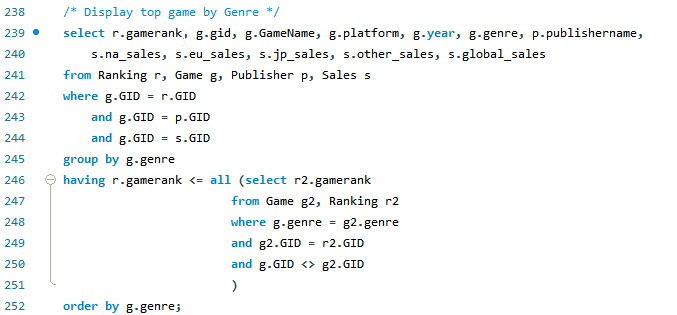
1. Display Top Game By Platform

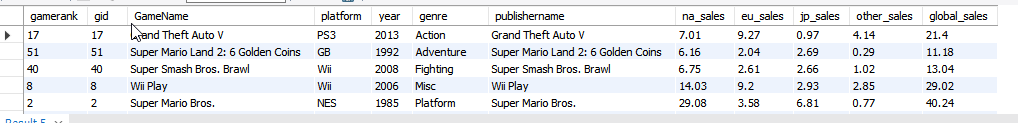




We tested this query by first grouping the games by their platform. Then for each distinct platform, we use a subquery to find the top rank for that platform. The image below shows the output of our SQL query. If our query runs correctly, an expected output will display distinct platform names and the top game for each unique platform. Clearly, the platform names are all unique and each platform has the highest global\_sales, since we only display the top game for each platform. Therefore, the output of our query corresponds to our expected output.

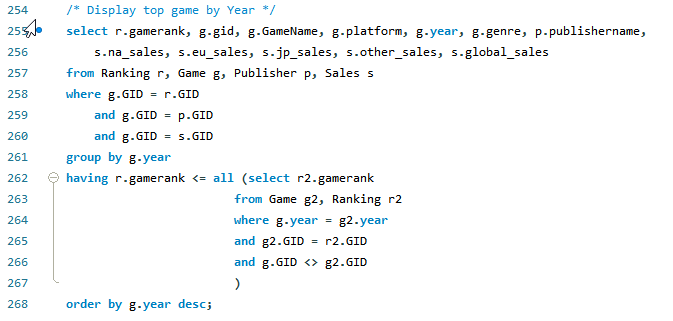
3. Display Top Game By Genre

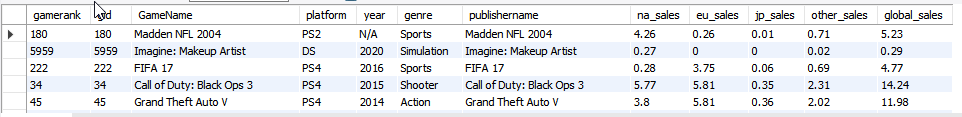




We tested this query by first grouping the games by their genre. Then for each distinct genre, we use a subquery to find the top rank for that genre. The image below shows the output of our SQL query. If our query runs correctly, an expected output will display distinct genres and the top game for each unique genre. Clearly, the genres are all unique and each genre has the highest global\_sales, since we only display the top game for each genre. Therefore, the output of our query corresponds to our expected output.

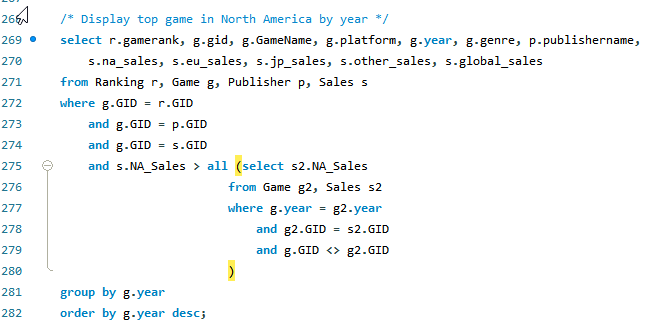
4. Display Top Game By Year

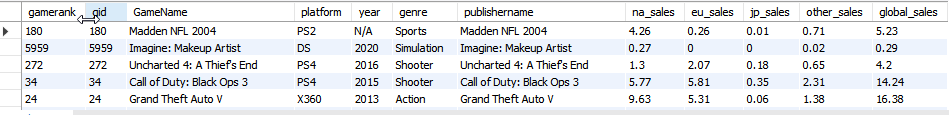




We tested this query by first grouping the games by their year. Then for each distinct year, we use a subquery to find the top rank for that year. The image below shows the output of our SQL query. If our query runs correctly, an expected output will display distinct years and the top game for each unique year. Clearly, the year values are all unique and each year has the highest global\_sales, since we only display the top game for each year. Therefore, the output of our query corresponds to our expected output.

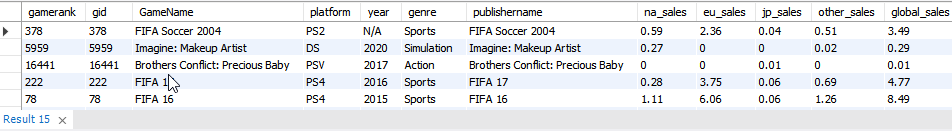
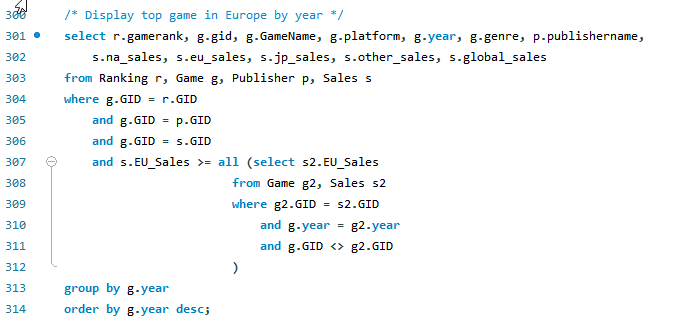
5. Display top game in North America by year





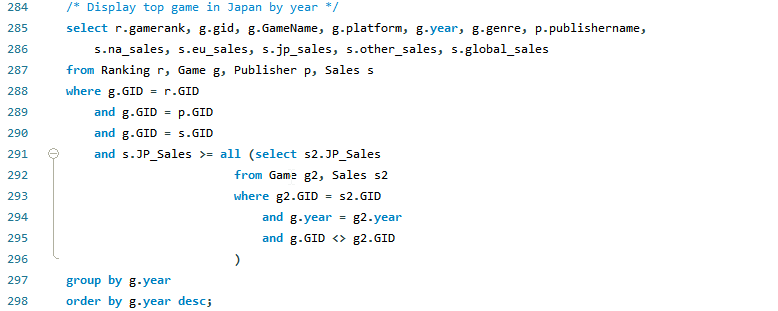
We tested this query by first sorting the game by the highest NA\_Sales which corresponds to the subquery. Then, we grouped the sorted games by each distinct year and ordered it in descending order of NA\_Sales. The image below shows the output of our SQL query. If our query runs correctly, an expected output will display tuples with games that have the highest NA\_Sales for each year. One way we tested this was to see if the returned year values were unique and then checking which game in each year actually had the highest NA\_Sales. Clearly, the year values are all unique, and each year had the corresponding highest NA\_Sales for that year, which matched the expected output.

6. Display top game in Europe by year



We tested this query by first sorting the game by the highest EU\_Sales which corresponds to the subquery. Then, we grouped the sorted games by each distinct year and ordered it in descending order of EU\_Sales. The image below shows the output of our SQL query. If our query runs correctly, an expected output will display tuples with games that have the highest EU\_Sales for each year. One way we tested this was to see if the returned year values were unique and then checking which game in each year actually had the highest EU\_Sales. Clearly, the year values are all unique, and each year had the corresponding highest EU\_Sales for that year, which matched the expected output.

7. Display Top Game in Japan



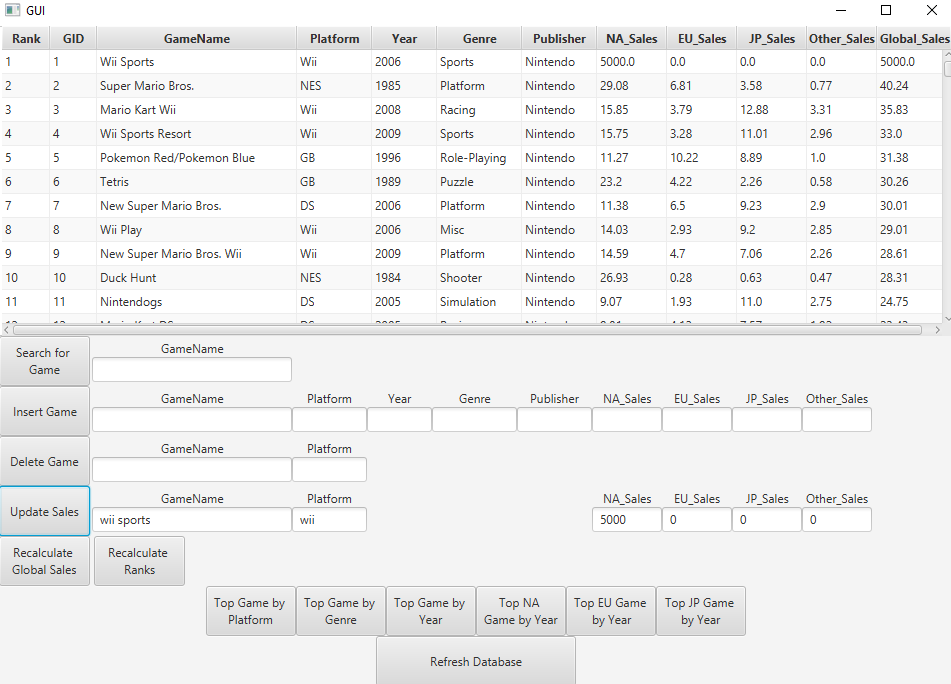
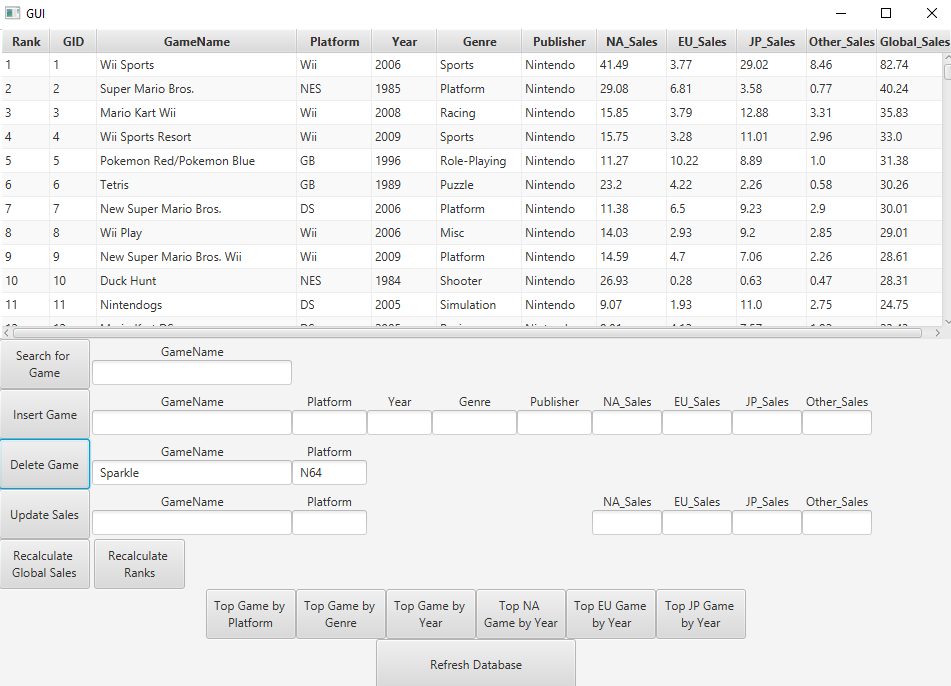
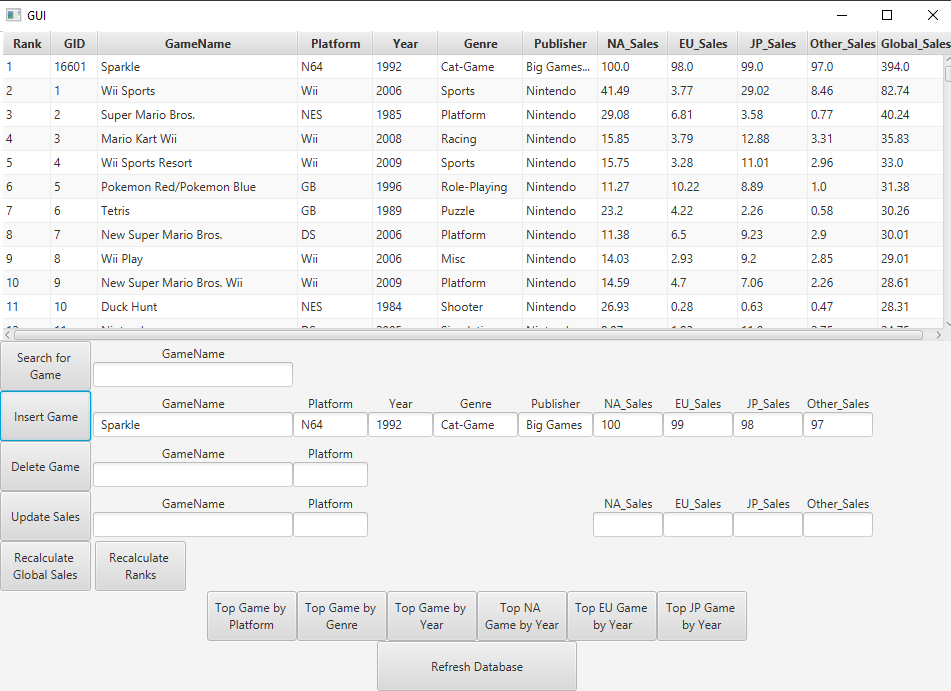
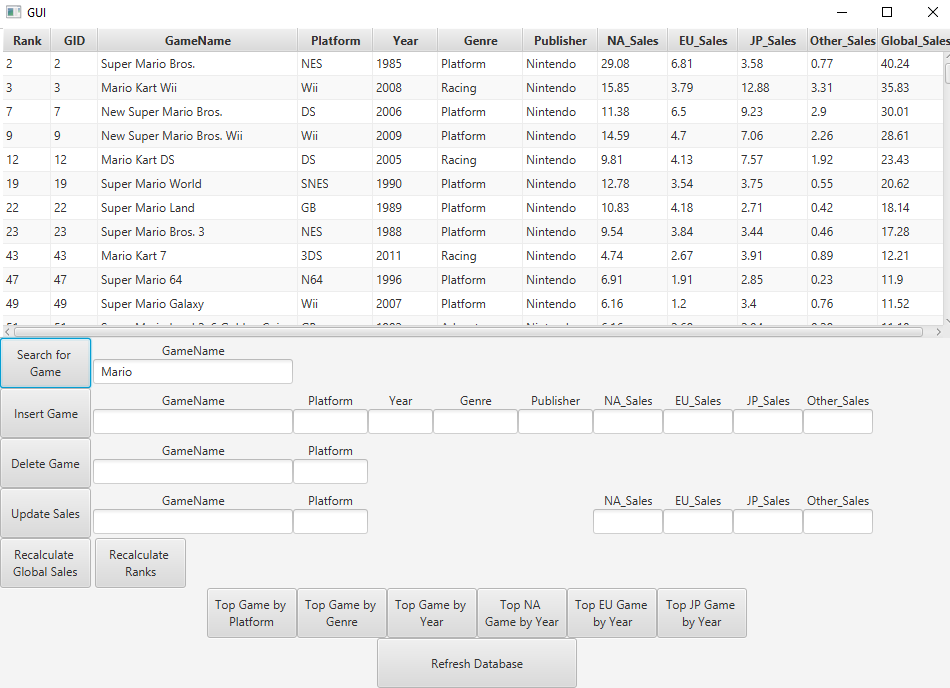


We tested this query by first sorting the game by the highest JP\_Sales which corresponds to the subquery. Then, we grouped the sorted games by each distinct year and ordered it in descending order of JP\_Sales. The image below shows the output of our SQL query. If our query runs correctly, an expected output will display tuples with games that have the highest JP\_Sales for each year. One way we tested this was to see if the returned year values were unique and then checking which game in each year actually had the highest JP\_Sales. Clearly, the year values are all unique, and each year had the corresponding highest JP\_Sales for that year, which matched the expected output.

Final evaluation of the application

* Objective
  + Make sure that games can be added, removed, searched for, global sales correctly. In each test case, we check if running each query increments regional sales, and updates rank consistently after changing sales data.
* Test case 1: Test that Global sales increases when the sales data in any region is incremented
  +   
    We decided to use ‘Wii Sports Resort’ to test if the application correctly updates changes made by user in the database. This is the initial data of the game before we change the value of sales in NA, EU, JP and other regions.
  +   
    We incremented all sales in all regions by 1.00. Then we check if all the sales are correctly updated according to the changes. We expected the global\_sales to be increased by 4, and we found the changes we made in the data are correctly updated in the database.
  +   
    Then we decremented all sales in all regions by 1.00. We expected the data of the game should be exactly the same as the data before any change is made. The table above indicates the changes of the sales are correctly updated because it proves the data after decrementing 1.0 in sales per region

GUI Examples: Search, Insert, Delete, and Update Sales

* + 

**III. Conclusion**

*What do you learn from this project (both interesting and uninteresting points)? Have you found any relevant database knowledge you have learned in this course helpful and have you encountered any database relevant issues that have been discussed in this course?*

**3.1 Lessons Learned and Knowledge Gained**

Back-end:We thought that the project gave us a broad/comprehensive overview of all the topics we had learned throughout the semester. We learned how to develop relational schema from a given dataset and learned how to normalize our data to eliminate data anomalies and duplicate data. As database implementers, we learned how to implement queries and stored procedures using MySQL. we learned how data is internally indexed/organized in the back-end by B+trees. Even though some of our functions did not work, we were able to get an idea on functionalities such as search, insert, and delete work on B+ trees

Front-end: At the front-end, we learned how to use the JDBC driver manager to connect our app with DB and interface between MySQL and the Java IDE. We also learned how to create a new cursor class for constructing objects which calls queries and stored procedures in MySQL. We also learned how to think from the user’s perspective, by thinking about what functionalities a typical user might need in the front-end and implementing it through the JavaFX GUI to enhance UI/UX. Lastly, we learned communication skills and how to work together as a team by modularization and incremental design.

**3.2 DB Relevant Issues**

We had a few challenges handling the dataset. Initially, we thought that the dataset was clean, but it was not entirely clean. There were some “gaps” in the data, and some of the data had anomalies, rounding errors, and duplicates. For example, the global sales attribute determines the rank for each game, but several ranks were missing, so we had to recalculate the game ranks. Another issue we encountered was with normalization. Next, we realized that global sales was not consistent with regional sales; the dataset had rounding errors or wrong global sales data for some games, so we knew that that needed to be recalculated based on the regional sales data. Lastly, we found that there was no way for the Games Table to be in either BCNF or 3NF without adding something to the table. GameName was a non-unique, games with the same name were released for multiple systems or in multiple years or even by multiple publishers. A candidate key for the Game table would have required a composite key consisting of the name, platform, year, and publisher, which is not in 3NF with the multiple functional dependencies derived from this. The only unique attribute was the game rank, but we knew that the game rank was going to change as the sales data updated.