DBMS 2 Midterm project
Games Store

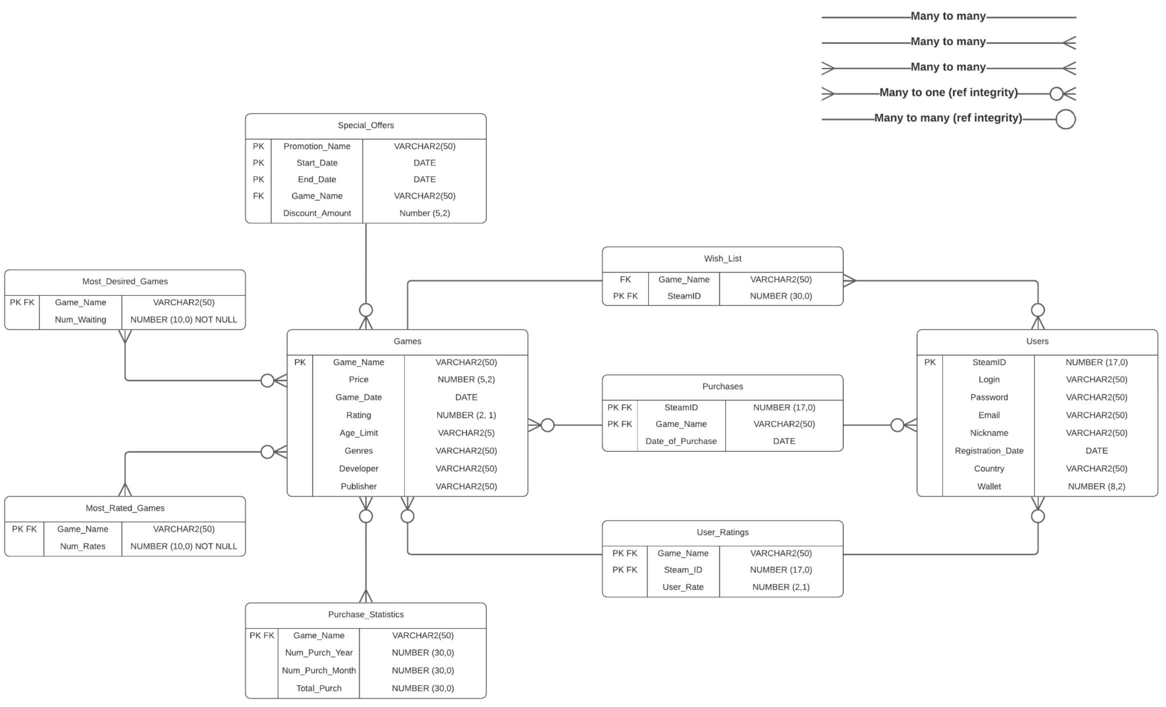
Made By:
Alikhan Zhakenov, Yerbatyr Yespanov,
Aidos Karakulov, Shyntas Bidashiyev.

Introduction:

The popularity of online games as a source of entertainment has caused the online gaming business to expand quickly in recent years. Online game shops now play a key role in the market by providing a variety of games and deals to draw in and keep clients. The topic this article wants to give is a thorough analysis of the online games shop database and its tables, focusing on the significant insights that may be obtained from doing so.

Link to Github https://github.com/Aidos-Karakulov/DBMS_2.git

1) ER Diagram and relations


Many to many
Many to many
Many to many
Many to one (ref integrity)
Many to many (ref integrity)
Special_Offers
PK Promotion_Name VARCHAR2(50)
PK Start_Date DATE
PK End_Date DATE
FK Game_Name VARCHAR2(50)
Discount_Amount Number (5,2)
Wish_List
FK Game_Name VARCHAR2(50)
PK FK SteamID NUMBER (30,0)
Most_Desired_Games
PK FK Game_Name VARCHAR2(50)
Num_Waiting NUMBER (10,0) NOT NULL
Games
PK Game_Name VARCHAR2(50)
Price NUMBER (5,2)
Game_Date DATE
Rating NUMBER (2, 1)
Age_Limit VARCHAR2(5)
Genres VARCHAR2(50)
Developer VARCHAR2(50)
Publisher VARCHAR2(50)
Purchases
PK FK SteamID NUMBER (17,0)
PK FK Game_Name VARCHAR2(50)
Date_of_Purchase DATE
Users
PK SteamID NUMBER (17,0)
Login VARCHAR2(50)
Password VARCHAR2(50)
Email VARCHAR2(50)
Nickname VARCHAR2(50)
Registration_Date DATE
Country VARCHAR2(50)
Wallet NUMBER (8,2)
Most_Rated_Games
PK FK Game_Name VARCHAR2(50)
Num_Rates NUMBER (10,0) NOT NULL
User_Ratings
PK FK Game_Name VARCHAR2(50)
PK FK Steam_ID NUMBER (17,0)
User_Rate NUMBER (2,1)
Purchase_Statistics
PK FK Game_Name VARCHAR2(50)
Num_Purch_Year NUMBER (30,0)
Num_Purch_Month NUMBER (30,0)
Total_Purch NUMBER (30,0)

In our main table Games we have one PK (Game_Name) and all of the attributes that describe information about the game and are dependent on it. This satisfies 3NF.

Why is the name of the game is key, after all the name is not always unique?

Game names are unique - after all, the repetition of the name entails copyright infringement. Because the games do not repeat the names of each other.

The little tables Most_Rated/Desired_Games are very simple, and as their names indicate, they determine the most desired and rated games.

They have only one attribute dependent on Game_Name – 3NF satisfied, each game may have only one number of rates and waiting people – atomicity also satisfied.

With Purchase_Statistics same situation – but with two extra attributes.

This table shows the number of purchases in the last month/year/all time.

Special_Offers have differences: we have huge PK, because the name is not guaranteed unique, and other offers may start or end in one time, but no offers can be at the same start-end time with the same name.

About attribute Discount_Amount – this is the percent of discount to the game.

Purchases and User_Ratings are the same: two-attribute key and one dependent attribute.

Why key is such?

In purchases one user may buy several games at once.

And with only ID we can’t find a certain purchase at a certain time.

In first this table had attribute price – but it turned out to be a mistake - a independent dependency appears:

SteamID, Game_Name --> Date_of_Purchase and Game_Name --> Price.

So we just delete the price – if it is so needed, we may make a query or view, that would include the price.

In ratings it is very simple: a certain user puts a rate on a certain game.

A certain user can rate a certain game.

Without the name of the game in the primary key, we simply will not be able to understand what rating is given to which game.

The Users table is like a games table: one PK and many attributes.

About relations: we made my own system, which stands in the corner of the diagram.

No, we didn't add anything new - we just labeled certain views that are similar to what they should look like.

This is also simple:

In tables Most_Desired_Games, Most_Rated_Games and Purchase_Statictics one game can appear only once.

In other tables connected with Games one game can appear many times: one game may be in some offers, wish lists, one game can be bought or rated by many users.

Almost all tables have a connection with referential integration to table Games.

Because if a game is present in the statistics/offer or rated, then it must be in the main table.

And this game should be represented in main table only once.

But only with Wish_List relation is “all”, because one game may appear in this table many times – one game may be in thousands of wish_lists.

Second big table – Users.

One user can buy or rate many games – but may have only one wish list.

But certain wish list/purchase/rate must belong only to one user.

2) Functional dependencies:

Games:

Game_Name --> Price, Game_Date, Rating, Age_Limit, Genres, Publisher, Developer

Special_Offers:

Promotion_Name, Start_Date, End_Date --> Game_Name, Discount_Amount

Purchase_Statistics:

Game_Name --> Num_Purch_Year, Num_Purch_Month, Total_Purch

Most_Rated_Games:

Game_Name --> Num_Rates

Most_Desired_Games:

Game_Name --> Num_Waiting

Users:

SteamID --> Login, Password, Email, Nickname, Registration_Date, Country, Wallet

Wish_List:

SteamID --> Game_Name

Purchases:

SteamID, Game_Name --> Date_of_Purchase

User_Ratings:

Steam_ID, Game_Name -> User_Rate

This is a boring theme – what we may tell about this, if all information was described in part about ERD?

but O-kay, it still shows well that all tables conform to normal forms.

3)Coding Part

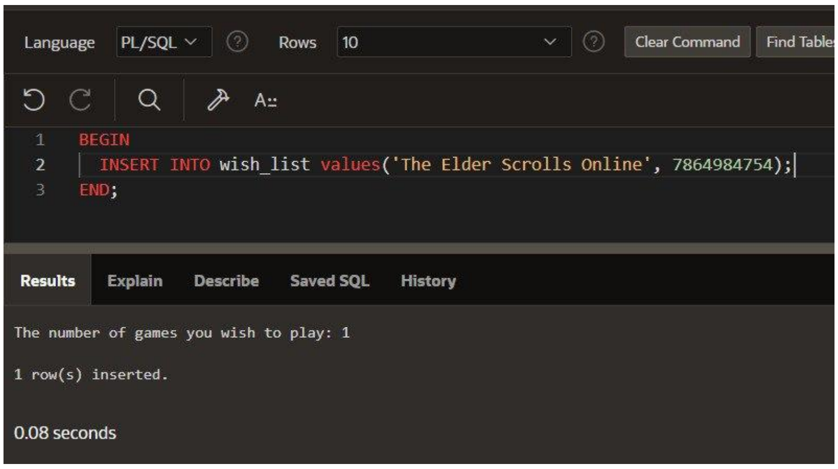
1. **Trigger Before Insert which will show the current number of rows in the table:**

```
create or replace TRIGGER num_of_waiting_games
BEFORE INSERT ON wish_list FOR EACH ROW
DECLARE
num_of_curr_rows NUMBER;
BEGIN
SELECT COUNT(*) INTO num_of_curr_rows FROM
wish_list where steamID = :NEW.steamID;
DBMS_OUTPUT.PUT_LINE('The number of games you wish
to play: ' || num_of_curr_rows);
END;
```

The triggers name “num_of_rows” and use syntax “CREATE OR REPLACE TRIGGER”
 “Before INSERT ON special_offers” Trigger mean should be work executed before a new row is inserted into table “special_offers”
 “Declare ” keyword is used to declare a variable “num_of_rows ” type of “Number”.
“Begin” the start Trigger
“Select Count(*) INTO num_of_rows From special_offers” statement queries the table “special_offers” count the number of rows in the table .Result is stored in the “num_of_rows” variable.
“DBMS_OUTPUT.PUT_LINE” statement prints a message to the console and shows the number of rows in the “special_offers” table before the insert operations.

"END" is already the end point of the trigger and this is the very creation or replacion

Result:



```
BEGIN
  INSERT INTO wish_list values('The Elder Scrolls Online', 7864984754);
END;
```

```
The number of games you wish to play: 1

1 row(s) inserted.

0.08 seconds
```

2. **Procedure which uses SQL%ROWCOUNT to determine the number of rows affected:**

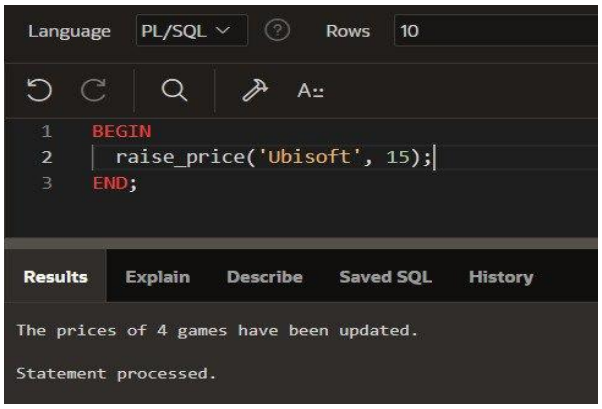
```
create or replace PROCEDURE raise_price(
publisher_name IN games.publisher%TYPE,
raise_procent IN NUMBER) AS
upd_affected NUMBER;
BEGIN
UPDATE Games
SET price = price + price * (raise_procent / 100)
WHERE publisher = publisher_name;
IF sql%notfound THEN
dbms_output.put_line('There is no such
publisher!');
ELSEIF sql%found THEN
upd_affected := SQL%ROWCOUNT;
dbms_output.put_line('The prices of ' ||
upd_affected || ' games have been updated.');
END IF;
END;
```

Update Games Set price = price +price*(raise_procent/100)
Where publisher = publisher_name;
The Update operator is needed to increase the price of all games from a given publisher, i.e. from "Publisher_name"

IF sql%notfound THEN dbms_output.put_line(‘There is no such publisher!’); Use the conditional IF statement in this row to check if there are any entries in the Games table If there is no error message is displayed.

ElseIF sql%found THEN upd_affected := SQLROWCOUNT; dbms_output.put_line(‘The prices of ‘ || upd_affected || ‘ games have been updated ‘); END IF;
If the UPDATE statement updated any data in the table, we store the number of updated records in a variable called "upd_affected".After that we show on the screen indicating that the prices for the specified number of games have been updated successfully

Result:



```
BEGIN
  raise_price('Ubisoft', 15);
END;
```

```
The prices of 4 games have been updated.

Statement processed.
```

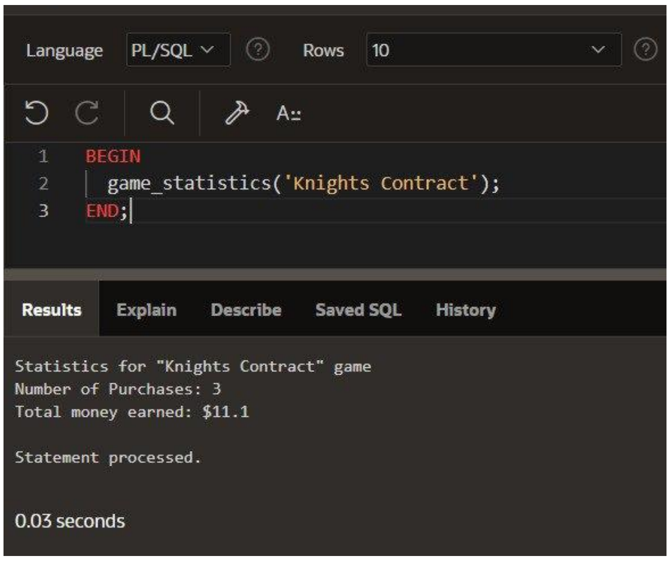
3. Procedure which does Group By information:

```
create or replace PROCEDURE game_statistics(game_name_stat IN
games.game_name%TYPE) AS
num_of_pur NUMBER;
price games.price%TYPE;
total_money NUMBER;
BEGIN
    SELECT g.price, count(p.steamID) INTO price, num_of_pur FROM Games g
    JOIN Purchases p ON g.Game_Name = p.Game_Name
    WHERE g.Game_Name = game_name_stat
    GROUP BY g.price, g.Game_Name;

    total_money := num_of_pur * price;
    DBMS_OUTPUT.PUT_LINE('Statistics for "' || game_name_stat || '" game');
    DBMS_OUTPUT.PUT_LINE('Number of Purchases: ' || num_of_pur);
    DBMS_OUTPUT.PUT_LINE('Total money earned: $' || total_money);
END;
```

The procedure shows how much the game has earned.
The name of the procedure is game_name_stat which is an attribute of the column called game_name of the table Games.
Inside the procedure Select is used to combine the tables Games and Purchases with the condition Where it selects the information of a certain game that you named passed as a parameter.
The SELECT result contains the price of the game and the number of purchases for that game, which is stored in a variable named price and num_of_pur
Before the end, the procedure calculates the total amount of money earned from selling the game by multiplying the number of games and stores it in a variable called total_money.
At the end the procedure outputs the result to the console via DBMS_OUTPUT.PUT_LINE

Result:



Language PL/SQL Rows 10

```
BEGIN
  game_statistics('Knights Contract');
END;
```

Results Explain Describe Saved SQL History

```
Statistics for "Knights Contract" game
Number of Purchases: 3
Total money earned: $11.1

Statement processed.
```

0.03 seconds

4. **User-defined exception which disallows to enter Promotion_Name to be less than 5 characters:**

```
create or replace procedure add_special_offer(
prom_name IN special_offers.Promotion_name%TYPE,
s_date IN special_offers.START_DATE%TYPE,
e_date IN special_offers.END_DATE%TYPE,
game_name IN special_offers.GAME_NAME%TYPE,
disc_amount IN
special_offers.DISCOUNT_AMOUNT%TYPE) AS
prom_name_exc EXCEPTION;
begin
IF length(prom_name) < 5 THEN
RAISE prom_name_exc;
ELSE
INSERT INTO Special_offers VALUES(prom_name,
s_date, e_date, game_name, disc_amount);
dbms_output.put_line('NEW Special Offer "' ||
prom_name || '" added! ');
dbms_output.put_line('The ' || disc_amount || '%
discount is valid for the "' || game_name || '"
game between ' || s_date || ' and ' || e_date);
END IF;
EXCEPTION
WHEN prom_name_exc THEN
dbms_output.put_line('The Promotion Name is too
short! It must be greater than 5. ');
WHEN others THEN
dbms_output.put_line('Error!');
end;
```

This exception describes how to add a new special offer to the Special_offers table.

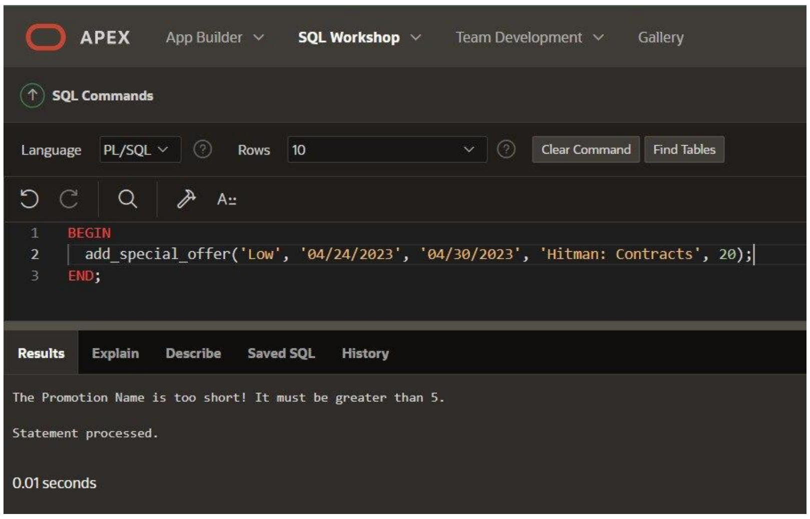
The procedure takes input parameters: promotion name (prom_name), start and end dates (s_date and e_date), game name (game_name), and discount amount (disc_amount).

If the offer name is shorter than 5 characters, then a prom_name_exc exception is generated.

Otherwise, an entry is added to the table and a message is displayed about adding a new special offer.

If any other error occurs, an error message is displayed.

Result:

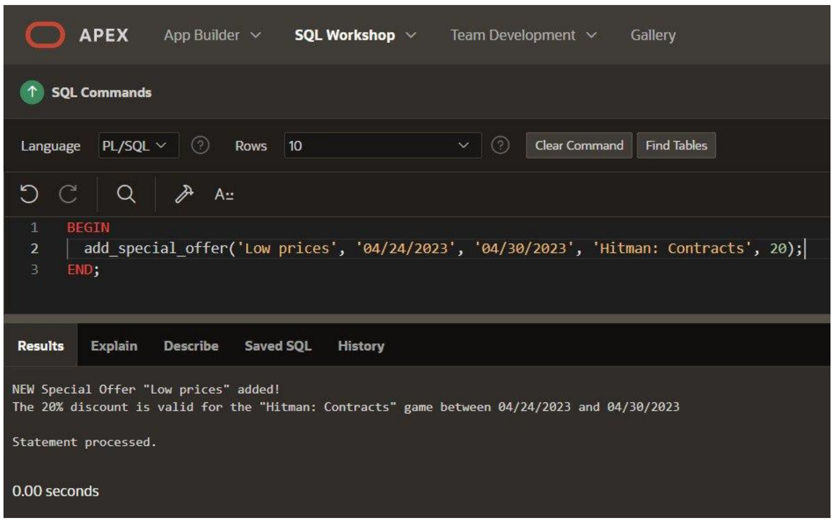


```
BEGIN
  add_special_offer('Low', '04/24/2023', '04/30/2023', 'Hitman: Contracts', 20);
END;
```

```
The Promotion Name is too short! It must be greater than 5.

Statement processed.
```

0.01 seconds



```
BEGIN
  add_special_offer('Low prices', '04/24/2023', '04/30/2023', 'Hitman: Contracts', 20);
END;
```

```
NEW Special Offer "Low prices" added!
The 20% discount is valid for the "Hitman: Contracts" game between 04/24/2023 and 04/30/2023

Statement processed.

0.00 seconds
```

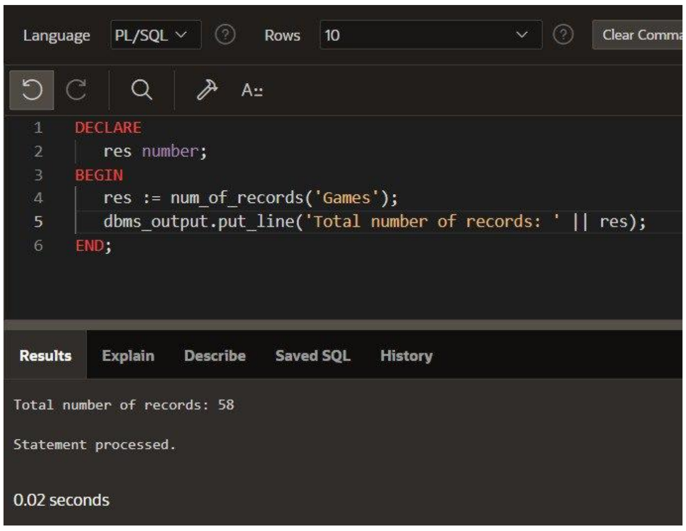
5. **Function which counts the number of records:**

```
CREATE OR REPLACE FUNCTION
num_of_records(table_name IN VARCHAR2)
RETURN NUMBER IS
   total_records NUMBER := 0;
BEGIN
   EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' ||
table_name INTO total_records;
   RETURN total_records;
END;
```

This function takes a single input parameter - table_name, which is a Varchar(2) data type. The purpose of this function is to return the total number of records present in a specified database table. The function uses dynamic SQL to execute a Select count(*) statement on the specified table. The Execute immediate statement allows the function to dynamically generate and execute SQL code at runtime. The result of the Select statement is then assigned to the total_records variable using the Into clause.

Finally, the function returns the value of the total_records variable.

Result:

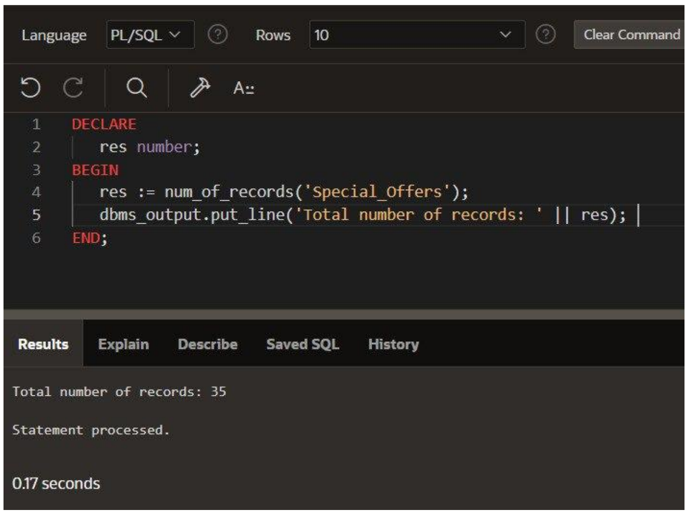


```
DECLARE
    res number;
BEGIN
    res := num_of_records('Games');
    dbms_output.put_line('Total number of records: ' || res);
END;
```

```
Total number of records: 58

Statement processed.

0.02 seconds
```



```
DECLARE
    res number;
BEGIN
    res := num_of_records('Special_Offers');
    dbms_output.put_line('Total number of records: ' || res);
END;
```

```
Total number of records: 35

Statement processed.

0.17 seconds
```