Console Games Database

ISM6218.901S22 Advanced Database Management

Group 2

Shailendra Singh Sherry Ingram Abhishanth Penta Kshitiz Kharel

Executive Summary

Introduction

A console game is a type of interactive multimedia software that uses a video game console to provide an interactive multimedia experience via a television of other display device. The game console generally consists of a handheld control device (although some use cameras to monitor user movements) and a computer that runs the game's software. The global console games market is about $26.8b in 2018 with key vendors in the market like Microsoft, Sony, and Nintendo.

Our group decided to create a database that serves as the backend for any application or services, which can leverage it to provide product recommendations (videogames) based on key attributes such as rating, genre, console compatibility, and so on. Our goal is to set up a working database that can also be used for data-science tasks such as predicting success of a new game based on analyzing similar games' information stored in our database.

Project Topic Areas and Weights

Topic Area	Description	Points
Database Design	This part includes a logical database design (for the relational model), using normalization to control redundancy and integrity constraints for data quality.	Weight: **30** Range: 20 - 30
Query Writing	This part demonstrates practical examples using SQL queries on the database for real-world scenarios or data request within an application/service.	Weight: **25** Range: 20 - 30
Performance Tuning	This section applies some performance tuning concepts such as indexing, etc., that improves the performance of the query executions.	Weight: **20** Range: 20 - 30
Data Visualization and ML Algorithms	This section attempts to demonstrate to use visualizations and data mining algorithms to analyze the data.	Weight: **25** Range: 10 - 40

Database Design

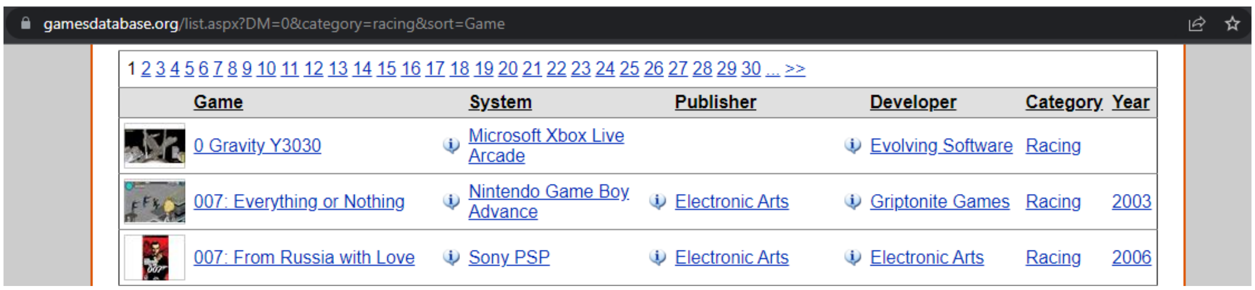
Section 1.1: Data Generation and Loading

Data Collection

The data was collected from two sources.

Games Data

The games data primarily comes from *Online Game System Repository* (https://www.gamesdatabase.org/). The following screenshot shows one such page:


gamesdatabase.org/list.aspx?DM=0&category=racing&sort=Game

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 ... >>

Game	System	Publisher	Developer	Category	Year
0 Gravity Y3030	Microsoft Xbox Live Arcade		Evolving Software	Racing	
007: Everything or Nothing	Nintendo Game Boy Advance	Electronic Arts	Griptonite Games	Racing	2003
007: From Russia with Love	Sony PSP	Electronic Arts	Electronic Arts	Racing	2006

Since the webpage has a defined structure and predictable formatting, we used python to web-scrap this data. We took only those categories that has more than 1000 games.

Users Data

For Users data, we used *Kaggle* dataset (https://www.kaggle.com/datasets/nathanlauga/nba-games). This dataset contains games and players information of NBA.

Data Cleaning

We used *Python* to clean the data and convert them into required CSV files. The following script shows the csv creation for games data.

```
import os


class Record:
   def __init__(self, line_txt):
      items = line_txt.split('\t')
      self.game = items[1]
      self.system = items[3]
      self.publisher = items[5]
      self.developer = items[7]
      self.category = items[8]
      self.year = items[9]
```

```python
      if not (self.game and self.system and self.publisher and self.developer and
self.category):
         raise ValueError

   def __str__(self):
      return "{},{},{},{},{},{}".format(self.game, self.system, self.publisher,
self.developer, self.category,
                                        self.year)


def get_records(file_name):
   records = []
   with open(file_name) as f:
      for line in f.readlines():
         if len(line.split('\t')) == 10:
            try:
               rec = Record(line)
               records.append(rec)
            except ValueError:
               pass
   return records


output_dir = r"D:\Practice\PythonWorkspace\DBMS Games Data Archive"
output_file = os.path.join(r'D:\Practice\PythonWorkspace\Notebooks',
"games_data.csv")
with open(output_file, 'w') as out:
   header = Record("  Game      System    Publisher     Developer  Category
Year\n")
   out.write(str(header))
   for dirName, subdirList, fileList in os.walk(output_dir):
      for fl in fileList:
         res = get_records(os.path.join(dirName, fl))
         res = [str(x) for x in res]
         out.writelines(res)

   print("Done")
```

This script read all the raw files and collated them into one final CSV file. The final csv file looks something like shown below:

```python
games_data_cleaned = pd.read_csv(r"D:\Practice\PythonWorkspace\Notebooks\games_data_cleaned.csv")
print("(Row,Col) = {}".format(games_data_cleaned.shape))
games_data_cleaned.head()
```

[20] ✓ 1.1s Python

```
(Row,Col) = (24931, 6)
```

	Game	System	Publisher	Developer	Category	Year
0	[PROTOTYPE]	Microsoft Xbox 360	Activision	Radical	Action	NaN
1	007: Agent Under Fire	Microsoft Xbox	Electronic Arts	Electronic Arts	Action	NaN
2	007: Agent Under Fire	Nintendo GameCube	Electronic Arts	Electronic Arts	Action	NaN
3	007: Everything or Nothing	Nintendo GameCube	Electronic Arts	Electronic Arts	Action	NaN
4	007: Everything or Nothing	Microsoft Xbox	Electronic Arts	Electronic Arts	Action	NaN

The Users data was cleaned in similar style. The following code shows the cleaning process for all the tables in the ER diagram (shown later in the document).

Load Libraries

```
import pandas as pd
import numpy as np
import random
```

Read NBA Kaggle Data Files

```
players = pd.read_csv('games_data/players.csv', on_bad_lines='skip')
games = pd.read_csv('games_data/games.csv', on_bad_lines='skip')
games_details = pd.read_csv('games_data/games_details.csv', on_bad_lines='skip')
ranking = pd.read_csv('games_data/ranking.csv', on_bad_lines='skip')
teams = pd.read_csv('games_data/teams.csv', on_bad_lines='skip')
console = pd.read_csv('games_data/games_data_cleaned.csv')
```

Create User Table

```
# Create Gamer Table
def get_tag(fname, lname=None):
   try:
      if lname:
         return fname[:3]+lname[:3]
   except:
      return fname[:3] + fname[:-3:-1]

   return fname[:3]+fname[:-3:-1]
gamer_table=games_details[['PLAYER_ID','PLAYER_NAME','NICKNAME','TEAM_CITY','PF','FG
_PCT']].drop_duplicates().sort_values(by='PLAYER_ID')
gamer_table=gamer_table.groupby('PLAYER_ID').first()
gamer_table=gamer_table.reset_index()
gamer_table['FirstName']=gamer_table['PLAYER_NAME'].str.split(' ')
gamer_table['LastName']=gamer_table['FirstName'].str[1]
gamer_table['FirstName']=gamer_table['FirstName'].str[0]

gamer_table['NICKNAME'] = gamer_table.apply(lambda x: get_tag(x.FirstName,
x.LastName), axis=1)
gamer_table['GamerScore']=round(gamer_table['FG_PCT']*100)
gamer_table['Age'] = np.random.randint(13, 45, gamer_table.shape[0])
gamer_table[gamer_table['GamerScore'].isnull()] =
gamer_table[gamer_table['GamerScore'].isnull()
                                                        ].apply(lambda x:
random.randint(11,99))
gamer_table['GamerScore']=gamer_table['GamerScore'].astype('int32')
gamer_table.rename(columns={'PLAYER_ID':'UID','NICKNAME':'GamerTag','TEAM_CITY':'Cit
y'}, inplace=True)
gamer_table =
gamer_table[['UID','FirstName','LastName','GamerTag','City','Age','GamerScore']].dro
p_duplicates().sort_values(by='UID')
```

Use standard data to get state from city names

```
cities = pd.read_csv('games_data/uscities.csv', on_bad_lines='skip')
cities = cities[['city','state_name']]
cities['City'] = cities['city']
cities = cities[['City','state_name']]
gamer_table=pd.merge(
    gamer_table,
    cities,
    on="City"
)
gamer_table=gamer_table.groupby('UID').first()
gamer_table=gamer_table.reset_index()
gamer_table['GamerTag']=gamer_table['GamerTag'].apply(str.lower)
gamer_table.rename(columns={'state_name':'State'}, inplace=True)
```

Further clean games data to match with project plan ER diagram

```
# Remove useless system values.
console=console[console['System']!='4']

# Fix System Column
console['System2']=console['System'].str.split(' ')
console['System2']=console['System2'].str[0]
console['System']=console['System2']
console=console.drop('System2',axis=1)
console.head()

# Fix Year Column
console['Year']=console['Year'].apply(pd.to_numeric)

# Create Categorical Column Codes
#------------------------------------
# Set all desired cols as category
console['CategoryCode'] = console['Category'].astype('category')
console['SystemCode'] = console['System'].astype('category')
console['PublisherCode'] = console['Publisher'].astype('category')
# Select all category columns and apply cat.codes attribute
cat_cols = console.select_dtypes(['category']).columns
console[cat_cols]=console[cat_cols].apply(lambda x : x.cat.codes+1)
```

Create Games Table

```
# Create Games Table
games_temp = games[['GAME_ID','REB_home','FT_PCT_away']].drop_duplicates()
games_temp.columns = ('GameId','Price','Rating')
console_temp = console[['Game','CategoryCode']].drop_duplicates()
console_temp = console_temp.reset_index()
games_temp = games_temp[:console_temp.shape[0]]
games_temp=games_temp.reset_index()
games_temp['index']=np.arange(1,games_temp.shape[0]+1,1)
console_temp['index']=np.arange(1,games_temp.shape[0]+1,1)
games_table = pd.merge(games_temp, console_temp,on="index")
games_table['Rating']=games_table['Rating']*5
games_table['Rating']=games_table['Rating'].round(2)
games_table = games_table[['GameId','Game','CategoryCode','Price','Rating']]
games_table.rename(columns={'Game':'Name','CategoryCode':'GenreId'}, inplace=True)
games_table['AgeRating']=np.random.randint(1, 4, games_table.shape[0])
```

Create Mapping table to preserve 3rd Normal Form

```
# Create GameConsoleAvailability
game_console_table = pd.merge(console[['Game','SystemCode']],
                              games_table[['Name','GameId']],
                              left_on="Game",
                              right_on="Name")
game_console_table = game_console_table[['GameId','SystemCode']].drop_duplicates()
game_console_table.columns = ('GameId','ConsoleId')
```

Create Dimension Table - Age Ratings

```
# Create Ratings Table
ratings_data = [[1,'Rated G','General audiences – All ages admitted.'],
                [2,'Rated PG','Parental guidance suggested – Some material may not
be suitable for pre-teenagers.'],
                [3,'Rated R','Restricted – Under 17 requires accompanying parent or
adult guardian.'],
                [4,'Rated X','No one under 17 admitted.']]

ratings_table = pd.DataFrame(ratings_data, columns = ['AgeRatingId',
'AgeRatingName','AgeRatingDescription'])
```

Create Console Table

```
# Create Console Table
console_table =
console[['System','SystemCode']].drop_duplicates().sort_values(by='SystemCode')
console_table = console_table.reset_index(drop=True)
console_table.rename(columns={'SystemCode':'ConsoleId', 'System':'ConsoleName'},
inplace=True)
console_table = console_table[['ConsoleId','ConsoleName']]
```

Create Genre Table

```
# Create Genre Table
genre_table =
console[['Category','CategoryCode']].drop_duplicates().sort_values(by='CategoryCode'
)
genre_table = genre_table.reset_index(drop=True)
genre_table.rename(columns={'Category':'Genre','CategoryCode':'GenreId'},
inplace=True)
genre_table = genre_table[['GenreId','Genre']]
```

Create Subscription Table

```
# Create Subscription Table
games_details_temp = games_details[['GAME_ID','PLAYER_ID']]
subs_table =
pd.merge(games_details_temp,gamer_table[['UID','GamerTag']],right_on="UID",left_on="
PLAYER_ID")
subs_table = subs_table[['UID','GamerTag','GAME_ID']]
subs_table =
pd.merge(subs_table,games_table[['GameId']],right_on="GameId",left_on="GAME_ID")
```

```
subs_table = subs_table[['UID','GamerTag','GameId']]
subs_table['SubscriptionId'] = np.random.randint(1, 3, subs_table.shape[0])
```

Create Mapping table to preserve 3rd Normal Form

```
# Create Game Subscription Type Table
subscription_data = [[1,'Free'],[2,'Freemium'],[3,'Paid']]
game_sub_type_table = pd.DataFrame(subscription_data, columns = ['SubscriptionId',
'SubscriptionName'])
```

Export respective tables to their CSV counterparts

```
from os import path
output_path =
r"C:\Users\Shail\Documents\PythonWorkspace\Notebooks\games_data\output"

# 1. Users Table
gamer_table.to_csv(path.join(output_path,"users.csv"), sep=',', encoding='utf-
8',index=False)
# 2. Games Table
games_table.to_csv(path.join(output_path,"games.csv"), sep=',', encoding='utf-
8',index=False)
# 3. Genre Table
genre_table.to_csv(path.join(output_path,"genre.csv"), sep=',', encoding='utf-
8',index=False)
# 4. Game Console Table
game_console_table.to_csv(path.join(output_path,"game_console.csv"), sep=',',
encoding='utf-8',index=False)
# 5. Age Ratings Table
ratings_table.to_csv(path.join(output_path,"age_ratings.csv"), sep=',',
encoding='utf-8',index=False)
# 6. Console Table
console_table.to_csv(path.join(output_path,"consoles.csv"), sep=',', encoding='utf-
8',index=False)
# 7. Game Subscription Type Table
game_sub_type_table.to_csv(path.join(output_path,"subscription_types.csv"), sep=',',
encoding='utf-8',index=False)
# 8. Subscription Table
subs_table.to_csv(path.join(output_path,"subscriptions.csv"), sep=',',
encoding='utf-8',index=False)
```

The last segment created following tables. The first line shows Row and Column count and the second shows table itself.

Age Rating

```
(Row,Col) = (4, 3)
```

	AgeRatingId	AgeRatingName	AgeRatingDescription
0	1	Rated G	General audiences – All ages admitted.
1	2	Rated PG	Parental guidance suggested – Some material ma...
2	3	Rated R	Restricted – Under 17 requires accompanying pa...
3	4	Rated X	No one under 17 admitted.

Consoles

```
(Row,Col) = (35, 2)
```

	ConsoleId	ConsoleName
0	1	Acorn
1	2	Amstrad
2	3	Apple
3	4	Arcade
4	5	Atari

Game Consoles

```
(Row,Col) = (23354, 2)
```

	GameId	ConsoleId
0	22101005	20
1	22101006	20
2	22101006	22
3	22101007	22
4	41000172	22

Games

```
(Row,Col) = (24931, 6)
```

	Game	System	Publisher	Developer	Category	Year
0	[PROTOTYPE]	Microsoft Xbox 360	Activision	Radical	Action	NaN
1	007: Agent Under Fire	Microsoft Xbox	Electronic Arts	Electronic Arts	Action	NaN
2	007: Agent Under Fire	Nintendo GameCube	Electronic Arts	Electronic Arts	Action	NaN
3	007: Everything or Nothing	Nintendo GameCube	Electronic Arts	Electronic Arts	Action	NaN
4	007: Everything or Nothing	Microsoft Xbox	Electronic Arts	Electronic Arts	Action	NaN

Genre

```
··· (Row,Col) = (20, 2)
```

	GenreId	Genre
0	1	Action
1	2	Adventure
2	3	Application
3	4	Arcade
4	5	Ball & Paddle

Subscription Types

```
··· (Row,Col) = (3, 2)
```

	SubscriptionId	SubscriptionName
0	1	Free
1	2	Freemium
2	3	Paid

Subscriptions

```
··· (Row,Col) = (317950, 4)
```

	UID	GamerTag	GameId	SubscriptionId
0	1627736	malbea	22101005	2
1	1626156	d'arus	22101005	2
2	1627752	taupri	22101005	1
3	1630233	natkni	22101005	1
4	1629130	dunrob	22101005	2

Users

```
(Row,Col) = (2151, 8)
```

	UID	FirstName	LastName	GamerTag	City	Age	GamerScore	State
0	15	Eric	Piatkowski	eripia	Chicago	44	67	Illinois
1	42	Monty	Williams	monwil	Philadelphia	39	25	Pennsylvania
2	43	Chris	Whitney	chrwhi	Washington	19	60	District of Columbia
3	56	Gary	Payton	garpay	Miami	24	40	Florida
4	57	Doug	Christie	douchr	Dallas	23	67	Texas

Data Loading

The respective CSV files were loaded using Oracle SQL Developers Import Wizard.

We first created all the tables, following SQL Shows CREATE Statements:

```sql
CREATE TABLE DB1SMOKE.AGE_RATINGS
(
AgeRatingId INT,
AgeRatingName VARCHAR2(10),
AgeRatingDescription VARCHAR2(100)
);

CREATE TABLE DB1SMOKE.CONSOLES
(
ConsoleId INT,
ConsoleName VARCHAR2(20)
);

CREATE TABLE DB1SMOKE.GAME_CONSOLES
(
GameId INT,
ConsoleId INT
);

CREATE TABLE DB1SMOKE.GAMES
(
GameId INT,
GameName VARCHAR2(100),
GenreId INT,
Price FLOAT,
Rating FLOAT,
AgeRating FLOAT
);

CREATE TABLE DB1SMOKE.GENRE
(
GenreId INT,
Genre VARCHAR2(20)
);

CREATE TABLE DB1SMOKE.SUBSCRIPTION_TYPES
```

```sql
(
SubscriptionId INT,
SubscriptionName VARCHAR2(10)
);

CREATE TABLE DB1SMOKE.SUBSCRIPTIONS
(
UserId INT,
GamerTag VARCHAR2(10),
GameId INT,
SubscriptionId INT
);

CREATE TABLE DB1SMOKE.USERS
(
UserId INT,
FirstName VARCHAR2(20),
LastName VARCHAR2(20),
GamerTag VARCHAR2(10),
City VARCHAR2(20),
Age INT,
GamerScore INT,
State VARCHAR2(30)
);
```

Section 1.2 Data Integrity

The integrity constraints necessary to help ensure data quality should be included in the design section. We implemented following constraints to ensure data integrity:

Primary Key Constraints

The following table are dimension tables in our database design. So, we need to create a Primary Keys on columns that are to be used as a Foreign Key in other tables.

```sql
ALTER TABLE AGE_RATINGS ADD CONSTRAINT PK_AGE_RATING_ID PRIMARY KEY
(AGERATINGID);

ALTER TABLE GENRE ADD CONSTRAINT PK_GENRE_ID PRIMARY KEY (GENREID);

ALTER TABLE SUBSCRIPTION_TYPES ADD CONSTRAINT PK_SUBSCRIPTION_ID PRIMARY KEY
(SUBSCRIPTIONID);

ALTER TABLE CONSOLES ADD CONSTRAINT PK_CONSOLE_ID PRIMARY KEY (CONSOLEID);
```

The following columns are unique identifiers in their respectable tables. This ensures that there are no duplicate records.

```
ALTER TABLE GAMES ADD CONSTRAINT PK_GAME_ID PRIMARY KEY (GAMEID);

ALTER TABLE USERS ADD CONSTRAINT PK_USER_ID PRIMARY KEY (USERID);
```

Foreign Key Constraints

We added foreign keys in our tables to ensure referential integrity across our database design. This constraint helps us to define the relationship between the tables.

For example, a foreign key reference in GAMES table to the GENRE table shows that the genre information can be acquired from the target table. This also helps with saving storage space as large string values are avoided being repeated and are replaced by their much shorter id counterparts.

```
ADD FOREIGN KEY constraints.
ALTER TABLE GAMES
ADD CONSTRAINT FK_GENRE_ID
FOREIGN KEY (GENREID) REFERENCES GENRE(GENREID);

ALTER TABLE SUBSCRIPTIONS
ADD CONSTRAINT FK_SUBSCRIPTION_ID
FOREIGN KEY (SUBSCRIPTIONID) REFERENCES SUBSCRIPTION_TYPES(SUBSCRIPTIONID);

ALTER TABLE GAME_CONSOLES
ADD CONSTRAINT FK_CONSOLE_GAME_ID
FOREIGN KEY (GAMEID) REFERENCES GAMES(GAMEID);

ALTER TABLE GAME_CONSOLES
ADD CONSTRAINT FK_CONSOLE_ID
FOREIGN KEY (CONSOLEID) REFERENCES CONSOLES(CONSOLEID);

ALTER TABLE GAMES
ADD CONSTRAINT FK_AGE_RATING
FOREIGN KEY (AGERATING) REFERENCES AGE_RATINGS(AGERATINGID);
```

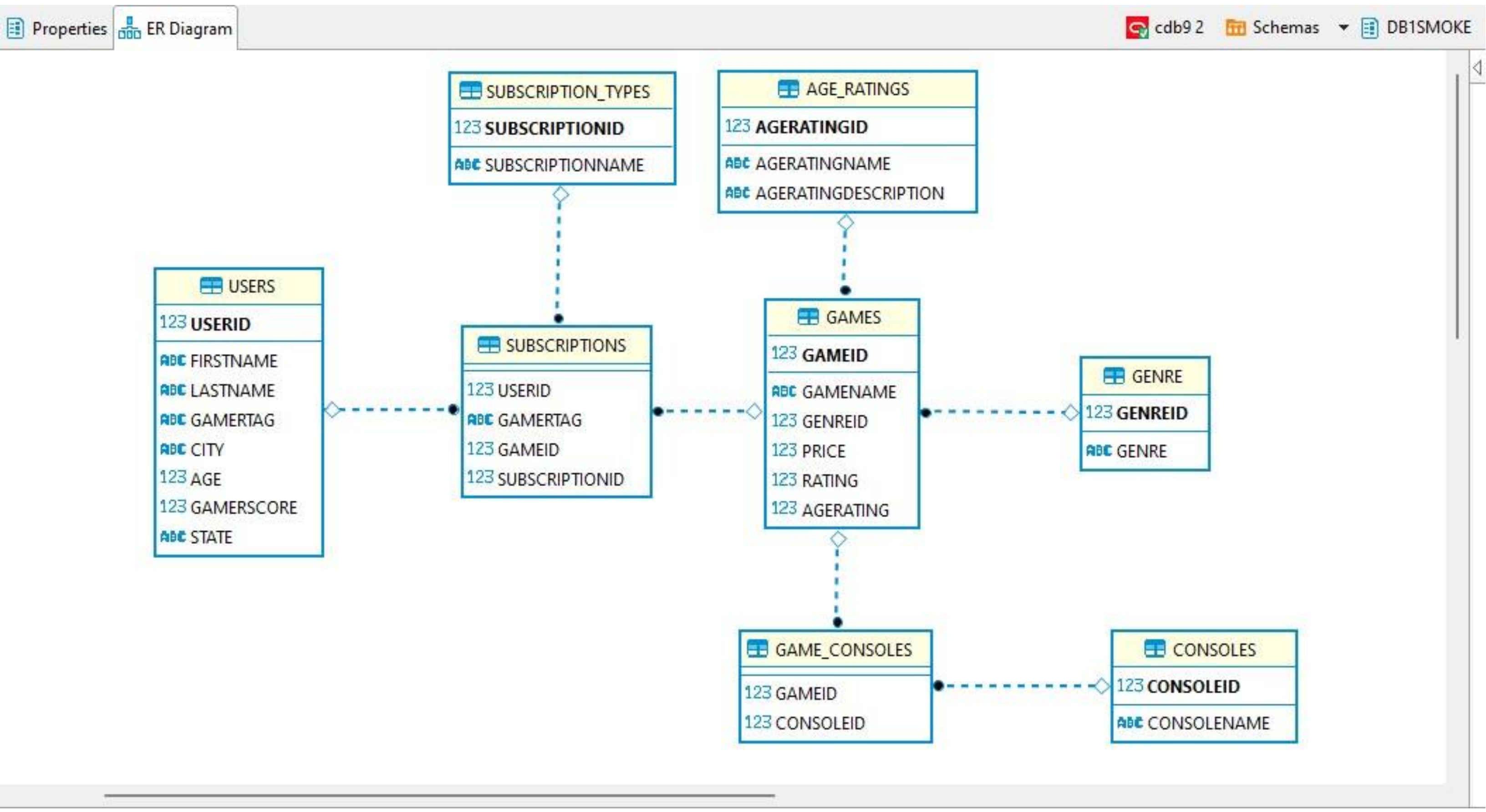
The foreign key also has one more advantage. It ensures the primary key constraint in the target table, and when used with index, it can significantly improve the performance of any operation that's pulling data from very large tables linked by indexed foreign keys. The subscription table has more than 300K records, and it would be useful to have foreign keys to ensure referential integrity as well as data integrity.

```
ALTER TABLE SUBSCRIPTIONS
ADD CONSTRAINT FK_GAME_ID
FOREIGN KEY (GAMEID) REFERENCES GAMES(GAMEID);

ALTER TABLE SUBSCRIPTIONS
ADD CONSTRAINT FK_USER_ID
FOREIGN KEY (USERID) REFERENCES USERS(USERID);
```

Section 1.3 E-R Diagram

Our final design looks as shown in below E-R Diagram. Due to the foreign key constraints, we can identify relationships among the tables. The primary keys are shown in bold and the foreign keys can be identified by the diamond symbol in the connecting lines.



Query Writing

This section demonstrates the practical examples where and how the data can be used. We provide several query examples that highlight the types of questions that can be answered using our database.

We present this in question-query format to demonstrate the practical usage.

```
Q-01. Display Minimum, Maximum, and Average Price per Game Genre. This can
be used, for example, to analyze financial aspects by genre and investing
appropriately.

SELECT
      g2.GENRE,
      MIN(g.PRICE) AS MIN_PRICE,
      MAX(g.PRICE) AS MAX_PRICE,
      AVG(g.PRICE) AS AVG_PRICE
FROM
      DB1SMOKE.GAMES g
INNER JOIN DB1SMOKE.GENRE g2
ON
      G.GENREID = G2.GENREID
GROUP BY
      g2.GENRE
```

	GENRE	MIN_PRICE	MAX_PRICE	AVG_PRICE
1	Family	26	62	42.7108433735
2	Racing	21	66	41.7217465753
3	Ball & Paddle	35	56	45.75
4	Breakout	30	55	41.7358490566
5	Educational	24	59	43.54375

```
Q-02. Find which cities would be good for hosting gaming competitions.
   1. Display the city's name as well as how many games are played by the
      users are in the city.
   2. Show cities with the most counts first.

SELECT
      u.CITY,
      COUNT(s.GAMEID) AS GAME_COUNTS
FROM
      DB1SMOKE.USERS u
INNER JOIN DB1SMOKE.SUBSCRIPTIONS s
ON
      u.USERID = s.USERID
GROUP BY
      u.CITY
```

```sql
ORDER BY COUNT(s.GAMEID) DESC
```

	CITY	GAME_COUNTS
1	Los Angeles	22,148
2	Dallas	17,650
3	San Antonio	17,080
4	Indiana	16,113
5	Houston	14,616

Q-03. Display all Games that contains “Need for Speed” somewhere in the name. Show the Games in decreasing order of their price.

```sql
SELECT
	g.GAMEID ,
	g.GAMENAME ,
	g.PRICE
FROM
	DB1SMOKE.GAMES g
WHERE
	g.GAMENAME LIKE '%Need for Speed%'
ORDER BY
	g.PRICE DESC
```

	GAMEID	GAMENAME	PRICE
1	22,000,049	Need for Speed Hot Pursuit	54
2	21,000,576	Need for Speed: Most Wanted	53
3	21,000,581	Need for Speed: V-Rally 2	52
4	21,000,596	Need for Speed III: Hot Pursuit	50
5	21,000,580	Need for Speed: Shift	49

Q-04. Label the cost category of a game based on its price. For each game display the game's name, price, and a textual label describing the cost category of the game.
The label should be

- "Very High" for a price more than 60
- "High" for a price range of 45 to 60
- "Average" for a price of 30 to 45, and
- “Low” for a price to be less than 30.

```sql
SELECT
	g.GAMENAME,
	g.PRICE,
	CASE
		WHEN g.PRICE >= 60 THEN 'Very High'
		WHEN g.PRICE >= 45 AND g.PRICE < 60 THEN 'High'
		WHEN g.PRICE >= 30 AND g.PRICE <45 THEN 'Average'
		ELSE 'Low'
```

```
	END AS "Cost Category"
FROM
	DB1SMOKE.GAMES g;
```

	GAMENAME	PRICE	Cost Category
1	Army Men: Air Attack 2	50	High
2	Army Men: Sarge's War	52	High
3	As Aventuras da TV Colosso	45	High
4	Asmik-kun Land	38	Average
5	Assassin	55	High
6	Assassin Special Edition	49	High
7	Assassin's Creed	54	High
8	Assassin's Creed Brotherhood	43	Average
9	Assassin's Creed II	50	High
10	Assassin's Creed Revelations	40	Average

Q-05. Find the top 10 most popular consoles. A console is popular when the number of games available exceed 100. Show the results with the most user count per console first.

```
WITH GameConsole AS
(
SELECT
	c.CONSOLEID,
	c.CONSOLENAME ,
	COUNT(gc.GAMEID) AS GAME_COUNT
FROM
	DB1SMOKE.CONSOLES c
INNER JOIN DB1SMOKE.GAME_CONSOLES gc
ON
	c.CONSOLEID = gc.CONSOLEID
GROUP BY
	c.CONSOLEID,
	c.CONSOLENAME
HAVING
	COUNT(gc.GAMEID) > 100
	)
SELECT
	c.CONSOLENAME,
	COUNT(u.USERID) AS "USAGE COUNT"
FROM
	GameConsole c
INNER JOIN DB1SMOKE.GAME_CONSOLES gc
ON
	c.CONSOLEID = gc.CONSOLEID
INNER JOIN DB1SMOKE.SUBSCRIPTIONS s
ON
	s.GAMEID = gc.GAMEID
```

```sql
INNER JOIN DB1SMOKE.USERS u
ON
      u.USERID = s.USERID
GROUP BY
      c.CONSOLENAME
ORDER BY
      COUNT(u.USERID) DESC;
```

	CONSOLENAME	USAGE COUNT
1	Arcade	69,019
2	Commodore	65,939
3	Nintendo	64,287
4	Atari	46,297
5	Microsoft	44,660
6	Sega	41,162
7	Sony	36,830
8	Sinclair	30,699
9	Amstrad	22,841
10	Valve	16,239

```sql
Q-06. Find the top 3 most experienced gamers. A gamer is experienced based
on the number of games they have played. Also show which city do they belong
to.

SELECT
      u.FIRSTNAME ,
      u.LASTNAME ,
      u.CITY,
      count(s.GAMEID) AS "Games Played"
FROM
      DB1SMOKE.USERS u
INNER JOIN DB1SMOKE.SUBSCRIPTIONS s
ON
      u.USERID = s.USERID
GROUP BY
      u.FIRSTNAME ,
      u.LASTNAME ,
      u.CITY
ORDER BY
      count(s.GAMEID) DESC
FETCH NEXT 3 ROWS ONLY
```

	FIRSTNAME	LASTNAME	CITY	Games Played
1	LeBron	James	Miami	1,045
2	Dwight	Howard	Los Angeles	1,003
3	Carmelo	Anthony	New York	945

Q-07. Find which age rating category has the most played games. Show user count by their age rating category in descending order.

```
SELECT
      ar.AGERATINGNAME,
      COUNT(s.USERID) AS "Usage Count"
FROM
      DB1SMOKE.GAMES g
INNER JOIN DB1SMOKE.SUBSCRIPTIONS s
ON
      g.GAMEID = s.GAMEID
INNER JOIN DB1SMOKE.AGE_RATINGS ar
ON
      g.AGERATING = ar.AGERATINGID
GROUP BY
      ar.AGERATINGNAME

ORDER  BY COUNT(s.USERID) DESC ;
```

	AGERATINGNAME	Usage Count
1	Rated R	107,434
2	Rated G	105,777
3	Rated PG	104,739

Q-08. Find which subscription model is the most popular. The popularity of a subscription model can be judged by the number of users play the games under that subscription type.

```
SELECT
      st.SUBSCRIPTIONNAME,
      COUNT(s.USERID) AS "Usage"
FROM
      DB1SMOKE.SUBSCRIPTIONS s
INNER JOIN DB1SMOKE.SUBSCRIPTION_TYPES st
ON
      s.SUBSCRIPTIONID = st.SUBSCRIPTIONID
GROUP BY
      st.SUBSCRIPTIONNAME
ORDER BY
      COUNT(s.USERID) DESC;
```

	SUBSCRIPTIONNAME	Usage
1	Free	159,135

Q-09. Find all duplicate users. A user is considered duplicate if they have the same first name, last name, city, and state.

```sql
SELECT
	u.FIRSTNAME,
	u.LASTNAME,
	u.CITY,
	u.STATE,
	COUNT(*) AS FREQ_CNT
FROM
	DB1SMOKE.USERS u
GROUP BY
	u.FIRSTNAME,
	u.LASTNAME,
	u.CITY,
	u.STATE
HAVING
	COUNT(*) > 1;
```

	FIRSTNAME	LASTNAME	CITY	STATE	FREQ_CNT
1	Josh	Akognon	Dallas	Texas	2
2	Tristan	Thompson	Cleveland	Ohio	2

```sql
Q-10. Find the top 3 most popular games.

SELECT
	g.GAMENAME,
	COUNT(s.USERID) AS "Usage"
FROM
	DB1SMOKE.GAMES g
INNER JOIN DB1SMOKE.SUBSCRIPTIONS s
ON
	g.GAMEID = s.GAMEID
GROUP BY
	g.GAMENAME
ORDER BY
	COUNT(s.USERID) DESC
FETCH NEXT 3 ROWS ONLY;
```

	GAMENAME	Usage
1	Missile Command	87
2	Warlords	87
3	Star Wars	80

Performance Tuning

Overview

Since this database contains elements of user and product information, it can grow quite huge overtime when new data is introduced. Database performance tuning will allow us to maximize resource utilization so that any application or service that runs on top of it will benefit from the critical database operations. This is important because even relatively minor database-related performance issues can impact the entire operation.

Indexing Strategy

Indexing is a very important strategy that can improve complex query performance involving joins, aggregation, etc. The benefit of indexes lies in the fact that it provides faster access to data for operations that return small number of a table's rows.

In our database, we have identified several columns that we suspect will be used more often than others. It is also a good idea to identify columns that are directly used in join conditions. Following is the description of all the indexes we created:

Column Name	Source Table	Reason
GameId ConsoleId	Game_Consoles	Game Console table is a mapping table. So, this will be the most frequently used table for connecting game id to console id. Console Id is also a frequently used column which has limited values since it is referencing a dimension table.
GenreId AgeRating	Games	These two columns are the primary keys of their respective dimension tables. Hence, they will offer value in filtering queries.
UserId GameId SubscriptionId GamerTag	Subscriptions	Subscription table is a central table that has many frequently used columns in join conditions.
GamerTag	Users	GamerTag works as a username so instead of finding a user by his/her name, this can act as pseudo-unique identifier.

Following lines of code shows the SQL statements for creating indexes on above mentioned table-column pairs:

```sql
-- Creating Indexes for Performance Tuining

CREATE INDEX IDX_GAMECONSOLE_GAMEID
ON
GAME_CONSOLES(GAMEID);

CREATE INDEX IDX_GAMECONSOLE_CONSOLEID
ON
GAME_CONSOLES(CONSOLEID);

CREATE INDEX IDX_GAMES_GENREID
ON
GAMES(GENREID);

CREATE INDEX IDX_GAMES_AGERATING
ON
GAMES(AGERATING);

CREATE INDEX IDX_SUBSCRIPTIONS_USERID
ON
SUBSCRIPTIONS(USERID);

CREATE INDEX IDX_SUBSCRIPTIONS_GAMEID
ON
SUBSCRIPTIONS(GAMEID);

CREATE INDEX IDX_SUBSCRIPTIONS_SUBSID
ON
SUBSCRIPTIONS(SUBSCRIPTIONID);

CREATE INDEX IDX_SUBSCRIPTIONS_GAMERTAG
ON
SUBSCRIPTIONS(GAMERTAG);

CREATE INDEX IDX_USERS_GAMERTAG
ON
USERS(GAMERTAG);
```

DB experiment for index performances

In previous section, we have created indexes on our main or production tables. Before loading the data into Oracle, we also had created the staging tables for backup. In this experiment, we will run a query on staging tables, that have no indexes on them, and on main tables, which have indexes created. We plan to compare the execution plans to see if they help in improving the query performance.

Staging Table Query

```sql
WITH GameConsole AS
(
SELECT
	c.CONSOLEID,
	c.CONSOLENAME ,
	COUNT(gc.GAMEID) AS GAME_COUNT
FROM
	DB1SMOKE.STG_CONSOLES c
INNER JOIN DB1SMOKE.STG_GAME_CONSOLES gc
ON
	c.CONSOLEID = gc.CONSOLEID
GROUP BY
	c.CONSOLEID,
	c.CONSOLENAME
HAVING
	COUNT(gc.GAMEID) > 100
	)
SELECT
	c.CONSOLENAME,
	COUNT(u.USERID) AS "USAGE COUNT"
FROM
	GameConsole c
INNER JOIN DB1SMOKE.STG_GAME_CONSOLES gc
ON
	c.CONSOLEID = gc.CONSOLEID
INNER JOIN DB1SMOKE.STG_SUBSCRIPTIONS s
ON
	s.GAMEID = gc.GAMEID
INNER JOIN DB1SMOKE.STG_USERS u
ON
	u.USERID = s.USERID
GROUP BY
	c.CONSOLENAME
ORDER BY
	COUNT(u.USERID) DESC;
```

Execution Plan Results

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			35	418
SORT		ORDER BY	35	418
HASH		GROUP BY	35	418
HASH JOIN			627254	388
Access Predicates				
U.USERID=S.USERID				
TABLE ACCESS	STG_USERS	FULL	2151	7
HASH JOIN			627254	379
Access Predicates				
S.GAMEID=GC.GAMEID				
HASH JOIN			29359	35
Access Predicates				
C.CONSOLEID=GC.CONSOLEID				
VIEW			44	20
FILTER				
Filter Predicates				
COUNT(GC.GAMEID)>100				
HASH		GROUP BY	44	20
HASH JOIN			23354	18
Access Predicates				
C.CONSOLEID=GC.CONSOLEID				
TABLE ACCE	STG_CONSOLES	FULL	35	3
TABLE ACCE	STG_GAME_CONSOLES	FULL	23354	15
TABLE ACCESS	STG_GAME_CONSOLES	FULL	23354	15
TABLE ACCESS	STG_SUBSCRIPTIONS	FULL	317950	343

Now we compare this plan with the one ran on main tables with indexes.

```
WITH GameConsole AS
(
SELECT
	c.CONSOLEID,
	c.CONSOLENAME ,
	COUNT(gc.GAMEID) AS GAME_COUNT
FROM
	DB1SMOKE.CONSOLES c
INNER JOIN DB1SMOKE.GAME_CONSOLES gc
ON
	c.CONSOLEID = gc.CONSOLEID
GROUP BY
	c.CONSOLEID,
	c.CONSOLENAME
HAVING
	COUNT(gc.GAMEID) > 100
	)
SELECT
	c.CONSOLENAME,
	COUNT(u.USERID) AS "USAGE COUNT"
FROM
	GameConsole c
INNER JOIN DB1SMOKE.GAME_CONSOLES gc
ON
	c.CONSOLEID = gc.CONSOLEID
INNER JOIN DB1SMOKE.SUBSCRIPTIONS s
ON
	s.GAMEID = gc.GAMEID
INNER JOIN DB1SMOKE.USERS u
ON
	u.USERID = s.USERID
```

```
GROUP BY
      c.CONSOLENAME
ORDER BY
      COUNT(u.USERID) DESC;
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			2	359
SORT		ORDER BY	2	359
HASH		GROUP BY	2	359
HASH JOIN			28512	356
Access Predicates				
S.GAMEID=GC.GAMEID				
NESTED LOOPS			28512	356
NESTED LOOPS				
STATISTICS COLLECTOR				
HASH JOIN			1335	37
Access Predicates				
C.CONSOLEID=GC.CONSOLEID				
VIEW			2	21
FILTER				
Filter Predicates				
COUNT(GC.GAMEID)>100				
HASH		GROUP BY	2	21
HASH JC			23354	19
Access Predicates				
C.CONSOLEID=GC.CONSOLEID				
TABI	CONSOLES	FULL	35	3
TABI	GAME_CONSOLES	FULL	23354	16
TABLE ACCESS	GAME_CONSOLES	FULL	23354	16
INDEX	IDX_SUBSCRIPTIONS_GAMEID	RANGE SCAN		
Access Predicates				
S.GAMEID=GC.GAMEID				
TABLE ACCESS	SUBSCRIPTIONS	BY INDEX ROWID	21	318
Filter Predicates				
S.USERID IS NOT NULL				
TABLE ACCESS	SUBSCRIPTIONS	FULL	317950	318
Filter Predicates				
S.USERID IS NOT NULL				

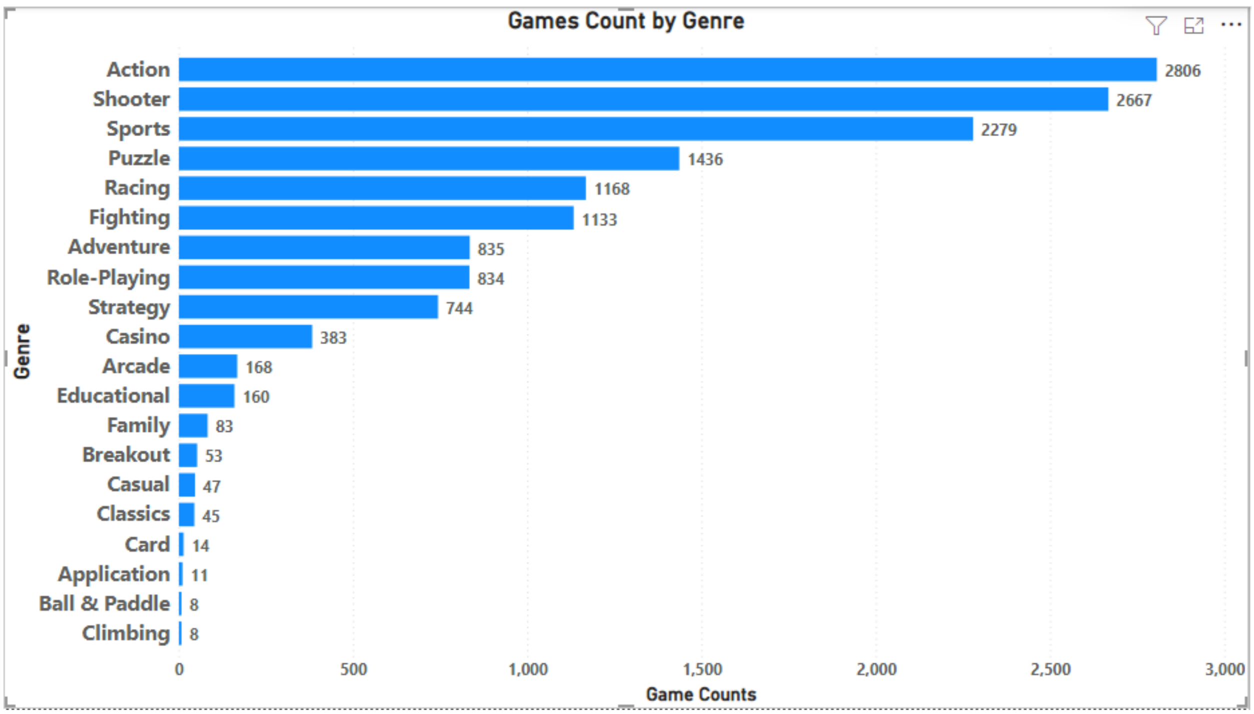
We can observe quite significant difference between the execution plans of the two queries. The last row where the table access occurs has a cost of 318 in indexed table as compared to the higher value of 343 in the non-indexed table. The performance gain occurred due to the BY INDEX ROWID section as pointed by the query coordinator.

Other Topics

Section 1.1: Data Visualization

Data visualization helps to tell stories by curating data into a form easier to understand, highlighting the trends and outliers. A good visualization tells a story, removing the noise from data and highlighting the useful information. We used Microsoft Power BI to visualize the data in our database.

Viz. 01 – Game Counts by Genre

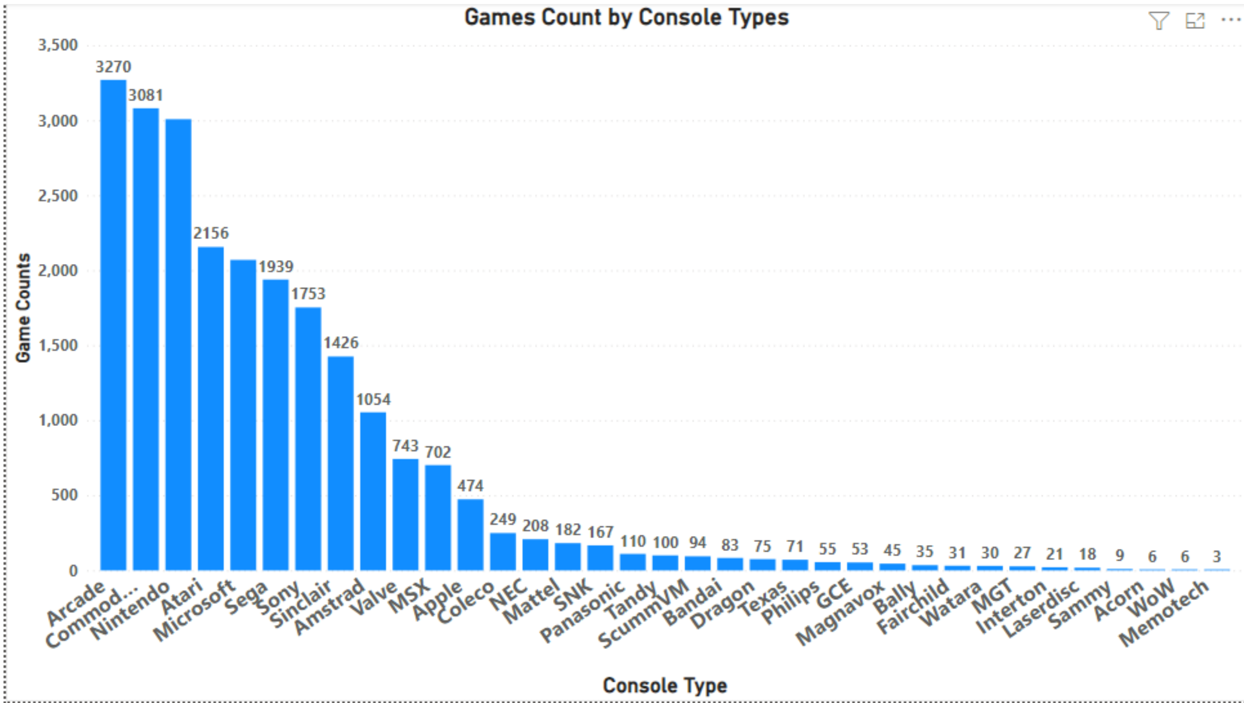


This visualization gives an overall distribution of games by their genres. This also tells what are the types of genres that are most popular among users and hence game developers target games in those categories.

Viz. 02 – Game Counts by Console Types

Similarly, we can observe the most popular consoles based on this data. Please note that this data is taken from a games archive that may not represent the most current trends. Here we see Sony, Microsoft Xbox, and Nintendo lagging some of the most older console systems. This again can change as we acquire more data.

The best thing about Power BI visuals is that they get updated if the data in their underlying tables change. Following visual helps to view this distribution:



Viz. 03 – Gamer's Location (Map Visualization)

To visualize location data, Power BI has a map visual, but it requires latitude and longitude data. To achieve that we use a standard dataset which has USA cities geo location data. The CSV looks something like below:

```python
cities = pd.read_csv(r'C:\Users\Shail\Downloads\simplemaps_uscities_basicv1.74/uscities.csv', on_bad_lines='skip')
print("(Row,Col) = {}".format(cities.shape))
cities.head()
```

[3] ✓ 0.4s Python

```
(Row,Col) = (28338, 17)
```

	city	city_ascii	state_id	state_name	county_fips	county_name	lat	lng	population	density	source	military	incorporated
0	New York	New York	NY	New York	36061	New York	40.6943	-73.9249	18713220	10715	polygon	False	True
1	Los Angeles	Los Angeles	CA	California	6037	Los Angeles	34.1139	-118.4068	12750807	3276	polygon	False	True

We imported this dataset into a table: `DB1SMOKE.CITIES`

On top of it we created following view:

```
CREATE VIEW DB1SMOKE.VW_USER_LOCATION_DATA AS
SELECT
      u.UserId,
      c.county_name AS Region,
      u.City,
      u.State,
      c.state_id AS StateCode,
      u.Age,
      u.GamerScore,
      c.lat AS Latitude,
      c.lng AS Longitude,
      'United States' AS Country,
      'USA' AS CountryCode
FROM
      DB1SMOKE.USERS u
INNER JOIN DB1SMOKE.CITIES c
ON
      u.City = c.city
      AND u.State = c.state_name;
```

Utilizing the above view, we could use “Latitude” and “Longitude” information to populate the map visualization. Following is the output for our console-games data:

Section 1.2: Linear Regression on Price

Let's create a custom view to populate the required data from our database. Our goal is to run a linear regression model for price and using rating, user count, console count as independent variables.

We create following VIEW:

```
CREATE VIEW DB1SMOKE.VW_GameStats AS
WITH game_stats AS
(
SELECT
      g.GAMENAME ,
      g.PRICE ,
      g.RATING ,
      COUNT(s.USERID) AS UserCounts,
      COUNT(gc.CONSOLEID) AS ConsoleCounts
FROM
      DB1SMOKE.GAMES g
INNER JOIN DB1SMOKE.SUBSCRIPTIONS s
ON
      g.GAMEID = s.GAMEID
INNER JOIN DB1SMOKE.GAME_CONSOLES gc
ON
      gc.GAMEID = g.GAMEID
GROUP BY
      g.GAMENAME ,
      g.PRICE ,
      g.RATING
)
SELECT
      GameName,
      ROUND(AVG(Price), 3) AS Price,
      ROUND(AVG(Rating), 3) AS Rating,
      ROUND(AVG(UserCounts), 3) AS UserCounts,
      ROUND(AVG(ConsoleCounts), 3) AS ConsoleCounts
FROM
      game_stats
GROUP BY
      GameName
```

We export this data into a CSV file so that we can read it in a different statistical analysis tool. We initialized a linear regression model with following parameters:

- Dependent Variable: Price
- Independent Variables: Rating, Console Counts, Game Counts

Following is the output of the linear model:

```
Call:
lm(formula = Price ~ Rating + UserCounts + ConsoleCounts, data = data)

Residuals:
     Min       1Q   Median       3Q      Max 
-27.3403  -4.4883  -0.2253   4.2104  27.9278 

Coefficients: (1 not defined because of singularities)
               Estimate Std. Error t value Pr(>|t|)    
(Intercept)   48.064067   0.416564 115.382   <2e-16 ***
Rating        -1.370347   0.107295 -12.772   <2e-16 ***
UserCounts     0.001122   0.002422   0.463    0.643    
ConsoleCounts        NA         NA      NA       NA    
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 6.522 on 14251 degrees of freedom
Multiple R-squared:  0.01132,Adjusted R-squared:  0.01118 
F-statistic:  81.6 on 2 and 14251 DF,  p-value: < 2.2e-16
```

We can observe that only Rating attribute shows the most significant factor in determining the price of the game. Number of Users playing, and Number of Console Counts has insignificant affect. This is as per the data in our database and might not represent the real-world scenarios.

Section 1.3: Initializing Machine Learning Models

Overview

In this experiment, our goal is to continue analyzing **Price** attribute. In one of the queries we created a **Cost Category** attribute that labels the Price attribute whether it is Very High, High, Average or Low. We will use this attribute to initialize two machine learning algorithms and compare the results.

Dataset Creation

We start by fetching the relevant data from our Database, to do this, we created a view that will be used to export this data into a CSV file:

```
CREATE VIEW [dbo].[VW_Game_Stats] AS
WITH stat AS
(
SELECT
      g.name AS GameName,
      g.price AS Price,
      g.rating AS Rating,
```

```
        count(s.userid) AS UserCounts,
        count(gc.consoleid) AS ConsoleCounts,
        CASE WHEN g.price >= 60 THEN 'Very High'
        WHEN g.price >=45 AND g.price < 60 THEN 'High'
        WHEN g.price >=30 AND g.price < 45 THEN 'Average'
        ELSE 'Low'
        END as CostCategory
FROM
        games g
INNER JOIN subscriptions s
ON
        g.gameid = s.gameid
INNER JOIN game_consoles gc
ON
        gc.gameid = g. gameid
GROUP BY
        g.name,
        g.price,
        g.rating
)
SELECT
        GameName,
        ROUND(AVG(Price), 3) AS Price,
        ROUND(AVG(Rating),3) AS Rating,
        ROUND(AVG(UserCounts),3) AS UserCounts,
        ROUND(AVG(ConsoleCounts),3) AS ConsoleCounts,
        CostCategory
FROM
        stat
GROUP BY GameName, CostCategory
```

Creating a ML Experiment

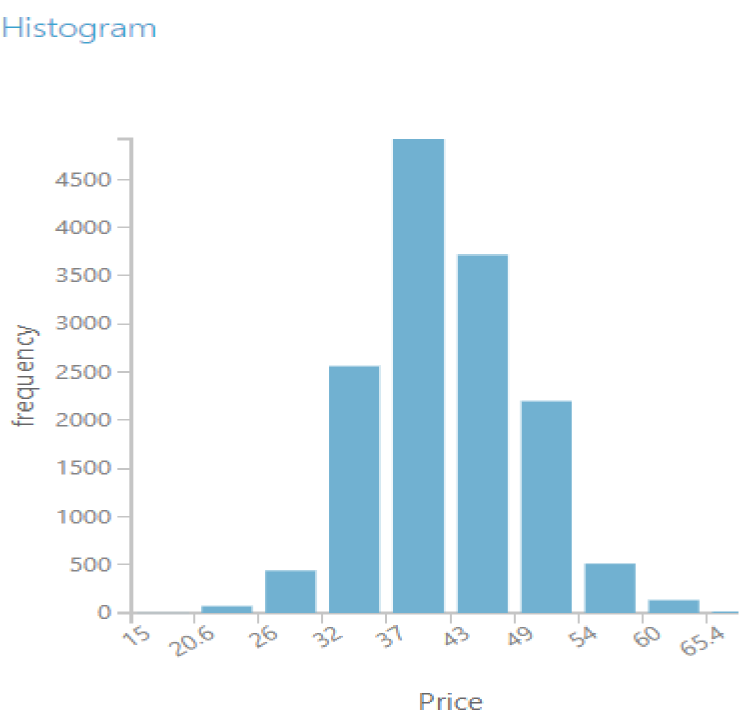
After creating the CSV, we import this dataset into ***Azure ML Studio*** (https://studio.azureml.net/). The dataset looks as shown in the Image below:

DBMS - Experiment 5/1/2022 › game_stats_new.csv › dataset

rows	columns
14564	6

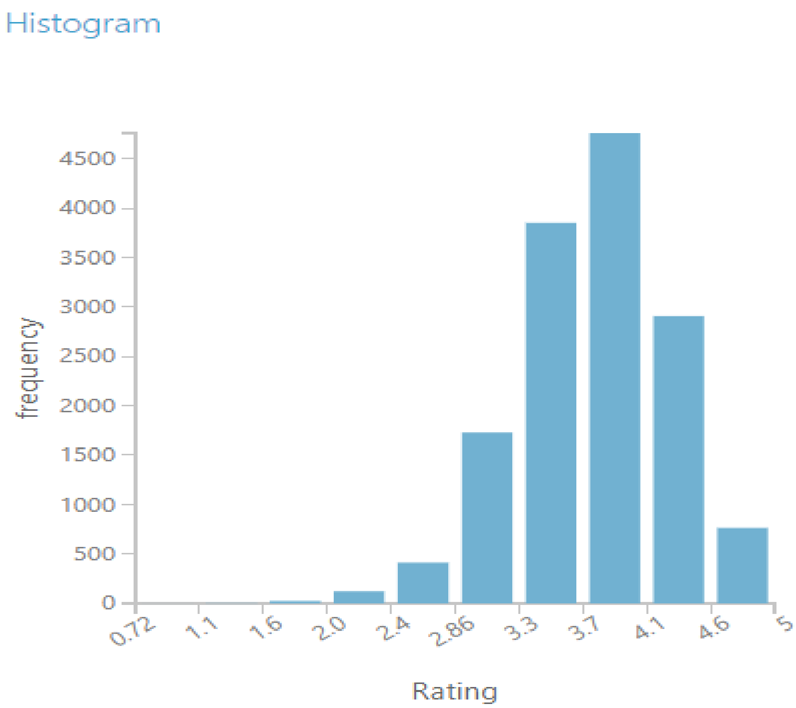
GameName	Price	Rating	UserCounts	ConsoleCounts	CostCategory
.38 Ambush Alley	40	5	19	19	Average
007 Legends	39	3.72	21	21	Average
007: A View to a Kill	43	3.12	25	25	Average
007: Licence to Kill	44	3.12	100	100	Average
007: Nightfire	37	3.89	38	38	Average
007: The Duel	40	4.2	17	17	Average
1 on 1 Government	41	3.34	22	22	Average

The following charts shows the data distribution of the independent variables:



Statistics

Mean	42.921
Median	43
Min	15
Max	71
Standard Deviation	6.6305
Unique Values	87
Missing Values	0
Feature Type	Numeric Featu



Statistics

Mean	3.7928
Median	3.82
Min	0.72
Max	5
Standard Deviation	0.5106
Unique Values	328
Missing Values	0
Feature Type	Numeric Feature

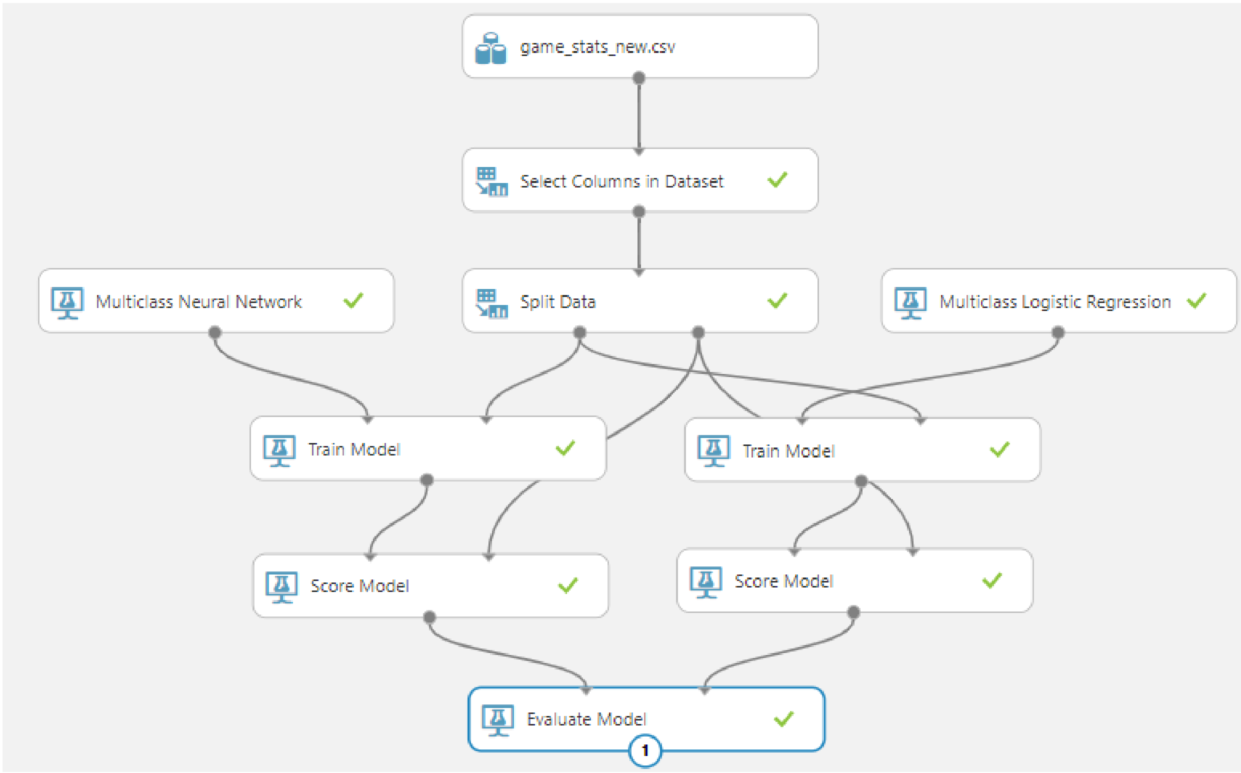
Initializing Model

We perform following steps in initializing model:

1. Import the dataset
2. Select the attributes to be used
    a. In this case, the dependent variable is a Categorical Variable: **Cost Category**
    b. The independent variables are **Price, Rating, Console Counts, Game Counts**
3. Split the train and test data by 70-30 split respectively.
4. Initialize Classification Algorithms
    a. Multi-Class Neural Networks
    b. Multi-Class Logistic Regression
5. Score the model using the test data and the Algorithm's outputs.
6. Evaluate the results for both for comparison.

We choose multi-Class algorithms because there are four classes viz: Very High, High, Average and Low. We choose classification algorithms because our dependent variable is a categorical variable.

The following figure the experiment created in Azure ML Studio:



Model Evaluation Results

DBMS - Experiment 5/1/2022 › Evaluate Model › Evaluation results

Metrics

Overall accuracy	0.984894
Average accuracy	0.992447
Micro-averaged precision	0.984894
Macro-averaged precision	0.991287
Micro-averaged recall	0.984894
Macro-averaged recall	0.777618

Metrics

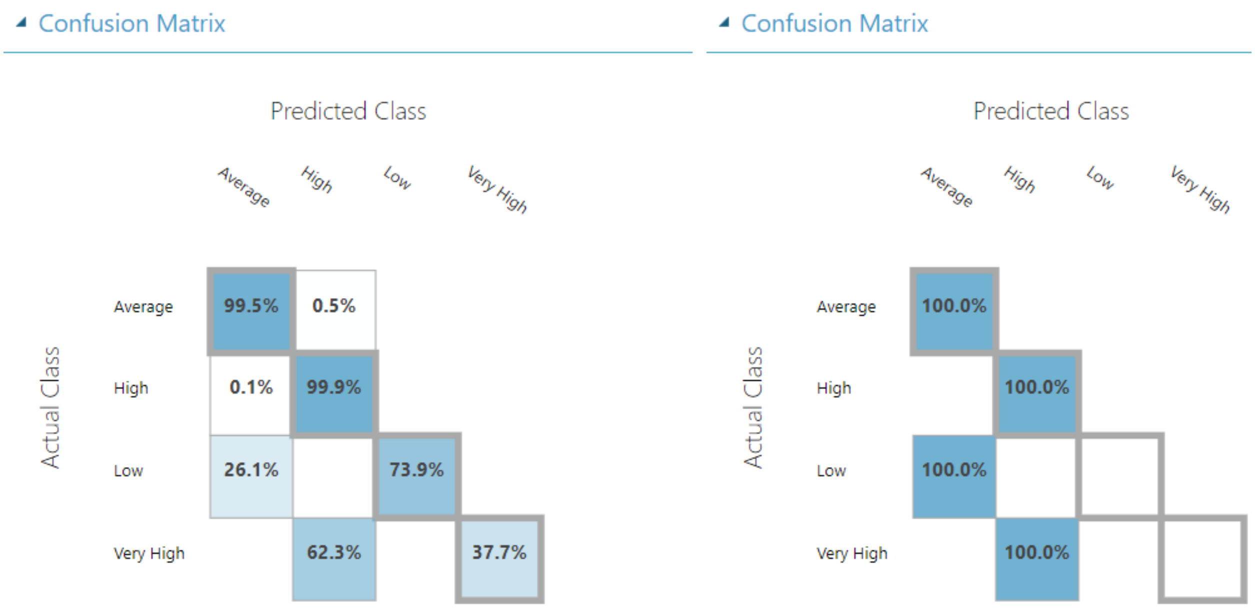
Overall accuracy	0.972076
Average accuracy	0.986038
Micro-averaged precision	0.972076
Macro-averaged precision	NaN
Micro-averaged recall	0.972076
Macro-averaged recall	0.5

Left: Neural Networks | Right: Logistic Regression

Both the algorithms performed well with Neural Networks showing a slightly better accuracy.

Let's evaluate the confusion matrix, which is a very popular measure used while solving classification problems. It can be applied to binary classification as well as for multiclass classification problems.

The following image the confusion matrix comparison between the two algorithms:



Left: Neural Networks | Right: Logistic Regression

We observe that the Logistic Regression didn't perform well as the number of false-positives and false-negatives are quite high. The Neural Networks did better job in this regard.