GAME ANALYSIS with SQL

Project By :

PREM NIMJE

Content

- Project Overview
- Dataset Description
- Entity Relationship Diagram
- Analysis (Queries)
- Summary

Project Overview

Decode Gaming Behavior" involves analyzing a gaming application's dataset with "Player Details" and "Level Details" tables. Its objective is to extract insights into player behavior and performance. Utilizing SQL queries, we aim to understand player engagement, skill progression, and areas for game experience enhancement. Key questions include player trends, level completion rates, and performance metrics analysis. Our goal is to provide actionable insights for informed decision-making in game development. The project encompasses data exploration, query formulation, result interpretation, and data visualization techniques. Through concise presentation, we facilitate stakeholders' understanding and decision-making in game development and management.

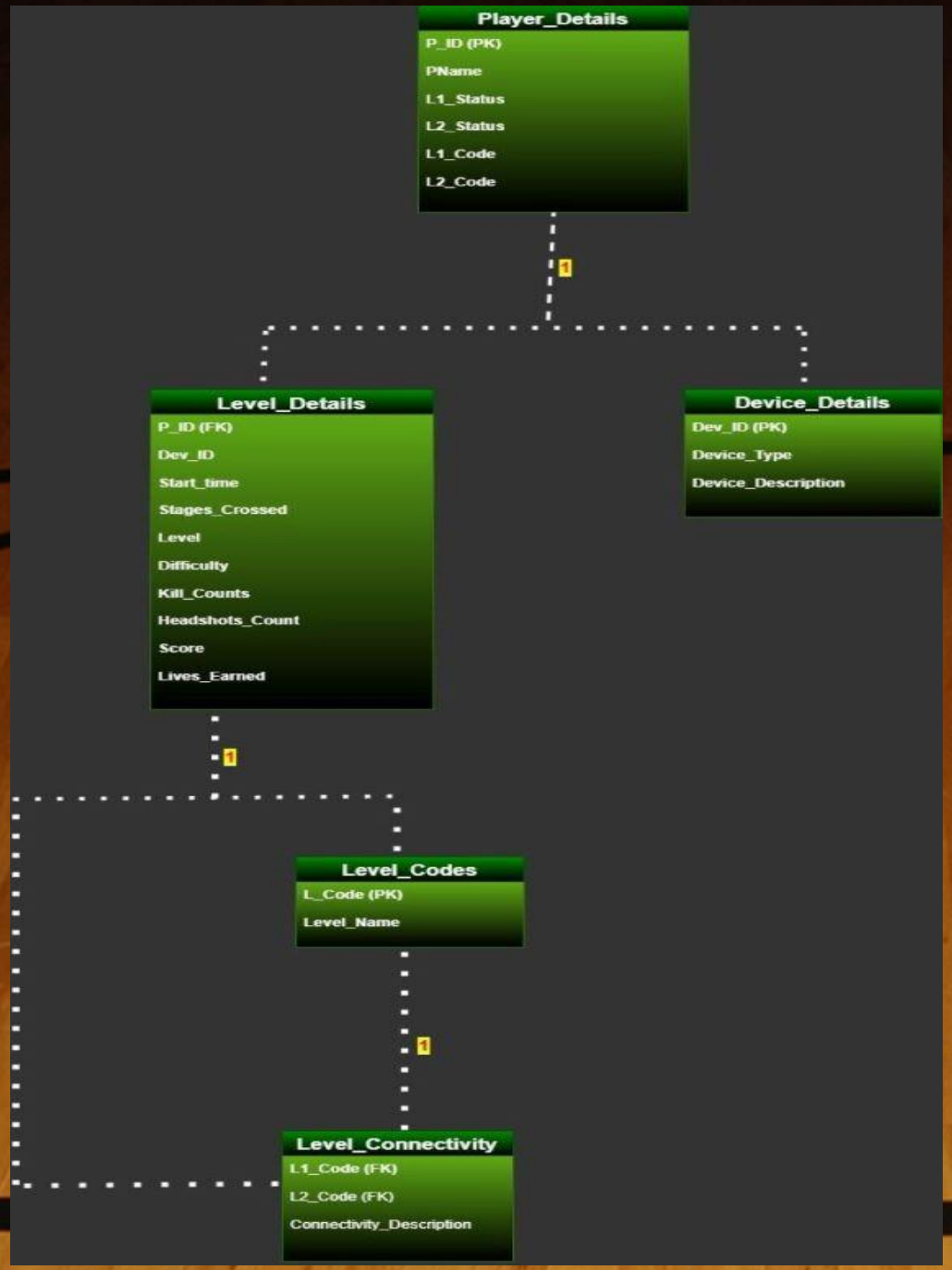
Dataset Description

The dataset includes two tables: `Player Details` and `Level Details`:

Player Details Table:

- **`P_ID`: Player ID**
- **`PName`: Player Name**
- **`L1_status`: Level 1 Status**
- **`L2_status`: Level 2 Status**
- **`L1_code`: System generated Level 1 Code**
- **`L2_code`: System generated Level 2 Code**

- **Level Details Table:**
- **`P_ID`: Player ID**
- **`Dev_ID`: Device ID**
- **`start_time`: Start Time**
- **`stages_crossed`: Stages Crossed**
- **`level`: Game Level**
- **`difficulty`: Difficulty Level**
- **`kill_count`: Kill Count**
- **`headshots_count`: Headshots Count**
- **`score`: Player Score**
- **`lives_earned`: Extra Lives Earned**



Entity Relationship Diagram

- We've added a new entity called "Device_Details" to capture information about the devices used by players.
- The "Level_Details" table now includes an attribute Dev_ID to indicate which device was used.
- Another entity called "Level_Codes" is introduced to store information about the codes associated with each level.
- "Level_Connectivity" represents the relationships between levels, using the codes from "Level_Codes" to indicate the connectivity between different levels.
- Arrows indicate the relationships between entities, with cardinality specified where necessary (1-to-many relationships).

Analysis (Queries)

Query – 1)

Extract P_ID, Dev_ID, PName and Difficulty_level of all players at level 0.

SELECT pd.P_ID, ld.Dev_ID, pd.PName, ld.difficulty

FROM Player_Details pd

INNER JOIN Level_Details ld ON pd.P_ID = ld.P_ID

WHERE ld.level = 0;

Analysis -- It performs an inner join on the Player ID column between the two tables to retrieve matching records based on the Player ID.

```
69  select pd.p_id, id.dev_id, pd.pname, id.difficulty
70  from player_details pd
71  inner join level_details id on pd.p_id = id.p_id
72  where id.level = 0;
```

Data Output | Messages | Notifications

	p_id bigint	dev_id character varying (50)	pname character varying (50)	difficulty character varying (50)
1	656	rf_013	sloppy-denim-wolfhound	Medium
2	632	bd_013	dorky-heliotrope-barracuda	Difficult
3	429	bd_013	flabby-firebrick-bee	Medium
4	310	bd_015	gloppy-tomato-wasp	Difficult
5	211	bd_017	breezy-indigo-starfish	Low
6	300	zm_015	lanky-asparagus-gar	Difficult
7	358	zm_017	skinny-grey-quetzal	Low
8	358	zm_013	skinny-grey-quetzal	Medium
9	641	rf_013	homey-alizarin-gar	Low
10	641	rf_015	homey-alizarin-gar	Medium

Total rows: 12 of 12 Query complete 00:00:00.139

Query – 2)

Find Level1_code wise Avg_Kill_Count where lives_earned is 2 and atleast 3 stages are crossed.

SELECT pd.L1_code, AVG(ld.kill_count) AS avg_kill_count

FROM Player_Details pd

INNER JOIN Level_Details ld ON pd.P_ID = ld.P_ID

WHERE ld.lives_earned = 2 AND ld.stages_crossed >= 3

GROUP BY pd.L1_code;

Analysis -- It performs an inner join on the Player ID column between Player_Details and Level_Details tables to retrieve matching records based on the Player ID. The result is grouped by L1_code.

```
70  SELECT pd.L1_code, AVG(ld.kill_count) AS avg_kill_count
71  FROM Player_Details pd
72  INNER JOIN Level_Details ld ON pd.P_ID = ld.P_ID
73  WHERE ld.lives_earned = 2 AND ld.stages_crossed >= 3
74  GROUP BY pd.L1_code;
75
76
```

Data Output | Messages | Notifications

	l1_code character varying (50)	avg_kill_count numeric
1	bulls_eye	22.2500000000000000
2	war_zone	19.2857142857142857
3	speed_blitz	19.3333333333333333

Query – 3)

Find the total number of stages crossed at each diffuculty level where for Level2 with players use zm_series devices. Arrange the result.

SELECT ld.difficulty, SUM(ld.stages_crossed) AS total_stages_crossed

FROM Level_Details ld

INNER JOIN Player_Details pd ON ld.P_ID = pd.P_ID

WHERE ld.level = 2 AND ld.Dev_ID LIKE 'zm_series%'

GROUP BY ld.difficulty

ORDER BY total_stages_crossed DESC;

Analysis -- It performs an inner join with the Player_Details table based on the Player ID. The result is grouped by difficulty and ordered by the total number of stages crossed in descending order.

```
69  SELECT ld.difficulty, SUM(ld.stages_crossed) AS total_stages_crossed
70  FROM Level_Details ld
71  INNER JOIN Player_Details pd ON ld.P_ID = pd.P_ID
72  WHERE ld.level = 2 AND ld.Dev_ID LIKE 'zm_series%'
73  GROUP BY ld.difficulty
74  ORDER BY total_stages_crossed DESC;
75
76
```

Data Output | Messages | Notifications

difficulty character varying (50)	total_stages_crossed bigint

Query – 4)

Extract P_ID and the total number of unique dates for those players who have played games on multiple days.

```
SELECT P_ID, COUNT(DISTINCT
DATE(start_datetime)) AS Unique_Dates

FROM Game_Data

GROUP BY P_ID

HAVING  COUNT(DISTINCT
DATE(start_datetime)) > 1;
```

Analysis -- It groups the results by P_ID and filters out the groups where the count of unique dates is greater than 1. This query helps identify players who have started games on multiple dates.

```
4  select p_id, count(distinct date(timestamp)) as unique_dates_count
5  from level_details
6  group by p_id
7  having count(distinct date(timestamp)) > 1;
```

Data Output | Messages | Notifications

	p_id integer	unique_dates_count bigint
1	211	4
2	224	2
3	242	2
4	292	2
5	300	3
6	310	3
7	368	2
8	483	3
9	590	3
10	632	3
11	641	2

Total rows: 14 of 14 Query complete 00:00:00.167

Query – 5)

Find P_ID and level wise sum of kill_counts where kill_count is greater than avg kill count for the Medium difficulty.

SELECT P_ID, Level, SUM(Kill_Count) AS Total_Kill_Count

FROM Game_Data

WHERE Kill_Count > (SELECT AVG(Kill_Count) FROM Game_Data WHERE Difficulty_level = 'Medium')

GROUP BY P_ID, Level;

Analysis -- It filters the data based on the condition that the Kill_Count is greater than the average Kill_Count for records with the Difficulty_level set to 'Medium'. Finally, it groups the results by P_ID and Level. This query helps identify players who have achieved above-average kill counts in levels classified as 'Medium' difficulty.

```
4  SELECT ld.P_ID, ld.level, SUM(ld.kill_count) AS total_kill_count
5  FROM Level_Details ld
6  INNER JOIN (
7      SELECT AVG(kill_count) AS avg_kill_count
8      FROM Level_Details
9      WHERE difficulty = 'Medium'
10 ) AS avg_table ON ld.kill_count > avg_table.avg_kill_count
11 GROUP BY ld.P_ID, ld.level;
12
```

Data Output | Messages | Notifications

	p_id integer	level integer	total_kill_count bigint
1	211	0	20
2	632	2	53
3	683	1	21
4	683	2	64
5	368	1	20
6	310	1	20
7	429	1	30

Total rows: 27 of 27 Query complete 00:00:00.222

Query – 6)

Find Level and its corresponding Level code wise sum of lives earned excluding level 0. Arrange in asecending order of level.

```
SELECT Level, Level_code,
SUM(lives_earned) AS Total_Lives_Earned

FROM Game_Data

WHERE Level > 0

GROUP BY Level, Level_code

ORDER BY Level ASC;
```

Analysis -- It filters the data to exclude Level 0, which typically represents the initial level or setup phase. Then, it calculates the sum of lives earned for each level and groups the results by Level and Level_code. Finally, it orders the results by Level in ascending order.

```
4  SELECT ld.level, pd.L1_code, SUM(ld.lives_earned) AS total_lives_earned
5  FROM Level_Details ld
6  INNER JOIN Player_Details pd ON ld.P_ID = pd.P_ID
7  WHERE ld.level > 0
8  GROUP BY ld.level, pd.L1_code
9  ORDER BY ld.level ASC;
10
11
```

Data Output | Messages | Notifications

	level integer	l1_code character varying (50)	total_lives_earned bigint
1	1	bulls_eye	5
2	1	leap_of_faith	0
3	1	speed_blitz	7
4	1	war_zone	11
5	2	bulls_eye	14
6	2	speed_blitz	20
7	2	war_zone	17

Total rows: 7 of 7 Query complete 00:00:00.122

Query – 7)

Find Top 3 score based on each dev_id and Rank them in increasing order using Row_Number. Display difficulty as well.

WITH TopScores AS (SELECT Dev_ID, Difficulty_level, Score, ROW_NUMBER() OVER(PARTITION BY Dev_ID ORDER BY Score DESC) AS Rank FROM Game_Data)

SELECT Dev_ID, Difficulty_level, Score, Rank

FROM TopScores

WHERE Rank <= 3;

Analysis -- By partitioning the data by Developer ID and ranking scores within each group, the query efficiently retrieves the highest scores. The main query then selects the Developer ID, Difficulty Level, Score, and Rank from the TopScores CTE, ensuring only the top three scores are included for each developer.

```
4  WITH RankedScores AS (
5      SELECT *,
6             ROW_NUMBER() OVER(PARTITION BY Dev_ID ORDER BY score DESC) AS rank
7      FROM Level_Details
8  )
9  SELECT Dev_ID, difficulty, score, rank
10 FROM RankedScores
11 WHERE rank <= 3;
12
```

Data Output | Messages | Notifications

	dev_id character varying (50)	difficulty character varying (50)	score integer	rank bigint
1	bd_013	Difficult	5300	1
2	bd_013	Difficult	4570	2
3	bd_013	Difficult	3370	3
4	bd_015	Difficult	5300	1
5	bd_015	Low	3200	2
6	bd_015	Difficult	1950	3
7	bd_017	Low	2400	1

Total rows: 30 of 30 Query complete 00:00:00.161

Query – 8)

Find first_login datetime for each device id.

SELECT Dev_ID, MIN(start_datetime) AS first_login

FROM Game_Data

GROUP BY Dev_ID;

Analysis -- The SQL query retrieves the earliest login timestamp for each developer by selecting the minimum start datetime grouped by the developer's ID from the Game_Data table

```
3  SELECT Dev_ID, MIN(timestamp) AS first_login
4  FROM Level_Details
5  GROUP BY Dev_ID;
6
7
```

Data Output | Messages | Notifications

	dev_id character varying (50)	first_login timestamp with time zone
1	rf_015	2022-10-11 19:34:00+05:30
2	zm_015	2022-10-11 14:05:00+05:30
3	wd_019	2022-10-12 23:19:00+05:30
4	rf_013	2022-10-11 05:20:00+05:30
5	zm_017	2022-10-11 14:33:00+05:30
6	bd_013	2022-10-11 02:23:00+05:30
7	bd_017	2022-10-12 07:30:00+05:30
8	bd_015	2022-10-11 18:45:00+05:30
9	zm_013	2022-10-11 13:00:00+05:30
10	rf_017	2022-10-11 09:28:00+05:30

Query – 9)

Find Top 5 score based on each difficulty level and Rank them in increasing order using Rank. Display dev_id as well.

WITH TopScores AS (SELECT Dev_ID, Difficulty_level, Score, RANK() OVER(PARTITION BY Difficulty_level ORDER BY Score DESC) AS Rank FROM Game_Data)

SELECT Dev_ID, Difficulty_level, Score, Rank FROM TopScores

WHERE Rank <= 5;

Analysis -- It assigns a rank to each score based on descending order. Then, it selects the developer ID, difficulty level, score, and rank from the TopScores CTE where the rank is less than or equal to 5.

```
4  WITH RankedScores AS (
5      SELECT *,
6             RANK() OVER(PARTITION BY difficulty ORDER BY score DESC) AS rank
7      FROM Level_Details
8  )
9  SELECT Dev_ID, difficulty, score, rank
10 FROM RankedScores
11 WHERE rank <= 5;
12
```

Data Output | Messages | Notifications

	dev_id character varying (50)	difficulty character varying (50)	score integer	rank bigint
1	zm_017	Difficult	5500	1
2	zm_017	Difficult	5500	1
3	bd_013	Difficult	5300	3
4	bd_015	Difficult	5300	3
5	rf_017	Difficult	5140	5
6	zm_015	Low	3470	1
7	zm_017	Low	3210	2

Query – 10)

Find the device ID that is first logged in(based on start_datetime) for each player(p_id). Output should contain player id, device id and first login datetime.

WITH FirstLogin AS (SELECT P_ID, Dev_ID, start_datetime, ROW_NUMBER() OVER(PARTITION BY P_ID ORDER BY start_datetime) AS RowNum FROM Game_Data)

SELECT P_ID, Dev_ID, start_datetime AS first_login FROM FirstLogin

WHERE RowNum = 1;

Analysis -- It assigns a row number to each login record within each player's data, ordered by the start_datetime. Then, it selects the player ID (P_ID), developer ID (Dev_ID), and the start_datetime corresponding to the first login (identified by RowNum = 1) from the FirstLogin CTE.

```
4  WITH FirstLogin AS (
5      SELECT P_ID, DEV_ID, MIN(timestamp) AS first_login
6      FROM Level_Details
7      GROUP BY P_ID, DEV_ID
8  )
9  SELECT fl.P_ID, fl.Dev_ID, fl.first_login
10 FROM FirstLogin fl
11 INNER JOIN Level_Details ld ON fl.P_ID = ld.P_ID AND fl.first_login = ld.timestamp
12
```

Data Output | Messages | Notifications

	p_id integer	dev_id character varying (50)	first_login timestamp with time zone
1	644	zm_015	2022-10-11 14:05:00+05:30
2	644	rf_015	2022-10-11 19:34:00+05:30
3	644	bd_017	2022-10-12 23:52:00+05:30
4	656	rf_013	2022-10-15 18:12:00+05:30
5	656	bd_015	2022-10-13 22:19:00+05:30
6	656	rf_017	2022-10-14 07:32:00+05:30
7	656	bd_013	2022-10-11 17:47:00+05:30

Query – 11)

For each player and date, how many kill_count played so far by the player. That is, the total number of games played by the player until that date.

A) window function

```
SELECT P_ID, start_datetime,
    SUM(Kill_Count) OVER(PARTITION BY P_ID ORDER BY start_datetime) AS Total_Kill_Count
FROM Game_Data;
```

Analysis -- It utilizes the window function SUM() with the OVER() clause to partition the data by P_ID and order it by start_datetime. This allows tracking the total kill count accumulated by each player as they progress through the game sessions, aiding in analyzing player performance trends and engagement levels over time.

```
1  SELECT P_ID, DATE(timestamp) AS date, SUM(kill_count)
2  OVER (PARTITION BY P_ID ORDER BY timestamp) AS total_kill_count
3  FROM Level_Details;
4
```

Data Output | Messages | Notifications

	p_id integer	date date	total_kill_count bigint
1	211	2022-10-12	20
2	211	2022-10-12	45
3	211	2022-10-13	75
4	211	2022-10-13	89
5	211	2022-10-14	98
6	211	2022-10-15	113
7	224	2022-10-14	20
8	224	2022-10-14	54
9	224	2022-10-15	84
10	224	2022-10-15	112
11	242	2022-10-13	21

Query – 11)

B) without window function

```
SELECT P_ID, start_datetime,
       (SELECT SUM(Kill_Count) FROM Game_Data sub WHERE sub.P_ID = main.P_ID AND sub.start_datetime <= main.start_datetime) AS Total_Kill_Count
FROM Game_Data main;
```

Analysis -- It utilizes a correlated subquery to sum the Kill_Count values from the Game_Data table for each player where the start_datetime is less than or equal to the start_datetime of the current row. This provides a running total of kill counts for each player as they progress through their gaming sessions.

```
1  SELECT P_ID, DATE(timestamp) AS date, SUM(kill_count) AS total_kill_count
2  FROM Level_Details
3  GROUP BY P_ID, DATE(timestamp);
4
5
```

Data Output | Messages | Notifications

	p_id integer	date date	total_kill_count bigint
1	368	2022-10-15	24
2	224	2022-10-15	58
3	656	2022-10-11	18
4	429	2022-10-11	99
5	644	2022-10-11	18
6	632	2022-10-12	73
7	656	2022-10-14	3
8	300	2022-10-12	18
9	368	2022-10-12	49
10	483	2022-10-11	70
11	296	2022-10-14	11

Total rows: 47 of 47 Query complete 00:00:00.221

Query – 12)

Find the cumulative sum of stages crossed over a start_datetime for each player id but exclude the most recent start_datetime

```
SELECT P_ID, start_time,
       SUM(stages_crossed) OVER (PARTITION BY P_ID ORDER BY start_time ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING) AS cumulative_stages_crossed
FROM Level_Details;
```

Analysis -- It utilizes the SUM() function with the window function OVER() to sum the stages_crossed values from the Game_Data table for each player. The ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING clause specifies the range of rows to include in the sum, which in this case is from the beginning of the partition (UNBOUNDED PRECEDING) up to the row immediately preceding the current row.

```
1  SELECT P_ID, timestamp, stages_crossed,
2         SUM(stages_crossed) OVER (PARTITION BY P_ID
3                                   ORDER BY timestamp
4                                   ROWS BETWEEN UNBOUNDED
5                                   PRECEDING AND 1 PRECEDING) AS cumulative_sum
6  FROM Level_Details;
7
```

Data Output | Messages | Notifications

	p_id integer	timestamp timestamp with time zone	stages_crossed integer	cumulative_sum bigint
1	211	2022-10-12 13:23:00+05:30	4	[null]
2	211	2022-10-12 18:30:00+05:30	5	4
3	211	2022-10-13 05:36:00+05:30	5	9
4	211	2022-10-13 22:30:00+05:30	5	14
5	211	2022-10-14 08:56:00+05:30	7	19
6	211	2022-10-15 11:41:00+05:30	8	26
7	224	2022-10-14 01:15:00+05:30	7	[null]
8	224	2022-10-14 08:21:00+05:30	5	7
9	224	2022-10-15 05:30:00+05:30	10	12
10	224	2022-10-15 13:43:00+05:30	4	22

Query – 13)

Extract top 3 highest sum of score for each device id and the corresponding player_id

WITH RankedScores AS (SELECT P_ID, Dev_ID, SUM(score) AS total_score, ROW_NUMBER() OVER (PARTITION BY Dev_ID ORDER BY SUM(score) DESC) AS rank FROM Level_Details GROUP BY P_ID, Dev_ID)

SELECT P_ID, Dev_ID, total_score

FROM RankedScores

WHERE rank <= 3;

Analysis -- It then assigns a rank to each player within each device based on their total score, with the highest scorer receiving rank 1. The results are filtered to include only the top 3 scorers for each device, showing their P_ID, Dev_ID, and total_score.

```
1  WITH RankedScores AS (
2      SELECT P_ID, Dev_ID, SUM(score) AS total_score,
3             ROW_NUMBER() OVER (PARTITION BY Dev_ID ORDER BY SUM(score) DESC) AS rank
4      FROM Level_Details
5      GROUP BY P_ID, Dev_ID
6  )
7  SELECT P_ID, Dev_ID, total_score
8  FROM RankedScores
9  WHERE rank <= 3;
10
```

Data Output | Messages | Notifications

	p_id integer	dev_id character varying (50)	total_score bigint
1	224	bd_013	9870
2	310	bd_013	3370
3	211	bd_013	3200
4	310	bd_015	5300
5	683	bd_015	3200
6	368	bd_015	1950
7	590	bd_017	2400
8	644	bd_017	1750
9	211	bd_017	390

Query – 14)

Find players who scored more than 50% of the avg score scored by sum of scores for each player_id.

```
SELECT P_ID FROM (SELECT P_ID, SUM(score) AS total_score FROM Level_Details
    GROUP BY P_ID) AS player_scores
WHERE total_score > 0.5 * (
    SELECT AVG(total_score) FROM (
        SELECT SUM(score) AS total_score
        FROM Level_Details
        GROUP BY P_ID
    ) AS avg_scores
);
```

Analysis -- It calculates the total score for each player in the inner subquery and then filters the results based on the condition specified.

```
1  SELECT P_ID
2  FROM (
3      SELECT P_ID, SUM(score) AS total_score
4      FROM Level_Details
5      GROUP BY P_ID
6  ) AS player_scores
7  WHERE total_score > 0.5 * (
8      SELECT AVG(total_score) FROM (
9          SELECT SUM(score) AS total_score
10         FROM Level_Details
11         GROUP BY P_ID
12     ) AS avg_scores
13 );
```

Data Output | Messages | Notifications

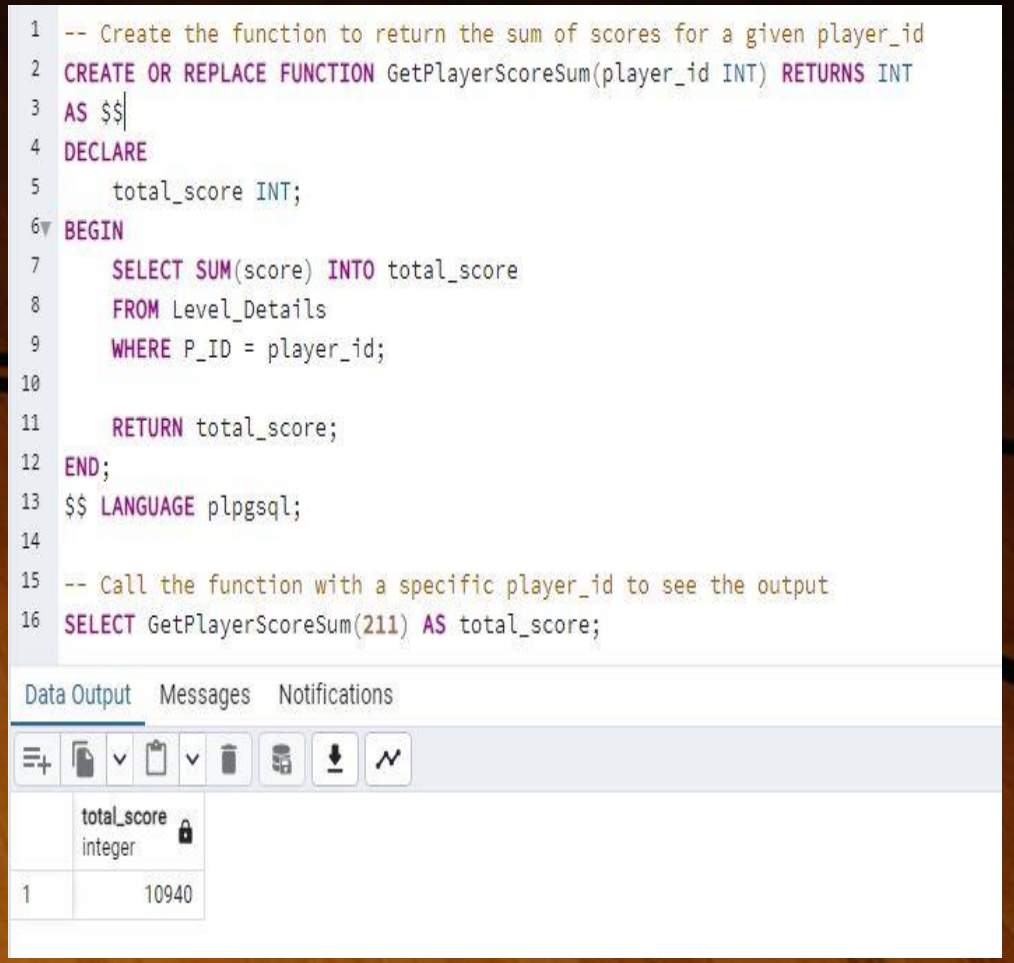
	p_id integer
1	429
2	590
3	663
4	211
5	224
6	310

Query – 15)

Create a function to return sum of Score for a given player_id.

```
CREATE OR REPLACE FUNCTION GetPlayerScoreSum(player_id INT) RETURNS INT
AS $$ DECLARE
    total_score INT;
BEGIN SELECT SUM(score) INTO total_score
    FROM Level_Details
    WHERE P_ID = player_id;
    RETURN total_score;
END;
$$ LANGUAGE plpgsql;
SELECT GetPlayerScoreSum(211) AS total_score;
```

Analysis -- returns the sum of scores for that player from the Level_Details table. The function is defined using PL/pgSQL language. After creating the function, it selects and displays the total score for a specific player ID (in this case, player ID 211) using the function.



```
1  -- Create the function to return the sum of scores for a given player_id
2  CREATE OR REPLACE FUNCTION GetPlayerScoreSum(player_id INT) RETURNS INT
3  AS $$
4  DECLARE
5      total_score INT;
6  BEGIN
7      SELECT SUM(score) INTO total_score
8      FROM Level_Details
9      WHERE P_ID = player_id;
10
11     RETURN total_score;
12 END;
13 $$ LANGUAGE plpgsql;
14
15 -- Call the function with a specific player_id to see the output
16 SELECT GetPlayerScoreSum(211) AS total_score;
```

Data Output | Messages | Notifications

	total_score integer
1	10940

Summary

- The project involved developing a database system for a gaming platform. Here's the key components and features:
- Database Schema: The project includes a well-structured relational database schema with tables such as `Player_Details`, `Level_Details`, and `Game_Data`, storing information about players, their game levels, and game statistics.
- Data Analysis Queries: Various SQL queries were implemented to perform data analysis tasks, such as calculating total scores, finding top scores, identifying players with specific characteristics, and computing cumulative statistics.
- Stored Procedures and Functions: PL/pgSQL stored procedures and functions were utilized to encapsulate complex SQL logic, improve code modularity, and enhance database performance. Functions like `GetPlayerScoreSum` were created to compute aggregated values based on input parameters.
- Window Functions: Window functions, such as `ROW_NUMBER()` and `SUM() OVER()`, were leveraged to perform advanced analytical operations like ranking scores, calculating cumulative sums, and retrieving data based on specific window partitions.
- Optimized Queries: Efforts were made to optimize SQL queries for efficiency and performance, ensuring that data retrieval and processing tasks are executed swiftly, even with large datasets.
- Overall, the project demonstrates proficiency in database design, SQL programming, and data analysis techniques, providing valuable insights into player behavior and game performance metrics.

THANK YOU