

Gaming Analysis with SQL

Internship Project by Anusiem Adaeze Comfort with Mentorness

BATCH: MIP-DA-06

Table of Contents

- ❖ About the project
- ❖ Dataset Description
- ❖ Outcomes
- ❖ New Learnings
- ❖ Analyze SQL problem statements
- ❖ Conclusion

About the project

In this project I worked with a dataset related to a game. The dataset includes two tables: 'Player Details' and 'Level Details'. Below is a brief description of the dataset

Player Details Table:

- ❖ 'P_ID': Player ID
- ❖ 'PName': Player Name
- ❖ 'L1_status': Level 1 Status
- ❖ 'L2_status': Level 2 Status
- ❖ 'L1_code': Systemgenerated Level 1 Code
- ❖ 'L2_code': Systemgenerated 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

Dataset Description

- ❖ Players play a game divided into 3-levels (L0,L1 and L2)
- ❖ Each level has 3 difficulty levels (Low,Medium,High)
- ❖ At each level,players have to kill the opponents using guns/physical fight
- ❖ Each level has multiple stages at each difficulty level.
- ❖ A player can only play L1 using its system generated L1_code.
- ❖ Only players who have played Level1 can possibly play Level2 using its system generated L2_code.
- ❖ By default a player can play L0.
- ❖ Each player can login to the game using a Dev_ID.
- ❖ Players can earn extra lives at each stage in a level.

Outcomes

- ❖ Gain strong analytical skills
- ❖ Enhance critical thinking abilities
- ❖ Enhance excellent communication skills
- ❖ Sharpen my ability to handle large and big data

New Learnings

- ❖ Stored Procedures are sets of SQL codes that can be created, stored and accessed often
- ❖ Ability to use window functions to perform a calculation on an aggregate value based on a set of rows and return multiple rows for each group
- ❖ How to use SQL Rank function and Row_Number function to find highest and lowest scores based on categories
- ❖ How to use nested queries in SQL

Analyze SQL problem statements

1. Extract `P_ID`, `Dev_ID`, `PName`, and `Difficulty_level` of all players at Level 0.

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** The tab is titled "game_analysis_project". It contains the following SQL code:

```
1 •    ALTER TABLE level_details2 DROP Row_Num;
2 •    ALTER TABLE player_details DROP Row_Num;
3 •    ALTER TABLE level_details2 RENAME COLUMN TimeStamp TO start_time;
4 -- Q1) Extract P_ID,Dev_ID,PName and Difficulty_level of all players
5 -- at level 0
6 •    SELECT
7     l.P_ID,
8     l.Dev_ID,
9     p.PName,
10    Difficulty
11   FROM level_details2 as l
12   LEFT JOIN player_details as p
13     USING(P_ID)
14   WHERE l.level = 0;
15
```

- Result Grid:** Below the editor, the results are displayed in a grid format. The columns are labeled P_ID, Dev_ID, PName, and Difficulty. The data is as follows:

	P_ID	Dev_ID	PName	Difficulty
▶	656	rf_013	sloppy-denim-wolfhound	Medium
	632	bd_013	dorky-heliotrope-barracuda	Difficult
	429	bd_013	flabby-firebrick-bee	Medium
	310	bd_015	gloppy-tomato-wasp	Difficult
	211	bd_017	breezy-indigo-starfish	Low
	300	zm_015	lanky-asparagus-gar	Difficult
	358	zm_017	skinny-grey-quetzal	Low
	358	zm_013	skinny-grey-quetzal	Medium
	641	rf_013	homey-alizarin-gar	Low
	641	rf_015	homey-alizarin-gar	Medium
	641	rf_013	homey-alizarin-gar	Difficult

- Status Bar:** At the bottom left, it says "Result 1".

2. Find `Level1_code` wise average `Kill_Count` where `lives_earned` is 2, and at least 3 stages are crossed.

The screenshot shows a MySQL Workbench interface with multiple tabs at the top: absenteeisms, triggers-storedprocedures-simpl, hrdata1, create-databases, demo 1*, and game_analysis_project (which is active). The query editor contains the following SQL code:

```
16      -- Q2) Find Level1_code wise Avg_Kill_Count where lives_earned is 2 and atleast
17      --      3 stages are crossed
18 •  SELECT
19     p.L1_code,
20     avg(l.Kill_Count) AS avg_kill_count,
21     lives_Earned,
22     Score
23   FROM level_details2 AS l
24   JOIN player_details AS p
25   USING(P_ID)
26   WHERE lives_Earned = 2 AND Stages_crossed >= 3
27   GROUP BY L1_Code;
28
29   -- Q3) Find the total number of stages crossed at each diffuculty level
30   -- where for Level2 with players use zm_series devices. Arrange the result
```

The results grid below the editor displays the following data:

L1_code	avg_kill_count	lives_Earned	Score
speed_blitz	19.3333	2	1750
war_zone	19.2857	2	3370
bulls_eye	22.2500	2	3470

3. Find the total number of stages crossed at each difficulty level for Level 2 with players using `zm_series` devices. Arrange the result in decreasing order of the total number of stages crossed.

The screenshot shows a MySQL Workbench interface. The top navigation bar includes tabs for 'absenteeisms', 'triggers-storedprocedures-simpl', 'hrdata1', 'create-databases', 'demo 1*', and 'game_analysis_project'. Below the tabs is a toolbar with various icons. The main area contains a SQL query:

```
28
29      -- Q3) Find the total number of stages crossed at each difficulty level
30      -- where for Level2 with players use zm_series devices. Arrange the result
31      -- in decsreasing order of total number of stages crossed.
32 • SELECT
33     sum(Stages_crossed) as 'Total_Stages_crossed',
34     Difficulty,
35     level,
36     DEV_ID
37   FROM level_details2
38   WHERE level = 2 AND Dev_ID LIKE 'zm%'
39   GROUP BY Difficulty
40   ORDER BY Total_Stages_crossed DESC;
41
42 -- Q4) Extract P_ID and the total number of unique dates for those players
```

The 'Result Grid' tab is selected, showing the following data:

	Total_Stages_crossed	Difficulty	level	DEV_ID
▶	46	Difficult	2	zm_017
	35	Medium	2	zm_015
	15	Low	2	zm_017

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

The screenshot shows a MySQL Workbench interface. The top navigation bar includes tabs for 'absenteeisms', 'triggers-storedprocedures-simpl', 'hrdata1', 'create-databases', 'demo 1*', and 'game_analysis_project'. Below the tabs is a toolbar with various icons. The main area contains a SQL query editor with the following code:

```
37     FROM level_details2
38     WHERE level = 2 AND Dev_ID LIKE 'zm%'
39     GROUP BY Difficulty
40     ORDER BY Total_Stages_crossed DESC;
41
42     -- Q4) Extract P_ID and the total number of unique dates for those players
43     -- who have played games on multiple days.
44 •   SELECT
45     DISTINCT P_ID,
46     count(DISTINCT DATE(start_time)) AS 'no_unique_dates'
47     FROM level_details2
48     GROUP BY P_ID
49     HAVING no_unique_dates > 1;
50
51     -- Q5) Find P_ID and level wise sum of kill_counts where kill_count
```

The code is numbered from 37 to 51. Lines 44 through 49 are highlighted with a blue selection box. The result grid below shows the output of the query:

P_ID	no_unique_dates
211	4
224	2
242	2
292	2
300	3
310	3
368	2
483	3
590	3
632	3

5. Find `P_ID` and levelwise sum of `kill_counts` where `kill_count` is greater than the average kill count for Medium difficulty.

```
absenteeisms triggers-storedprocedures-simpl hrdata1 create-databases demo 1* game_analysis_project × level_details2 player_details
49      HAVING no_unique_dates > 1;
50
51      -- Q5) Find P_ID and level wise sum of kill_counts where kill_count
52      -- is greater than avg kill count for the Medium difficulty.
53 •   SELECT
54      *,
55      SUM(Kill_Count) as total_kill_count,
56      AVG(Kill_Count) AS 'avg_kill_count'
57      FROM level_details2
58      GROUP BY P_ID, level
59      HAVING Kill_Count > avg_kill_count AND Difficulty = 'Medium';
60
61      -- Q6) Find Level and its corresponding Level code wise sum of lives earned
62      -- excluding level 0. Arrange in asecending order of level.
63 •   SELECT
<
Result Grid | Filter Rows: | Export: | Wrap Cell Content:


| P_ID | Dev_ID | start_time          | Stages_crossed | Level | Difficulty | Kill_Count | Headshots_Count | Score | Lives_Earned | total_kill_count | avg_kill_count |
|------|--------|---------------------|----------------|-------|------------|------------|-----------------|-------|--------------|------------------|----------------|
| 644  | zm_015 | 2022-10-11 14:05:08 | 3              | 1     | Medium     | 11         | 5               | 350   | 1            | 18               | 9.0000         |
| 483  | zm_017 | 2022-10-11 14:33:27 | 10             | 2     | Medium     | 50         | 43              | 5490  | 3            | 94               | 31.3333        |
| 368  | zm_015 | 2022-10-12 01:14:34 | 7              | 1     | Medium     | 20         | 18              | 2040  | 0            | 34               | 17.0000        |
| 683  | rf_015 | 2022-10-13 22:30:17 | 5              | 2     | Medium     | 25         | 18              | 2800  | 0            | 74               | 18.5000        |


```

6. Find 'Level' and its corresponding 'Level_code' wise sum of lives earned, excluding Level 0. Arrange in ascending order of level

The screenshot shows a MySQL Workbench interface. The top navigation bar has tabs for 'absenteeisms', 'triggers-storedprocedures-simpl', 'hrdata1', 'create-databases', 'demo 1*', and 'game_analysis_project'. Below the tabs is a toolbar with icons for file operations, search, and database management. The main area contains a SQL query editor with the following code:

```
61      -- Q6) Find Level and its corresponding Level code wise sum of lives earned
62      -- excluding level 0. Arrange in asecending order of level.
63 •   SELECT
64     l.level,
65     p.L1_code,
66     p.L2_code,
67     sum(l.Lives_Earned) as total_lives_earned
68     FROM level_details2 as l INNER JOIN player_details as p
69     USING (P_ID)
70     GROUP BY l.level
71     HAVING level != 0
72     ORDER BY level;
73
74      -- Q7) Find Top 3 score based on each dev_id and Rank them in increasing order
75      -- using Row_Number. Display difficulty as well.
```

Below the code editor is a 'Result Grid' section with the following table:

	level	L1_code	L2_code	total_lives_earned
▶	1	speed_blitz	cosmic_vision	23
	2	speed_blitz	cosmic_vision	51

7. Find the top 3 scores based on each 'Dev_ID' and rank them in increasing order using 'Row_Number'. Display the difficulty as well.

The screenshot shows a MySQL Workbench interface with several tabs at the top: absenteeisms, triggers-storedprocedures-simpl, hrdata1, create-databases, demo 1*, and game_analysis_project. The game_analysis_project tab is active. In the query editor, a SQL script is written to solve the problem:

```
70     GROUP BY l.level
71     HAVING level != 0
72     ORDER BY level;
73
74     -- Q7) Find Top 3 score based on each dev_id and Rank them in increasing order
75     -- using Row_Number. Display difficulty as well.
76 • WITH top3 AS
77     (SELECT Dev_ID, Score, Difficulty, ROW_NUMBER()
78      OVER(PARTITION BY Dev_ID ORDER BY Score DESC) AS rn FROM level_details2)
79     SELECT * FROM top3 WHERE rn <=3;
80
81     -- Q8) Find first_login datetime for each device id
82 • SELECT
83     distinct Dev_ID,
84     min((start_time)) as 'first_login_datetime'
```

The result grid below the editor displays the following data:

	Dev_ID	Score	Difficulty	rn
▶	bd_013	5300	Difficult	1
	bd_013	4570	Difficult	2
	bd_013	3370	Difficult	3
	bd_015	5300	Difficult	1
	bd_015	3200	Low	2
	bd_015	1950	Difficult	3
	bd_017	2400	Low	1
	bd_017	1750	Medium	2
	bd_017	390	Low	3
	rf_013	2970	Difficult	1
	rf_013	2780	Medium	2

8. Find the `first_login` datetime for each device ID.

The screenshot shows the MySQL Workbench interface with the following details:

- Query Editor:** Displays the SQL code for finding the first login datetime for each device ID. The code uses a subquery to find the top 3 rows and then groups by Dev_ID to get the minimum start_time as the first_login_datetime.
- Results Grid:** Shows the output of the query, which lists 10 device IDs and their corresponding first login datetimes.

```
absenteeisms triggers-storedprocedures-simpl hrdata1 create-databases demo 1* game_analysis_project
79     SELECT * FROM top3 WHERE rn <=3;
80
81     -- Q8) Find first_login datetime for each device id
82 •     SELECT
83         distinct Dev_ID,
84         min((start_time)) as 'first_login_datetime'
85     FROM level_details2
86     GROUP BY Dev_ID;
87
88     -- Q9) Find Top 5 score based on each difficulty level and Rank them in
89     -- increasing order using Rank. Display dev_id as well.
90 •     WITH score_point AS (
91         SELECT
92             Score,
93             Difficulty,
```

Dev_ID	first_login_datetime
zm_015	2022-10-11 14:05:08
rf_015	2022-10-11 19:34:25
bd_017	2022-10-12 07:30:18
rf_013	2022-10-11 05:20:40
bd_015	2022-10-11 18:45:55
rf_017	2022-10-11 09:28:56
bd_013	2022-10-11 02:23:45
zm_017	2022-10-11 14:33:27
zm_013	2022-10-11 13:00:22
wd_019	2022-10-12 23:19:17

9. Find the top 5 scores based on each difficulty level and rank them in increasing order using 'Rank'. Display 'Dev_ID' as well.

The screenshot shows a MySQL Workbench interface with a query editor and a result grid. The query editor contains the following SQL code:

```
88 -- Q9) Find Top 5 score based on each difficulty level and Rank them in
89 -- increasing order using Rank. Display dev_id as well.
90 WITH score_point AS (
91     SELECT Score,
92            Difficulty,
93            Dev_ID,
94            RANK() OVER (PARTITION BY Difficulty ORDER BY Score) AS score_rank
95     FROM level_details2
96 )
97     SELECT Score,
98            Difficulty,
99            Dev_ID,
100           score_rank
101        FROM score_point
102       WHERE score_rank <= 5;
```

The result grid displays the following data:

	Score	Difficulty	Dev_ID	score_rank
▶	100	Difficult	bd_013	1
	100	Difficult	zm_017	1
	100	Difficult	wd_019	1
	100	Difficult	bd_013	1
	235	Difficult	rf_013	5
	50	Low	zm_017	1
	70	Low	zm_017	2

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.

The screenshot shows a MySQL Workbench interface with a query editor and a result grid. The query editor contains a SQL script to find the first login device ID for each player. The result grid displays the output of the query.

```
game_analysis_project × absenteeisms triggers-storedprocedures-simpl hrdatal1 create-databases
106
107      -- Q10) Find the device ID that is first logged in(based on start_datetime)
108      -- for each player(p_id). Output should contain player id, device id and
109      -- first login datetime.
110 •   SELECT
111     P_ID,
112     Dev_ID,
113     min((start_time)) AS 'first_login_date'
114   FROM level_details2
115   GROUP BY P_ID;
116
117   -- Q11) For each player and date, how many kill_count played so far by the p
118   -- by the player until that date.
```

P_ID	Dev_ID	first_login_date
644	zm_015	2022-10-11 14:05:08
656	rf_013	2022-10-11 17:47:09
296	zm_017	2022-10-14 15:15:15
632	bd_013	2022-10-12 16:30:30
428	bd_015	2022-10-15 18:00:00
429	rf_017	2022-10-11 09:28:56
310	rf_017	2022-10-11 15:15:15
211	bd_017	2022-10-12 13:23:45
319	zm_017	2022-10-12 14:20:40
547	rf_013	2022-10-15 02:19:27
300	bd_013	2022-10-11 05:20:40
224	rf_017	2022-10-14 01:15:56
242	bd_013	2022-10-13 01:14:29

11. For each player and date, determine how many `kill_counts` were played by the player so far.
a) Using window functions

The screenshot shows a MySQL Workbench interface. The top navigation bar includes tabs for 'game_analysis_project', 'absenteeisms', 'triggers-storedprocedures-simpl', 'hrdata1', 'create-databases', 'demo 1*', 'level_details2', and 'player_details'. Below the tabs is a toolbar with various icons. The main area contains a SQL query editor with numbered lines of code. Lines 115 and 116 show a 'GROUP BY P_ID' clause. Lines 117 through 119 are comments explaining the goal: '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.' Line 119 indicates 'a) window function'. Lines 120 and 121 begin a 'SELECT' statement with columns 'P_ID' and 'Date'. Line 122 adds 'Date(start_time) as Date'. Line 123 includes a window function 'sum(Kill_Count) OVER (PARTITION BY P_ID ORDER BY start_time)' with an alias 'total_number_of_games'. Line 124 specifies the table 'FROM level_details2'. Lines 125 and 126 are comments for 'b) without window function'. Line 127 begins another 'SELECT' statement. Below the editor is a 'Result Grid' showing the query results. The grid has columns 'P_ID', 'Date', and 'total_number_of_games'. The data shows multiple rows for player 211 across dates from 2022-10-12 to 2022-10-15, with values increasing from 20 to 113. Other players like 224 and 242 are also listed with their respective game counts.

P_ID	Date	total_number_of_games
211	2022-10-12	20
211	2022-10-12	45
211	2022-10-13	75
211	2022-10-13	89
211	2022-10-14	98
211	2022-10-15	113
224	2022-10-14	20
224	2022-10-14	54
224	2022-10-15	84
224	2022-10-15	112
242	2022-10-13	21
242	2022-10-14	58
292	2022-10-12	21

11. For each player and date, determine how many `kill_counts` were played by the player so far.
b) Without window functions

The screenshot shows a MySQL Workbench interface with the following details:

- Query Editor:** The code is written in SQL. It includes a comment -- b) without window function and a SELECT statement that groups by P_ID and Date, summing Kill_Count.
- Result Grid:** The results are displayed in a table with columns P_ID, Date, and Total_Kill_Count. The data is as follows:

	P_ID	Date	Total_Kill_Count
▶	644	2022-10-11	18
	644	2022-10-12	24
	656	2022-10-15	15
	656	2022-10-13	19
	656	2022-10-14	3
	656	2022-10-11	18
	296	2022-10-14	11
	632	2022-10-12	73
	632	2022-10-13	27
	632	2022-10-14	30
	428	2022-10-15	5
	429	2022-10-11	99
	310	2022-10-11	20

12. Find the cumulative sum of stages crossed over `start_datetime` for each `P_ID`, excluding the most recent `start_datetime`.

The screenshot shows a MySQL Workbench interface with the following details:

- Connections:** game_analysis_project, absenteeisms, triggers-storedprocedures-simpl, hrdata1, create-databases, demo 1*, level_details2
- Toolbar:** Includes icons for file operations, search, and database management.
- Query Editor:** Displays the following SQL code:

```
133
134      -- Q12) Find the cumulative sum of an stages crossed over a start_datetime
135      -- for each player id but exclude the most recent start_datetime
136 •   SELECT
137     P_ID,
138     start_time as Datetime,
139     sum(Stages_crossed) OVER (PARTITION BY P_ID ORDER BY start_time) as cumulative_sum_of_stages_crossed
140   FROM level_details2;
141
142      -- Q13) Extract top 3 highest sum of score for each device id and the corresponding player_id
143 •   SELECT
144     DISTINCT Dev_ID,
145     P_ID,
```
- Result Grid:** Shows the output of the query:

P_ID	Datetime	cumulative_sum_of_stages_crossed
211	2022-10-12 13:23:45	4
211	2022-10-12 18:30:30	9
211	2022-10-13 05:36:15	14
211	2022-10-13 22:30:18	19
211	2022-10-14 08:56:24	26
211	2022-10-15 11:41:19	34
224	2022-10-14 01:15:56	7
224	2022-10-14 08:21:49	12
224	2022-10-15 05:30:28	22
224	2022-10-15 13:43:50	26
242	2022-10-13 01:14:29	6
242	2022-10-14 04:38:50	14
292	2022-10-12 04:29:45	4

13. Extract the top 3 highest sums of scores for each `Dev_ID` and the corresponding `P_ID`.

The screenshot shows a MySQL Workbench interface. The query editor tab is titled 'game_analysis_project' and contains the following SQL code:

```
142 -- Q13) Extract top 3 highest sum of score for each device id and the corresponding player_id
143 • SELECT
144     DISTINCT Dev_ID,
145     P_ID,
146     sum(Score) as highest_score
147     FROM level_details2
148     GROUP BY Dev_ID
149     ORDER BY highest_score DESC
150     LIMIT 3;
151
152 -- Q14) Find players who scored more than 50% of the avg score scored by sum of
153 -- scores for each player_id
154 • SELECT
```

The results grid below the editor shows the output of the query:

	Dev_ID	P_ID	highest_score
▶	zm_017	296	34460
	bd_013	656	27110
	zm_015	644	21690

14. Find players who scored more than 50% of the average score, scored by the sum of scores for each 'P_ID'.

The screenshot shows a MySQL Workbench interface. The query editor tab is titled 'game_analysis_project'. The code is as follows:

```
-- Q14) Find players who scored more than 50% of the avg score scored by sum of
-- scores for each player_id
SELECT P_ID,
       sum(Score) as total_score
  FROM level_details2
 GROUP BY P_ID
 HAVING total_score >
       (
           SELECT AVG(Score)*0.5
             FROM level_details2
       );
```

The results grid shows the following data:

P_ID	total_score
644	2250
656	4820
296	1140
632	10750
429	13220
310	13810
211	10940
547	3450
300	4860
224	16310
242	6310
292	2560
590	8000

15. Create a stored procedure to find the top `n` `headshots_count` based on each `Dev_ID` and rank them in increasing order using `Row_Number`. Display the difficulty as well.

The screenshot shows the MySQL Workbench interface with the 'game_analysis_project' database selected. The code editor contains the following SQL script:

```
164
165    -- Q15) Create a stored procedure to find top n headshots_count based on each dev_id and Rank them in increasing order
166    -- using Row_Number. Display difficulty as well.
167    DELIMITER $$

168 • CREATE PROCEDURE top_n_headshots_count(IN parameter_rn INT)
169 BEGIN
170     WITH top_N AS
171     (SELECT Dev_ID, Score, Difficulty, ROW_NUMBER()
172      OVER(PARTITION BY Dev_ID ORDER BY Score DESC) AS rn FROM level_details2)
173     SELECT * FROM top_N WHERE rn <= parameter_rn;
174 END $$

175 DELIMITER ;
176 • -- CALLING THE PROCEDURE TO FIND TOP N HEAD_SHOTS
177 CALL top_n_headshots_count(5);
```

The result grid below the code editor displays the following data:

	Dev_ID	Score	Difficulty	rn
▶	bd_013	5300	Difficult	1
	bd_013	4570	Difficult	2
	bd_013	3370	Difficult	3
	bd_013	3200	Difficult	4
	bd_013	2840	Low	5
	bd_015	5300	Difficult	1
	bd_015	3200	Low	2
	bd_015	1950	Difficult	3
	bd_015	1450	Low	4
	bd_015	1300	Difficult	5
	bd_017	2400	Low	1

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

The screenshot shows a MySQL Workbench interface with a query editor and a result grid.

Query Editor:

```
game_analysis_project    absenteeisms    triggers-storedprocedures-simpl    hrdata1    create-databases
172     OVER(PARTITION BY Dev_ID ORDER BY Score DESC) AS rn FROM level_details2)
173     SELECT * FROM top_N WHERE rn <= parameter_rn;
174 END $$*
175 DELIMITER ;
176 -- CALLING THE PROCEDURE TO FIND TOP N HEAD_SHOTS
177 CALL top_n_headshots_count(5);
178 -- Q16) Create a function to return sum of Score for a given player_id.
179 *
SELECT
  DISTINCT P_ID,
  sum(Score) AS Total_Score
FROM player_details AS p LEFT JOIN level_details2 AS l
  USING(P_ID)
GROUP BY P_ID
ORDER BY Total_Score DESC
```

Result Grid:

P_ID	Total_Score
683	18140
483	17230
224	16310
310	13810
429	13220
211	10940
632	10750
663	10750
368	8710
590	8000
242	6310

Conclusion

SQL is needed to seamlessly navigate through simple and complex datasets, it is used to efficiently execute queries, edit database tables and extract meaningful insights from a dataset to make decisions.

In this game analysis project, using SQL I was able to solve some statement problems, query necessary data to identify pattern and behaviors of the game, to predict gaming outcomes and scores, identify excellent players and most suitable devices they used and many others.