Game Decoder SQl Presentation

There are 2 Tables which are player_details and level_details.
Here is the description of both tables:

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

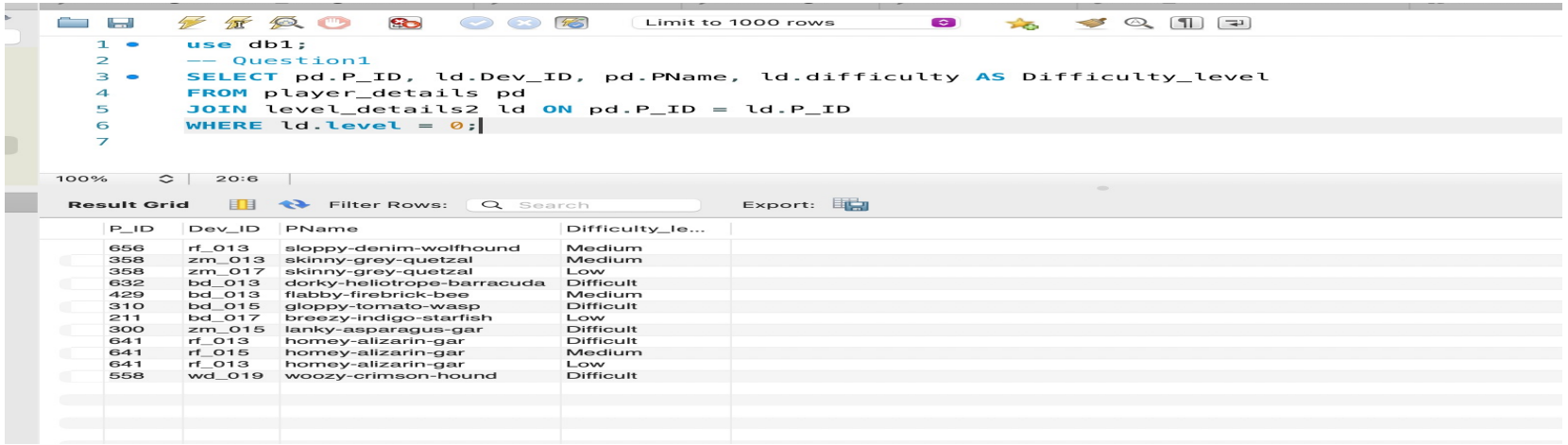
`score`: Player Score

`lives_earned`: Extra Lives Earned

Question 1:

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

- # Sql_Statement and Response:



```
Limit to 1000 rows
1  use db1;
2  -- Question1
3  SELECT pd.P_ID, ld.Dev_ID, pd.PName, ld.difficulty AS Difficulty_level
4  FROM player_details pd
5  JOIN level_details2 ld ON pd.P_ID = ld.P_ID
6  WHERE ld.level = 0;
7
```

100% 20:6

Result Grid | Filter Rows: Search | Export:

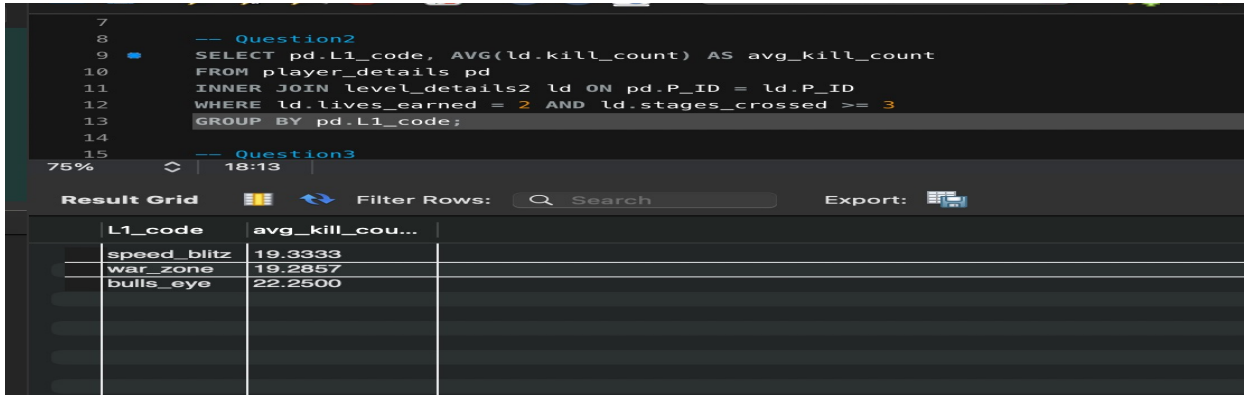
P_ID	Dev_ID	PName	Difficulty_le...
656	rf_013	sloppy-denim-wolfhound	Medium
358	zm_013	skinny-grey-quetzal	Medium
358	zm_017	skinny-grey-quetzal	Low
632	bd_013	dorky-heliotrope-barracuda	Difficult
429	bd_013	flabby-firebrick-bee	Medium
310	bd_015	gloppy-tomato-wasp	Difficult
211	bd_017	breezy-indigo-starfish	Low
300	zm_015	lanky-asparagus-gar	Difficult
641	rf_013	homey-alizarin-gar	Difficult
641	rf_015	homey-alizarin-gar	Medium
641	rf_013	homey-alizarin-gar	Low
558	wd_019	woozy-crimson-hound	Difficult

- The Explanation:

1. SELECT pd.P_ID, ld.Dev_ID, pd.PName, ld.difficulty AS Difficulty_level: This part of the query selects the columns we want to retrieve. We select P_ID from the player_details table (pd alias), Dev_ID from the level_details2 table (ld alias), PName from the player_details table, and difficulty from the level_details2 table. We also use an alias Difficulty_level for the difficulty column.
2. FROM player_details pd: This specifies the table from which we are selecting data and assigns it the alias pd.
3. JOIN level_details2 ld ON pd.P_ID = ld.P_ID: This part of the query performs a join operation between the player_details table (pd) and the level_details2 table (ld) based on the common column P_ID. It joins the rows from both tables where the P_ID values match.
4. WHERE ld.level = 0: This part of the query filters the joined rows to include only those where the level column in the level_details2 table has a value of 0.

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

- # Sql_Statement and Response:



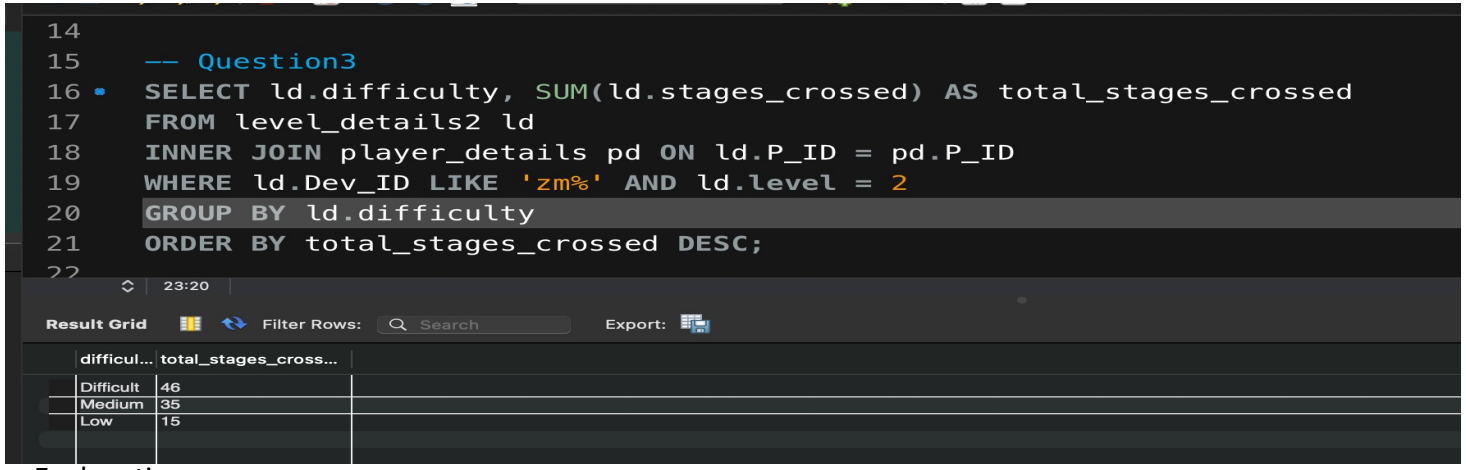
- The Explanation:

1. SELECT pd.L1_code, AVG(ld.kill_count) AS avg_kill_count: This part of the query selects the L1_code column from the player_details table (pd) and calculates the average kill_count from the level_details2 table (ld). The AS avg_kill_count alias is used to rename the calculated average.
2. FROM player_details pd: This specifies the table from which we are selecting data and assigns it the alias pd.
3. INNER JOIN level_details2 ld ON pd.P_ID = ld.P_ID: This part of the query performs an inner join operation between the player_details table (pd) and the level_details2 table (ld) based on the common column P_ID. It joins the rows from both tables where the P_ID values match.
4. WHERE ld.lives_earned = 2 AND ld.stages_crossed >= 3: This part of the query filters the joined rows to include only those where the lives_earned column in the level_details2 table has a value of 2 and the stages_crossed column has a value greater than or equal to 3.
5. GROUP BY pd.L1_code: This part of the query groups the results by the L1_code column, so that the average kill_count is calculated for each unique L1_code value.

Question 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.

- Sql_Statement and Response:



```
14
15    -- Question3
16    SELECT ld.difficulty, SUM(ld.stages_crossed) AS total_stages_crossed
17    FROM level_details2 ld
18    INNER JOIN player_details pd ON ld.P_ID = pd.P_ID
19    WHERE ld.Dev_ID LIKE 'zm%' AND ld.level = 2
20    GROUP BY ld.difficulty
21    ORDER BY total_stages_crossed DESC;
22
```

23:20

Result Grid | Filter Rows: Search | Export:

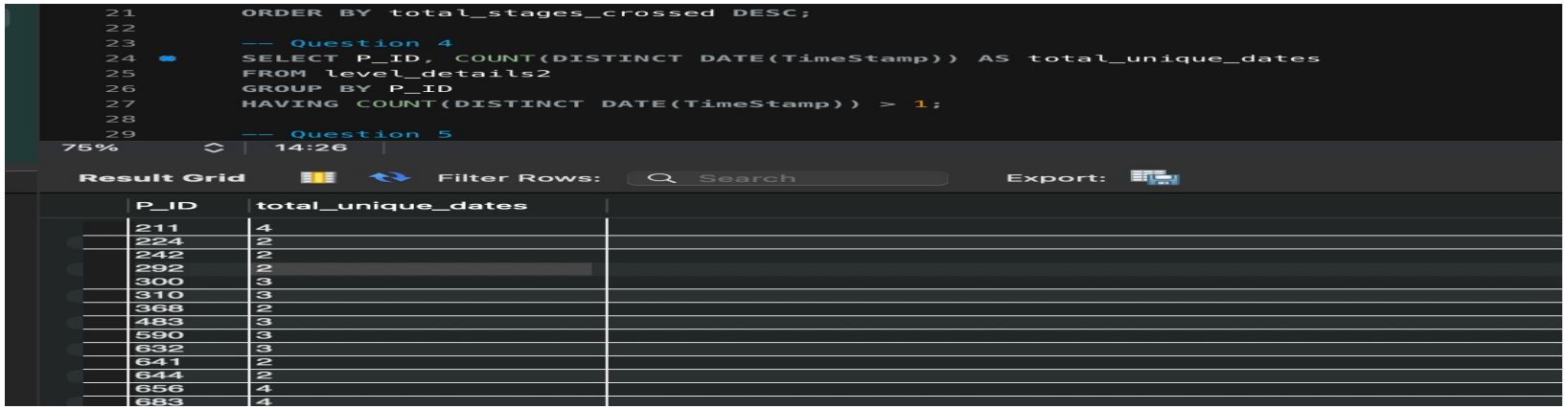
difficul...	total_stages_cross...
Difficult	46
Medium	35
Low	15

- The Explanation:

1. SELECT ld.difficulty, SUM(ld.stages_crossed) AS total_stages_crossed: This part of the query selects the difficulty column from the level_details2 table (ld) and calculates the sum of the stages_crossed column for each difficulty level. The AS total_stages_crossed alias is used to rename the calculated sum.
2. FROM level_details2 ld: This specifies the table from which we are selecting data and assigns it the alias ld.
3. INNER JOIN player_details pd ON ld.P_ID = pd.P_ID: This part of the query performs an inner join operation between the level_details2 table (ld) and the player_details table (pd) based on the common column P_ID. It joins the rows from both tables where the P_ID values match.
4. WHERE ld.Dev_ID LIKE 'zm%' AND ld.level = 2: This part of the query filters the joined rows to include only those where the Dev_ID column in the level_details2 table starts with 'zm' and the level column has a value of 2.
5. GROUP BY ld.difficulty: This part of the query groups the results by the difficulty column, so that the sum of stages_crossed is calculated for each unique difficulty level.
6. ORDER BY total_stages_crossed DESC: This part of the query orders the results by the total number of stages crossed (total_stages_crossed) in descending order.

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

- Sql_Statement and Response:



- The Explanation:

1. SELECT P_ID, COUNT(DISTINCT DATE(TimeStamp)) AS total_unique_dates: This part of the query selects the P_ID column from the level_details2 table and calculates the count of distinct dates extracted from the TimeStamp column. The AS total_unique_dates alias is used to rename the calculated count.
2. FROM level_details2: This specifies the table from which we are selecting data.
3. GROUP BY P_ID: This part of the query groups the results by the P_ID column, so that the count of unique dates is calculated for each unique player.
4. HAVING COUNT(DISTINCT DATE(TimeStamp)) > 1: This part of the query filters the grouped results to include only those players who have played games on multiple days, specifically those where the count of distinct dates is greater than 1.

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

- Sql_Statement and Response:

```
-- Question 5
WITH MediumAvg AS (
    SELECT AVG(kill_count) AS avg_kill_count
    FROM Level_Details2
    WHERE difficulty = 'Medium'
)
SELECT ld.P_ID, ld.level, SUM(ld.kill_count) AS sum_kill_count
FROM Level_Details2 ld
JOIN MediumAvg ma ON ld.difficulty = 'Medium' AND ld.kill_count > ma.avg_kill_count
GROUP BY ld.P_ID,ld.level;

-- Question 6
```

75% | 14:29

Result Grid | Filter Rows: Search | Export:

P_ID	level	sum_kill_cou...
644	2	24
632	1	28
632	2	23
211	1	30
300	1	23
224	1	20
483	1	40
483	2	50
368	1	20
663	1	28
663	2	23
683	2	25

- The Explanation:

1. WITH MediumAvg AS (...): This part of the query defines a Common Table Expression (CTE) named MediumAvg, which calculates the average kill_count for the difficulty level 'Medium’

2. SELECT AVG(kill_count) AS avg_kill_count FROM Level_Details2 WHERE difficulty = 'Medium': This subquery calculates the average kill_count for the difficulty level 'Medium’

3. SELECT ld.P_ID, ld.level, SUM(ld.kill_count) AS sum_kill_count FROM Level_Details2 ld: This part of the query selects the P_ID, level, and calculates the sum of kill_count from the Level_Details2 table.

4. JOIN MediumAvg ma ON ld.difficulty = 'Medium' AND ld.kill_count > ma.avg_kill_count: This part of the query joins the MediumAvg CTE with the Level_Details2 table based on the condition that the difficulty is 'Medium' and the kill_count is greater than the average kill_count for the 'Medium' difficulty level.

5. GROUP BY ld.P_ID, ld.level: This groups the results by P_ID and level.

Question 6:

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

- Sql_Statement and Response:

```
GROUP BY ld.P_ID,ld.level;

-- Question 6
SELECT ld.level, pd.L1_code AS Level1_code, pd.L2_code AS Level2_code, SUM(ld.lives_earned) AS total_lives_earned
FROM Level_Details2 ld
JOIN Player_Details pd ON ld.P_ID = pd.P_ID
WHERE ld.level > 0
GROUP BY ld.level, pd.L1_code, pd.L2_code
ORDER BY ld.level ASC;
```

14:45

Result Grid | Filter Rows: Search | Export:

level	Level1_code	Level2_code	total_lives_earn...
1	bulls_eye		3
1	bulls_eye	cosmic_vision	1
1	bulls_eye	resurgence	1
1	leap_of_faith		0
1	speed_blitz		0
1	speed_blitz	cosmic_vision	4
1	speed_blitz	splippery_slope	3
1	war_zone		4
1	war_zone	resurgence	0
1	war_zone	splippery_slope	7
2	bulls_eye	cosmic_vision	6
2	bulls_eye	resurgence	8
2	speed_blitz	cosmic_vision	6
2	speed_blitz	splippery_slope	14

Result 26

The Explanation:

1.SELECT:

1. ld.level: Selects the level column from the Level_Details2 table.
2. pd.L1_code AS Level1_code: Selects the L1_code column from the Player_Details table and aliases it as Level1_code.
3. pd.L2_code AS Level2_code: Selects the L2_code column from the Player_Details table and aliases it as Level2_code.
4. SUM(ld.lives_earned) AS total_lives_earned: Calculates the sum of the lives_earned column from the Level_Details2 table and aliases it as total_lives_earned.

2.FROM:

1. Specifies the tables being queried, Level_Details2 and Player_Details.
2. Joins these tables based on the P_ID column in Level_Details2 and Player_Details.

1. WHERE:

1. Filters the rows where the level in the Level_Details2 table is greater than 0.

2. GROUP BY:

1. Groups the result set by ld.level, pd.L1_code, and pd.L2_code. This ensures that the aggregation function (SUM) is applied to distinct combinations of these columns.

3. ORDER BY:

1. Orders the result set by ld.level in ascending order

Question 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.

- ## Sql_Statement and Response:

```
-- Question 7
WITH RankedScores AS (
    SELECT ld.P_ID, ld.Dev_ID, ld.Level, ld.score, ld.Difficulty,
        ROW_NUMBER() OVER (PARTITION BY ld.Dev_ID ORDER BY ld.score DESC) AS score_rank
    FROM Level_Details2 ld
)
SELECT rs.Dev_ID, rs.score, rs.Difficulty
FROM RankedScores rs
WHERE rs.score_rank <= 3;

-- Question 8
```

75% 12:55

Result Grid | Filter Rows: Search | Export:

Dev_ID	score	Difficul...
bd_013	5300	Difficult
bd_013	4570	Difficult
bd_013	3370	Difficult
bd_015	5300	Difficult
bd_015	3200	Low
bd_015	1950	Difficult
bd_017	2400	Low
bd_017	1750	Medium
bd_017	390	Low
rf_013	2970	Difficult
rf_013	2700	Medium
rf_013	2300	Medium
rf_015	3950	Difficult
rf_015	2800	Medium
rf_015	900	Medium

Result 27

- The Explanation:

1. Common Table Expression (CTE) - RankedScores:

1. The Level_Details2 table is aliased as ld.
2. The query selects P_ID, Dev_ID, Level, score, and Difficulty from the Level_Details2 table.
3. The ROW_NUMBER() window function is used to assign a rank to each row within each partition defined by Dev_ID, ordered by score in descending order.

2.SELECT:

1. Selects Dev_ID, score, and Difficulty from the RankedScores CTE.

3. WHERE:

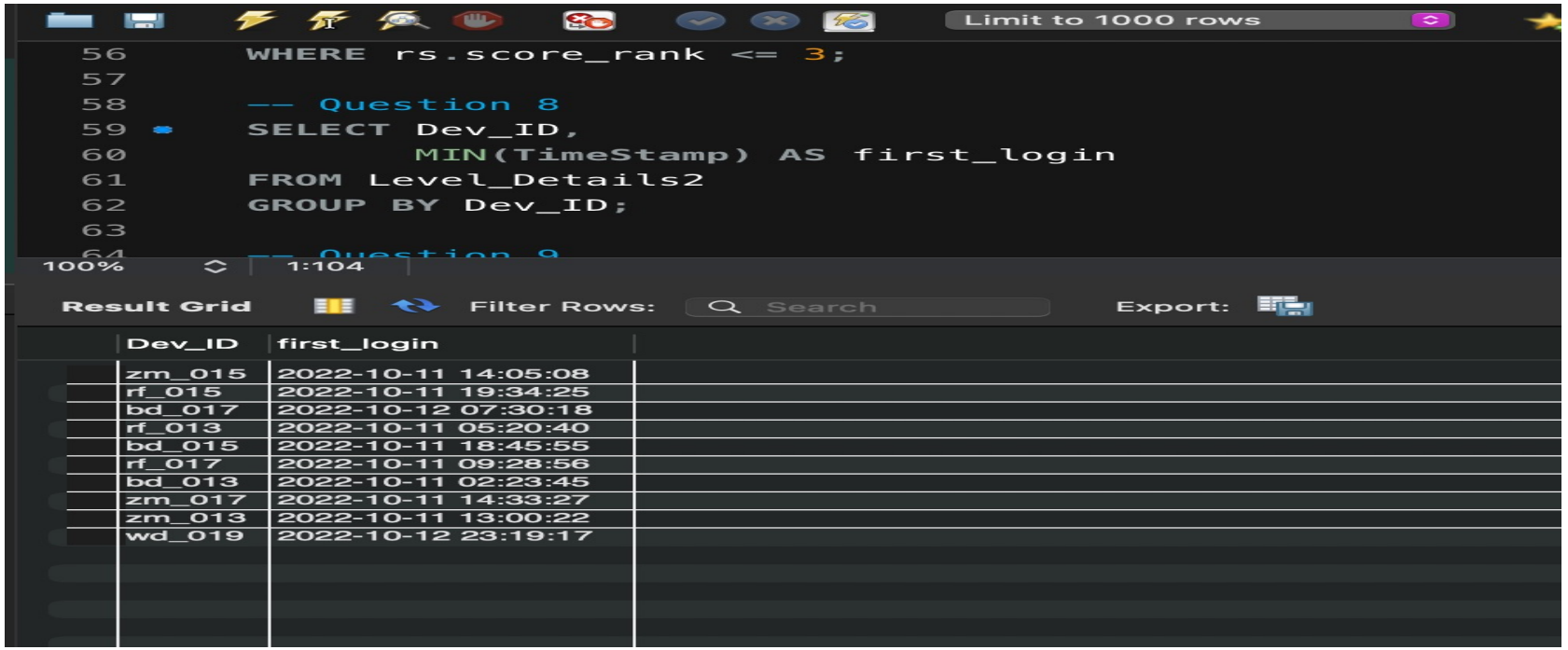
1. Filters the result set to include only rows where the score_rank is less than or equal to 3, effectively selecting the top 3 scores for each Dev_ID.

- This query efficiently retrieves the top 3 scores for each Dev_ID, along with their corresponding Difficulty levels.

Question 8:

Find the `first_login` datetime for each device ID.

- Sql_Statement and Response:



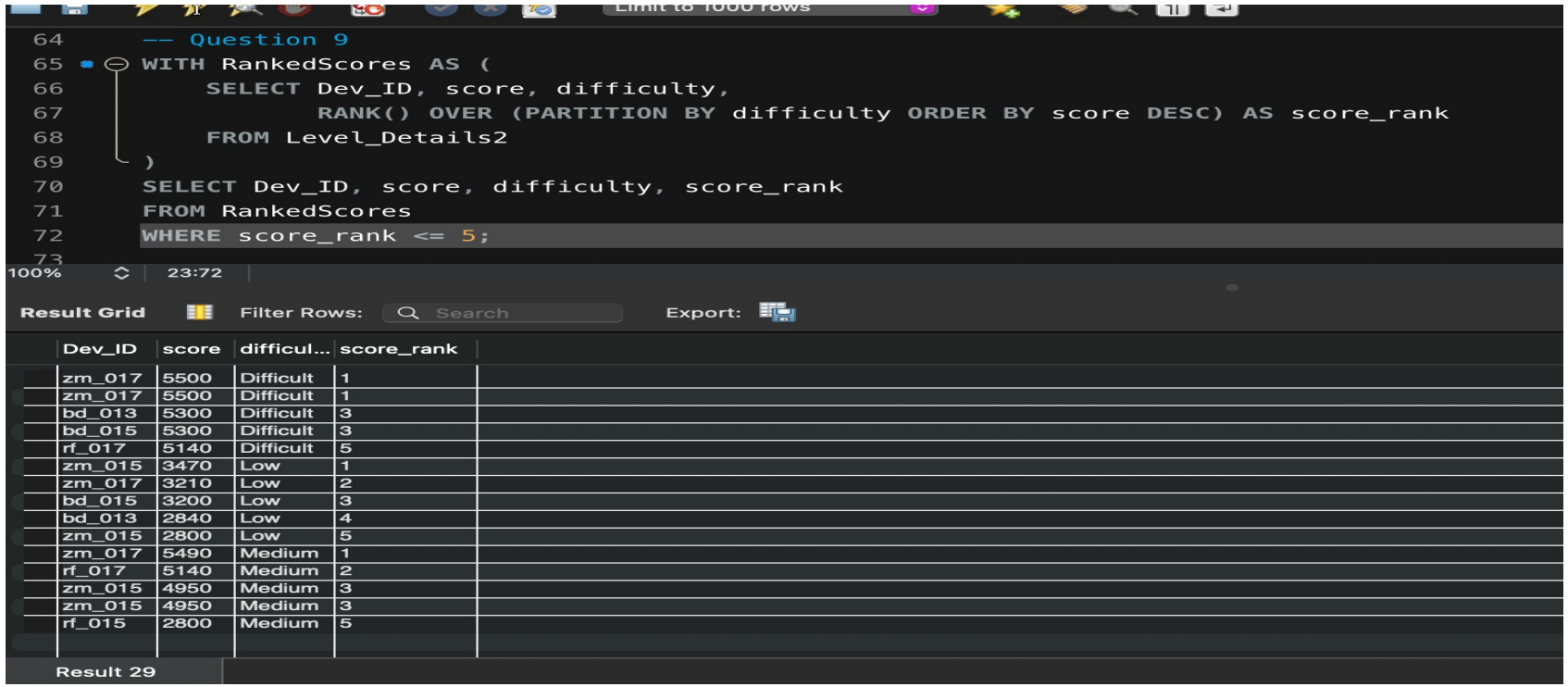
- The Explanation:

This query effectively finds the earliest login timestamp for each device ID recorded in the Level_Details2 table.

Question 9:

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

- Sql_Statement and Response:



```sql
-- Question 9
WITH RankedScores AS (
    SELECT Dev_ID, score, difficulty,
           RANK() OVER (PARTITION BY difficulty ORDER BY score DESC) AS score_rank
    FROM Level_Details2
)
SELECT Dev_ID, score, difficulty, score_rank
FROM RankedScores
WHERE score_rank <= 5;
```

100% 23:72

Result Grid Filter Rows: Search Export:

Dev_ID	score	difficul...	score_rank
zm_017	5500	Difficult	1
zm_017	5500	Difficult	1
bd_013	5300	Difficult	3
bd_015	5300	Difficult	3
rf_017	5140	Difficult	5
zm_015	3470	Low	1
zm_017	3210	Low	2
bd_015	3200	Low	3
bd_013	2840	Low	4
zm_015	2800	Low	5
zm_017	5490	Medium	1
rf_017	5140	Medium	2
zm_015	4950	Medium	3
zm_015	4950	Medium	3
rf_015	2800	Medium	5

Result 29

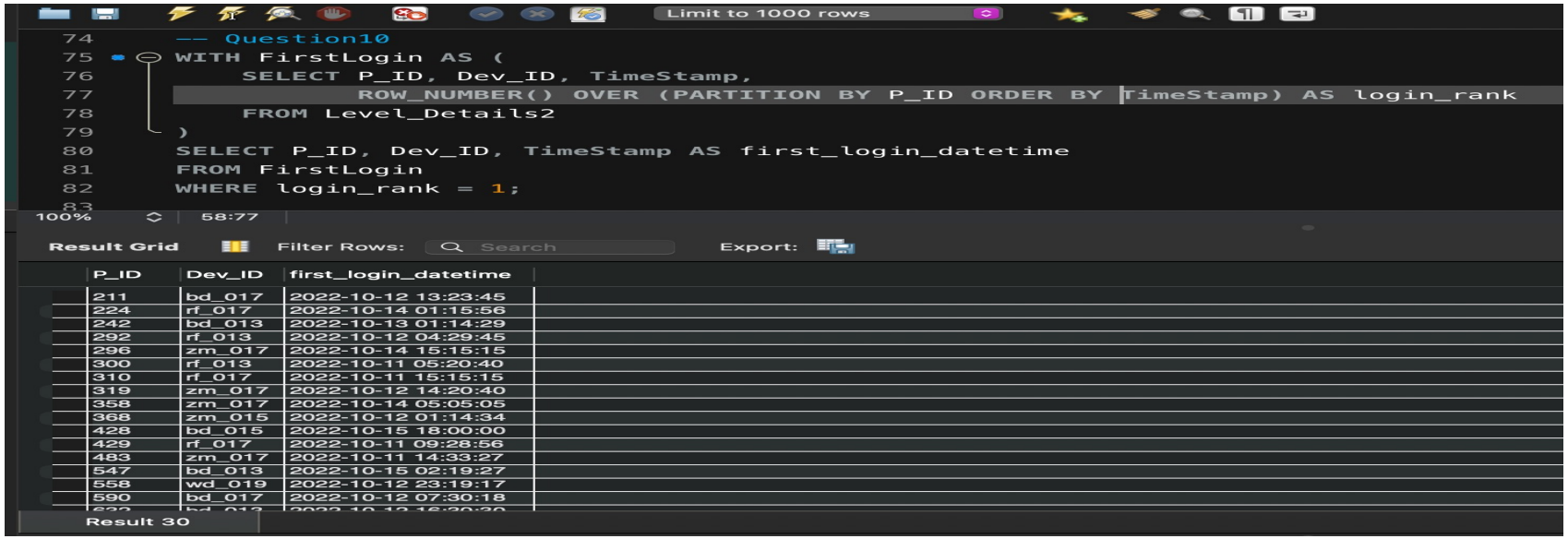
- The Explanation:

- **WITH RankedScores**: This defines a Common Table Expression (CTE) named RankedScores. It computes the rank of each score within its difficulty level using the RANK() function.
  - Dev_ID: The device ID associated with the score.
  - score: The score achieved.
  - difficulty: The difficulty level of the game.
  - RANK() OVER (PARTITION BY difficulty ORDER BY score DESC) AS score_rank: Assigns a rank to each score within its difficulty level, with higher scores receiving lower ranks.
- **SELECT**: Specifies the columns to be selected in the final result.
  - Dev_ID, score, difficulty: Selects the device ID, score, and difficulty level from the RankedScores CTE.
  - score_rank: Includes the score rank computed in the CTE.
- **FROM RankedScores**: Specifies the source of data for the SELECT statement, which is the RankedScores CTE.
- **WHERE score_rank <= 5**: Filters the results to include only scores with ranks up to and including 5. This ensures that only the top 5 scores for each difficulty level are returned.
- This query effectively retrieves the top 5 scores for each difficulty level, along with their corresponding ranks, from the Level_Details2 table.
-

Question 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.

- Sql_Statement and Response:



```
74  -- Question10
75  WITH FirstLogin AS (
76      SELECT P_ID, Dev_ID, TimeStamp,
77             ROW_NUMBER() OVER (PARTITION BY P_ID ORDER BY TimeStamp) AS login_rank
78      FROM Level_Details2
79  )
80  SELECT P_ID, Dev_ID, TimeStamp AS first_login_datetime
81  FROM FirstLogin
82  WHERE login_rank = 1;
83
```

Limit to 1000 rows

100% 58:77

Result Grid Filter Rows: Search Export:

P_ID	Dev_ID	first_login_datetime
211	bd_017	2022-10-12 13:23:45
224	rf_017	2022-10-14 01:15:56
242	bd_013	2022-10-13 01:14:29
292	rf_013	2022-10-12 04:29:45
296	zm_017	2022-10-14 15:15:15
300	rf_013	2022-10-11 05:20:40
310	rf_017	2022-10-11 15:15:15
319	zm_017	2022-10-12 14:20:40
358	zm_017	2022-10-14 05:05:05
368	zm_015	2022-10-12 01:14:34
428	bd_015	2022-10-15 18:00:00
429	rf_017	2022-10-11 09:28:56
483	zm_017	2022-10-11 14:33:27
547	bd_013	2022-10-15 02:19:27
558	wd_019	2022-10-12 23:19:17
590	bd_017	2022-10-12 07:30:18

Result 30

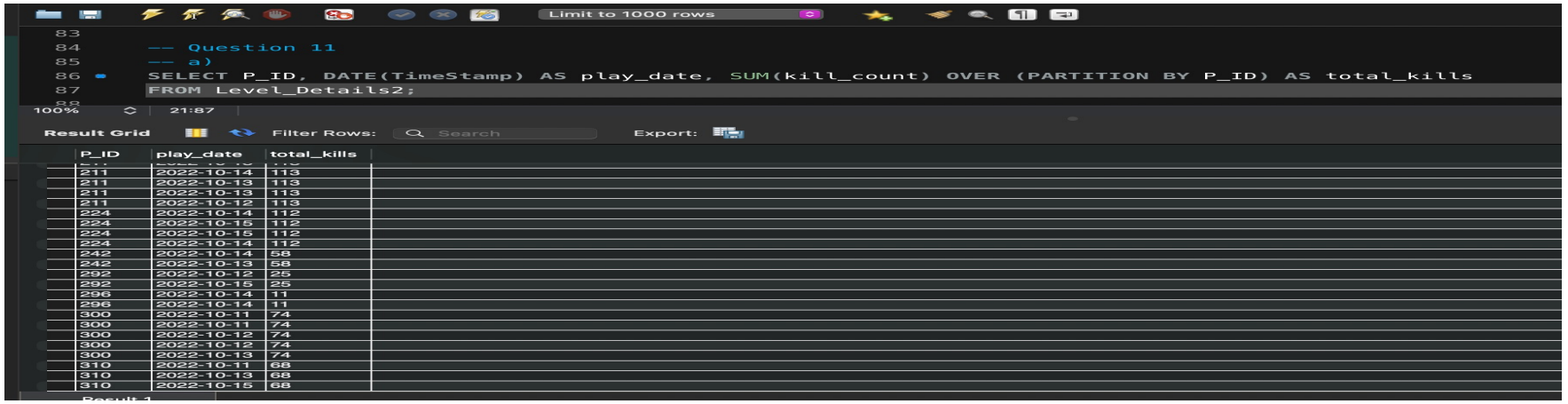
- The Explanation:
- **WITH FirstLogin**: This defines a Common Table Expression (CTE) named FirstLogin. It selects the player ID (P_ID), device ID (Dev_ID), and login timestamp (TimeStamp) from the Level_Details2 table. It also assigns a row number to each record within each player ID partition based on the login timestamp, ordered in ascending order.
    - ROW_NUMBER() OVER (PARTITION BY P_ID ORDER BY TimeStamp) AS login_rank: Assigns a row number to each record within each player ID partition, ordered by the login timestamp. This row number indicates the sequence of logins for each player.
- **SELECT**: Specifies the columns to be selected in the final result.
    - P_ID, Dev_ID, TimeStamp AS first_login_datetime: Selects the player ID, device ID, and login timestamp from the FirstLogin CTE. The login timestamp is renamed as first_login_datetime.
- **FROM FirstLogin**: Specifies the source of data for the SELECT statement, which is the FirstLogin CTE.
- **WHERE login_rank = 1**: Filters the results to include only records where the login rank is 1. This ensures that only the first login timestamp for each player ID is returned.
- This query effectively retrieves the first login datetime for each device ID and player ID from the Level_Details2 table.

Question 11:

11. For each player and date, determine how many `kill_counts` were played by the player so far.

a) Using window functions

- Sql_Statement and Response:



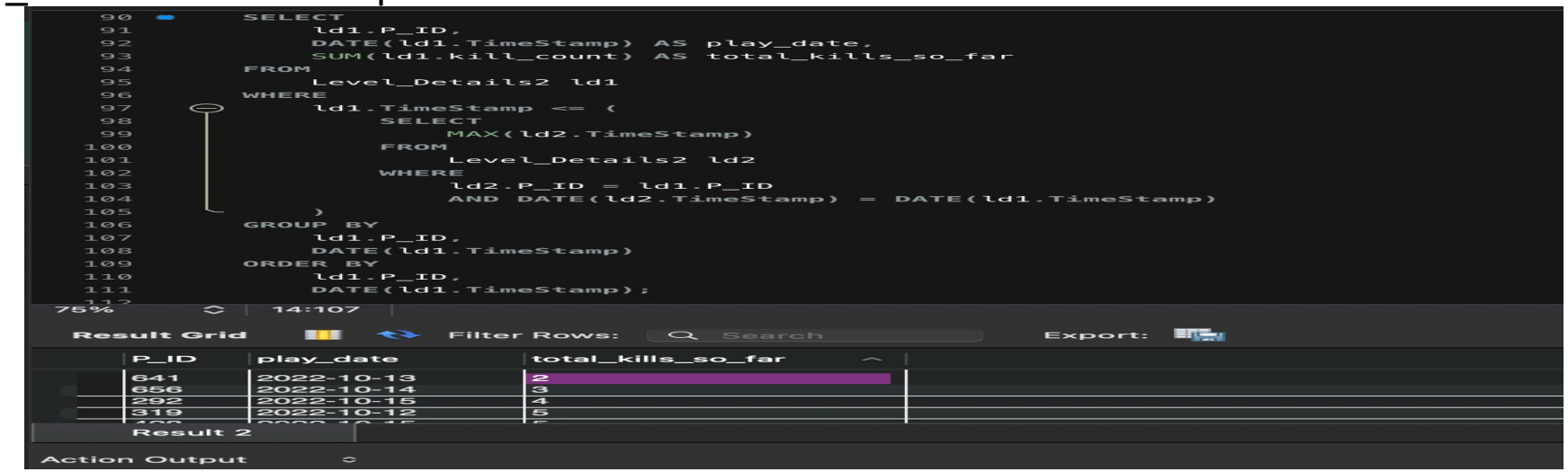
- The Explanation:
- **SELECT**: Specifies the columns to be selected in the final result.
  - P_ID: Selects the player ID.
  - DATE(TimeStamp) AS play_date: Selects the date component of the timestamp, represented as play_date.
  - SUM(kill_count) OVER (PARTITION BY P_ID ORDER BY TimeStamp) AS total_kills: Calculates the cumulative sum of kill counts for each player (P_ID) ordered by the timestamp (TimeStamp). The SUM function is used in combination with the OVER clause to perform a windowed aggregation. The PARTITION BY P_ID clause divides the data into partitions based on the player ID, and the ORDER BY TimeStamp clause specifies the order within each partition.
- This query provides a result set containing the player ID (P_ID), the date of play (play_date), and the cumulative sum of kill counts up to each play date (total_kills).

Question 11:

11. For each player and date, determine how many `kill_counts` were played by the player so far.

b) Without window functions

- Sql_Statement and Response:



```
SELECT
    ld1.P_ID,
    DATE(ld1.TimeStamp) AS play_date,
    SUM(ld1.kill_count) AS total_kills_so_far
FROM
    Level_Details2 ld1
WHERE
    ld1.TimeStamp <= (
        SELECT
            MAX(ld2.TimeStamp)
        FROM
            Level_Details2 ld2
        WHERE
            ld2.P_ID = ld1.P_ID
            AND DATE(ld2.TimeStamp) = DATE(ld1.TimeStamp)
    )
GROUP BY
    ld1.P_ID,
    DATE(ld1.TimeStamp)
ORDER BY
    ld1.P_ID,
    DATE(ld1.TimeStamp);
```

75% | 14:107

Result Grid | Filter Rows: Search | Export:

P_ID	play_date	total_kills_so_far
641	2022-10-13	2
656	2022-10-14	3
292	2022-10-15	4
319	2022-10-12	5

Result 2

Action Output

The Explanation:

- **SELECT**: Specifies the columns to be selected in the final result.
  - ld1.P_ID: Selects the player ID.
  - DATE(ld1.TimeStamp) AS play_date: Selects the date component of the timestamp, represented as play_date.
  - SUM(ld1.kill_count) AS total_kills_so_far: Calculates the total sum of kill counts played by the player up to the current date. This is achieved by summing the kill_count column.
- **FROM**: Specifies the source table for the query (Level_Details2).
- **WHERE**: Filters the rows based on the condition that the timestamp (ld1.TimeStamp) should be less than or equal to the maximum timestamp for the same player (P_ID) and date (play_date).
- **GROUP BY**: Groups the results by player ID and play date.
- **ORDER BY**: Orders the results by player ID and play date.

Question 12:

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

- Sql_Statement and Response:

```sql
-- Question 12
SELECT P_ID,
       TimeStamp,
       stages_crossed,
       SUM(stages_crossed) OVER (PARTITION BY P_ID ORDER BY TimeStamp ASC ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING) AS cumulative_stages_crossed
FROM Level_Details2
WHERE TimeStamp NOT IN (
    SELECT MAX(TimeStamp)
    FROM Level_Details2 ld2
    WHERE ld2.P_ID = Level_Details2.P_ID
);
```

75% 25:121

Result Grid | Filter Rows: Search | Export:

P_ID	play_date	total_kills_so_far
211	2022-10-12	45
211	2022-10-13	44
211	2022-10-14	9
211	2022-10-15	15
224	2022-10-14	54
224	2022-10-15	58
242	2022-10-13	21
242	2022-10-14	37
292	2022-10-12	21
292	2022-10-15	4
296	2022-10-14	11
300	2022-10-11	48
300	2022-10-12	18
300	2022-10-13	8
310	2022-10-11	20
310	2022-10-13	34
310	2022-10-15	14

Result 3

- The Explanation:
- **SELECT**: Specifies the columns to be selected in the final result.
  - P_ID: Selects the player ID.
  - TimeStamp: Selects the timestamp.
  - stages_crossed: Selects the number of stages crossed.
  - SUM(stages_crossed) OVER (...): Uses a window function to calculate the cumulative sum of stages crossed for each P_ID, excluding the current row (ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING). The PARTITION BY P_ID ensures that the sum is calculated separately for each player.
- **FROM**: Specifies the source table for the query (Level_Details2).
- **WHERE**: Filters the rows based on the condition that the TimeStamp is not the maximum TimeStamp for the corresponding P_ID. This ensures that the most recent TimeStamp is excluded from the calculation.

Question 13:

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

- Sql_Statement and Response:

```
126         -- Question 13
127
128     WITH RankedScores AS (
129         SELECT Dev_ID, P_ID, SUM(score) AS total_score,
130                ROW_NUMBER() OVER (PARTITION BY Dev_ID ORDER BY SUM(score) DESC) AS score_rank
131         FROM Level_Details2
132         GROUP BY Dev_ID, P_ID
133     )
134     SELECT Dev_ID, P_ID, total_score
135     FROM RankedScores
136     WHERE score_rank <= 3;
137
```

75% | 33:134

Result Grid | Filter Rows: Search | Export:

Dev_ID	P_ID	total_sco...
bd_013	224	9870
bd_013	310	3370
bd_013	211	3200
bd_015	310	5300
bd_015	683	3200
bd_015	368	1950
bd_017	590	2400
bd_017	644	1750
bd_017	211	390
rf_013	368	2970
rf_013	211	2700
rf_013	300	2300
rf_015	483	3950
rf_015	683	2800
rf_015	590	900
rf_017	310	5140

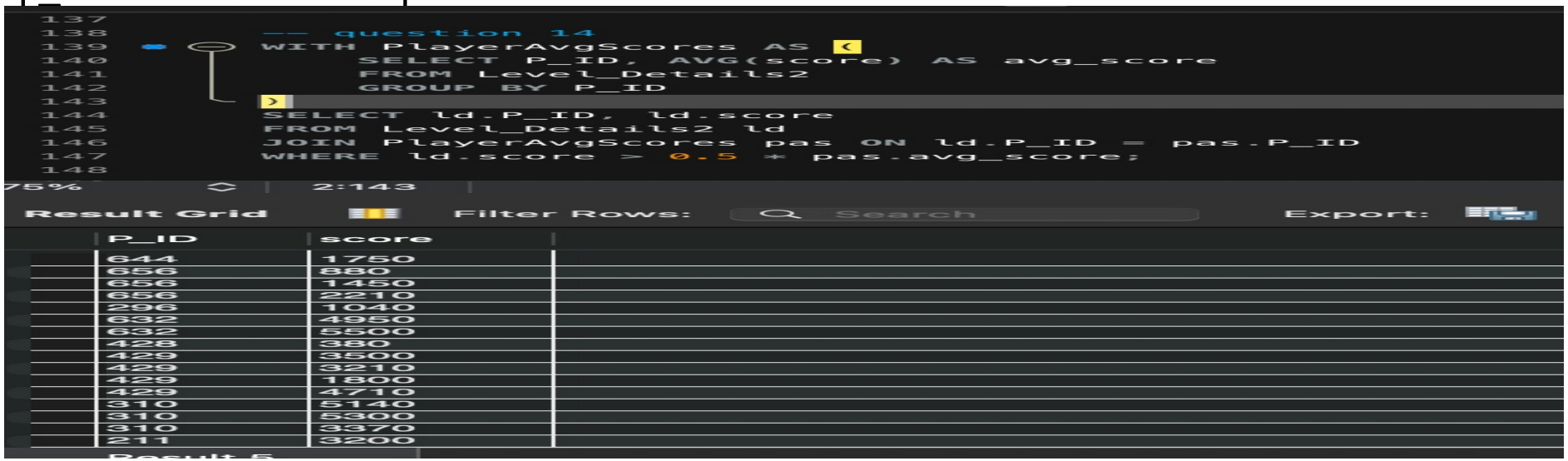
Result 4

- The Explanation:
- **WITH Clause (Common Table Expression - CTE)**:
  - RankedScores: Calculates the total score for each combination of Dev_ID and P_ID and ranks them based on the total score within each Dev_ID partition.
    - Dev_ID: Device ID.
    - P_ID: Player ID.
    - SUM(score) AS total_score: Calculates the total score for each Dev_ID and P_ID combination.
    - ROW_NUMBER() OVER (PARTITION BY Dev_ID ORDER BY SUM(score) DESC) AS score_rank: Assigns a rank to each combination of Dev_ID and P_ID based on the total score, with the highest total score receiving a rank of 1 within each Dev_ID partition.
- **SELECT**:
  - Selects the columns to be displayed in the final result set.
  - Dev_ID: Device ID.
  - P_ID: Player ID.
  - total_score: Total score calculated for each Dev_ID and P_ID combination.
- **FROM**:
  - Specifies the source table or CTE (RankedScores).
- **WHERE Clause**:
  - Filters the results to only include rows where the score rank is less than or equal to 3, effectively selecting the top 3 total scores for each Dev_ID..

Question 14:

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

- ## Sql_Statement and Response:



```
-- question 14
WITH PlayerAvgScores AS (
    SELECT P_ID, AVG(score) AS avg_score
    FROM Level_Details2
    GROUP BY P_ID
)
SELECT ld.P_ID, ld.score
FROM Level_Details2 ld
JOIN PlayerAvgScores pas ON ld.P_ID = pas.P_ID
WHERE ld.score > 0.5 * pas.avg_score;
```

Result Grid | Filter Rows: Search | Export:

P_ID	score
644	1750
656	880
656	1450
656	2210
296	1040
632	4950
632	5500
428	880
429	3500
429	3210
429	1800
429	4710
310	5140
310	5300
310	3370
211	3200

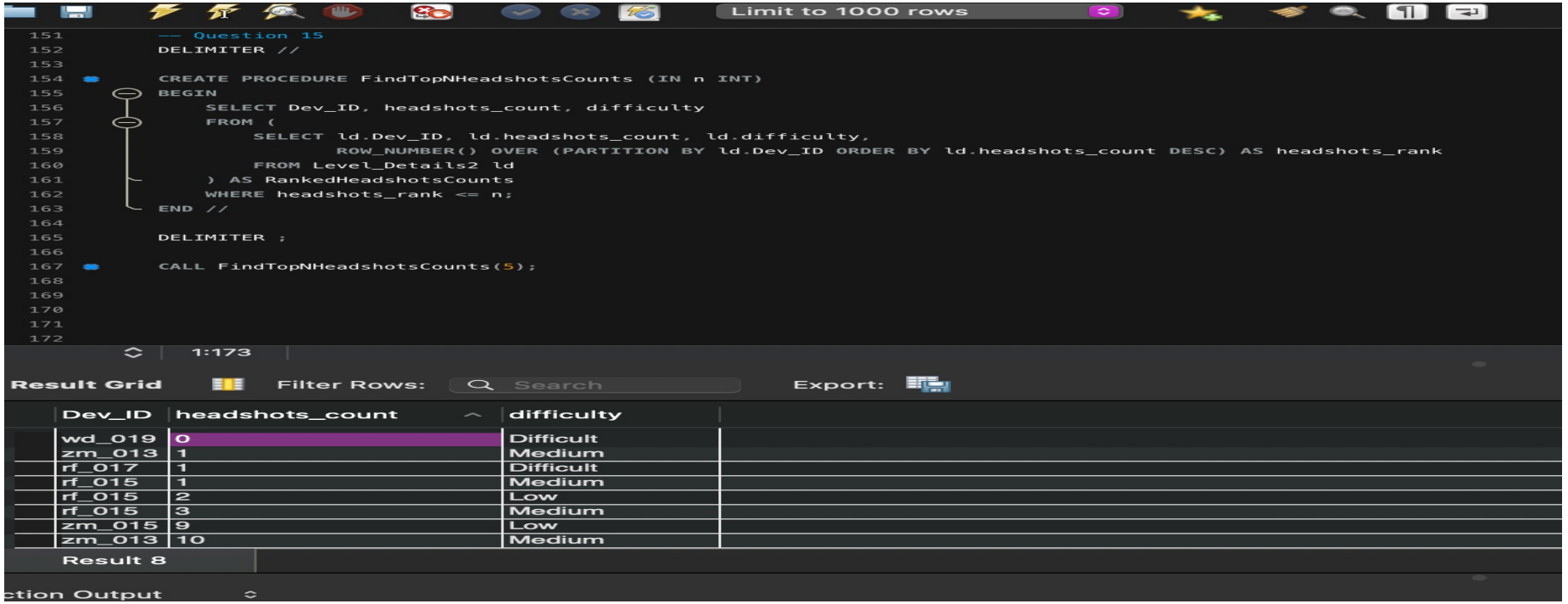
- The Explanation:

1. **Common Table Expression (CTE) - PlayerAvgScores**:
    1. This CTE calculates the average score for each player (P_ID) from the Level_Details2 table using the AVG function.
    2. It groups the data by P_ID.
2. **Main Query**:
    1. It selects the P_ID and score columns from the Level_Details2 table.
    2. It joins the Level_Details2 table with the PlayerAvgScores CTE on the P_ID column.
    3. The join ensures that each row in the Level_Details2 table is matched with the corresponding average score calculated for the player.
    4. The WHERE clause filters the result to include only rows where the score is greater than 50% of the player's average score.

Question 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.

- Sql_Statement and Response:



- The Explanation:

- FindTopNHeadshotsCounts is designed to find the top N headshots counts for each Dev_ID and rank them in descending order based on headshots_count. Here's a breakdown of the procedure:

1. **Input Parameter**:
    1. The procedure accepts a single input parameter n, which specifies the number of top headshots counts to retrieve for each Dev_ID.
2. **Query Explanation**:
    1. The main query selects Dev_ID, headshots_count, and difficulty from a subquery.
    2. The subquery calculates the row number (headshots_rank) for each Dev_ID partitioned by Dev_ID and ordered by headshots_count in descending order.
    3. It selects Dev_ID, headshots_count, difficulty, and headshots_rank.
    4. The outer query filters the result to include only rows where headshots_rank is less than or equal to the input parameter n.
3. **Usage**:
    1. This procedure can be invoked by passing an integer value for n, specifying the desired number of top headshots counts.
    2. For example, calling CALL FindTopNHeadshotsCounts(5) would retrieve the top 5 headshots counts for each Dev_ID.