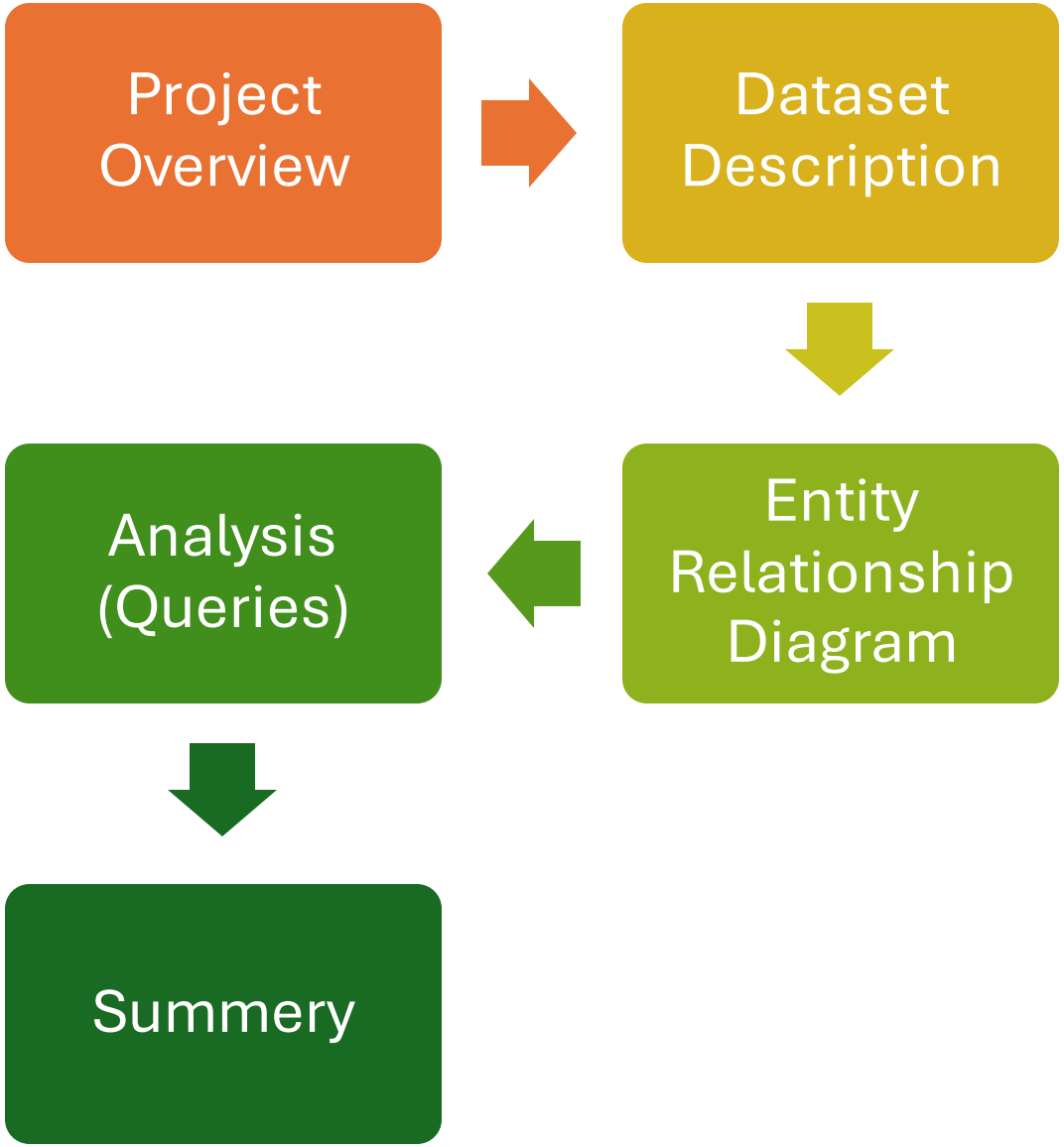
GAME ANALYSIS with SQL

MENTORNESS

Internship Project by

Content


Project Overview
Dataset Description
Entity Relationship Diagram
Analysis (Queries)
Summery

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 progession, and areas for game experience enhancement.
- ✓ Key questions include player trend, 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 table : **'Player Details'** and **'Level Details'**

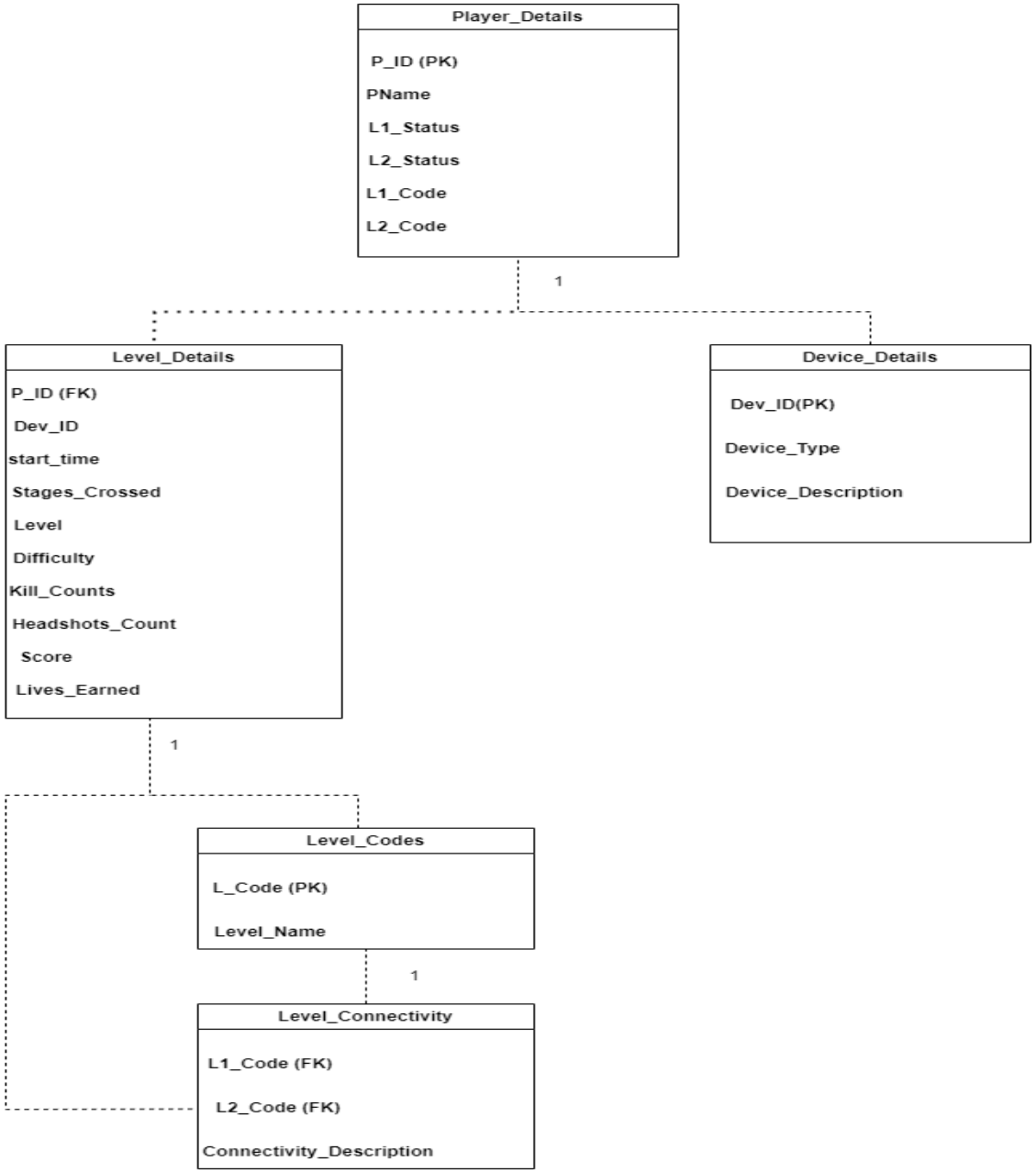
PLAYER DETAILS TABLE:

- **'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



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 included 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,ID.DEV_ID,PD.PNAME,ID.DIFFICULTYFROM PLAYER_DETAILS PDINNER JOIN LEVEL_DETAILS2 ID ON PD.P_ID= ID.P_IDWHERE ID.LEVEL=0;

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

```
SELECT PD.P_ID,ID.DEV_ID,PD.PNAME,ID.DIFFICULTY
FROM PLAYER_DETAILS PD
INNER JOIN LEVEL_DETAILS2 ID ON PD.P_ID = ID.P_ID
WHERE ID.LEVEL=0;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

P_ID	DEV_ID	PNAME	DIFFICULTY
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

Result 1

Query – 2

Final Level1_code wise Avg_Kill_Count where lives_earned is 2 and at least 3 stages are crossed

SELECT PD.L1_CODE,AVG(LD.KILL_COUNT) AS AVG_KILL_COUNTFROM PLAYER_DETAILS PDINNER JOIN LEVEL_DETAILS2 LD ON PD.P_ID = LD.P_IDWHERE LD.LIVES_EARNED=2 AND LD.STAGES_CROSSED >=3GROUP BY PD.L1_CODE;

Analysis – it performs an inner join onn 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.

```
SELECT PD.L1_CODE,AVG(LD.KILL_COUNT) AS AVG_KILL_COUNT
FROM PLAYER_DETAILS PD
INNER JOIN LEVEL_DETAILS2 LD ON PD.P_ID = LD.P_ID
WHERE LD.LIVES_EARNED=2 AND LD.STAGES_CROSSED >=3
GROUP BY PD.L1_CODE;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

L1_CODE	AVG_KILL_COUNT
speed_blitz	19.3333
war_zone	19.2857
bulls_eye	22.2500

Query – 3

Find the total number of stages crossed at each difficulty level where for Level2 with players use zm_series device. 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 on with 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.

```sql
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;
```

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(TIMESTAMP)) AS UNIQUE_DATE_COUNTFROM LEVEL_DETAILS2GROUP BY P_IDHAVING COUNT(DISTINCT DATE(TIMESTAMP)) >1;

```
SELECT P_ID,COUNT(DISTINCT DATE(TIMESTAMP)) AS UNIQUE_DATE_COUNT
FROM LEVEL_DETAILS2
GROUP BY P_ID
HAVING COUNT(DISTINCT DATE(TIMESTAMP)) >1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

P_ID	UNIQUE_DATE_COUNT
211	4
224	2
242	2
292	2
300	3
310	3
368	2
483	3
590	3
632	3
641	2
644	2
656	4

Analysis – It groups the results by p_ID and filter 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

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 LD.P_ID,LD.LEVEL,SUM(LD.KILL_COUNT) AS TOTAL_KILL_COUNTFROM LEVEL_DETAILS2 LDINNER JOIN(SELECT AVG(KILL_COUNT) AS AVG_KILL_COUNT FROM LEVEL_DETAILS2 WHERE DIFFICULTY = 'Medium') AS AVG_TABLE ON LD.KILL_COUNT > AVG_TABLE.AVG_KILL_COUNTGROUP BY LD.P_ID,LD.LEVEL;

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

```sql
SELECT LD.P_ID,LD.LEVEL,SUM(LD.KILL_COUNT) AS TOTAL_KILL_COUNT
FROM LEVEL_DETAILS2 LD
INNER JOIN(
    SELECT AVG(KILL_COUNT) AS AVG_KILL_COUNT
    FROM LEVEL_DETAILS2
    WHERE DIFFICULTY = 'Medium' )
    AS AVG_TABLE ON LD.KILL_COUNT > AVG_TABLE.AVG_KILL_COUNT
GROUP BY LD.P_ID,LD.LEVEL;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

P_ID	LEVEL	TOTAL_KILL_COUNT
644	2	24
656	1	37
632	0	45
632	1	28
632	2	53
429	1	30
429	2	55
310	1	20
310	0	34

Result 5

Query – 6

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

SELECT LD.LEVEL,PD.L1_CODE,SUM(LD.LIVES_EARNED) AS TOTAL_LIVES_EARNEDFROM LEVEL_DETAILS2 LDINNER JOIN PLAYER_DETAILS PD ON LD.P_ID = PD.P_IDWHERE LD.LEVEL>0GROUP BY LD.LEVEL,PD.L1_CODEORDER BY LD.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.

```
SELECT LD.LEVEL,PD.L1_CODE,SUM(LD.LIVES_EARNED) AS TOTAL_LIVES_EARNED
FROM LEVEL_DETAILS2 LD
INNER JOIN PLAYER_DETAILS PD ON LD.P_ID = PD.P_ID
WHERE LD.LEVEL>0
GROUP BY LD.LEVEL,PD.L1_CODE
ORDER BY LD.LEVEL ASC;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

LEVEL	L1_CODE	TOTAL_LIVES_EARNED
1	bulls_eye	5
1	leap_of_faith	0
1	speed_blitz	7
1	war_zone	11
2	bulls_eye	14
2	speed_blitz	20
2	war_zone	17

Query – 7

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

SELECT rs.Dev_ID, rs.P_ID, rs.score, rs.difficulty, rs.ScoreRankFROM(SELECT ld.P_ID,ld.Dev_ID,ld.score,ld.difficulty, @rn := IF(@prevDevID = ld.Dev_ID, @rn + 1, 1) AS ScoreRank, @prevDevID := ld.Dev_ID FROM (SELECT * FROM Level_Details2 ORDER BY Dev_ID, score DESC) ld, (SELECT @rn := 0, @prevDevID := '') AS vars) rsWHERE rs.ScoreRank <= 3ORDER BY rs.Dev_ID, rs.ScoreRank;

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.

```
SELECT rs.Dev_ID, rs.P_ID, rs.score, rs.difficulty, rs.ScoreRank
FROM ( SELECT ld.P_ID,ld.Dev_ID,ld.score,ld.difficulty,
        @rn := IF(@prevDevID = ld.Dev_ID, @rn + 1, 1) AS ScoreRank,
        @prevDevID := ld.Dev_ID
    FROM
        (SELECT * FROM Level_Details2 ORDER BY Dev_ID, score DESC) ld,
        (SELECT @rn := 0, @prevDevID := '') AS vars
) rs
WHERE rs.ScoreRank <= 3
ORDER BY rs.Dev_ID, rs.ScoreRank;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Dev_ID	P_ID	score	difficulty	ScoreRank
bd_013	224	5300	Difficult	1
bd_013	224	4570	Difficult	2
bd_013	310	3370	Difficult	3
bd_015	310	5300	Difficult	1
bd_015	683	3200	Low	2
bd_015	368	1950	Difficult	3
bd_017	590	2400	Low	1
bd_017	644	1750	Medium	2
bd_017	211	390	Low	3
rf_013	368	2970	Difficult	1
rf_013	211	2700	Medium	2
rf_013	300	2300	Medium	3
rf_015	483	3950	Difficult	1

Result 7

Query – 8

Final first_login datetime for each device id.

SELECT DEV_ID,MIN(TIMESTAMP) AS FIRST_LOGINFROM LEVEL_DETAILS2GROUP 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 Player_Details.

```
SELECT DEV_ID,MIN(TIMESTAMP) AS FIRST_LOGIN
FROM LEVEL_DETAILS2
GROUP BY DEV_ID;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

DEV_ID	FIRST_LOGIN
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

Query – 9

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

SELECT ld.Dev_ID,ld.P_ID,ld.score,ld.difficulty, (SELECT COUNT(*) FROM Level_Details2 ld2 WHERE ld2.difficulty = ld.difficulty AND ld2.score >= ld.score) AS ScoreRank FROM Level_Details2 ldWHERE (SELECT COUNT(*) FROM Level_Details2 ld2 WHERE ld2.difficulty = ld.difficulty AND ld2.score >= ld.score) <= 5ORDER BY ld.difficulty, ScoreRank;

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.

```sql
SELECT ld.Dev_ID,ld.P_ID,ld.score,ld.difficulty,
    ( SELECT COUNT(*)
        FROM Level_Details2 ld2
        WHERE ld2.difficulty = ld.difficulty AND ld2.score >= ld.score
    ) AS ScoreRank FROM Level_Details2 ld
WHERE ( SELECT COUNT(*) FROM Level_Details2 ld2
    WHERE ld2.difficulty = ld.difficulty AND ld2.score >= ld.score
) <= 5
ORDER BY ld.difficulty, ScoreRank;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Dev_ID	P_ID	score	difficulty	ScoreRank
zm_017	632	5500	Difficult	2
zm_017	663	5500	Difficult	2
bd_015	310	5300	Difficult	4
bd_013	224	5300	Difficult	4
rf_017	310	5140	Difficult	5
zm_015	242	3470	Low	1
zm_017	429	3210	Low	2
bd_015	683	3200	Low	3
bd_013	242	2840	Low	4
zm_015	211	2800	Low	5
zm_017	483	5490	Medium	1
rf_017	224	5140	Medium	2
zm_015	632	4950	Medium	4
zm_015	663	4950	Medium	4
rf_015	683	2800	Medium	5

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,MIN(timestamp) AS FIRST_LOGINFROM LEVEL_DETAILS2GROUP BY P_ID,DEV_ID)SELECT FL.P_ID,FL.DEV_ID,FL.FIRST_LOGINFROM FIRSTLOGIN FLINNER JOIN LEVEL_DETAILS2 LD ON FL.P_ID = LD.P_ID AND FL.FIRST_LOGIN=LD.TIMESTAMP;

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.

```
WITH FIRSTLOGIN AS(
SELECT P_ID,DEV_ID,MIN(timestamp) AS FIRST_LOGIN
FROM LEVEL_DETAILS2
GROUP BY P_ID,DEV_ID
)
SELECT FL.P_ID,FL.DEV_ID,FL.FIRST_LOGIN
FROM FIRSTLOGIN FL
INNER JOIN LEVEL_DETAILS2 LD ON FL.P_ID = LD.P_ID AND FL.FIRST_LOGIN=LD.TIMESTAMP;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

P_ID	DEV_ID	FIRST_LOGIN
644	zm_015	2022-10-11 14:05:08
644	rf_015	2022-10-11 19:34:25
644	bd_017	2022-10-12 23:52:18
656	rf_013	2022-10-15 18:12:50
656	bd_015	2022-10-13 22:19:45
656	rf_017	2022-10-14 07:32:00
656	bd_013	2022-10-11 17:47:09
296	zm_017	2022-10-14 15:15:15
296	zm_015	2022-10-14 19:35:49
632	bd_013	2022-10-12 16:30:30
632	rf_013	2022-10-12 19:36:40
632	zm_017	2022-10-13 06:30:20
632	zm_015	2022-10-13 10:56:17
428	bd_015	2022-10-15 18:00:00
429	rf_017	2022-10-11 09:28:56
429	zm_017	2022-10-11 21:39:00

Result 10

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,DATE(TIMESTAMP) AS DATE,SUM(KILL_COUNT)OVER (PARTITION BY P_ID ORDER BY TIMESTAMP) AS TOTAL_KILL_COUNTFROM LEVEL_DETAILS2;
```

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 tracing the total kill count accumulated by each player as they progress through the game sessions, adding in analyzing player performance trends and engagement levels over time.

```
SELECT P_ID,DATE(TIMESTAMP) AS DATE,SUM(KILL_COUNT)
OVER (PARTITION BY P_ID ORDER BY TIMESTAMP) AS TOTAL_KILL_COUNT
FROM LEVEL_DETAILS2;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

P_ID	DATE	TOTAL_KILL_COUNT
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
292	2022-10-15	25
296	2022-10-14	7
296	2022-10-14	11

Result 11

Query – 11

A. Without Window function

```
SELECT P_ID,DATE(TIMESTAMP) AS DATE,SUM(KILL_COUNT) AS TOTAL_KILL_COUNTFROM LEVEL_DETAILS2GROUP BY P_ID,DATE(TIMESTAMP);
```

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 progess through their gaming session

```
SELECT P_ID,DATE(TIMESTAMP) AS DATE,SUM(KILL_COUNT) AS TOTAL_KILL_COUNT
FROM LEVEL_DETAILS2
GROUP BY P_ID,DATE(TIMESTAMP);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

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
310	2022-10-13	34
310	2022-10-15	14
211	2022-10-12	45

Result 12

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,TIMESTAMP,STAGES_CROSSED,SUM(STAGES_CROSSED) OVER (PARTITION BY P_I ORDER BY timestamp ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING)FROM LEVEL_DETAILS2;

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.

```
SELECT P_ID,TIMESTAMP,STAGES_CROSSED,
SUM(STAGES_CROSSED) OVER ( PARTITION BY P_ID
                           ORDER BY timestamp
                           ROWS BETWEEN UNBOUNDED
                           PRECEDING AND 1 PRECEDING)
FROM LEVEL_DETAILS2;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

P_ID	TIMESTAMP	STAGES_CROSSED	SUM(STAGES_CROSSED) OVER (PARTITION BY P_ID ORDER BY timestamp ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING)
211	2022-10-12 13:23:45	4	NULL
211	2022-10-12 18:30:30	5	4
211	2022-10-13 05:36:15	5	9
211	2022-10-13 22:30:18	5	14
211	2022-10-14 08:56:24	7	19
211	2022-10-15 11:41:19	8	26
224	2022-10-14 01:15:56	7	NULL
224	2022-10-14 08:21:49	5	7
224	2022-10-15 05:30:28	10	12
224	2022-10-15 13:43:50	4	22
242	2022-10-13 01:14:29	6	NULL

Result 13

Query – 13

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

WITH RankedStages AS (SELECT P_ID, timestamp,stages_crossed, ROW_NUMBER() OVER (PARTITION BY P_ID ORDER BY timestamp DESC) AS rn FROM Level_Details2)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_crossedFROM RankedStagesWHERE rn > 1;

Analysis – It then assigns a rank to each player within each device based on their total score, with the highest scorer receiving rank1.

The results are filtered to include only the top 3 scorers for each device, showing their P_ID, Dev_ID and total_score.

```sql
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;
```

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_DETAILS2      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_DETAILS2   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.

```
SELECT P_ID FROM (
        SELECT P_ID,SUM(SCORE) AS TOTAL_SCORE
        FROM LEVEL_DETAILS2
        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_DETAILS2
    GROUP BY P_ID
    ) AS AVG_SCORES);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

P_ID
656
632
429
310
211
300
224
242
590
483
368
663
683

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.

```
-- Create the function to return the sum of scores 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;

-- Call the function with a specific player_id to see the output
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 task, such as calculating total scores, finding top socres, identifying players with specific characteristics, and computing cumulative statistics.

Stored Procedures and Functions: PL/pgSQL stired procedures and function were utilized to encapsulate complex SQL logic, Improve code modularity and enhance database performance. Functions like 'GetPlayerScore SUm' were created to compute aggregated values based on input parameters.

Windows Function : Windows functions, such as 'ROW_NUMBER ()' and 'SUM()' , 'OVER()', were leveraged to perform advanced analytical operatons like ranking scores, calculating cumulative sums, and retrieving data based on specific windows partiions.

Optimized Queries : Effort were made to optimize SQL queries for efficiency and performance, ensuring that data retrieval and precessing task are executed swiftly, even with large datasets.

Overall the project demonstrates proficiency in database design. SQL Porgramming and data analysis techniques. Providing valuable insights into player behavior and game performance metrics.

Thank You!