GAME ANALYSIS

USING SQL

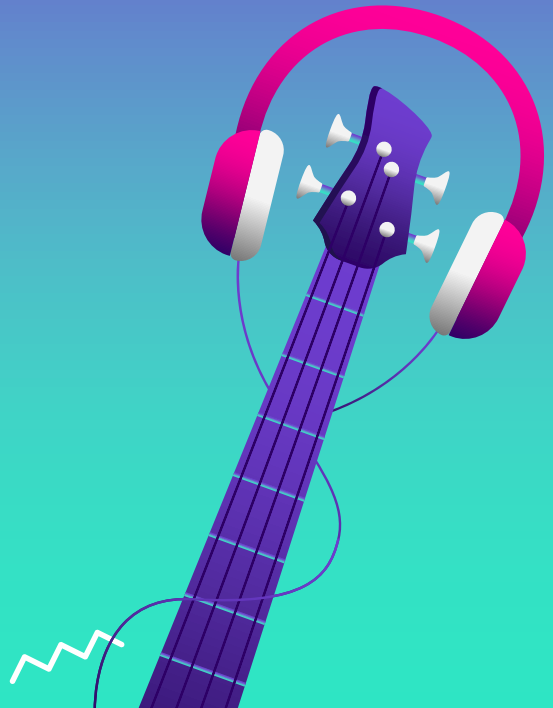
TITLE: DECODING GAME BEHAVIOR

By- Ayush Kumar

CONTENTS OF THIS PRESENTATION

1. Introduction
2. Data Source and Methodology
3. Uploading CSVs into PostgreSQL using Python
4. Insight Slides (SQL queries to solve all 15 Problem statements & their significance)
5. Conclusion

INTRODUCTION



This project dives into the world of gaming to analyze player behavior. We've studied gameplay data, focusing on key metrics to uncover patterns and understand how game elements influence strategies.

Our aim is to provide insights that can help developers enhance gaming experiences. Let's explore this exciting journey of decoding game behavior together!

DATA SOURCE & METHODOLOGY

The data for this project was provided by **Mentorness**, which served as the primary source for our analysis. The datasets were uploaded into a **PostgreSQL** database using **Python** to **automate the process, ensuring efficiency and accuracy**.

As part of the data preparation process, certain columns were **dropped** and others were **restructured** for efficient use. This step was crucial in ensuring that the **data was clean, relevant, and easy to work with**.

The methodology involved **solving 15 problem statements using SQL queries**. These problem statements, also provided by Mentorness, guided the analysis and helped uncover **key insights** into game behavior. The use of SQL allowed for **robust data manipulation** and querying capabilities, enabling a thorough exploration of the dataset.

This approach ensured a systematic and rigorous analysis of the data, leading to valuable insights and conclusions.

Uploading CSVs using Python

Pandas & SqlAlchemy

```
import pandas as pd
from sqlalchemy import create_engine

# Define the connection string
conn_string =
'postgresql://username:password@localhost/database_name'

# Create a database engine
db = create_engine(conn_string)

# Establish a connection
conn = db.connect()

# List of file paths
file_paths = [
    r"path_to_your_file1",
    r"path_to_your_file2"
]     ......
```

```
......
for file_path in file_paths:
    # Read the CSV file into a DataFrame
    df = pd.read_csv(file_path)

    # Extract table name from file name
    table_name = file_path.split('\\')[-1].split('.')[0]

    # Save the DataFrame to the database
    df.to_sql(table_name, con=conn, if_exists='replace',
index=False)

# Close the database connection
conn.close()
```

Insights, Queries & their Significance

01

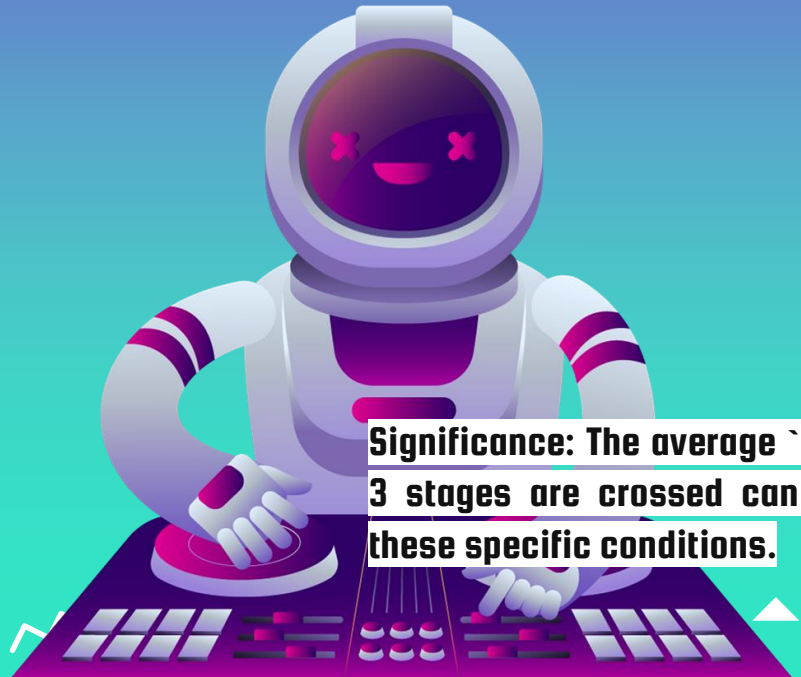
Extract `P_ID`, `Dev_ID`, `PName`, and `Difficulty_level` of all players at Level 0

```
select p.p_id, l.dev_id, p.p_name, l.difficulty as Difficulty_Level
from player_details as p
join level_details as l
on p.p_id = l.p_id
where l.level = 0
order by p.p_id;
```

Significance: Understanding the details of players at Level 0 can help identify new or inexperienced players, which could be useful for player support or marketing efforts.

02

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



```
select p.l1_code, l.kill_count
from player_details as p
join level_details as l
on p.p_id = l.p_id
where l.lives_earned = 2 and l.stages_crossed = 3;
```

Significance: The average `Kill_Count` where `lives_earned` is 2, and at least 3 stages are crossed can provide insights into player performance under these specific conditions.

03

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

```
select difficulty as difficulty_level, count(stages_crossed) as total_no_of_stages
from level_details
where level = 2 and dev_id like 'zm_%'
group by difficulty, stages_crossed
order by stages_crossed desc;
```

Significance: Knowing the total number of stages crossed at each difficulty level for Level 2 with players using `zm_series` devices can help understand player engagement and device performance.

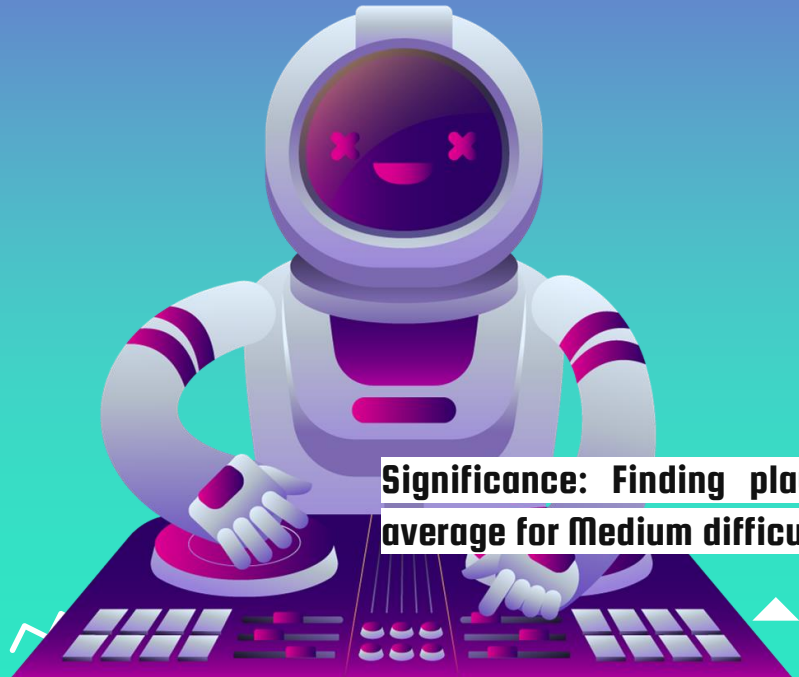
04 Extract `P_ID` and the total number of unique dates for those players who have played games on multiple days

```
select p_id, count(distinct date(timestamp)) as total_unique_dates
from level_details
group by p_id
having count(distinct date(timestamp)) > 1;
```

Significance: Identifying players who have played games on multiple days can help understand player retention and engagement over time.

05

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



```
select p_id, level, sum(kill_count) as total_kills
from level_details
where difficulty = 'medium'group by p_id, level
having sum(kill_count) > (
                    select avg(kill_count)
                    from level_details
                    where difficulty = 'medium')
order by p_id, level;
```

Significance: Finding players who have a `kill_count` greater than the average for Medium difficulty can help identify skilled players.

06

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

```
select l.level, p.l1_code, p.l2_code, sum(lives_earned) as sum_of_lives_earned
from level_details as l
join player_details as p on p.p_id = l.p_id
where l.level > 0
group by l.level, p.l1_code, p.l2_code
order by l.level;
```

Significance: Understanding the sum of lives earned at each level, excluding Level 0, can provide insights into player survival rates at different levels.

07

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

```
select * from (
                select dev_id, score, difficulty, row_number() over (partition by dev_id
                order by score desc) as rank
                from level_details
) as subquery
where rank <= 3
order by dev_id, rank;
```

Significance: Ranking the top 3 scores based on each `Dev_ID` can help understand device performance and player skill.

Find the `first_login` datetime for each device ID

08

```
select dev_id, min(timestamp) as first_login
from level_details
group by dev_id;
```

Significance: Knowing the `first_login` datetime for each device ID can provide insights into device usage patterns.

09

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

```
select * from (
                select difficulty as difficulty_level, score, dev_id, rank() over (partition
                by difficulty order by score desc) as rank
                from level_details
) as subquery
where rank <= 5
order by difficulty_level, rank;
```

Significance: Ranking the top 5 scores based on each difficulty level can help understand player performance across different difficulty levels.

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

```
select p.p_id, l.dev_id, p.first_login_datetime
from (
                select p_id, min(timestamp) as first_login_datetime
                from level_details
                group by p_id
) as p
join level_details as l on p.p_id = l.p_id and p.first_login_datetime = l.timestamp
order by p.p_id;
```

Significance: Identifying the device ID that is first logged in for each player can provide insights into player device preferences.

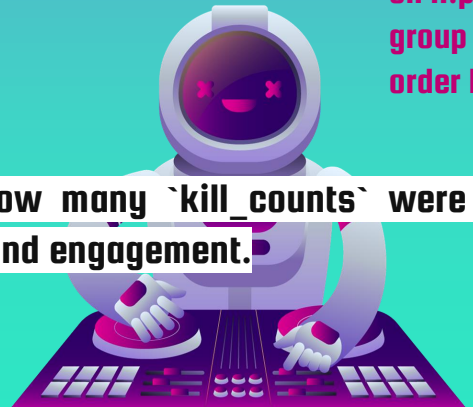
For each player and date, determine how many `kill_counts` were played by the player so far

a) Using window functions

```
select p_id, timestamp, sum(kill_count) over (partition
by p_id order by timestamp rows between unbounded
preceding and current row) as cumulative_kill_count
from level_details;
```

b) Without window functions

```
select l1.p_id, l1.timestamp, sum(l2.kill_count) as
cumulative_kill_count
from level_details l1
join level_details l2
on l1.p_id = l2.p_id and l2.timestamp <= l1.timestamp
group by l1.p_id, l1.timestamp
order by l1.p_id, l1.timestamp;
```



Significance: Determining how many `kill_counts` were played by the player so far can help understand player progress and engagement.

12

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

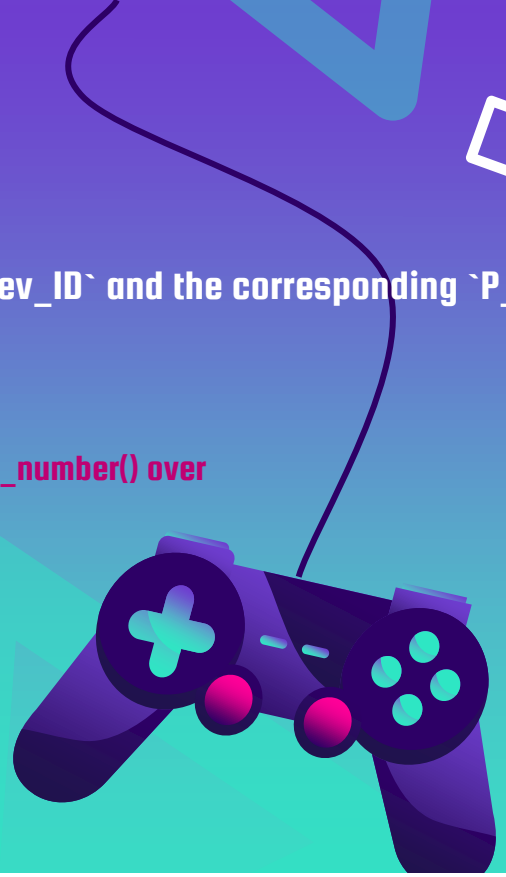
```
select p_id, timestamp, sum(stages_crossed) over (partition by p_id order by
timestamp rows between unbounded preceding and 1 preceding) as
cumulative_stages_crossed
from level_details
order by p_id, timestamp;
```

Significance: Finding the cumulative sum of stages crossed over `start_datetime` for each `P_ID`, excluding the most recent `start_datetime`, can provide insights into player progress over time.

13

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

```
select dev_id, p_id, sum_of_scores
from (
        select dev_id, p_id, sum(score) as sum_of_scores, row_number() over
        (partition by dev_id order by sum(score) desc) as rank
        from level_details
        group by dev_id, p_id
) as subquery
where rank <= 3
order by dev_id, rank;
```



Significance: Extracting the top 3 highest sums of scores for each `Dev_ID` and the corresponding `P_ID` can help understand player performance and device performance.

14

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

```
select p_id, sum(score) as sum_of_scores
from level_details
group by p_id
having sum(score) > (
        select 0.5 * avg(sum_of_scores)
        from (
                select p_id, sum(score) as sum_of_scores
                from level_details
                group by p_id
        ) as subquery
)
order by p_id;
```

Significance: Identifying players who scored more than 50% of the average score can help identify above-average players.

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

```
CREATE OR REPLACE FUNCTION get_top_headshots(n integer)
RETURNS TABLE (dev_id text, headshots_count bigint, difficulty text, rank bigint) AS $$
BEGIN
        RETURN QUERY
        SELECT * FROM (
            SELECT l.dev_id, l.headshots_count, l.difficulty, ROW_NUMBER() OVER (PARTITION BY l.dev_id ORDER
            BY l.headshots_count DESC) as rank
            FROM level_details AS l
        ) AS subquery
        WHERE subquery.rank <= n
        ORDER BY dev_id, subquery.rank;
END; $$
LANGUAGE plpgsql;
```

Significance: Creating a stored procedure to find the top `n` `headshots_count` based on each `Dev_ID` can provide a reusable tool for analyzing player performance.

TOP THREE INSIGHTS, WHY?

These insights cover a broad range of areas including player identification, engagement analysis, and performance evaluation, making them valuable for a diverse audience.

INSIGHT 1

Understanding the details of players at Level 0 can help identify new or inexperienced players, which could be useful for player support or marketing efforts.

INSIGHT 4

Identifying players who have played games on multiple days can help understand player retention and engagement over time.

INSIGHT 13

Extracting the top 3 highest sums of scores for each `Dev_ID` and the corresponding `P_ID` can help understand player performance and device performance.

CONCLUSION

Our project leveraged Game Analysis data and Python for efficient database management. We prepared the data rigorously and used SQL to solve 15 problem statements, revealing key insights into game behavior.

These insights, including player identification, retention, and performance evaluation, have implications for player support and game improvement. The project showcased SQL's power in handling large datasets and the importance of systematic analysis in driving industry strategies.

The outcomes of this project can guide future strategies and decision-making processes in the gaming industry.

THANKS!