Game Play Analytics with SQL and Pandas: A Hands-On Guide Using LeetCode 511, 512, 534, 550

Contents

Contents

Chapter 1 — First Login Date (LeetCode 511: Game Play Analysis I) 1.1 Problem Restatement

We are given a table Activity:

Column Name	Type
player_id	int
device_id	int
event_date	date
games_played	int

(player_id, event_date) together form the primary key. Each row represents a login session. We must determine the first login date (earliest event_date) for every player. Goal:

Return a table with the following structure:

player_id	first_login
int	date

1.2 SQL Solution (Canonical)

SELECT player_id, MIN(event_date) AS first_login FROM Activity GROUP BY player_id;

Explanation GROUP BY player_id partitions the table by each player. MIN(event_date) extracts the earliest login date within the group. Simple and efficient since event_date is comparable and primary key ensures uniqueness.

1.3 Pandas Solution (Canonical)

import pandas as pd

def game_play_analysis_I(activity: pd.DataFrame) -> pd.DataFrame: result = (activity .groupby("player_id", as_index=False)["event_date"] .min() .rename(columns={"event_date": "first_login"})) return result

Explanation (line-by-line) groupby("player_id") → partitions the DataFrame by player. ["event_date"].min() → computes earliest login date. rename() ensures the result column matches expected output format.

1.4 Alternative SQL Approaches Approach A — Window Function + DISTINCT

SELECT DISTINCT player_id, FIRST_VALUE(event_date) OVER (PARTITION BY player_id ORDER BY event_date) AS first_login FROM Activity;

Approach B — Correlated Subquery

SELECT a.player_id, (SELECT MIN(event_date) FROM Activity AS b WHERE a.player_id = b.player_id) AS first_login FROM Activity AS a GROUP BY a.player_id;

Approach C — Self-Join on Minimum Date

SELECT a.player_id, a.event_date AS first_login FROM Activity AS a JOIN (SELECT player_id, MIN(event_date) AS min_date FROM Activity GROUP BY player_id) AS x ON a.player_id = x.player_id AND a.event_date = x.min_date;

Canonical solution is still the fastest and most readable.

1.5 Alternative Pandas Approaches Approach A — sort + drop_duplicates

def game_play_analysis_I_alt1(activity): df = activity.sort_values(["player_id", "event_date"]) return df.drop_duplicates("player_id")[["player_id", "event_date"]].rename(columns={"event_date": "first_login"})

Approach B — idxmin

def game_play_analysis_I_alt2(activity): idx = activity.groupby("player_id")["event_date"].idxmin() return activity.loc[idx, ["player_id", "event_date"]].rename(columns={"event_date": "first_login"})

1.6 Edge Cases & Discussion

1.6.1 A player logs in once Both SQL and Pandas handle this naturally—MIN(event_date) is that single date. 1.6.2 Multiple sessions on same date Not possible due to PK constraint (player_id, event_date). 1.6.3 Extremely large tables SQL engines use indexes and query planners → generally O(n log n). Pandas loads entire dataset into memory → potential memory bottlenecks. 1.6.4 NULL values Assumed not present (LeetCode guarantees valid input). If NULL dates existed, MIN would ignore NULLs unless configured otherwise.

1.7 Time & Space Complexity Analysis

SQL Query Complexity

Operation	Complexity
Grouping on player_id	O(N log N) typically
Aggregation (MIN)	O(1) per group
Total	O(N log N)

If the DB index exists on (player_id, event_date), the query may become nearly O(N). Pandas Complexity groupby + min

Stage	Complexity
Hash-based groupby	O(N)
Min aggregation	O(1) per group
Total	O(N) average

Stage	Complexity

Memory usage:

Pandas must hold the entire dataset in RAM → cost   size of Activity table. 1.8 Code Review Commentary (Optimization Perspective)

SQL Code Review Notes   Simple, readable, minimal operations.   Uses built-in aggregate MIN → optimal.   Add an index on (player_id, event_date) for very large datasets.   Avoid window functions because unnecessary for this problem.

Pandas Code Review Notes   Uses groupby + min (optimal approach).   Avoids unnecessary sorting or merging.   Ensure event_date column is converted to datetime type: activity[“event_date”] = pd.to_datetime(activity[“event_date”])

Scalability Considerations For massive datasets (> 50M rows), SQL is preferable due to on-disk processing. Pandas solution could be rewritten with: Polars for lazy execution Dask for distributed processing

1.9 Example Walkthrough Input

player_id | event_date

1 | 2016-03-01 1 | 2016-05-02 2 | 2017-06-25 3 | 2016-03-02 3 | 2018-07-03

Processing Player 1 → MIN = 2016-03-01 Player 2 → MIN = 2017-06-25 Player 3 → MIN = 2016-03-02 Output player_id | first_login ———————— 1 | 2016-03-01 2 | 2017-06-25 3 | 2016-03-02

End of Chapter 1

Chapter 2 — First Login Device

LeetCode 512: Game Play Analysis II

2.1 Problem Restatement

We are given the following table:

“‘text +————–+———+ | Column Name | Type | +————–+———+ | player_id | int | | device_id | int | | event_date | date | | games_played | int | +————–+———+ PRIMARY KEY: (player_id, event_date)

Each row corresponds to a login session.

Our goal: For each player, report the device_id that was used on their FIRST login date.

Expected output: +———–+———–+ | player_id | device_id | +———–+———–+ Because (player_id, event_date) is a primary key, there will always be exactly one device_id corresponding to the earliest date.

2.2 SQL Solution (Canonical) The natural SQL solution uses an aggregate to find the earliest date per player and then joins back to get the device:

SELECT a.player_id, a.device_id FROM Activity AS a JOIN (SELECT player_id, MIN(event_date) AS first_login FROM Activity GROUP BY player_id) AS f ON a.player_id = f.player_id AND a.event_date = f.first_login;

Why this works The inner query extracts the earliest login date. Joining on (player_id, event_date) correctly matches the unique login row. Guaranteed 1-to-1 mapping due to primary key constraint.

2.3 Pandas Solution (Canonical)

import pandas as pd

def game_play_analysis_II(activity: pd.DataFrame) -> pd.DataFrame: first_dates = (activity .groupby(“player_id”, as_index=False)[“event_date”] .min() .rename(columns={“event_date”: “first_login”}))

```
merged = first_dates.merge(
    activity,
    left_on=["player_id", "first_login"],
    right_on=["player_id", "event_date"],
    how="left"
)

return merged[["player_id", "device_id"]]
```

Explanation groupby + min → finds earliest date. merge maps each earliest date back to the single row containing the matching device. Output is trimmed to required columns.

2.4 Alternative SQL Solutions

A. Window Function (ROW_NUMBER)

SELECT player_id, device_id FROM (SELECT player_id, device_id, event_date, ROW_NUMBER() OVER (PARTITION BY player_id ORDER BY event_date) AS rn FROM Activity) AS t WHERE rn = 1;

B. FIRST_VALUE Window Function

SELECT DISTINCT player_id, FIRST_VALUE(device_id) OVER (PARTITION BY player_id ORDER BY event_date) AS device_id FROM Activity;

C. Correlated Subquery (less efficient)

SELECT player_id, device_id FROM Activity AS a WHERE event_date = (SELECT MIN(event_date) FROM Activity AS b WHERE b.player_id = a.player_id);

2.5 Alternative Pandas Solutions

A. sort + drop_duplicates def game_play_analysis_II_alt1(activity): df = activity.sort_values([“player_id”, “event_date”]) df = df.drop_duplicates(“player_id”) return df[[“player_id”, “device_id”]]

B. idxmin to retrieve rows directly def game_play_analysis_II_alt2(activity): idx = activity.groupby(“player_id”)[“event_da return activity.loc[idx, [“player_id”, “device_id”]]

2.6 Edge Cases and Discussion

A. Player logs in only once Output is trivial: the one device recorded.

B. Multiple sessions with same event_date Impossible because of primary key (player_id, event_date).

C. Ties on earliest date Not possible.

D. Large datasets SQL handles grouping efficiently using indexes. Pandas may hit memory limits because it loads entire table into RAM.

E. Time zone considerations LeetCode always uses plain dates; no UTC/offset concerns.

2.7 Complexity Analysis SQL Complexity

Operation	Complexity
MIN(event_date) per player	O(N)
Join with main table	O(N log N) or better with indexing

Operation	Complexity
Total	O(N log N) or O(N)

Pandas Complexity

Operation	Complexity
groupby + min	O(N)
merge	O(N)
total	O(N)

Memory cost is proportional to the full table size.

2.8 Code Review Commentary

SQL Review Notes

The canonical solution is optimal and widely used. Readable and easy to maintain. Adding an index on (player_id, event_date) greatly improves performance. Window function version is clear when more ranking logic is needed.

Pandas Review Notes

Canonical solution is clean and idiomatic. Avoid unnecessary sorting unless using the drop_duplicates approach. Ensure event_date is a proper datetime type.

Scalability for Very Large Data

SQL is preferred for data > tens of millions of rows. Pandas alternatives: Polars for lazy query-like execution. Dask for distributed DataFrame computing. DuckDB if SQL is desired within Python.

2.9 Example Walkthrough

Input

player_id	device_id	event_date
1 1 2 3 3	2 2 3 1 4	2016-03-01 2016-05-02 2017-06-25 2016-03-02 2018-07-03

First login dates

player_id | first_login

1 | 2016-03-01 2 | 2017-06-25 3 | 2016-03-02

Match corresponding devices

player_id | device_id

1 | 2 2 | 3 3 | 1

End of Chapter 2

Chapter 3 — Cumulative Games Played So Far

LeetCode 534: Game Play Analysis III

3.1 Problem Restatement

We are given the table:

“‘text +————–+———+ | Column Name | Type | +————–+———+ | player_id | int | | device_id | int | | event_date | date | | games_played | int | +————–+———+

PRIMARY KEY: (player_id, event_date)

Each record corresponds to a player’s login on a given date, including how many games they played on that day. Goal: For each (player_id, event_date), compute: Total number of games played by that player from their first login up to that date (i.e., a running cumulative sum ordered by date) Expected output: +———–+————+——————–+ | player_id | event_date | games_played_so_far | +———–+——––+——————–+ 3.2 SQL Solution (Canonical — Window Function) SELECT player_id, event_date, SUM(games_played) OVER (PARTITION BY player_id ORDER BY event_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS games_played_so_far FROM Activity; Explanation The SUM(…) OVER window computes a cumulative running total. Ordering by event_date ensures correctness. UNBOUNDED PRECEDING → start from earliest event. CURRENT ROW → include the current row’s games. This is the standard and optimal SQL formulation. 3.3 Pandas Solution (Canonical) import pandas as pd

def game_play_analysis_III(activity: pd.DataFrame) -> pd.DataFrame: # Sort to establish event order activity_sorted = activity.sort_values([“player_id”, “event_date”])

```
# Compute cumulative games
activity_sorted["games_played_so_far"] = (
    activity_sorted
    .groupby("player_id")["games_played"]
    .cumsum()
)

return activity_sorted[["player_id", "event_date", "games_played_so_far"]]
```

Explanation Sorting ensures chronological order. groupby(…).cumsum() performs cumulative summation within each player partition. 3.4 Alternative SQL Approaches A. Correlated Subquery (inefficient but valid) SELECT a.player_id, a.event_date, (SELECT SUM(b.games_played) FROM Activity AS b WHERE b.player_id = a.player_id AND b.event_date <= a.event_date) AS games_played_so_far FROM Activity AS a; Complexity: O(N²) worst case → not recommended for large tables. B. Self-Join Prefix Sum SELECT a.player_id, a.event_date, SUM(b.games_played) AS games_played_so_far FROM Activity a JOIN Activity b ON a.player_id = b.player_id AND b.event_date <= a.event_date GROUP BY a.player_id, a.event_date; Better than correlated query, but still not as efficient as window functions. 3.5 Alternative Pandas Approaches A. Using expanding() def game_play_analysis_III_alt1(activity): df = activity.sort_values([“player_id”, “event_date”]) df[“games_played_so_far”] = (df.groupby(“player_id”)[“games_played”] .expanding() .sum() .reset_index(level=0, drop=True)) return df[[“player_id”, “event_date”, “games_played_so_far”]] B. Using .apply (slower) def game_play_analysis_III_alt2(activity): df = activity.sort_values([“player_id”, “event_date”]) df[“games_played_so_far”] = (df.groupby(“player_id”)[“games_played”] .apply(lambda s: s.cumsum()) .reset_index(level=0, drop=True)) return df[[“player_id”, “event_date”, “games_played_so_far”]] 3.6 Edge Cases & Subtleties A. Zero games played games_played = 0 Still contributes to the cumulative sum (no increment). B. Sparse login days Players do not log in every day. Missing days do not appear in results. C. Ordering concerns event_date must be strictly increasing per player due to primary key constraint. D. Extremely large datasets SQL window function is optimized and highly scalable. Pandas requires entire dataset in RAM → may hit memory limits. E. Negative values Not applicable in the LeetCode problem, but cumulative logic still works.

3.7 Time & Space Complexity Analysis SQL

Operation	Complexity
Windowed SUM	O(N) within partitions
Sorting within partitions	~O(N log N) total
Overall	O(N log N)

Indexes on (player_id, event_date) improve performance significantly.

Pandas

Stage	Complexity
Sorting	O(N log N)
groupby + cumsum	O(N)
Total	O(N log N)

Memory cost: must hold entire table in memory.

3.8 Code Review Commentary

SQL Review Notes

Window-function approach is optimal and idiomatic. Avoid correlated subqueries in production systems. Ensure index: INDEX(player_id, event_date) This accelerates both sorting and partitioning.

Pandas Review Notes Always sort before cumsum to guarantee correct cumulative order. Avoid .apply() unless necessary; it is slower and less readable. Consider using Polars for large-scale cumulative operations: df.sort([“player_id”, “event_date”]).with_columns(pl.col(“games_played”).cumsum().over(“player_id”))

3.9 Example Walkthrough

Input +———–+———–+————+————-+ | player_id | device_id | event_date | games_played | +———–+———–+————+————-+ | 1 | 2 | 2016-03-01 | 5 | | 1 | 2 | 2016-05-02 | 6 | | 1 | 3 | 2017-06-25 | 1 | | 3 | 1 | 2016-03-02 | 0 | | 3 | 4 | 2018-07-03 | 5 | +———–+———–+————+————-+

Processing Player 1: 2016-03-01 → 5 2016-05-02 → 5 + 6 = 11 2017-06-25 → 5 + 6 + 1 = 12 Player 3: 2016-03-02 → 0 2018-07-03 → 0 + 5 = 5

Final Output

player_id	event_date	games_played_so_far
1 1 1 3 3	2016-03-01 2016-05-02 2017-06-25 2016-03-02 2018-07-03	5 11 12 0 5

End of Chapter 3

Chapter 4 — Next-Day Retention After First Login

LeetCode 550: Game Play Analysis IV

4.1 Problem Restatement

We are given the following table:

“‘text +—————+———+ | Column Name | Type | +—————+———+ | player_id | int | | device_id | int | | event_date | date | | games_played | int | +—————+———+ PRIMARY KEY: (player_id, event_date)

Each row represents a player’s login activity for a given day.

Goal: Compute the fraction of players who logged in exactly on the day after their first login date.

Output:

fraction

Where:

fraction = retained_players / total_players (rounded to 2 decimals)

4.2 SQL Solution (Canonical)

WITH first_login AS (SELECT player_id, MIN(event_date) AS first_login FROM Activity GROUP BY player_id), next_day_login AS (SELECT DISTINCT f.player_id FROM first_login f JOIN Activity a ON a.player_id = f.player_id AND a.event_date = DATE_ADD(f.first_login, INTERVAL 1 DAY)) SELECT ROUND(COUNT(next_day_login.player_id) / COUNT(first_login.player_id), 2) AS fraction FROM first_login LEFT JOIN next_day_login ON first_login.player_id = next_day_login.player_id;

Explanation

first_login → earliest login date for each player. next_day_login → players who appeared again on first_login + 1 day. Final query computes fraction with rounding.

4.3 Pandas Solution (Canonical)

import pandas as pd

def game_play_analysis_IV(activity: pd.DataFrame) -> pd.DataFrame: # First login date per player first = (activity .groupby(“player_id”, as_index=False)[“event_date”] .min() .rename(columns={“event_date”: “first_login”}))

```
# Merge with original table
df = activity.merge(first, on="player_id", how="left")

# Compute next day
df["next_day"] = df["first_login"] + pd.Timedelta(days=1)

# Identify retained players
retained_players = df.loc[
    df["event_date"] == df["next_day"],
    "player_id"
].drop_duplicates()

total_players = first["player_id"].nunique()
retained_count = retained_players.nunique()

fraction = round(retained_count / total_players + 1e-9, 2)

return pd.DataFrame({"fraction": [fraction]})
```

Explanation

groupby().min() finds first login. Merging aligns each user's later activity with first login date. We check if ANY event_date equals first_login + 1 day. Round to 2 decimals, using small epsilon to avoid float instability.

4.4 Alternative SQL Approaches

A. Using EXISTS

WITH first_login AS (SELECT player_id, MIN(event_date) AS first_login FROM Activity GROUP BY player_id) SELECT ROUND(SUM(EXISTS (SELECT 1 FROM Activity a WHERE a.player_id = f.player_id AND a.event_date = DATE_ADD(f.first_login, INTERVAL 1 DAY))) / COUNT(*), 2) AS fraction FROM first_login f;

B. Using Window Functions

WITH ordered AS (SELECT player_id, event_date, ROW_NUMBER() OVER (PARTITION BY player_id ORDER BY event_date) AS rn FROM Activity), first AS (SELECT player_id, event_date AS first_login FROM ordered WHERE rn = 1) SELECT ROUND(COUNT(a.player_id) / COUNT(f.player_id), 2) AS fraction FROM first f LEFT JOIN Activity a ON a.player_id = f.player_id AND a.event_date = DATE_ADD(f.first_login, INTERVAL 1 DAY);

4.5 Alternative Pandas Approaches

A. Pivot-like direct membership test

def game_play_analysis_IV_alt1(activity): first = activity.groupby("player_id")["event_date"].min() next_day = first + pd.Timedelta(days=1)

```
df = activity.set_index(["player_id", "event_date"])

retained = [
    p for p in first.index
    if (p, next_day[p]) in df.index
]

return pd.DataFrame({"fraction": [round(len(retained)/len(first), 2)]})
```

B. Using merge with adjusted dates

def game_play_analysis_IV_alt2(activity): first = activity.groupby("player_id")["event_date"].min().reset_index() first["next_day"] = first["event_date"] + pd.Timedelta(days=1)

```
merged = first.merge(
    activity,
    left_on=["player_id", "next_day"],
    right_on=["player_id", "event_date"],
    how="left"
)

retained = merged["event_date"].notna().sum()
total = len(first)

return pd.DataFrame({"fraction": [round(retained / total, 2)]})
```

4.6 Edge Cases & Discussion

A. Player logs in only once

Cannot be retained; contributes to denominator only.

B. Player logs in many times

We only care about the day immediately following their first login.

C. No player logs in the next day

Output is: fraction = 0.00

D. All players log in next day

Output is: fraction = 1.00

E. Time gaps

If a player logs in on first day, then 5 days later → not retained.

F. Large datasets

SQL versions scale extremely well. Pandas version must load entire dataset into memory → may require Dask/Polars for large workloads.

4.7 Complexity Analysis

SQL

Operation	Complexity
MIN(event_date) groupby	O(N log N)
Join / Exists lookup	O(N) to O(N log N)
Overall	O(N log N)

Pandas

Operation	Complexity
groupby min	O(N)
merge	O(N)
filter	O(N)
Overall	**O(N)**

Memory usage: proportional to full table size.

4.8 Code Review Commentary (Optimization Perspective)

SQL Review Notes

Canonical two-CTE solution is clean and optimal. EXISTS version can be more efficient in some engines. Ensure indexing: INDEX(player_id, event_date) Improves join and filtering performance.

Pandas Review Notes

Avoid row-by-row loops; use vectorized merge logic. Ensure event_date dtype is datetime for correct arithmetic. Use .drop_duplicates() to avoid counting players multiple times.

Large-Scale Python Processing

Use: Polars for high-speed lazy queries DuckDB for SQL-on-local-files Dask for distributed execution

4.9 Example Walkthrough

Input

player_id	event_date
1 1 2 2 3	2020-01-01 2020-01-02 2020-05-10 2020-05-12 2020-08-20

First logins

player_id | first_login

1 | 2020-01-01 2 | 2020-05-10 3 | 2020-08-20 Next-day logins player_id | next_day_login

1 | yes 2 | no 3 | no

So: retained_players = 1 total_players = 3 fraction = 0.33 → rounded to 0.33

Output

fraction
0.33

End of Chapter 4