SQL Hard Problems with Pandas Solutions

Table of Contents

- 2720. Popularity Percentage
- 2752. Customers with Maximum Number of Transactions on Consecutive Days
- 2793. Status of Flight Tickets
- 2991. Top Three Wineries
- 3057. Employees Project Allocation
- 3060. User Activities within Time Bounds
- 3061. Calculate Trapping Rain Water
- 3103. Find Trending Hashtags II
- 3156. Employee Task Duration and Concurrent Tasks
- 3188. Find Top Scoring Students II
- 3214. Year on Year Growth Rate
- 3236. CEO Subordinate Hierarchy
- 3268. Find Overlapping Shifts II
- 3384. Team Dominance by Pass Success
- 3401. Find Circular Gift Exchange Chains
- 3451. Find Invalid IP Addresses

185. Department Top Three Salaries

Description:

Table 1: Employee

Column Name	Type
id name salary	int varchar
departmentId	int int

id is the primary key (column with unique values) for this table. departmentId is a foreign key (reference column) of the ID from the Department table. Each row of this table indicates the ID, name, and salary of an employee. It also contains the ID of their department.

Table 2: Department

Column Name	Type
id name	int varchar

id is the primary key (column with unique values) for this table. Each row of this table indicates the ID of a department and its name.

A company's executives are interested in seeing who earns the most money in each of the company's departments. A high earner in a department is an employee who has a salary in the top three unique salaries for that department.

Write a solution to find the employees who are high earners in each of the departments.

Return the result table in any order.

Solution:

```
import pandas as pd

def top_three_salaries(employee: pd.DataFrame, department: pd.DataFrame) -> pd.DataFrame:
    # Step 1: Merge the employee and department dataframes
    merged_df = employee.merge(department,
                               left_on='departmentId',
                               right_on='id',
                               suffixes=('_employee', '_department'))

    # Step 2: Group by department and calculate dense rank of salaries in each department
    merged_df['salary_rank'] = merged_df.groupby('departmentId')['salary'].rank(method='dens

    # Step 3: Filter for employees with salary rank in top 3
    result = merged_df[merged_df['salary_rank'] <= 3]

    # Step 4: Select and rename the required columns
    result = result[['name_department', 'name_employee', 'salary']]
    result.columns = ['Department', 'Employee', 'Salary']

    return result
```

Explanation:

- This solution identifies the highest-paid employees in each department, specifically those with salaries in the top three within their department.
- The approach uses the following steps:
    1. Merge the Employee and Department tables to get department names along with employee details.
    2. Use Pandas' `rank()` method with `dense` parameter to assign ranks to salaries within each department.
    3. Filter for employees with a salary rank of 3 or less, capturing the top three salary tiers in each department.
    4. Format the result by selecting and renaming the required columns.
- The `dense` ranking method ensures that if multiple employees have the same salary, they get the same rank and all count toward the top three.
- For example, if two employees have the highest salary in a department, both get rank 1, and the employee with the next highest salary gets rank 2.

- Time complexity: O(n log n) where n is the number of employees, due to the sorting operations in the ranking function.
- Space complexity: O(n) for the merged dataframe and result set.
- This approach handles departments of different sizes efficiently without requiring complex subqueries.
- For large datasets, we could improve performance by:
  - Pre-filtering the employee dataframe to include only relevant columns before the merge
  - Using a more memory-efficient approach if the dataframes are very large
  - Potentially using a more performant merge strategy if needed

262. Trips and Users

Description:

Table 3: Trips

Column Name	Type
id client_id	int int int int
driver_id city_id	enum varchar
status request_at	

id is the primary key for this table. The table holds all taxi trips. Each trip has a unique id, while client_id and driver_id are foreign keys to the users_id at the Users table. Status is an ENUM (category) type of ('completed', 'cancelled_by_driver', 'cancelled_by_client').

Table 4: Users

Column Name	Type
users_id banned	int enum
role	enum

users_id is the primary key for this table. The table holds all users. Each user has a unique users_id, and role is an ENUM type of ('client', 'driver', 'partner'). banned is an ENUM (category) type of ('Yes', 'No').

The cancellation rate is computed by dividing the number of canceled (by client or driver) requests with unbanned users by the total number of requests with unbanned users on that day.

Write a solution to find the cancellation rate of requests with unbanned users (both client and driver must not be banned) each day between "2013-10-01" and "2013-10-03" with at least one trip. Round Cancellation Rate to two decimal points.

Return the result table in any order.

Solution:

```
import pandas as pd

def trips_and_users(trips: pd.DataFrame, users: pd.DataFrame) -> pd.DataFrame:
    # Step 1: Filter unbanned users
    unbanned_users = users[users['banned'] == 'No']['users_id'].tolist()

    # Step 2: Filter trips with unbanned clients and drivers
    filtered_trips = trips[
        (trips['client_id'].isin(unbanned_users)) &
        (trips['driver_id'].isin(unbanned_users)) &
        (trips['request_at'] >= '2013-10-01') &
        (trips['request_at'] <= '2013-10-03')
    ]

    # Step 3: Group by date and calculate cancellation rate
    # Count total trips and cancelled trips per day
    daily_stats = filtered_trips.groupby('request_at').agg(
        total_trips=('id', 'count'),
        cancelled_trips=('status', lambda x: sum(x.str.startswith('cancelled')))
    ).reset_index()

    # Step 4: Calculate cancellation rate and round to 2 decimal places
    daily_stats['Cancellation Rate'] = (daily_stats['cancelled_trips'] / daily_stats['total_t

    # Step 5: Format the result
    result = daily_stats[['request_at', 'Cancellation Rate']]
    result.columns = ['Day', 'Cancellation Rate']

    return result
```

Explanation:

- This query calculates the daily cancellation rate for taxi trips between Oct 1-3, 2013, where both client and driver are not banned.
- The solution uses a four-step approach:

1. Identify unbanned users by filtering the Users table for banned = 'No'.
2. Filter the Trips table to include only rides where both driver and client are unbanned and within the date range.
3. Group the filtered trips by date and calculate both the total trips and cancelled trips.
4. Calculate the cancellation rate by dividing cancelled trips by total trips for each day.

- The cancellation rate is calculated by dividing cancelled trips count by total trips count and rounding to two decimal places.
- For cancellation detection, we use a lambda function that checks if the status starts with 'cancelled'.
- Time complexity: O(n + m) where n is the number of trips and m is the number of users, due to the filtering and grouping operations.
- Space complexity: O(n) for the filtered trips and result dataframes.
- The approach efficiently handles the multiple filtering conditions in a clear, step-by-step manner.
- For large datasets, performance could be improved by:
  - Using dictionary-based lookups for unbanned users instead of lists
  - Pre-filtering the trips dataframe by date before other conditions
  - Using more efficient string matching for cancelled status detection
- This solution correctly handles the requirement to round the cancellation rate to two decimal places.

569. Median Employee Salary

Description:

Table 5: Employee

Column Name	Type
id company salary	int varchar int

id is the primary key (column with unique values) for this table. Each row of this table indicates the company and the salary of one employee.

Write a solution to find the rows that contain the median salary of each company. While calculating the median, when you sort the salaries of the company, break the ties by id.

Return the result table in any order.

The result format is in the following example.

Solution:

```
import pandas as pd
import numpy as np

def median_employee_salary(employee: pd.DataFrame) -> pd.DataFrame:
    # Step 1: Sort the data by company, salary, and id to break ties
    employee_sorted = employee.sort_values(by=['company', 'salary', 'id'])

    # Step 2: Assign row numbers within each company
    employee_sorted['row_num'] = employee_sorted.groupby('company').cumcount() + 1

    # Step 3: Calculate the total number of employees per company
    company_sizes = employee_sorted.groupby('company').size().reset_index(name='total_employe

    # Step 4: Merge the row numbers and company sizes
    employee_with_stats = employee_sorted.merge(company_sizes, on='company')

    # Step 5: Find the median positions
    # For odd number of employees: (n+1)/2
    # For even number of employees: n/2 and n/2+1
    employee_with_stats['is_median'] = employee_with_stats.apply(
        lambda row: (row['total_employees'] % 2 == 1 and row['row_num'] == (row['total_emplo
                    (row['total_employees'] % 2 == 0 and (row['row_num'] == row['total_emplo
                                                          row['row_num'] == row['total_emplo
        axis=1
    )

    # Step 6: Filter for median rows and select required columns
    result = employee_with_stats[employee_with_stats['is_median']][['id', 'company', 'salary

    return result
```

Explanation:

- This solution identifies rows containing the median salary for each company, breaking ties by employee ID when sorting salaries.
- The approach follows these steps:

    1. Sort the employee data by company, salary, and ID to establish a precise ordering.
    2. Assign sequential row numbers to employees within each company group.
    3. Calculate the total number of employees in each company.

4. Combine the row numbers with company sizes to determine median positions.
5. Apply a formula to identify which rows correspond to median positions:
   - For companies with odd employee counts: the middle row ((n+1)/2)
   - For companies with even employee counts: both middle rows (n/2 and n/2+1)
6. Filter for only the rows identified as medians and return the required columns.

- Time complexity: O(n log n) due to the sorting operations required for proper row numbering.
- Space complexity: O(n) where n is the number of employees, as we need to store all employee records with additional information.
- The solution correctly handles both odd and even employee counts:
  - For 5 employees, the 3rd ranked employee (position (5+1)/2) is selected.
  - For 6 employees, both the 3rd and 4th ranked employees (positions 6/2 and 6/2+1) are selected.
- The sort order by company, salary, and ID ensures that ties in salary are properly broken by ID.
- For large datasets, performance could be improved by:
  - Using more memory-efficient operations
  - Potentially splitting the calculation by company for parallel processing
  - Using specialized median calculation functions if available
- This approach maintains data integrity by working with the original ID, company, and salary values throughout the process.

571. Find Median Given Frequency of Numbers

Description:

Table 6: Numbers

Column Name	Type
num	int
frequency	int

num is the primary key for this table. Each row of this table shows the frequency of a number in the database.

The median is the value separating the higher half from the lower half of a data set.

Write an SQL query to report the median of all the numbers in the database after decompressing the Numbers table. Round the median to one decimal place.

Solution:

```python
import pandas as pd
import numpy as np

def find_median_given_frequency(numbers: pd.DataFrame) -> pd.DataFrame:
    # Step 1: Decompress the numbers table
    expanded_numbers = []
    for _, row in numbers.iterrows():
        expanded_numbers.extend([row['num']] * row['frequency'])

    # Step 2: Calculate the median
    median = np.median(expanded_numbers)

    # Step 3: Round to one decimal place
    rounded_median = round(float(median), 1)

    # Step 4: Create result DataFrame
    result = pd.DataFrame({'median': [rounded_median]})

    return result
```

Explanation:

- This solution calculates the median of a set of numbers where each number appears with a specific frequency.
- The approach consists of four main steps:
    1. "Decompress" the Numbers table by creating an expanded list where each number is repeated according to its frequency.
    2. Calculate the median of the expanded list using NumPy's median function.
    3. Round the result to one decimal place as required.
    4. Format the result as a DataFrame with a single column and value.
- For example, if the row is (3, 5), the number 3 will appear 5 times in the expanded dataset.
- Time complexity: O(n), where n is the total count of numbers after expansion (sum of all frequencies).
- Space complexity: O(n) for the expanded list of numbers.
- For large datasets with high frequencies, this approach could be memory-intensive as it materializes all repeated instances.
- An alternative approach for very large datasets would be to:
    1. Sort the numbers
    2. Calculate cumulative frequencies
    3. Find the position that represents the median

4. Perform linear interpolation if needed

- The NumPy median function automatically handles both odd and even counts, performing linear interpolation for even-count datasets.
- The solution works for any valid input within the constraints of the problem.

579. Find Cumulative Salary of an Employee

Description:

Table 7: Employee

Column Name	Type
id month salary	int int int

(id, month) is the primary key for this table. Each row in the table indicates the salary of an employee in one month. If the employee did not receive any salary for a month, there will not be an entry with id and month.

Write an SQL query to calculate the cumulative salary summary for every employee.

The cumulative salary summary for an employee can be calculated as follows:

- For each month that the employee worked, sum up the salaries in that month and the previous two months. This is their 3-month sum for that month. If an employee didn't work for some month, exclude that month from the calculation.
- Do not include the 3-month sum for the most recent month that the employee worked for in the result table.
- Do not include the 3-month sum for any month the employee didn't get any salary.

Return the result table ordered by id in ascending order. In case of a tie, order it by month in descending order.

Solution:

```
import pandas as pd

def cumulative_salary(employee: pd.DataFrame) -> pd.DataFrame:
    # Step 1: Sort the data by id and month
    employee_sorted = employee.sort_values(by=['id', 'month'])

    # Step 2: For each employee, calculate the rolling 3-month sum of salaries
    def calculate_3month_sum(group):
```

```
        # Sort by month for each employee
        group = group.sort_values('month')

        # Calculate rolling 3-month sum
        group['3month_sum'] = group['salary'].rolling(window=3, min_periods=1).sum()

        # Find the most recent month for each employee
        max_month = group['month'].max()

        # Exclude the most recent month
        group = group[group['month'] != max_month]

        return group

    # Apply the function to each employee group
    result = employee_sorted.groupby('id').apply(calculate_3month_sum).reset_index(drop=True)

    # Step 3: Select and rename required columns, sort as specified
    result = result[['id', 'month', '3month_sum']]
    result.columns = ['id', 'month', 'Salary']
    result = result.sort_values(by=['id', 'month'], ascending=[True, False])

    return result
```

Explanation:

- This solution calculates a rolling 3-month salary sum for each employee, excluding the most recent month.
- The approach uses the following steps:
  1. Sort the employee data by ID and month to establish chronological order.
  2. Define a function to process each employee's data:
     - Calculate a rolling 3-month sum using Pandas' `rolling()` window function
     - Identify and exclude the most recent month for each employee
  3. Format the result with the required columns and sort order.
- The `min_periods=1` parameter in `rolling()` ensures that the first month has just its own salary, the second month has the sum of two months, and subsequent months have the full 3-month sum.
- Time complexity: O(n log n) where n is the number of employee-month records, due to the sorting operations.
- Space complexity: O(n) for the processed dataframe.
- This solution handles edge cases elegantly:

  - For the first month of employment, only that month's salary is counted.
  - For the second month, only the first two months' salaries are counted.
  - Non-consecutive months are handled correctly, as missing months simply don't appear in the calculation.

- The approach efficiently uses Pandas' groupby and apply functions to process each employee's data separately.
- For large datasets, performance could be improved by:
  - Using a vectorized approach instead of apply() if possible
  - Optimizing the sorting operations
  - Using a more memory-efficient implementation for very large datasets
- The solution correctly follows the requirements to exclude the most recent month and sort by ID ascending, then month descending.

601. Human Traffic of Stadium

Description:

Table 8: Stadium

Column Name	Type
id visit_date people	int date int

id is the primary key for this table. Each row of this table contains the visit date and visit id to the stadium with the number of people during the visit. No two rows will have the same visit_date. The date of a visit is unique.

Write an SQL query to display the records with three or more consecutive rows where the number of people is greater than or equal to 100.

Return the result table ordered by visit_date in ascending order.

Solution:

```
import pandas as pd

def human_traffic(stadium: pd.DataFrame) -> pd.DataFrame:
    # Step 1: Filter for visits with 100 or more people
    high_traffic = stadium[stadium['people'] >= 100].copy()

    # Step 2: Sort by ID to ensure chronological order
    high_traffic = high_traffic.sort_values('id')
```

```
    # Step 3: Create a group identifier for consecutive IDs
    high_traffic['diff'] = high_traffic['id'] - high_traffic.index
    high_traffic['group'] = high_traffic['diff'].factorize()[0]

    # Step 4: Count rows in each group
    group_counts = high_traffic.groupby('group').size().reset_index(name='count')

    # Step 5: Filter for groups with at least 3 consecutive days
    valid_groups = group_counts[group_counts['count'] >= 3]['group'].tolist()

    # Step 6: Filter rows that belong to valid groups
    result = high_traffic[high_traffic['group'].isin(valid_groups)][['id', 'visit_date', 'pe

    # Step 7: Sort by visit_date
    result = result.sort_values('visit_date')

    return result
```

Explanation:

- This solution identifies consecutive days with stadium attendance of 100 or more people.
- The approach uses a "gaps and islands" technique to identify consecutive sequences in the data:
    1. Filter for days with 100+ visitors to focus only on high-traffic days.
    2. Sort by ID to ensure chronological order.
    3. Create a group identifier for consecutive sequences by subtracting the row index from the ID.
        - When IDs are consecutive, this difference remains constant, creating a "group ID" for consecutive records.
        - For example, if IDs 3, 4, 5 have indices 0, 1, 2, then (3-0), (4-1), (5-2) all equal 3, identifying them as part of the same group.
    4. Count the number of rows in each group to find sequences of consecutive days.
    5. Filter for groups with at least 3 consecutive days.
    6. Select only the rows that belong to these valid groups.
    7. Sort the result by visit_date as required.
- Time complexity: O(n log n) where n is the number of stadium visits, due to the sorting operations.
- Space complexity: O(n) for the high traffic dataframe and result set.
- This solution efficiently handles the "consecutive" requirement without complex joins or window function chains.

- The technique works correctly even with gaps in the ID sequence, as long as the IDs are still in chronological order.
- For large datasets, performance could be improved by:
  - Using more vectorized operations
  - Potentially using a different approach for identifying consecutive sequences
  - Pre-filtering the data if possible
- The factorize() function efficiently converts the diff column into group identifiers, handling potential non-integer differences.

615. Average Salary: Departments VS Company

Description:

Table 9: Salary

Column Name	Type
id employee_id	int int
amount	int date
pay_date	

id is the primary key column for this table. Each row of this table indicates the salary of an employee in one month. employee_id is a foreign key from the Employee table.

Table 10: Employee

Column Name	Type
employee_id	int int
department_id	

employee_id is the primary key column for this table. Each row of this table indicates the department of an employee.

Write an SQL query to report the comparison result (higher/lower/same) of the average salary of employees in a department to the company's average salary.

Return the result table in any order.

The comparison result is: - 'higher' when the average salary of the department is higher than the company's average salary. - 'lower' when the average salary of the department is lower than the company's average salary. - 'same' when the average salary of the department is the same as the company's average salary.

Solution:

```
import pandas as pd

def department_salary_comparison(salary: pd.DataFrame, employee: pd.DataFrame) -> pd.DataFra
    # Step 1: Extract month from pay_date
    salary['pay_month'] = pd.to_datetime(salary['pay_date']).dt.strftime('%Y-%m')

    # Step 2: Calculate the company average salary per month
    company_avg = salary.groupby('pay_month')['amount'].mean().reset_index(name='company_avg

    # Step 3: Merge employee and salary data
    merged_data = salary.merge(employee, on='employee_id')

    # Step 4: Calculate department average salary per month
    dept_avg = merged_data.groupby(['pay_month', 'department_id'])['amount'].mean().reset_in

    # Step 5: Merge department averages with company averages
    comparison = dept_avg.merge(company_avg, on='pay_month')

    # Step 6: Determine comparison result
    comparison['comparison'] = comparison.apply(
        lambda row: 'higher' if row['dept_avg'] > row['company_avg']
                    else 'lower' if row['dept_avg'] < row['company_avg']
                    else 'same',
        axis=1
    )

    # Step 7: Format the result
    result = comparison[['pay_month', 'department_id', 'comparison']]

    return result
```

Explanation:

- This solution compares each department's average monthly salary against the company-wide average for the same month.
- The approach follows these steps:

    1. Extract the month from the pay_date to group by month.
    2. Calculate the company-wide average salary for each month.
    3. Merge the Employee and Salary tables to associate departments with salaries.
    4. Calculate the average salary for each department in each month.

5. Merge the department averages with company averages to enable comparison.
6. Apply a formula to determine if each department's average is higher, lower, or the same as the company average.
7. Format the result with the required columns.

- The comparison logic uses a simple lambda function that compares the department average against the company average for the same month.
- Time complexity: O(n log n) where n is the number of salary records, due to the merging and grouping operations.
- Space complexity: O(d × m) where d is the number of departments and m is the number of months.
- This solution handles edge cases appropriately:
  - Departments that exactly match the company average are labeled as 'same'.
  - The comparison is done month-by-month, so a department could be higher in one month and lower in another.
- For large datasets, performance could be improved by:
  - Using more efficient merge strategies
  - Pre-filtering the data if possible
  - Using more vectorized operations instead of apply()
- The use of strftime('%Y-%m') ensures proper month-level grouping regardless of the specific days when salaries were paid.

618. Students Report By Geography

Description:

Table 11: Student

Column Name	Type
name	varchar
continent	varchar

There is no primary key for this table. It may contain duplicate rows. Each row of this table indicates the name of a student and the continent they came from.

A school has students from Asia, Europe, and America.

Write an SQL query to pivot the continent column in the Student table so that each name is sorted alphabetically and displayed underneath its corresponding continent. The output headers should be America, Asia, and Europe, respectively.

The test cases are generated so that the student number from America is not less than either Asia or Europe.

Solution:

```
import pandas as pd

def students_report_by_geography(student: pd.DataFrame) -> pd.DataFrame:
    # Step 1: Sort names within each continent
    america_students = student[student['continent'] == 'America'].sort_values('name')
    asia_students = student[student['continent'] == 'Asia'].sort_values('name')
    europe_students = student[student['continent'] == 'Europe'].sort_values('name')

    # Step 2: Assign row numbers to create alignment
    america_students.reset_index(drop=True, inplace=True)
    asia_students.reset_index(drop=True, inplace=True)
    europe_students.reset_index(drop=True, inplace=True)

    # Step 3: Select only the name column from each continent
    america_names = america_students[['name']].rename(columns={'name': 'America'})
    asia_names = asia_students[['name']].rename(columns={'name': 'Asia'})
    europe_names = europe_students[['name']].rename(columns={'name': 'Europe'})

    # Step 4: Concatenate horizontally to create pivot
    # Start with America (largest by problem statement)
    result = america_names.copy()

    # Join with Asia
    if not asia_names.empty:
        result = result.join(asia_names, how='left')
    else:
        result['Asia'] = None

    # Join with Europe
    if not europe_names.empty:
        result = result.join(europe_names, how='left')
    else:
        result['Europe'] = None

    return result
```

Explanation:

- This solution creates a pivot table that reorganizes student data from rows (grouped by continent) into columns (one for each continent).
- The approach follows these steps:

    1. Filter and sort students from each continent separately.
    2. Reset indices to create aligned positions across the continents.
    3. Extract and rename the name columns for each continent.
    4. Combine the continent dataframes horizontally using join operations.

- The problem specifies that America has at least as many students as Asia or Europe, so we start with America and join the others to it.
- Time complexity: O(n log n) where n is the number of students, due to the sorting operations.
- Space complexity: O(n) for the filtered datasets and result set.
- This solution handles cases where continents have different numbers of students:
    - When joining, the 'left' join ensures all rows from America are preserved.
    - For any missing positions in Asia or Europe, NULL values will appear in the result.
- For large datasets, performance could be improved by:
    - Using more vectorized operations
    - Potentially using a more efficient pivoting approach if available
- The explicit handling of empty continent dataframes ensures the solution works even if a continent has no students.
- This approach is straightforward and doesn't require complex reshaping operations, making it easier to understand and maintain.
- An alternative approach would be to use pandas' built-in pivot functions, but this direct approach gives more control over the sorting and alignment.