## SQL Hard

### 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:**

```sql
WITH base AS (
    SELECT d.name AS Department, e.name AS Employee, e.salary AS Salary, DENSE_RANK() OVER (
    FROM Employee e INNER JOIN Department d
        ON e.departmentId = d.id
    GROUP BY d.name, e.name, e.salary
)

SELECT Department, Employee, Salary
FROM base
WHERE rank <= 3;
```

**Explanation:**

- This query identifies the highest-paid employees in each department, specifically those with salaries in the top three within their department.
- The solution uses a Common Table Expression (CTE) with a window function to rank employees within each department based on salary.
- `DENSE_RANK() OVER (PARTITION BY e.departmentId ORDER BY e.salary DESC)` assigns a rank to each employee's salary within their department, with 1 being the highest.
- Unlike regular RANK(), DENSE_RANK() doesn't skip ranks for ties, which is important when multiple employees have the same salary.
- The main query then filters for employees with a rank of 3 or less, capturing the top three salary tiers in each department.
- The solution joins the Employee and Department tables to include department names in the output.
- Time complexity: O(n log n) where n is the number of employees, due to the sorting operations in the window function.
- Space complexity: O(n) for the CTE and result set.
- The DENSE_RANK() function 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.
- This approach efficiently handles departments of different sizes without requiring complex subqueries.
- For large datasets, indexes on departmentId and salary would improve join and sorting performance.

### 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 (column with unique values) 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 (column with unique values) 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:**

```
-- Write your PostgreSQL query statement below
WITH unbanned AS (
    SELECT *
    FROM Users
    WHERE banned = 'No'
),

cancelled AS (
    SELECT *
    FROM Trips
    WHERE LEFT(status, 3) LIKE 'can'
)

SELECT t.request_at AS "Day", ROUND(COUNT(c.id)::numeric / COUNT(t.id), 2) AS "Cancellation R
FROM Trips t LEFT JOIN cancelled c
    ON t.id = c.id
WHERE t.client_id IN (SELECT users_id FROM unbanned)
    AND t.driver_id IN (SELECT users_id FROM unbanned)
    AND t.request_at BETWEEN '2013-10-01' AND '2013-10-03'
GROUP BY t.request_at
```

**Explanation:**

- This query calculates the daily cancellation rate for taxi trips between Oct 1-3, 2013, where both client and driver are not banned.
- It creates two Common Table Expressions (CTEs) to simplify the logic: one for unbanned users and another for cancelled trips.
- The first CTE (`unbanned`) filters the Users table to include only those who aren't banned.
- The second CTE (`cancelled`) identifies all trips with status starting with "can" (cancelled trips).
- The main query joins all trips with the cancelled trips using a LEFT JOIN to preserve all eligible trips.
- The WHERE clause ensures both driver and client are unbanned by using IN subqueries against the unbanned CTE.
- Date filtering with BETWEEN ensures only trips from Oct 1-3, 2013 are included.
- The cancellation rate is calculated by dividing cancelled trips count by total trips count for each day.
- The ROUND function with type casting to numeric ensures the result is formatted to two decimal places.
- Time complexity: O(n + m) where n is the number of trips and m is the number of users, due to the table scans and joins.
- Space complexity: O(u + c) where u is the number of unbanned users and c is the number of cancelled trips.
- Using CTEs improves query readability and maintenance compared to multiple subqueries.
- The LEFT JOIN is crucial as it preserves all trips even if they weren't cancelled, ensuring correct denominator in the rate calculation.
- Using `LEFT(status, 3) LIKE 'can'` efficiently captures both cancellation types in a single condition.
- For large datasets, indexes on users_id, client_id, driver_id, and request_at would significantly improve query performance.
- Explicitly casting to numeric when calculating the rate prevents integer division issues that would result in truncated values.

### 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:**

```sql
-- Write your PostgreSQL query statement below
WITH company_counts AS (
    SELECT company, COUNT(*) as emp_count
    FROM Employee
    GROUP BY company
),
ranked_salaries AS (
    SELECT e.id, e.company, e.salary,
           ROW_NUMBER() OVER (PARTITION BY e.company ORDER BY e.salary, e.id) as row_rank,
           cc.emp_count
    FROM Employee e
    JOIN company_counts cc ON e.company = cc.company
)
SELECT id, company, salary
FROM ranked_salaries
WHERE
    (emp_count % 2 = 1 AND row_rank = (emp_count + 1) / 2) OR  -- Odd count: middle value
    (emp_count % 2 = 0 AND (row_rank = emp_count / 2 OR row_rank = emp_count / 2 + 1))  -- E
```

**Explanation:**

- This query identifies rows containing the median salary for each company, breaking ties by employee ID when sorting salaries.
- It uses a two-stage approach with Common Table Expressions (CTEs) to first count employees and then assign precise rankings.
- The first CTE calculates the total number of employees for each company, which is crucial for determining median positions.
- The second CTE employs ROW_NUMBER() to assign a sequential rank to each employee within their company, ordering first by salary and then by ID.
- For companies with an odd number of employees, only the middle row is selected (e.g., for 5 employees, the 3rd ranked employee).
- For companies with an even number of employees, both middle rows are included (e.g., for 6 employees, both the 3rd and 4th ranked employees).

- The WHERE clause uses a conditional expression to filter for the appropriate rows based on whether the employee count is odd or even.
- The formula (emp_count + 1) / 2 finds the median position for odd counts, while emp_count / 2 and emp_count / 2 + 1 identify both median positions for even counts.
- Time complexity is O(n log n) due to the sorting operations required within each company partition for ranking.
- Space complexity is O(n) where n is the number of employees, as we need to store all employee records with additional ranking information.
- Using ROW_NUMBER() instead of RANK() ensures unique positions even when salaries are identical, allowing the tie-breaking by ID to work correctly.
- The solution efficiently handles the multiple requirements: partitioning by company, ordering by salary, breaking ties by ID, and selecting median values.
- For large employee datasets, indexes on company and salary columns would significantly improve query performance.
- The modulo operation (%) provides a clean way to differentiate between companies with odd and even employee counts.
- This approach maintains data integrity by passing through the original employee IDs, company names, and salary values without manipulation.
- The solution is adaptable to handle additional requirements like excluding certain employees or adding weighted calculations.

### 571. Find Median Given Frequency of Numbers

**Description:**

Table 6: Numbers

| Column Name | Type |
|---|---|
| num frequency | int 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:**

```
WITH expanded AS (
    SELECT num
    FROM Numbers
```

```
    CROSS JOIN GENERATE_SERIES(1, frequency) as s
)

SELECT ROUND(
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY num)::numeric,
    1
) AS median
FROM expanded;
```

**Explanation:**

- This query calculates the median of a set of numbers where each number appears with a specific frequency.
- The solution uses a Common Table Expression (CTE) and PostgreSQL's specific functions:
  - The `expanded` CTE effectively "decompresses" the Numbers table using CROSS JOIN with GENERATE_SERIES to repeat each number according to its frequency value.
  - For example, if the row is (3, 5), the number 3 will appear 5 times in the expanded dataset.
  - GENERATE_SERIES(1, frequency) creates a sequence from 1 to the frequency value for each row.
- The main query uses `PERCENTILE_CONT(0.5)` window function, which is specifically designed to calculate percentiles (with 0.5 representing the median).
- The `WITHIN GROUP (ORDER BY num)` clause organizes the data in ascending order for the percentile calculation.
- The result is cast to numeric and rounded to one decimal place using the `ROUND` function.
- Time complexity: O(n) where n is the total count of numbers after expansion.
- Space complexity: O(n) for the expanded table.
- For large datasets with high frequencies, this approach may be memory-intensive as it materializes all repeated instances.
- An alternative approach for very large datasets would be to use a weighted percentile calculation, but PostgreSQL's PERCENTILE_CONT handles this scenario elegantly.
- The solution handles both odd and even counts correctly, as PERCENTILE_CONT automatically performs linear interpolation for even-count datasets.
- The query is compatible with PostgreSQL and takes advantage of its specific functions for percentile calculation.

### 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:**

```sql
-- Write your PostgreSQL query statement below
WITH base AS (
    SELECT id, month, salary
    , CASE WHEN month-1 = LAG(month, 1) OVER (PARTITION BY id ORDER BY month)
        THEN LAG(salary, 1) OVER (PARTITION BY id ORDER BY month) ELSE 0 END AS sal1
    , CASE WHEN month-2 = LAG(month, 2) OVER (PARTITION BY id ORDER BY month)
        THEN LAG(salary, 2) OVER (PARTITION BY id ORDER BY month) ELSE 0 END AS sal2
    FROM Employee
    GROUP BY id, month, salary
)

SELECT id, month, (salary + sal1 + sal2) AS "Salary"
FROM base
WHERE (id, month) NOT IN (SELECT id, MAX(month) FROM employee GROUP BY id)
ORDER BY id ASC, month DESC
```

**Explanation:**

- This query calculates a rolling 3-month salary sum for each employee, excluding their most recent month.
- The solution uses window functions and CASE statements to access previous months' salary values without self-joins.
- The `base` CTE retrieves each employee's monthly salary and uses `LAG` window functions to access salaries from previous months as long as they are consecutive
- `LAG(salary, 1)` gets the salary from 1 month before
- `LAG(salary, 2)` gets the salary from 2 months before
- `MAX(month)` identifies the most recent month for each employee, which will be excluded from the results
- The main query adds the current month's salary to the previous two months to calculate the 3-month cumulative sum.
- The `WHERE (id, month) NOT IN...` subquery excludes the most recent month for each employee from the results.
- Results are ordered by employee ID ascending and month descending, as specified in the requirements.
- Time complexity: O(n log n) where n is the number of employee-month records, due to the sorting in window functions.
- Space complexity: O(n) for the CTE and result set.
- This solution handles edge cases elegantly:
  - For the first month of employment, only that month's salary is counted (as previous months are 0).
  - 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 is efficient for large datasets as it avoids multiple self-joins that would otherwise be needed.
- For optimal performance, indexes on (id, month) would be beneficial.
- Note that this solution assumes that months are represented as integers (e.g., 1-12) rather than dates.
- For real-world scenarios with large datasets, partitioning the employee table by id could further improve performance.

### 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:**

```sql
WITH base AS (
    SELECT id, visit_date, people, id - ROW_NUMBER() OVER (ORDER BY id) AS group_id
    FROM Stadium
    WHERE people >= 100
)

SELECT id, visit_date, people
FROM base
WHERE group_id IN (
    SELECT group_id
    FROM base
    GROUP BY group_id
    HAVING COUNT(group_id) >= 3
)
ORDER BY visit_date ASC
```

**Explanation:**

- This query identifies consecutive days with stadium attendance of 100 or more people.
- The solution uses a "gaps and islands" technique to identify consecutive sequences in the data.
- The first CTE, `high_traffic`, filters for days with 100+ visitors and creates a grouping identifier for consecutive days.
- The key insight is using `id - ROW_NUMBER() OVER (ORDER BY id)` to create a constant value for consecutive sequences.
- When IDs are consecutive, subtracting their row numbers (which also increment by 1) produces the same value, creating a "group ID" for consecutive records.
- For example, if IDs 3, 4, 5 have row numbers 1, 2, 3, then (3-1), (4-2), (5-3) all equal 2, identifying them as part of the same group.
- The main query then selects only records from groups that have 3 or more consecutive days of high traffic.
- We filter using a subquery that counts records in each group and keeps only those with counts >= 3.

- Results are ordered 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 CTE and result set.
- This solution efficiently handles the "consecutive" requirement without complex self-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, an index on the people column would improve performance of the initial filtering.
- The approach is database-agnostic and works well in PostgreSQL, MySQL, SQL Server, and other major SQL dialects.
- An alternative approach could use window functions like LAG() and LEAD() to check adjacent days, but this "gaps and islands" technique is more elegant for identifying consecutive sequences.
- For very large datasets, partitioning by date ranges could improve query performance.

### 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:**

```
with avgs as (
  select
    to_char(pay_date, 'YYYY-MM') as pay_month,
    department_id,
    avg(amount) over (partition by department_id,to_char(pay_date, 'YYYY-MM')) as dept_avg,
    avg(amount) over (partition by to_char(pay_date, 'YYYY-MM')) as co_avg
  from Salary s join Employee e using (employee_id)
)

select
  distinct pay_month,
  department_id,
  CASE
    WHEN dept_avg > co_avg THEN 'higher'
    WHEN dept_avg < co_avg THEN 'lower'
    ELSE 'same'
  END as comparison
from avgs
```

**Explanation:**

- Need an explanation

### 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:**

```
-- Write your PostgreSQL query statement below
WITH ranked_students AS (
    SELECT
        name,
        continent,
        ROW_NUMBER() OVER(PARTITION BY continent ORDER BY name) AS row_num
    FROM Student
)

SELECT
    MAX(CASE WHEN continent = 'America' THEN name END) AS "America",
    MAX(CASE WHEN continent = 'Asia' THEN name END) AS "Asia",
    MAX(CASE WHEN continent = 'Europe' THEN name END) AS "Europe"
FROM ranked_students
GROUP BY row_num
ORDER BY "America", "Asia",  "Europe";
```

**Explanation:**

- This query creates a pivot table that reorganizes student data from rows (grouped by continent) into columns (one for each continent).
- The solution uses a two-step approach with a CTE and conditional aggregation to achieve the pivoting.
- The `ranked_students` CTE first assigns sequential row numbers to students within each continent, ordered alphabetically by name.
- This is done using the `ROW_NUMBER() OVER(PARTITION BY continent ORDER BY name)` window function.
- For example, if America has students ['Jane', 'John', 'Mike'] after sorting, they would get row numbers 1, 2, and 3 within the America partition.
- The main query then uses conditional aggregation with `CASE` expressions to pivot the data:

  - Each `CASE WHEN continent = X THEN name END` returns the name when the continent matches, otherwise NULL.
  - The `MAX` function effectively selects the non-NULL value in each group (there will be at most one non-NULL value per continent in each row_num group).

- By grouping by row_num, we align students across continents based on their position within each continent's alphabet-sorted list.
- Time complexity: O(n log n) where n is the number of students, due to the sorting operations.
- Space complexity: O(n) for the CTE and result set.
- This approach handles the case where continents have different numbers of students - any "missing" positions will show as NULL in the results.
- The query automatically adjusts to any number of students and works correctly regardless of data distribution.
- For large datasets, indexes on the continent and name columns would improve performance.
- This solution is particularly elegant as it doesn't require hard-coding the names of all possible continents in the query logic.
- An alternative approach could use the CROSSTAB function in PostgreSQL, but this method is more portable across different SQL dialects.
- Note that the problem specifies that America will always have at least as many students as Asia or Europe, which ensures the pivot will have meaningful results for all rows.
- For performance reasons with very large datasets, consider materializing the ranked_students CTE as a temporary table.

### 1097. Gameplay Analysis V

**Description:**

Table 12: Activity

| Column Name | Type |
|---|---|
| player_id | int int date |
| device_id | int |
| event_date | |
| games_played | |

(player_id, event_date) is the primary key of this table. This table shows the activity of players of some games. Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on someday using some device.

The install date of a player is the first login day of that player.

We define day 1 retention of some date X to be the number of players whose install date is X and they logged in again on the day right after X, divided by the number of players whose install date is X, rounded to 2 decimal places.

Write an SQL query to report for each install date, the number of players that installed the game on that day, and the day 1 retention.

**Solution:**

```
WITH install_dates AS (
    SELECT player_id, MIN(event_date) AS install_dt
    FROM Activity
    GROUP BY player_id
),

day1_retention AS (
    SELECT i.install_dt, COUNT(DISTINCT i.player_id) AS installs
        , COUNT(DISTINCT a.player_id) AS retained
    FROM install_dates i
    LEFT JOIN Activity a ON i.player_id = a.player_id
        AND a.event_date = i.install_dt + 1
    GROUP BY i.install_dt
)

SELECT install_dt,installs
    , ROUND(COALESCE(retained::NUMERIC / installs, 0), 2) AS Day1_retention
FROM day1_retention
ORDER BY install_dt;
```

**Explanation:**

- This query calculates "day 1 retention rate" which is the percentage of players who return to play on the day immediately after their first login (install date).
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The first CTE, `install_dates`, identifies each player's first login (install date) using the MIN function.
  - The second CTE, `day1_retention`, counts both total installs and the number of retained players for each install date.
- For each install date, we count:
  - The number of distinct players who installed on that date (total installs)
  - The number of these players who returned the very next day (retained players)

- We use a LEFT JOIN to ensure we capture all install dates, even if no players returned the next day.
- The main query calculates the day 1 retention rate by dividing retained count by install count.
- The COALESCE function handles cases where there are no retained players (avoiding division by zero).
- We convert to NUMERIC type and use ROUND to format the result to 2 decimal places as required.
- Time complexity: O(n) where n is the number of activity records, as we scan the table once to find install dates and once to check for retention.
- Space complexity: O(p) where p is the number of players, as we store each player's install date.
- This approach efficiently handles players who never return after installation and multiple game sessions on the same day.
- For large datasets, indexes on player_id and event_date would significantly improve performance.

### 1127. User Purchase Platform

**Description:**

Table 13: Spending

| Column Name | Type |
|---|---|
| user_id | int date |
| spend_date | enum int |
| platform amount | |

The table records the user spending data on different platforms (mobile, desktop) on different dates. (user_id, spend_date, platform) is the primary key of this table. The platform column is an ENUM type of ('mobile', 'desktop').

Write an SQL query to find the total number of users and the total amount spent using the mobile only, desktop only, and both platforms for each date.

The query result format is in the following example:

Spending table:

| user_id | spend_date | platform | amount |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 1 2 2 3 3 | 2019-07-01 | mobile | 100 100 |
| | 2019-07-01 | desktop | 100 100 |
| | 2019-07-01 | mobile | 100 100 |
| | 2019-07-02 | mobile | |
| | 2019-07-01 | desktop | |
| | 2019-07-02 | desktop | |

Result table:

| spend_date | platform | total_amount | total_users |
|---|---|---|---|
| 2019-07-01 | desktop | 100 100 200 100 | 1 1 1 1 1 0 |
| 2019-07-01 | mobile both | 100 0 | |
| 2019-07-01 | desktop | | |
| 2019-07-02 | mobile both | | |
| 2019-07-02 | | | |
| 2019-07-02 | | | |

**Solution:**

```
-- Write your PostgreSQL query statement below
WITH user_platform AS (
    SELECT spend_date, user_id
    , SUM(CASE WHEN platform = 'mobile' THEN amount ELSE 0 END) AS mobile_amount
    , SUM(CASE WHEN platform = 'desktop' THEN amount ELSE 0 END) AS desktop_amount
    FROM Spending
    GROUP BY spend_date, user_id
),

user_platform_type AS (
    SELECT spend_date, user_id
        , CASE WHEN mobile_amount > 0 AND desktop_amount > 0 THEN 'both'
            WHEN mobile_amount > 0 THEN 'mobile' ELSE 'desktop' END AS platform
        , mobile_amount + desktop_amount AS amount
    FROM user_platform
),

all_dates_platforms AS (
    SELECT DISTINCT spend_date, p.*
    FROM Spending CROSS JOIN (
        SELECT 'mobile' AS platform
```

```
        UNION ALL SELECT 'desktop'
        UNION ALL SELECT 'both'
    ) p
)

SELECT a.spend_date, a.platform, COALESCE(SUM(u.amount), 0) AS total_amount
    , COUNT(u.user_id) AS total_users
FROM all_dates_platforms a LEFT JOIN  user_platform_type u
    ON a.spend_date = u.spend_date AND a.platform = u.platform
GROUP BY a.spend_date, a.platform
```

**Explanation:**

- This query analyzes user spending across different platforms (mobile, desktop, or both) for each date.
- The solution uses a multi-step approach with three CTEs to transform and aggregate the data:
  - The first CTE, `user_platform`, aggregates spending by user and date, using conditional sums to separate mobile and desktop amounts.
  - The second CTE, `user_platform_type`, categorizes each user on each date as using 'mobile', 'desktop', or 'both' platforms.
  - The third CTE, `all_dates_platforms`, creates a complete cartesian product of all dates with all three platform types, ensuring we have rows for all combinations.
- The main query joins these prepared data sets to calculate the total amount and total users for each date-platform combination.
- We use LEFT JOIN to ensure all date-platform combinations appear in the result, even those with zero users.
- COALESCE handles NULL values for total_amount when there are no users in a specific category.
- The COUNT(user_id) function counts only non-NULL values, giving us the correct user count.
- The results are ordered by date and then by platform in a specific order (both, desktop, mobile).
- Time complexity: O(n log n) where n is the number of spending records, due to the grouping and sorting operations.
- Space complexity: O(d $\times$ p) where d is the number of distinct dates and p is the number of platform types (3).
- The CROSS JOIN ensures we have complete reporting for all possible platform categories, including combinations with zero users.
- For large datasets, indexes on user_id, spend_date, and platform would improve performance.

### 1159. Market Analysis II

**Description:**

Table 16: Users

| Column Name | Type |
|---|---|
| user_id join_date | int date |
| favorite_brand | varchar |

user_id is the primary key of this table. This table has the info of the users of an online shopping website where users can sell and buy items.

Table 17: Orders

| Column Name | Type |
|---|---|
| order_id | int date int |
| order_date item_id | int int |
| buyer_id seller_id | |

order_id is the primary key of this table. item_id is a foreign key to the Items table. buyer_id and seller_id are foreign keys to the Users table.

Table 18: Items

| Column Name | Type |
|---|---|
| item_id | int varchar |
| item_brand | |

item_id is the primary key of this table.

Write an SQL query to find for each user whether the brand of their second item (ordered by date) sold is their favorite brand. If a user sold less than two items, report the answer for that user as "no" (case-insensitive).

Return the result table in any order.

**Solution:**

```sql
-- Write your PostgreSQL query statement below
WITH ranked_sales AS (
    SELECT seller_id, item_id
        , ROW_NUMBER() OVER (PARTITION BY seller_id ORDER BY order_date) AS sale_rank
    FROM Orders
),

second_sales AS (
    SELECT rs.seller_id, i.item_brand
    FROM ranked_sales rs JOIN Items i
        ON rs.item_id = i.item_id
    WHERE rs.sale_rank = 2
)

SELECT
    u.user_id AS seller_id
    , CASE WHEN ss.item_brand = u.favorite_brand THEN 'yes' ELSE 'no'
    END AS "2nd_item_fav_brand"
FROM Users u LEFT JOIN  second_sales ss
        ON u.user_id = ss.seller_id
ORDER BY u.user_id;
```

**Explanation:**

- This query determines if a user's second sold item matches their favorite brand.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The first CTE, `ranked_sales`, assigns a sequential rank to each sale by a seller, ordered by sale date using ROW_NUMBER().
  - The second CTE, `second_sales`, joins the ranked sales with the Items table to get the brand of each seller's second sale.
- The main query joins the Users table with the second_sales CTE to compare each user's favorite brand with their second sale's brand.
- We use LEFT JOIN to include all users, even those with fewer than two sales.
- The CASE expression handles the comparison, returning 'yes' when the brands match and 'no' otherwise.
- When a user has fewer than two sales, the second_sales join will return NULL, and the CASE statement will return 'no' as required.
- Time complexity: O(n log n) where n is the number of orders, due to the sorting operation for ranking sales.
- Space complexity: O(s) where s is the number of sellers who have made at least one sale.

- The solution efficiently handles users who have made fewer than two sales and correctly identifies the chronological second sale.
- For large datasets, indexes on seller_id and order_date in the Orders table would improve performance.
- Breaking down the logic into CTEs improves readability compared to using complex nested subqueries.

### 1194. Tournament Winners

**Description:**

Table 19: Players

| Column Name | Type |
|---|---|
| player_id | int int |
| group_id | |

player_id is the primary key of this table. Each row indicates the group of each player.

Table 20: Matches

| Column Name | Type |
|---|---|
| match_id | int int int |
| first_player | int int |
| second_player | |
| first_score | |
| second_score | |

match_id is the primary key of this table. Each row is a record of a match, first_player and second_player contain the player_id of each player. first_score and second_score contain the number of points of the first_player and second_player respectively. You may assume that, in each match, players belong to the same group.

The winner in each group is the player who scored the maximum total points within the group. In the case of a tie, the lowest player_id wins.

Write an SQL query to find the winner in each group.

**Solution:**

```sql
WITH player_scores AS (
    -- Calculate scores from first_player perspective
    SELECT first_player AS player_id, first_score AS score
    FROM Matches

    UNION ALL
    -- Calculate scores from second_player perspective
    SELECT second_player AS player_id, second_score AS score
    FROM Matches
),

total_scores AS (
    SELECT p.player_id, p.group_id
        , COALESCE(SUM(ps.score), 0) AS total_score
    FROM Players p
    LEFT JOIN player_scores ps ON p.player_id = ps.player_id
    GROUP BY p.player_id, p.group_id
),

ranked_scores AS (
    SELECT player_id, group_id, total_score
        , RANK() OVER (PARTITION BY group_id ORDER BY total_score DESC, player_id) AS player_
    FROM total_scores
)

SELECT group_id, player_id
FROM ranked_scores
WHERE player_rank = 1;
```

**Explanation:**

- This query identifies the winner in each player group based on total points scored.
- The solution uses a three-step approach with Common Table Expressions (CTEs):
  - The first CTE, `player_scores`, collects all scores for all players using UNION ALL to combine points scored as both first_player and second_player.
  - The second CTE, `total_scores`, calculates the total score for each player by joining Players with the aggregated scores and summing them.
  - The third CTE, `ranked_scores`, ranks players within each group using the RANK() window function, ordering by total_score (DESC) and breaking ties with player_id.
- The main query selects only the top-ranked player from each group (player_rank = 1).

- Time complexity: O(n log n) where n is the number of players and matches, due to the sorting operations.
- Space complexity: O(p) where p is the number of players, as we store scores for each player.
- COALESCE handles players who haven't played any matches (giving them a score of 0).
- The LEFT JOIN ensures all players are included in the result, even if they haven't played any matches.
- RANK() is crucial for handling ties according to the specified tiebreaker rule (lowest player_id).
- The solution correctly handles cases where a player may appear as both first_player and second_player in different matches.
- For large datasets, indexes on player_id in both tables would improve performance.

### 1225. Report Contiguous Dates

**Description:**

Table 21: Failed

| Column Name | Type |
|---|---|
| fail_date | date |

Primary key for this table is fail_date. Failed table contains the days of failed tasks.

Table 22: Succeeded

| Column Name | Type |
|---|---|
| success_date | date |

Primary key for this table is success_date. Succeeded table contains the days of succeeded tasks.

A system is running one task every day. Every task is independent of the previous tasks. The tasks can fail or succeed.

Write an SQL query to generate a report of period_state for each continuous interval of days in the period from 2019-01-01 to 2019-12-31.

period_state is 'failed' if tasks in this interval failed or 'succeeded' if tasks in this interval succeeded. Interval of days are retrieved as start_date and end_date.

Return the result table ordered by start_date.

**Solution:**

```sql
-- Write your PostgreSQL query statement below
WITH base AS (
    SELECT fail_date AS "dates", 'failed' AS "outcome"
    FROM Failed
    WHERE fail_date BETWEEN '2019-01-01' AND '2019-12-31'
    UNION
    SELECT success_date AS "dates", 'succeeded' AS "outcome"
    FROM Succeeded
    WHERE success_date BETWEEN '2019-01-01' AND '2019-12-31'
    ORDER BY "dates"
),

periods AS (
    SELECT dates, outcome, ROW_NUMBER() OVER (ORDER BY dates) - ROW_NUMBER() OVER (PARTITION
    FROM base
)

SELECT outcome AS period_state, MIN(dates) AS "start_date", MAX(dates) AS "end_date"
FROM periods
GROUP BY "period_state", group_id
ORDER BY "start_date"
```

**Explanation:**

- This query identifies continuous intervals of 'failed' or 'succeeded' days throughout 2019.
- The solution uses a "gaps and islands" technique with Common Table Expressions (CTEs):
  - The first CTE, `all_dates`, combines dates from both tables with their respective states ('failed' or 'succeeded').
  - The second CTE, `ranked_dates`, implements the core of the technique by calculating a constant value for consecutive dates.
- The key insight is calculating `date - ROW_NUMBER()` which creates the same value for consecutive dates within each state:
  - For continuous sequences, this difference remains constant
  - When there's a gap, the difference changes, creating a new group_id
- The main query then finds the minimum and maximum dates for each group_id within each state to identify continuous periods.
- Time complexity: O(n log n) where n is the total number of dates in both tables, due to the sorting operations.
- Space complexity: O(n) for the combined dates and result set.

- This approach efficiently handles interleaved periods of success and failure and multiple distinct periods of the same state.
- The "gaps and islands" technique is particularly elegant for this problem as it naturally identifies continuous sequences.
- Results are ordered by start_date as required, showing a chronological view of how the system alternated between success and failure.
- For large datasets, indexes on date columns would improve performance.

### 1336. Number of Transactions per Visit

**Description:**

Table 23: Visits

| Column Name | Type |
|---|---|
| user_id visit_date | int date |

(user_id, visit_date) is the primary key for this table. Each row of this table indicates that user_id has visited the bank in visit_date.

Table 24: Transactions

| Column Name | Type |
|---|---|
| user_id | int date int |
| transaction_date | |
| amount | |

There is no primary key for this table, it may contain duplicates. Each row of this table indicates that user_id has done a transaction of amount in transaction_date. It is guaranteed that the user has visited the bank in the transaction_date.(i.e The Visits table contains (user_id, transaction_date) in one row)

Write an SQL query to find how many users visited the bank and didn't do any transactions, how many visited the bank and did one transaction and so on.

The result table will contain two columns: - transactions_count which is the number of transactions done in one visit. - visits_count which is the corresponding number of visits with transactions_count transactions.

Return the result table ordered by transactions_count.

**Solution:**

```sql
-- Write your PostgreSQL query statement below
WITH transactions_per_visit AS (
    SELECT v.user_id, v.visit_date, COUNT(t.transaction_date) AS t_count
    FROM Visits v LEFT JOIN Transactions t
        ON v.user_id = t.user_id AND v.visit_date = t.transaction_date
    GROUP BY v.user_id, v.visit_date
),

max_transactions AS (
    SELECT MAX(t_count) AS max_t
    FROM transactions_per_visit
),

t_counts AS (
    SELECT t_count, COUNT(*) AS v_count
    FROM transactions_per_visit
    GROUP BY t_count
),

all_counts AS (
    SELECT GENERATE_SERIES(0, max_t) AS transactions_count
    FROM max_transactions
)

SELECT a.transactions_count, COALESCE(tc.v_count, 0) AS visits_count
FROM all_counts a LEFT JOIN t_counts tc
    ON a.transactions_count = tc.t_count
ORDER BY a.transactions_count
```

**Explanation:**

- This query calculates how many users made different numbers of transactions per visit to the bank, including those who made no transactions.
- The solution uses a multi-step approach with several Common Table Expressions (CTEs):
  - The first CTE, `transactions_per_visit`, counts the number of transactions each visitor made on each visit date.
  - Using LEFT JOIN ensures we include visits with zero transactions.
  - The second CTE, `max_transactions`, finds the maximum number of transactions made in any single visit.
  - The third CTE, `transaction_counts`, counts visits grouped by their transaction count.

  - The fourth CTE, `all_counts`, generates a sequence of numbers from 0 to the maximum transaction count.

- The main query joins the transaction counts with the generated sequence to ensure all possible transaction counts appear in the results.
- COALESCE(tc.visits_count, 0) handles transaction counts that didn't occur in the data, displaying a count of 0.
- Results are ordered by transactions_count as specified in the requirements.
- Time complexity: O(n log n) where n is the number of visits, due to the grouping and sorting operations.
- Space complexity: O(m) where m is the maximum number of transactions made in any visit.
- The use of PostgreSQL's generate_series() function elegantly creates a complete series of transaction counts.
- For large datasets, indexes on user_id and visit_date/transaction_date would improve join performance.
- An important edge case handled by this solution is when there are gaps in the transaction counts (e.g., some visitors made 0, 1, or 3 transactions but none made exactly 2).
- The LEFT JOIN in the main query ensures these gaps are filled with zeros in the final output.
- The COALESCE function in the max_transactions CTE handles the edge case where there are no transactions at all.
- This approach is particularly efficient as it avoids complex recursive CTEs or self-joins that might be needed in other SQL dialects.

### 1369. Get the Second Most Recent Activity

**Description:**

Table 25: UserActivity

| Column Name | Type |
|---|---|
| username activity | varchar |
| startDate endDate | varchar date |
| | date |

There is no primary key for this table. It may contain duplicates. This table contains information about the activity performed by each user in a period of time. A person with username performed an activity from startDate to endDate.

Write an SQL query to show the second most recent activity of each user.

If the user only has one activity, return that one. A user cannot perform more than one activity at the same time.

Return the result table in any order.

**Solution:**

```sql
-- Write your PostgreSQL query statement below
WITH base AS (
    SELECT *, RANK() OVER (PARTITION BY username ORDER BY startDate DESC) AS act_rank
        , COUNT(*) OVER (PARTITION BY username) AS act_count
    FROM UserActivity
    GROUP BY username, activity, startDate, endDate
)

SELECT username, activity, startdate AS "startDate", enddate AS "endDate"
FROM base
WHERE act_rank = 2 OR act_count = 1
ORDER BY username
```

**Explanation:**

- This query finds the second most recent activity for each user, or their only activity if they have just one.
- The solution uses a single Common Table Expression (CTE) with window functions to rank activities and count them for each user:
  - `ROW_NUMBER() OVER (PARTITION BY username ORDER BY startDate DESC)` assigns a rank to each activity based on recency.
  - `COUNT(*) OVER (PARTITION BY username)` counts the total number of activities for each user.
- The main query then filters the results with two conditions:
  - For users with only one activity (activity_count = 1), return that activity.
  - For users with multiple activities, return the second most recent (activity_rank = 2).
- Time complexity: O(n log n) where n is the number of activities, due to the sorting operations.
- Space complexity: O(n) for the CTE and result set.
- The problem statement guarantees that users cannot perform more than one activity at the same time, which simplifies the solution.
- ROW_NUMBER() is the appropriate window function here as we need the exact second most recent activity.
- The startDate is used for ordering to determine recency, as specified in the problem.
- For large datasets, an index on username and startDate would improve performance.

- The solution handles the edge case of users with only one activity by explicitly checking for this condition.
- This approach is more efficient than using subqueries to filter for the second most recent activity.
- The WHERE clause uses a logical OR to combine the two conditions, making the query more readable than using a CASE expression.
- If the activities table is very large, this solution scales well as it only requires a single pass through the data.
- The window functions eliminate the need for self-joins or multiple aggregations, resulting in a more efficient query.

### 1384. Total Sales Amount by Year

**Description:**

Table 26: Product

| Column Name | Type |
|---|---|
| product_id | int varchar |
| product_name | |

product_id is the primary key for this table. product_name is the name of the product.

Table 27: Sales

| Column Name | Type |
|---|---|
| product_id period_start | int date |
| period_end | date int |
| average_daily_sales | |

product_id is the primary key for this table. period_start and period_end indicate the start and end date for the sales period, and both dates are inclusive. The average_daily_sales column holds the average daily sales amount of the items for the period.

Write an SQL query to report the total sales amount of each item for each year, with corresponding product_name, product_id, product_name, and report_year.

Return the result table ordered by product_id and report_year.

**Solution:**

```
WITH RECURSIVE year_ranges AS (
    SELECT
        EXTRACT(YEAR FROM MIN(period_start)) AS start_year,
        EXTRACT(YEAR FROM MAX(period_end)) AS end_year
    FROM
        Sales
),
years AS (
    SELECT start_year AS year
    FROM year_ranges

    UNION ALL

    SELECT year + 1
    FROM years
    WHERE year < (SELECT end_year FROM year_ranges)
),
sales_by_year AS (
    SELECT
        s.product_id,
        p.product_name,
        y.year::int AS report_year,
        CASE
            WHEN EXTRACT(YEAR FROM s.period_start) = EXTRACT(YEAR FROM s.period_end) THEN
                s.average_daily_sales * (s.period_end - s.period_start + 1)
            WHEN y.year = EXTRACT(YEAR FROM s.period_start) THEN
                s.average_daily_sales * (DATE_TRUNC('year', s.period_start + interval '1 yea
            WHEN y.year = EXTRACT(YEAR FROM s.period_end) THEN
                s.average_daily_sales * (s.period_end - DATE_TRUNC('year', s.period_end) + 1
            ELSE
                s.average_daily_sales * (DATE_TRUNC('year', MAKE_DATE(y.year::int + 1, 1, 1)
        END AS total_amount
    FROM
        Sales s
    JOIN
        Product p ON s.product_id = p.product_id
    JOIN
        years y ON y.year BETWEEN EXTRACT(YEAR FROM s.period_start) AND EXTRACT(YEAR FROM s.p
)

SELECT
    product_id,
```

```
    product_name,
    report_year::text AS report_year,
    total_amount
FROM
    sales_by_year
ORDER BY
    product_id, report_year;
```

**Explanation:**

- This query calculates the total sales amount for each product, broken down by year, for periods that may span multiple years.
- The solution uses a three-step approach with recursive Common Table Expressions (CTEs):
  - The first CTE, `year_ranges`, determines the overall range of years covered by the sales data.
  - The second CTE, `years`, recursively generates a list of all years in that range.
  - The third CTE, `sales_by_year`, calculates sales for each product in each year using a complex CASE expression.
- The CASE expression handles four scenarios:
  - When the entire sales period is within a single year, simply multiply average_daily_sales by the total number of days.
  - When calculating for the start year of a multi-year period, compute days from period_start to the end of that year.
  - When calculating for the end year of a multi-year period, compute days from the start of that year to period_end.
  - For complete years in the middle of a period, use the full year's days (accounting for leap years).
- The main query formats the results and orders them by product_id and report_year.
- Time complexity: O(n × y) where n is the number of sales records and y is the number of years in the date range.
- Space complexity: O(n × y) for the generated result set with each product-year combination.
- This solution properly handles sales periods spanning multiple years, a key challenge of this problem.
- The recursive CTE to generate years is more efficient than using a calendar table for small to medium datasets.
- DATE_TRUNC and MAKE_DATE functions handle the year boundaries precisely, accounting for leap years.

- Adding 1 in the date difference calculations ensures inclusive date ranges as specified in the problem.
- The conversion of year to text in the final query ensures consistent formatting of the report_year column.
- For large datasets with sales periods spanning many years, this solution may create many rows, but remains efficient.
- An index on product_id and the period_start/period_end columns would improve join performance.
- This approach correctly handles edge cases like sales periods starting or ending on year boundaries.
- The solution works for any date range, not just specific years, making it adaptable to different datasets.

### 1412. Find the Quiet Students in All Exams

**Description:**

Table 28: Student

| Column Name | Type |
|---|---|
| student_id student_name | int varchar |

student_id is the primary key for this table. student_name is the name of the student.

Table 29: Exam

| Column Name | Type |
|---|---|
| exam_id | int int int |
| student_id score | |

(exam_id, student_id) is the primary key for this table. Each row of this table indicates that the student with student_id had a score of score in the exam with id exam_id.

A quiet student is the one who took at least one exam and did not score the highest or the lowest score in any of the exams.

Write an SQL query to report the students (student_id, student_name) being quiet in all exams. Don't return the student who has never taken any exam.

Return the result table ordered by student_id.

**Solution:**

```sql
WITH ranking AS (
    SELECT
        e.exam_id,
        e.student_id,
        e.score,
        RANK() OVER (PARTITION BY e.exam_id ORDER BY e.score ASC) AS low_rank,
        RANK() OVER (PARTITION BY e.exam_id ORDER BY e.score DESC) AS high_rank,
        COUNT(*) OVER (PARTITION BY e.exam_id) AS exam_count
    FROM
        Exam e
),
extremes AS (
    SELECT DISTINCT
        student_id
    FROM
        ranking
    WHERE
        (low_rank = 1 OR high_rank = 1)
        AND exam_count > 1  -- Only consider exams with multiple students
)

SELECT
    s.student_id,
    s.student_name
FROM
    Student s
JOIN
    Exam e ON s.student_id = e.student_id
LEFT JOIN
    extremes ex ON s.student_id = ex.student_id
WHERE
    ex.student_id IS NULL
GROUP BY
    s.student_id, s.student_name
ORDER BY
    s.student_id;
```

**Explanation:**

- This query finds "quiet students" who took at least one exam but never scored the highest or lowest in any exam they took.
- The solution uses a two-step approach with Common Table Expressions (CTEs):

  - The first CTE, `ranking`, assigns both low and high ranks to each student's score within each exam using RANK().
  - The second CTE, `extremes`, identifies students who scored either the highest or lowest in any exam.

- RANK() is used instead of ROW_NUMBER() to handle ties properly—multiple students can share the highest or lowest score.
- The main query joins the Student table with Exam to ensure we only include students who took at least one exam.
- The LEFT JOIN with the extremes CTE, combined with WHERE ex.student_id IS NULL, filters out students who ever had extreme scores.
- The GROUP BY ensures each student appears only once in the results, even if they took multiple exams.
- Time complexity: O(n log n) where n is the number of exam records, due to the sorting operations.
- Space complexity: O(n) for the CTEs and result set.
- The solution properly handles edge cases:
  - Exams with only one student are handled by checking exam_count $> 1$ in the extremes CTE.
  - Students who tied for highest/lowest scores are all excluded, as required.
- The COUNT(*) OVER window function efficiently counts students in each exam without additional grouping.
- For large datasets, indexes on student_id and exam_id would improve join performance.
- The DISTINCT in the extremes CTE ensures that if a student scored highest in one exam and lowest in another, they're only counted once.
- This solution efficiently excludes students who never took any exam by using an INNER JOIN with Exam in the main query.
- The query follows the requirement to order results by student_id.
- An alternative approach could use NOT EXISTS subqueries, but the CTE method is more readable and often performs better.

### 1479. Sales by Day of the Week

**Description:**

Table 30: Orders

| Column Name | Type |
|---|---|
| order_id | int int date |
| customer_id | varchar int |
| order_date item_id | |
| quantity | |

(order_id, item_id) is the primary key for this table. This table contains information on the orders placed. order_date is the date when the order was placed. item_id is the id of the item. quantity is the number of items ordered.

Table 31: Items

| Column Name | Type |
|---|---|
| item_id item_name | varchar |
| item_category | varchar |
| | varchar |

item_id is the primary key for this table. item_name is the name of the item. item_category is the category of the item.

Write an SQL query to report how many units in each category were ordered on each day of the week.

Return the result table ordered by category.

**Solution:**

```sql
WITH daily_sales AS (
    SELECT
        i.item_category AS Category,
        EXTRACT(DOW FROM o.order_date) AS day_of_week,
        COALESCE(SUM(o.quantity), 0) AS total_quantity
    FROM
        Items i
    LEFT JOIN
        Orders o ON i.item_id = o.item_id
    GROUP BY
        i.item_category, EXTRACT(DOW FROM o.order_date)
)

SELECT
    Category,
    COALESCE(SUM(CASE WHEN day_of_week = 0 THEN total_quantity ELSE 0 END), 0) AS Monday,
    COALESCE(SUM(CASE WHEN day_of_week = 1 THEN total_quantity ELSE 0 END), 0) AS Tuesday,
    COALESCE(SUM(CASE WHEN day_of_week = 2 THEN total_quantity ELSE 0 END), 0) AS Wednesday,
    COALESCE(SUM(CASE WHEN day_of_week = 3 THEN total_quantity ELSE 0 END), 0) AS Thursday,
    COALESCE(SUM(CASE WHEN day_of_week = 4 THEN total_quantity ELSE 0 END), 0) AS Friday,
    COALESCE(SUM(CASE WHEN day_of_week = 5 THEN total_quantity ELSE 0 END), 0) AS Saturday,
    COALESCE(SUM(CASE WHEN day_of_week = 6 THEN total_quantity ELSE 0 END), 0) AS Sunday
```

```
FROM
    daily_sales
GROUP BY
    Category
ORDER BY
    Category;
```

**Explanation:**

- This query reports the number of units ordered for each item category, broken down by day of the week.
- The solution uses a pivot table approach with a Common Table Expression (CTE):
  - The `daily_sales` CTE calculates the total quantity of items ordered in each category for each day of the week.
  - EXTRACT(DOW FROM order_date) converts the order date to a day of week number (0-6, where 0 is Sunday).
  - The LEFT JOIN ensures all item categories are included, even if they have no orders.
- The main query then pivots the data using CASE expressions and SUM aggregation:
  - Each day gets its own column by filtering on the day_of_week value.
  - COALESCE handles NULL values by converting them to 0 as required.
- Results are ordered by Category as specified in the requirements.
- Time complexity: O(n log n) where n is the number of order records, due to the grouping and sorting operations.
- Space complexity: O(c $\times$ 7) where c is the number of item categories, as we aggregate into a fixed structure.
- Note: PostgreSQL's EXTRACT(DOW) function numbers days from 0 (Sunday) to 6 (Saturday), but the query adapts by mapping these values to the correct day names.
- The solution handles cases where a category has no orders on specific days by using COALESCE to convert NULLs to 0.
- For large datasets, indexes on item_id in both tables would improve join performance.
- This approach effectively transforms row-level data into a column-based report format, a common requirement in business reporting.
- Using a CTE first to aggregate the data and then pivoting it is often more efficient than pivoting directly from the base tables.
- The solution correctly accounts for multiple orders of the same item on the same day by summing quantities.
- For very large datasets, consider partitioning the Orders table by date range to improve query performance.

### 1635. Hopper Company Queries I

**Description:**

Table 32: Drivers

| Column Name | Type |
|---|---|
| driver_id | int |
| join_date | date |

driver_id is the primary key for this table. Each row of this table contains the driver's ID and the date they joined the Hopper company.

Table 33: Rides

| Column Name | Type |
|---|---|
| ride_id | int |
| user_id | int |
| requested_at | date |

ride_id is the primary key for this table. Each row of this table contains the ID of a ride, the user's ID that requested it, and the day they requested it. There may be some ride requests in this table that were not accepted.

Table 34: AcceptedRides

| Column Name | Type |
|---|---|
| ride_id | int |
| driver_id | int |
| ride_distance | int |
| ride_duration | int |

ride_id is the primary key for this table. Each row of this table contains information about an accepted ride. It is guaranteed that each accepted ride exists in the Rides table.

Write an SQL query to report the following statistics for each month of 2020:

- The number of drivers currently with the Hopper company by the end of the month (active_drivers).
- The number of accepted rides in that month (accepted_rides).

Return the result table ordered by month in ascending order, where month is the month's number (January is 1, February is 2, etc.).

**Solution:**

```sql
WITH RECURSIVE months AS (
    SELECT 1 AS month
    UNION ALL
    SELECT month + 1 FROM months WHERE month < 12
),
active_drivers_by_month AS (
    SELECT
        month,
        COUNT(driver_id) AS active_drivers
    FROM
        months m
    CROSS JOIN
        Drivers d
    WHERE
        EXTRACT(YEAR FROM d.join_date) < 2020 OR
        (EXTRACT(YEAR FROM d.join_date) = 2020 AND EXTRACT(MONTH FROM d.join_date) <= m.month
    GROUP BY
        month
),
accepted_rides_by_month AS (
    SELECT
        EXTRACT(MONTH FROM r.requested_at) AS month,
        COUNT(ar.ride_id) AS accepted_rides
    FROM
        Rides r
    JOIN
        AcceptedRides ar ON r.ride_id = ar.ride_id
    WHERE
        EXTRACT(YEAR FROM r.requested_at) = 2020
    GROUP BY
        EXTRACT(MONTH FROM r.requested_at)
)

SELECT
    m.month,
    COALESCE(ad.active_drivers, 0) AS active_drivers,
    COALESCE(ar.accepted_rides, 0) AS accepted_rides
FROM
    months m
LEFT JOIN
    active_drivers_by_month ad ON m.month = ad.month
LEFT JOIN
```

```
    accepted_rides_by_month ar ON m.month = ar.month
ORDER BY
    m.month;
```

**Explanation:**

- This query generates monthly statistics for a ride-sharing company, focusing on active drivers and accepted rides for each month of 2020.
- The solution uses a three-step approach with recursive Common Table Expressions (CTEs):
  - The first CTE, `months`, generates all 12 months of the year as a base for our report.
  - The second CTE, `active_drivers_by_month`, calculates the cumulative count of drivers who joined by the end of each month.
  - The third CTE, `accepted_rides_by_month`, counts the accepted rides for each month in 2020.
- For active drivers, we include both drivers who joined in previous years (EXTRACT(YEAR FROM join_date) $<$ 2020) and those who joined in 2020 up to the current month.
- The CROSS JOIN ensures each driver is evaluated against every month of the year.
- The main query joins all three CTEs to produce the final report, ordered by month.
- LEFT JOINs are used to ensure all months appear in the result, even if they have no active drivers or accepted rides.
- COALESCE handles NULL values by converting them to 0 as required.
- Time complexity: O((m $\times$ d) + r) where m is the number of months (12), d is the number of drivers, and r is the number of rides.
- Space complexity: O(m) for the result set (12 rows).
- This solution correctly handles edge cases:
  - Months with no accepted rides show 0 for accepted_rides
  - The cumulative count of drivers is maintained across months
- For large datasets, indexes on join_date in Drivers and requested_at in Rides would improve performance.
- The recursive CTE approach to generate months is more elegant and portable than using a calendar table for this specific report.
- EXTRACT() functions efficiently isolate the year and month components from dates for filtering and grouping.
- The query is designed to work in PostgreSQL, using its specific date functions and recursive CTE syntax.

### 1645. Hopper Company Queries II

**Description:**

Table 35: Drivers

| Column Name | Type |
|---|---|
| driver_id | int date |
| join_date | |

driver_id is the primary key for this table. Each row of this table contains the driver's ID and the date they joined the Hopper company.

Table 36: Rides

| Column Name | Type |
|---|---|
| ride_id user_id | int int date |
| requested_at | |

ride_id is the primary key for this table. Each row of this table contains the ID of a ride, the user's ID that requested it, and the day they requested it. There may be some ride requests in this table that were not accepted.

Table 37: AcceptedRides

| Column Name | Type |
|---|---|
| ride_id driver_id | int int int |
| ride_distance | int |
| ride_duration | |

ride_id is the primary key for this table. Each row of this table contains information about an accepted ride. It is guaranteed that each accepted ride exists in the Rides table.

Write an SQL query to report the percentage of working drivers (working_percentage) for each month of 2020 where:

```
Working Percentage = (100 * Number of drivers that accepted at least one ride during the mon
```

An available driver is a driver who joined the company before or during the current month and has not quit the company before or during the current month.

Return the result table ordered by month in ascending order, where month is the month's number (January is 1, February is 2, etc.). Round working_percentage to the nearest 2 decimal places.

**Solution:**

```
WITH RECURSIVE months AS (
    SELECT 1 AS month
    UNION ALL
    SELECT month + 1 FROM months WHERE month < 12
),
available_drivers AS (
    SELECT
        m.month,
        COUNT(DISTINCT d.driver_id) AS available_drivers_count
    FROM
        months m
    CROSS JOIN
        Drivers d
    WHERE
        EXTRACT(YEAR FROM d.join_date) < 2020 OR
        (EXTRACT(YEAR FROM d.join_date) = 2020 AND EXTRACT(MONTH FROM d.join_date) <= m.month
    GROUP BY
        m.month
),
working_drivers AS (
    SELECT
        EXTRACT(MONTH FROM r.requested_at) AS month,
        COUNT(DISTINCT ar.driver_id) AS active_drivers_count
    FROM
        Rides r
    JOIN
        AcceptedRides ar ON r.ride_id = ar.ride_id
    WHERE
        EXTRACT(YEAR FROM r.requested_at) = 2020
    GROUP BY
        EXTRACT(MONTH FROM r.requested_at)
)

SELECT
    m.month,
    CASE
        WHEN ad.available_drivers_count = 0 THEN 0.00
        ELSE ROUND((COALESCE(wd.active_drivers_count, 0) * 100.0) / ad.available_drivers_coun
    END AS working_percentage
FROM
    months m
```

```sql
LEFT JOIN
    available_drivers ad ON m.month = ad.month
LEFT JOIN
    working_drivers wd ON m.month = wd.month
ORDER BY
    m.month;
```

**Explanation:**

- This query calculates the monthly percentage of active drivers (those who accepted at least one ride) out of all available drivers for each month of 2020.
- The solution uses a three-step approach with Common Table Expressions (CTEs):
  - The first CTE, `months`, recursively generates all 12 months of the year as a basis for the report.
  - The second CTE, `available_drivers`, counts distinct drivers available in each month (joined before or during the month).
  - The third CTE, `working_drivers`, counts distinct drivers who actually accepted rides each month.
- The main query calculates the working percentage using the formula: (active drivers ÷ available drivers) × 100.
- ROUND(…, 2) ensures the percentages are rounded to two decimal places as required.
- The LEFT JOINs ensure all months appear in the result, even if they have no working drivers.
- The CASE expression handles the potential division by zero if there are no available drivers in a month.
- Time complexity: O((m × d) + r) where m is the number of months (12), d is the number of drivers, and r is the number of rides.
- Space complexity: O(m) for the result set (12 rows).
- The DISTINCT keyword in COUNT(DISTINCT driver_id) ensures each driver is counted only once per month, even if they accepted multiple rides.
- For large datasets, indexes on join_date, requested_at, and driver_id would improve performance.
- The solution correctly handles edge cases:
  - Months with no active drivers show 0.00%
  - Months with no available drivers show 0.00% (avoiding division by zero)
- The COALESCE function ensures proper calculation even if a month has no working drivers.
- Using 100.0 (not 100) in the calculation ensures decimal division rather than integer division, preserving precision.
- The query follows the requirement to order results by month in ascending order.

### 1651. Hopper Company Queries III

**Description:**

Table 38: Drivers

| Column Name | Type |
|---|---|
| driver_id | int |
| join_date | date |

driver_id is the primary key for this table. Each row of this table contains the driver's ID and the date they joined the Hopper company.

Table 39: Rides

| Column Name | Type |
|---|---|
| ride_id | int |
| user_id | int |
| requested_at | date |

ride_id is the primary key for this table. Each row of this table contains the ID of a ride, the user's ID that requested it, and the day they requested it. There may be some ride requests in this table that were not accepted.

Table 40: AcceptedRides

| Column Name | Type |
|---|---|
| ride_id | int |
| driver_id | int |
| ride_distance | int |
| ride_duration | int |

ride_id is the primary key for this table. Each row of this table contains information about an accepted ride. It is guaranteed that each accepted ride exists in the Rides table.

Write an SQL query to compute the average_ride_distance and average_ride_duration of every 3-month window starting from January - March 2020 to October - December 2020. Round average_ride_distance and average_ride_duration to the nearest two decimal places.

The average_ride_distance is calculated by summing up the total ride_distance values from the three months and dividing it by 3. The average_ride_duration is calculated in a similar way.

Return the result table ordered by month in ascending order, where month is the starting month's number (January is 1, February is 2, etc.).

**Solution:**

```
WITH RECURSIVE months AS (
    SELECT generate_series(1, 10) AS month
),
monthly_stats AS (
    SELECT
        EXTRACT(MONTH FROM r.requested_at) AS month,
        SUM(ar.ride_distance) AS total_distance,
        SUM(ar.ride_duration) AS total_duration,
        COUNT(ar.ride_id) AS ride_count
    FROM
        Rides r
    JOIN
        AcceptedRides ar ON r.ride_id = ar.ride_id
    WHERE
        EXTRACT(YEAR FROM r.requested_at) = 2020
    GROUP BY
        EXTRACT(MONTH FROM r.requested_at)
),
rolling_window AS (
    SELECT
        m1.month,
        COALESCE(ms1.total_distance, 0) +
        COALESCE(ms2.total_distance, 0) +
        COALESCE(ms3.total_distance, 0) AS three_month_distance,

        COALESCE(ms1.total_duration, 0) +
        COALESCE(ms2.total_duration, 0) +
        COALESCE(ms3.total_duration, 0) AS three_month_duration,

        CASE
            WHEN COALESCE(ms1.ride_count, 0) + COALESCE(ms2.ride_count, 0) + COALESCE(ms3.ri
            THEN 1
            ELSE 3
        END AS divisor
    FROM
        months m1
    LEFT JOIN
        monthly_stats ms1 ON m1.month = ms1.month
    LEFT JOIN
        monthly_stats ms2 ON m1.month + 1 = ms2.month
    LEFT JOIN
```

```
        monthly_stats ms3 ON m1.month + 2 = ms3.month
    WHERE
        m1.month <= 10  -- Only start windows that can be completed in 2020
)

SELECT
    month,
    ROUND(CASE WHEN three_month_distance = 0 THEN 0.00 ELSE three_month_distance / divisor EL
    ROUND(CASE WHEN three_month_duration = 0 THEN 0.00 ELSE three_month_duration / divisor EL
FROM
    rolling_window
ORDER BY
    month;
```

**Explanation:**

- This query calculates three-month rolling window averages for ride distances and durations throughout 2020.
- The solution uses a three-step approach with Common Table Expressions (CTEs):
  - The first CTE, `months`, generates the first 10 months of the year (as we need complete 3-month windows ending in December).
  - The second CTE, `monthly_stats`, aggregates the total distance, duration, and count of rides for each month.
  - The third CTE, `rolling_window`, joins each month with the following two months to create 3-month windows.
- For each window, we sum the total distances and durations across all three months.
- The CASE expression for the divisor handles windows with zero rides (to avoid division by zero).
- The query only includes starting months up to October (month <= 10) to ensure each window has three complete months in 2020.
- ROUND(..., 2) ensures the averages are rounded to two decimal places as required.
- Time complexity: O(r + m) where r is the number of rides and m is the number of months (10).
- Space complexity: O(m) for the result set (10 rows).
- This solution correctly handles edge cases:
  - Months with no rides show 0.00 for both average metrics
  - The COALESCE function handles missing months in the data
- For large datasets, indexes on requested_at in Rides and ride_id in AcceptedRides would improve performance.

- Using LEFT JOINs between months and stats ensures all months appear in the results, even those with no rides.
- The generate_series() function provides a concise way to create the sequence of months in PostgreSQL.
- The solution efficiently computes rolling windows without needing window functions or complex self-joins.
- The query follows the requirement to order results by starting month in ascending order.

### 1767. Find the Subtasks That Did Not Execute

**Description:**

Table 41: Tasks

| Column Name | Type |
|---|---|
| task_id | int int |
| subtasks_count | |

task_id is the primary key for this table. Each row in this table indicates that task_id contains subtasks_count subtasks.

Table 42: Executed

| Column Name | Type |
|---|---|
| task_id subtask_id | int int |

(task_id, subtask_id) is the primary key for this table. Each row in this table indicates that the subtask subtask_id from task task_id was executed.

Write an SQL query to report the IDs of the missing subtasks for each task_id.

Task_id is the ID of the task, and subtask_id is the ID of the subtask. Subtask IDs are numbered from 1 to subtasks_count.

Return the result table in any order.

**Solution:**

```
WITH RECURSIVE all_subtasks AS (
    SELECT
        t.task_id,
        generate_series(1, t.subtasks_count) AS subtask_id
    FROM
```

```
        Tasks t
)

SELECT
    a.task_id,
    a.subtask_id
FROM
    all_subtasks a
LEFT JOIN
    Executed e ON a.task_id = e.task_id AND a.subtask_id = e.subtask_id
WHERE
    e.task_id IS NULL
ORDER BY
    a.task_id, a.subtask_id;
```

**Explanation:**

- This query identifies subtasks that were expected to be executed (based on the subtasks_count) but do not appear in the Executed table.
- The solution uses a single Common Table Expression (CTE) with PostgreSQL's generate_series function:
  - The `all_subtasks` CTE generates all expected subtask IDs for each task based on its subtasks_count.
  - generate_series(1, subtasks_count) creates a sequence from 1 to the total number of subtasks for each task.
- The main query then uses a LEFT JOIN to find subtasks that don't have a matching record in the Executed table.
- The WHERE clause filters for rows where no match was found (e.task_id IS NULL), identifying subtasks that weren't executed.
- Results are ordered by task_id and subtask_id for readability.
- Time complexity: O(n × s) where n is the number of tasks and s is the maximum subtasks_count.
- Space complexity: O(n × s) for the full set of potential subtasks.
- This solution efficiently handles both executed and missing subtasks, even for tasks with a large number of subtasks.
- For large datasets, indexes on (task_id, subtask_id) in both tables would improve join performance.
- The generate_series function is a PostgreSQL feature that elegantly solves the problem of generating sequences.
- An alternative approach in databases without generate_series would require a numbers or tally table, making this PostgreSQL solution particularly concise.

- The query correctly handles edge cases:
  - Tasks with all subtasks executed won't appear in the results
  - Tasks with no subtasks executed will have all their subtasks in the results
  - Tasks with partially executed subtasks will show only the missing ones
- This approach avoids complex recursive queries that might be needed in other database systems.

### 1892. Page Recommendations II

**Description:**

Table 43: Friendship

| Column Name | Type |
|---|---|
| user1_id user2_id | int int |

(user1_id, user2_id) is the primary key for this table. Each row of this table indicates that there is a friendship relation between user1_id and user2_id.

Table 44: Likes

| Column Name | Type |
|---|---|
| user_id page_id | int int |

(user_id, page_id) is the primary key for this table. Each row of this table indicates that user_id likes page_id.

Write an SQL query to recommend pages to the users using the following approach:

If a user A likes a page P1, another user B likes a page P2, and both users A and B are friends, then page P2 should be recommended to user A. Note that page P1 should not be recommended to user B since user B already likes it.

Return the result table in any order.

**Solution:**

```
WITH friendship_normalized AS (
    -- Normalize friendship to ensure consistent directional representation
    SELECT user1_id AS user_id, user2_id AS friend_id FROM Friendship
    UNION
    SELECT user2_id AS user_id, user1_id AS friend_id FROM Friendship
```

```
),
friend_likes AS (
    -- Find pages liked by friends
    SELECT
        fn.user_id,
        l.page_id,
        COUNT(fn.friend_id) AS friends_count
    FROM
        friendship_normalized fn
    JOIN
        Likes l ON fn.friend_id = l.user_id
    LEFT JOIN
        Likes ul ON fn.user_id = ul.user_id AND l.page_id = ul.page_id
    WHERE
        ul.user_id IS NULL  -- User doesn't already like this page
    GROUP BY
        fn.user_id, l.page_id
)

SELECT
    user_id,
    page_id,
    friends_count
FROM
    friend_likes
ORDER BY
    user_id, friends_count DESC, page_id;
```

**Explanation:**

- This query recommends pages to users based on what their friends like, but only if the user doesn't already like that page.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The first CTE, `friendship_normalized`, converts the bidirectional friendship relation into a normalized directional format.
  - The second CTE, `friend_likes`, identifies pages liked by a user's friends that the user doesn't already like.
- The UNION operation in the first CTE ensures that friendships are represented in both directions, as they are bidirectional.
- The JOIN between friendship_normalized and Likes finds pages liked by friends.
- The LEFT JOIN with another instance of Likes checks if the user already likes the page.

- The WHERE clause with ul.user_id IS NULL filters out pages the user already likes.
- GROUP BY with COUNT aggregates recommendations by friends, counting how many friends like each page.
- Time complexity: O(f × l) where f is the number of friendships and l is the number of likes.
- Space complexity: O(f + r) where r is the number of recommendations.
- This solution correctly handles edge cases:
  - Users with no friends or whose friends have no likes will receive no recommendations
  - Users who already like all the pages their friends like will receive no recommendations
- For large datasets, indexes on user_id, friend_id, and page_id would significantly improve join performance.
- The final ORDER BY clause sorts recommendations first by user_id, then by popularity (friends_count in descending order), and finally by page_id.
- The solution avoids complex self-joins by normalizing the friendship relation first.
- The approach efficiently handles the mutual friendship relation which is often a challenging aspect of social network queries.
- Using COUNT(friend_id) provides additional context about the recommendation's strength (how many friends like it).

### 1917. Leetcodify Friends Recommendations

**Description:**

Table 45: Listens

| Column Name | Type |
|---|---|
| user_id song_id day | int int date |

There is no primary key for this table. It may contain duplicates. Each row of this table indicates that the user user_id listened to the song song_id on the day day.

Table 46: Friendship

| Column Name | Type |
|---|---|
| user1_id user2_id | int int |

(user1_id, user2_id) is the primary key for this table. Each row of this table indicates that the users user1_id and user2_id are friends. Note that user1_id < user2_id.

Write an SQL query to recommend friends to Leetcodify users. We recommend user x to user y if:

- Users x and y are not friends, and
- Users x and y listened to the same three or more different songs on the same day.

Note that friend recommendations are unidirectional, meaning if user x is recommended to user y, user y is not necessarily recommended to user x.

Return the result table in any order.

**Solution:**

```sql
WITH user_song_days AS (
    -- Get distinct user-song-day combinations to avoid duplicates
    SELECT DISTINCT
        user_id,
        song_id,
        day
    FROM
        Listens
),
common_songs AS (
    -- Count common songs listened to by user pairs on the same day
    SELECT
        u1.user_id AS user1_id,
        u2.user_id AS user2_id,
        u1.day,
        COUNT(DISTINCT u1.song_id) AS song_count
    FROM
        user_song_days u1
    JOIN
        user_song_days u2 ON u1.song_id = u2.song_id AND u1.day = u2.day
    WHERE
        u1.user_id <> u2.user_id
    GROUP BY
        u1.user_id, u2.user_id, u1.day
    HAVING
        COUNT(DISTINCT u1.song_id) >= 3
),
friendship_normalized AS (
    -- Normalize friendship to consider both directions
    SELECT user1_id, user2_id FROM Friendship
    UNION
```

```
    SELECT user2_id, user1_id FROM Friendship
)

SELECT
    cs.user1_id AS user_id,
    cs.user2_id AS recommended_id
FROM
    common_songs cs
LEFT JOIN
    friendship_normalized fn ON cs.user1_id = fn.user1_id AND cs.user2_id = fn.user2_id
WHERE
    fn.user1_id IS NULL
GROUP BY
    cs.user1_id, cs.user2_id;
```

**Explanation:**

- This query finds potential friend recommendations based on music listening patterns, specifically recommending users who listened to at least 3 of the same songs on the same day but are not already friends.
- The solution uses a three-step approach with Common Table Expressions (CTEs):
  - The first CTE, `user_song_days`, gets distinct user-song-day combinations to avoid counting duplicated listens.
  - The second CTE, `common_songs`, identifies pairs of users who listened to at least 3 of the same songs on the same day.
  - The third CTE, `friendship_normalized`, normalizes the friendship relation to consider it bidirectional.
- The JOIN in the common_songs CTE pairs users who listened to the same song on the same day.
- The HAVING clause ensures we only consider pairs with at least 3 common songs.
- The main query excludes pairs that are already friends using a LEFT JOIN and WHERE fn.user1_id IS NULL.
- The final GROUP BY eliminates any potential duplicates in the recommendations.
- Time complexity: $O(n^2)$ where n is the number of user-song-day combinations, due to the self-join operation.
- Space complexity: $O(n^2 + f)$ where f is the number of friendships.
- This solution correctly handles edge cases:
  - Users who listened to the same song multiple times on the same day (handled by DISTINCT in the first CTE)
  - Bidirectional friendship relations (handled by the UNION in the friendship_normalized CTE)

- For large datasets, indexes on user_id, song_id, and day in the Listens table would significantly improve join performance.
- The query ensures that recommendations are unidirectional as specified in the requirements.
- The approach efficiently identifies common listening patterns without complex window functions or recursive queries.
- The DISTINCT keyword in COUNT(DISTINCT song_id) ensures each common song is counted only once.
- For particularly large datasets, consider partitioning the Listens table by day to improve query performance.

### 1919. Leetcodify Similar Friends

**Description:**

Table 47: Listens

| Column Name | Type |
|---|---|
| user_id song_id day | int int date |

There is no primary key for this table. It may contain duplicates. Each row of this table indicates that the user user_id listened to the song song_id on the day day.

Table 48: Friendship

| Column Name | Type |
|---|---|
| user1_id user2_id | int int |

(user1_id, user2_id) is the primary key for this table. Each row of this table indicates that the users user1_id and user2_id are friends. Note that user1_id < user2_id.

Write an SQL query to report the similar friends of Leetcodify users. A user x and user y are similar friends if:

- Users x and y are friends, and
- Users x and y listened to the same three or more different songs on the same day.

Return the result table in any order.

**Solution:**

```sql
WITH user_song_days AS (
    -- Get distinct user-song-day combinations to avoid duplicates
    SELECT DISTINCT
        user_id,
        song_id,
        day
    FROM
        Listens
),
common_songs AS (
    -- Count common songs listened to by user pairs on the same day
    SELECT
        CASE WHEN u1.user_id < u2.user_id THEN u1.user_id ELSE u2.user_id END AS user1_id,
        CASE WHEN u1.user_id < u2.user_id THEN u2.user_id ELSE u1.user_id END AS user2_id,
        u1.day,
        COUNT(DISTINCT u1.song_id) AS song_count
    FROM
        user_song_days u1
    JOIN
        user_song_days u2 ON u1.song_id = u2.song_id AND u1.day = u2.day
    WHERE
        u1.user_id <> u2.user_id
    GROUP BY
        CASE WHEN u1.user_id < u2.user_id THEN u1.user_id ELSE u2.user_id END,
        CASE WHEN u1.user_id < u2.user_id THEN u2.user_id ELSE u1.user_id END,
        u1.day
    HAVING
        COUNT(DISTINCT u1.song_id) >= 3
)

SELECT
    cs.user1_id,
    cs.user2_id
FROM
    common_songs cs
JOIN
    Friendship f ON cs.user1_id = f.user1_id AND cs.user2_id = f.user2_id
GROUP BY
    cs.user1_id, cs.user2_id;
```

**Explanation:**

- This query identifies pairs of friends who are "similar" because they listened to at least 3 of the same songs on the same day.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The first CTE, `user_song_days`, gets distinct user-song-day combinations to avoid counting duplicated listens.
  - The second CTE, `common_songs`, identifies pairs of users who listened to at least 3 of the same songs on the same day, ensuring user1_id < user2_id for consistency.
- The CASE expressions in common_songs ensure that user pairs are ordered correctly (smaller ID first) to match the Friendship table's convention.
- The main query joins with the Friendship table to ensure we only include pairs who are actually friends.
- The GROUP BY eliminates any potential duplicates if friends have multiple days with 3+ common songs.
- Time complexity: $O(n^2)$ where n is the number of user-song-day combinations, due to the self-join operation.
- Space complexity: $O(n^2)$ for the potential user pairs.
- This solution correctly handles edge cases:
  - Users who listened to the same song multiple times on the same day (handled by DISTINCT in the first CTE)
  - The ordering of user IDs in the friendship relation (handled by the CASE expressions)
- For large datasets, indexes on user_id, song_id, and day in the Listens table would significantly improve join performance.
- The DISTINCT keyword in COUNT(DISTINCT song_id) ensures each common song is counted only once.
- This query efficiently builds on the previous problem (Leetcodify Friends Recommendations) but with the additional constraint that the users must be friends.
- The approach avoids unnecessary data processing by joining with the Friendship table only after identifying user pairs with common songs.
- The solution maintains the requirement that user1_id < user2_id in the final results, matching the convention in the Friendship table.

### 1972. First and Last Call On the Same Day

**Description:**

Table 49: Calls

| Column Name | Type |
|---|---|

| | |
|---|---|
| caller_id | int int |
| recipient_id | varchar |
| call_time | |

(caller_id, recipient_id, call_time) is the primary key for this table. Each row contains information about a call made between a caller_id and a recipient_id at call_time. call_time is of the format 'HH:MM:SS'.

Write an SQL query to report the IDs of the users whose first and last calls on any day were with the same person. Calls are counted regardless of being the caller or the recipient.

Return the result table in any order.

**Solution:**

```
WITH normalized_calls AS (
    -- Normalize the calls to handle both caller and recipient perspectives
    SELECT
        caller_id AS user_id,
        recipient_id AS other_id,
        CAST(call_time AS TIME) AS call_time,
        DATE_TRUNC('day', TO_TIMESTAMP(call_time, 'HH24:MI:SS')) AS call_date
    FROM
        Calls

    UNION ALL

    SELECT
        recipient_id AS user_id,
        caller_id AS other_id,
        CAST(call_time AS TIME) AS call_time,
        DATE_TRUNC('day', TO_TIMESTAMP(call_time, 'HH24:MI:SS')) AS call_date
    FROM
        Calls
),
daily_extremes AS (
    -- Find first and last call for each user on each day
    SELECT
        user_id,
        call_date,
        FIRST_VALUE(other_id) OVER (
            PARTITION BY user_id, call_date
```

```
            ORDER BY call_time
            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
        ) AS first_call_id,
        LAST_VALUE(other_id) OVER (
            PARTITION BY user_id, call_date
            ORDER BY call_time
            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
        ) AS last_call_id
    FROM
        normalized_calls
)

SELECT DISTINCT
    user_id
FROM
    daily_extremes
WHERE
    first_call_id = last_call_id
GROUP BY
    user_id;
```

**Explanation:**

- This query identifies users whose first and last calls on any day were with the same person, regardless of who initiated the call.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The first CTE, `normalized_calls`, flattens the call data to consider both perspectives (caller and recipient).
  - The second CTE, `daily_extremes`, uses window functions to identify the first and last call for each user on each day.
- The UNION ALL in normalized_calls ensures each call is represented from both users' perspectives.
- CAST and DATE_TRUNC functions handle the time and date components of call_time.
- FIRST_VALUE and LAST_VALUE window functions efficiently identify the first and last call within each day.
- The ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING frame ensures the entire partition is considered.
- The main query filters for users where first_call_id equals last_call_id.
- Time complexity: O(n log n) where n is the number of calls, due to the sorting operations in the window functions.
- Space complexity: O(n) for the normalized calls.

- This solution correctly handles edge cases:
  - Users who made only one call on a day (first and last are the same)
  - Users who called the same person multiple times
  - Users who had calls across multiple days
- For large datasets, indexes on caller_id, recipient_id, and call_time would improve performance.
- The DISTINCT keyword in the final selection ensures each user is reported only once, even if they had matching first/last calls on multiple days.
- The solution elegantly handles the bidirectional nature of calls without complex self-joins.
- TO_TIMESTAMP with the format string 'HH24:MI:SS' properly parses the time strings in PostgreSQL.
- Alternative approaches might use MIN and MAX functions, but the window functions provide a more elegant solution.

### 2004. The Number of Seniors and Juniors to Join the Company

**Description:**

Table 50: Candidates

| Column Name | Type |
|---|---|
| employee_id | int enum |
| experience salary | int |

employee_id is the primary key column for this table. experience is an enum with one of the values ('Senior', 'Junior'). Each row of this table indicates the id of a candidate, their experience, and their salary.

A company wants to hire new employees. The budget of the company for the salaries is $70000. The company's hiring criteria are:

1. Keep hiring the senior with the smallest salary until you cannot hire any more seniors.
2. Use the remaining budget to hire the junior with the smallest salary.
3. Keep hiring the junior with the smallest salary until you cannot hire any more juniors.

Write an SQL query to find the employee_id of the seniors and juniors who will be hired.

Return the result table in any order.

**Solution:**

```sql
WITH senior_budgeting AS (
    SELECT
        employee_id,
        experience,
        salary,
        SUM(salary) OVER (
            PARTITION BY experience
            ORDER BY salary, employee_id
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        ) AS running_total
    FROM
        Candidates
    WHERE
        experience = 'Senior'
),
junior_budgeting AS (
    SELECT
        employee_id,
        experience,
        salary,
        SUM(salary) OVER (
            PARTITION BY experience
            ORDER BY salary, employee_id
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        ) AS running_total
    FROM
        Candidates
    WHERE
        experience = 'Junior'
),
senior_hired AS (
    SELECT
        employee_id,
        salary,
        running_total
    FROM
        senior_budgeting
    WHERE
        running_total <= 70000
),
senior_budget_used AS (
    SELECT COALESCE(MAX(running_total), 0) AS used
```

```
    FROM senior_hired
),
junior_hired AS (
    SELECT
        jb.employee_id,
        jb.salary
    FROM
        junior_budgeting jb,
        senior_budget_used sbu
    WHERE
        jb.running_total <= (70000 - sbu.used)
)

SELECT employee_id
FROM senior_hired

UNION ALL

SELECT employee_id
FROM junior_hired
ORDER BY employee_id;
```

**Explanation:**

- This query implements a hiring strategy based on experience and salary within a fixed budget of $70,000.
- The solution uses a multi-step approach with Common Table Expressions (CTEs):
  - The first two CTEs, `senior_budgeting` and `junior_budgeting`, calculate running totals of salaries for seniors and juniors separately.
  - The third CTE, `senior_hired`, identifies seniors who can be hired within the budget.
  - The fourth CTE, `senior_budget_used`, calculates the total budget used for seniors.
  - The fifth CTE, `junior_hired`, identifies juniors who can be hired with the remaining budget.
- The SUM() OVER() window function with the ROWS frame creates a running total of salaries in order of increasing salary.
- The WHERE clauses in senior_hired and junior_hired ensure we stay within the budget limits.
- The final query combines hired seniors and juniors using UNION ALL.
- Time complexity: O(n log n) where n is the number of candidates, due to the sorting operations.

- Space complexity: O(n) for the intermediate results.
- This solution correctly implements the hiring strategy:
  - Prioritizes seniors with smallest salaries
  - Uses remaining budget for juniors with smallest salaries
  - Handles the case where no seniors or no juniors are hired
- For large datasets, indexes on experience and salary would improve performance.
- COALESCE(MAX(running_total), 0) handles the edge case where no seniors are hired.
- The solution efficiently handles the budget allocation without complex recursive logic.
- The ORDER BY within the window functions ensures ties in salary are broken by employee_id.
- This approach directly translates the business rules into SQL operations.
- The final ORDER BY ensures results are returned in employee_id order, though this is not required by the problem.

### 2010. The Number of Seniors and Juniors to Join the Company II

**Description:**

Table 51: Candidates

| Column Name | Type |
|---|---|
| employee_id | int enum |
| experience salary | int |

employee_id is the primary key column for this table. experience is an enum with one of the values ('Senior', 'Junior'). Each row of this table indicates the id of a candidate, their experience, and their salary.

A company wants to hire new employees. The company's maximum budget for salaries is $70000.

The company wants to first use the budget to hire the most senior employees. Then, if they still have budget left, they will start hiring the most junior employees. If they still have budget left, they will repeat this hiring process.

Write an SQL query to find the number of seniors and juniors the company can hire under the budget.

Return the result as a single row with 2 columns: senior_count representing the number of seniors the company can hire, and junior_count representing the number of juniors the company can hire.

**Solution:**

```sql
WITH senior_budgeting AS (
    SELECT
        employee_id,
        salary,
        SUM(salary) OVER (
            ORDER BY salary, employee_id
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        ) AS running_total
    FROM
        Candidates
    WHERE
        experience = 'Senior'
),
junior_budgeting AS (
    SELECT
        employee_id,
        salary,
        SUM(salary) OVER (
            ORDER BY salary, employee_id
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        ) AS running_total
    FROM
        Candidates
    WHERE
        experience = 'Junior'
),
senior_hires AS (
    SELECT
        COUNT(*) AS senior_count,
        COALESCE(MAX(running_total), 0) AS senior_budget
    FROM
        senior_budgeting
    WHERE
        running_total <= 70000
),
junior_hires AS (
    SELECT
        COUNT(*) AS junior_count
    FROM
        junior_budgeting jb,
        senior_hires sh
    WHERE
```

```
        jb.running_total <= (70000 - sh.senior_budget)
)

SELECT
    sh.senior_count,
    COALESCE(jh.junior_count, 0) AS junior_count
FROM
    senior_hires sh
LEFT JOIN
    junior_hires jh ON 1=1;
```

**Explanation:**

- This query determines the maximum number of senior and junior employees that can be hired within a $70,000 budget, prioritizing seniors by salary.
- The solution uses a four-step approach with Common Table Expressions (CTEs):
  - The first two CTEs, `senior_budgeting` and `junior_budgeting`, calculate running totals of salaries for each experience level.
  - The third CTE, `senior_hires`, counts how many seniors can be hired and calculates the total budget used.
  - The fourth CTE, `junior_hires`, counts how many juniors can be hired with the remaining budget.
- The SUM() OVER() window function creates running totals of salaries ordered by increasing salary.
- The main query joins the senior_hires and junior_hires CTEs to produce the final counts.
- Time complexity: O(n log n) where n is the number of candidates, due to the sorting operations.
- Space complexity: O(n) for the intermediate results.
- This solution correctly handles edge cases:
  - No seniors or no juniors in the candidate pool
  - Not enough budget to hire any candidates
  - Budget sufficient to hire all candidates
- For large datasets, indexes on experience and salary would improve performance.
- COALESCE handles NULL values that might occur if no juniors can be hired.
- The LEFT JOIN ensures the query returns a result even if no juniors can be hired.
- The ORDER BY within the window functions ensures ties in salary are broken by employee_id.
- The solution efficiently implements the hiring strategy without complex recursive logic.
- The approach directly models the business requirements in SQL operations.
- The condition "1=1" in the LEFT JOIN is a PostgreSQL technique to create a cross join with a LEFT JOIN syntax.

### 2118. Build the Equation

**Description:**

Table 52: Terms

| Column Name | Type |
|---|---|
| power | int |
| factor | int |

power is the primary key column for this table. Each row of this table contains information about one term of the equation. power is an integer in the range [0, 100]. factor is an integer in the range [-100, 100] and cannot be zero.

You have a very powerful program that can solve any equation of one variable in the world. The equation passed to the program must be formatted as follows:

- The left-hand side (LHS) should contain all the terms.
- The right-hand side (RHS) should be zero.
- Each term of the LHS should follow the format "X^" where:
  - is either "+" or "-".
  - is the absolute value of the factor.
  - X is the uppercase letter "X".
  - is the value of the power.
- If the power is 0, do not include "^".
- If the power is 1, include "X" but do not include "^".
- Terms must be sorted by the value of the power in descending order.
- There should be no leading "+" sign for the first term.
- There should be a single space between "+" or "-" and the term.
- There should be no extra spaces.

Write an SQL query to build the equation.

**Solution:**

```
WITH ordered_terms AS (
    SELECT
        power,
        factor,
        CASE
            WHEN power = 0 THEN ''
            WHEN power = 1 THEN 'X'
            ELSE CONCAT('X^', power)
        END AS power_string,
```

```
        ROW_NUMBER() OVER (ORDER BY power DESC) AS term_order
    FROM
        Terms
)

SELECT
    CONCAT(
        STRING_AGG(
            CASE
                WHEN term_order = 1 THEN
                    CASE
                        WHEN factor > 0 THEN CONCAT(factor, power_string)
                        ELSE CONCAT('-', ABS(factor), power_string)
                    END
                ELSE
                    CASE
                        WHEN factor > 0 THEN CONCAT('+ ', factor, power_string)
                        ELSE CONCAT('- ', ABS(factor), power_string)
                    END
            END,
            ' '
        ),
        ' = 0'
    ) AS equation
FROM
    ordered_terms;
```

**Explanation:**

- This query builds a mathematical equation from terms stored in a table, following specific formatting rules.
- The solution uses a Common Table Expression (CTE) approach:
  - The `ordered_terms` CTE processes each term to generate the proper power notation and orders them by power descending.
  - CASE expressions handle special cases for powers of 0 (no "X" term) and 1 (just "X" without exponent).
  - ROW_NUMBER() identifies the first term, which has special formatting (no leading sign).
- The main query uses STRING_AGG to concatenate all terms with proper spacing and signs.

- For the first term, we simply use the factor value (or its negation) without a leading "+" sign.
- For subsequent terms, we add either "+" or "-" before the factor value.
- The ABS() function ensures we only show the absolute value of the factor, handling the sign separately.
- Finally, we append " = 0" to complete the equation as required.
- Time complexity: O(n log n) where n is the number of terms, due to the sorting operation.
- Space complexity: O(n) for the intermediate expressions and final equation.
- This solution correctly handles all the formatting requirements:
  - Descending order of powers
  - Special cases for powers of 0 and 1
  - Proper sign handling with spacing
  - No leading "+" for the first term
- For large datasets, an index on the power column would improve performance.
- STRING_AGG is a PostgreSQL function that elegantly handles the concatenation with separator.
- The approach directly translates the mathematical notation rules into SQL string manipulation.
- The solution works for the full range of valid inputs (powers 0-100, factors -100 to 100 except 0).
- Nested CASE expressions provide a clean way to handle the multiple conditional formatting rules.

### 2153. The Number of Passengers in Each Bus II

**Description:**

Table 53: Buses

| Column Name | Type |
|---|---|
| bus_id | int int |
| arrival_time | int |
| capacity | |

bus_id is the primary key column for this table. Each row of this table contains information about the arrival time of a bus at the LeetCode station and its capacity. No two buses will arrive at the same time.

Table 54: Passengers

| Column Name | Type |
|---|---|

| | |
|---|---|
| passenger_id | int int |
| arrival_time | |

passenger_id is the primary key column for this table. Each row of this table contains information about the arrival time of a passenger at the LeetCode station.

Buses and passengers arrive at the LeetCode station. If a bus arrives at the station at time tbus and a passenger arrived at time tpassenger where tpassenger <= tbus, the passenger will take the bus only if the passenger did not take any other bus before that.

Write an SQL query to report the number of passengers that took each bus. Return the result table ordered by bus_id in ascending order.

**Solution:**

```
WITH ordered_buses AS (
    SELECT
        bus_id,
        arrival_time,
        capacity,
        LAG(arrival_time, 1, 0) OVER (ORDER BY arrival_time) AS prev_arrival_time,
        ROW_NUMBER() OVER (ORDER BY arrival_time) AS bus_order
    FROM
        Buses
),
eligible_passengers AS (
    SELECT
        p.passenger_id,
        p.arrival_time,
        MIN(b.bus_order) OVER (PARTITION BY p.passenger_id) AS first_available_bus
    FROM
        Passengers p
    JOIN
        ordered_buses b ON p.arrival_time <= b.arrival_time
),
passengers_per_bus AS (
    SELECT
        ob.bus_id,
        ob.bus_order,
        ob.capacity,
        (
            SELECT COUNT(*)
```

```
            FROM eligible_passengers ep
            WHERE ep.first_available_bus = ob.bus_order
        ) AS raw_passengers
    FROM
        ordered_buses ob
),
bus_allocation AS (
    SELECT
        bus_id,
        bus_order,
        capacity,
        raw_passengers,
        SUM(raw_passengers) OVER (ORDER BY bus_order) AS cumulative_passengers,
        SUM(capacity) OVER (ORDER BY bus_order) AS cumulative_capacity
    FROM
        passengers_per_bus
),
final_counts AS (
    SELECT
        ba.bus_id,
        CASE
            WHEN ba.bus_order = 1 THEN LEAST(ba.raw_passengers, ba.capacity)
            ELSE LEAST(
                ba.raw_passengers,
                ba.capacity - GREATEST(0, prev_ba.cumulative_passengers - prev_ba.cumulative_
            )
        END AS passengers_count
    FROM
        bus_allocation ba
    LEFT JOIN
        bus_allocation prev_ba ON ba.bus_order = prev_ba.bus_order + 1
)

SELECT
    bus_id,
    passengers_count
FROM
    final_counts
ORDER BY
    bus_id;
```

**Explanation:**

- This query calculates how many passengers can fit on each bus, considering capacity limits and the rule that passengers take the first available bus after their arrival.
- The solution uses a multi-step approach with Common Table Expressions (CTEs):
  - The `ordered_buses` CTE arranges buses by arrival time and identifies the previous bus's arrival time.
  - The `eligible_passengers` CTE matches passengers to their first available bus using window functions.
  - The `passengers_per_bus` CTE calculates how many passengers would initially be assigned to each bus.
  - The `bus_allocation` CTE tracks cumulative passengers and capacity to handle overflow.
  - The `final_counts` CTE adjusts passenger counts based on capacity constraints and overflow from previous buses.
- The LEAST function ensures no bus exceeds its capacity.
- The combination of window functions and self-join handles the complex logic of passenger allocation across buses.
- Time complexity: O((b+p) log (b+p)) where b is the number of buses and p is the number of passengers.
- Space complexity: O(b+p) for the intermediate results.
- This solution correctly handles edge cases:
  - Buses with excess capacity
  - Buses that cannot accommodate all waiting passengers
  - Overflows to subsequent buses
- For large datasets, indexes on arrival_time in both tables would improve join performance.
- The approach efficiently models the "first available bus" requirement using window functions.
- The LEFT JOIN in the final_counts CTE handles the first bus, which has no previous bus.
- The solution carefully tracks both cumulative passengers and cumulative capacity to handle capacity constraints.
- The GREATEST(0, …) function prevents negative overflow calculations if early buses have excess capacity.
- The final ORDER BY ensures results are returned in bus_id order as required.

### 2173. Longest Winning Streak

**Description:**

Table 55: Matches

| Column Name | Type |
|---|---|
| player_id | int date |
| match_day | enum |
| result | |

(player_id, match_day) is the primary key for this table. Each row of this table contains the ID of a player, the day of the match they played, and the result of that match. The result column is an ENUM type of ('Win', 'Draw', 'Lose').

Write an SQL query to report the longest winning streak for each player. If there is a tie, report all of them.

A winning streak is a consecutive sequence of matches where a player wins. The streak ends once the player loses or has a draw.

Return the result table ordered by player_id.

**Solution:**

```
WITH match_groups AS (
    SELECT
        player_id,
        match_day,
        result,
        match_day - ROW_NUMBER() OVER (
            PARTITION BY player_id, result
            ORDER BY match_day
        )::integer AS group_id
    FROM
        Matches
    WHERE
        result = 'Win'
),
streak_lengths AS (
    SELECT
        player_id,
        group_id,
        COUNT(*) AS streak_length
    FROM
        match_groups
    GROUP BY
        player_id, group_id
```

```
),
max_streaks AS (
    SELECT
        player_id,
        MAX(streak_length) AS longest_streak
    FROM
        streak_lengths
    GROUP BY
        player_id
),
all_players AS (
    SELECT DISTINCT player_id
    FROM Matches
)

SELECT
    ap.player_id,
    COALESCE(ms.longest_streak, 0) AS longest_streak
FROM
    all_players ap
LEFT JOIN
    max_streaks ms ON ap.player_id = ms.player_id
ORDER BY
    ap.player_id;
```

**Explanation:**

- This query identifies each player's longest winning streak (consecutive matches won).
- The solution uses a four-step approach with Common Table Expressions (CTEs):
  - The `match_groups` CTE uses the "islands and gaps" technique to identify consecutive win sequences.
  - The `streak_lengths` CTE counts the length of each winning streak.
  - The `max_streaks` CTE finds the maximum streak length for each player.
  - The `all_players` CTE ensures all players are included in the results, even those without wins.
- The key insight is using "match_day - ROW_NUMBER()" to create a constant group_id for consecutive wins.
- When dates are consecutive and only wins are included, this difference creates the same value for a streak.
- The main query joins all_players with max_streaks to include players with no wins (who get a longest_streak of 0).

- Time complexity: O(n log n) where n is the number of matches, due to the sorting operations.
- Space complexity: O(n) for the intermediate results.
- This solution correctly handles edge cases:
  - Players with no wins
  - Players with multiple winning streaks of the same length
  - Non-consecutive match days
- For large datasets, indexes on player_id and match_day would improve performance.
- COALESCE ensures players without any wins show a longest_streak of 0 rather than NULL.
- The WHERE clause in match_groups efficiently filters for wins only, simplifying the grouping logic.
- The LEFT JOIN ensures all players appear in the results, regardless of their win record.
- The approach elegantly handles the streak identification without complex self-joins or recursive queries.
- The query follows the requirement to order results by player_id.
- The ROW_NUMBER() window function partitions by both player_id and result to create proper groupings within each player's wins.

### 2199. Finding the Topic of Each Post

**Description:**

Table 56: Keywords

| Column Name | Type |
|---|---|
| topic_id word | int varchar |

(topic_id, word) is the primary key for this table. Each row of this table contains the id of a topic and a word that belongs to that topic.

Table 57: Posts

| Column Name | Type |
|---|---|
| post_id content | int varchar |

post_id is the primary key for this table. Each row of this table contains the ID of a post and its content.

The topics of a post are the topics that the post's content belongs to. The topic of a content is detected by the following rules:

If the content contains the exact word from a keyword of a topic, then the content has the topic of that keyword. Same content can have multiple topics.

Write an SQL query to find the topics of each post according to the rules above. Return the result table ordered by post_id in ascending order. In case of a tie, order by topic_id in ascending order.

Note that:

- The word in the Keywords table is case-sensitive, and the content should match exactly the word in the Keywords table.
- A string like "Leetcode" is not equal to "leetcode".

**Solution:**

```
WITH topic_matches AS (
    SELECT
        p.post_id,
        k.topic_id
    FROM
        Posts p
    JOIN
        Keywords k ON p.content ~* CONCAT('\\m', k.word, '\\M')
    GROUP BY
        p.post_id, k.topic_id
)

SELECT
    post_id,
    COALESCE(
        (
            SELECT STRING_AGG(topic_id::text, ',')
            FROM (
                SELECT topic_id
                FROM topic_matches tm
                WHERE tm.post_id = p.post_id
                ORDER BY topic_id
            ) AS ordered_topics
        ),
        'Ambiguous!'
    ) AS topic
FROM
    Posts p
```

```
ORDER BY
    post_id;
```

**Explanation:**

- This query identifies the topics of each post based on keyword matching, with special attention to exact word matches.
- The solution uses a Common Table Expression (CTE) approach:
  - The `topic_matches` CTE finds all matches between posts and keywords using regular expression matching.
  - The PostgreSQL pattern `\\m` and `\\M` match word boundaries, ensuring exact word matches.
- The main query aggregates all matched topics for each post using STRING_AGG, ordering them by topic_id.
- The COALESCE function handles posts with no matched topics, labeling them as 'Ambiguous!'.
- Time complexity: O(p × k) where p is the number of posts and k is the number of keywords.
- Space complexity: O(p × k) in the worst case if all posts match all topics.
- This solution correctly handles edge cases:
  - Posts matching multiple topics
  - Posts matching no topics
  - Case-sensitive matching requirements
- For large datasets, a full-text search index would significantly improve performance over regular expression matching.
- The `~*` operator in PostgreSQL performs case-insensitive pattern matching.
- The word boundary patterns `\\m` and `\\M` ensure only complete words are matched, not partial matches within larger words.
- The GROUP BY in the topic_matches CTE ensures each post-topic pair is counted only once, even if a keyword appears multiple times.
- STRING_AGG with proper ordering ensures consistent results for posts with multiple topics.
- The subquery with ORDER BY inside STRING_AGG ensures topics are sorted before aggregation.
- The final ORDER BY ensures results are returned in post_id order as required.

### 2252. Dynamic Pivoting of a Table

**Description:**

Table 58: Products

| Column Name | Type |
|---|---|
| product_id store | int varchar |
| price | int |

(product_id, store) is the primary key for this table. Each row of this table indicates the price of product_id in store.

Write an SQL query to pivot the table so that each row shows the product_id and a separate column for the price in each store. If the price is not available in a particular store, set the price to null.

Return the result table in any order.

**Solution:**

```
WITH stores AS (
    SELECT DISTINCT store
    FROM Products
    ORDER BY store
),
store_columns AS (
    SELECT
        STRING_AGG(
            FORMAT('MAX(CASE WHEN store = ''%s'' THEN price END) AS "%s"', store, store),
            ', '
        ) AS store_columns
    FROM
        stores
)

SELECT format('
    SELECT
        product_id,
        %s
    FROM
        Products
    GROUP BY
        product_id
', (SELECT store_columns FROM store_columns)) AS pivot_query;
```

**Explanation:**

- This query dynamically generates a pivot table SQL statement based on the unique store values in the Products table.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The `stores` CTE extracts all distinct store names, ordered alphabetically.
  - The `store_columns` CTE generates the dynamic column expressions for each store.
- The query uses STRING_AGG to concatenate all the CASE expressions that will form the pivot columns.
- The FORMAT function creates properly quoted SQL statements with the store names.
- The main query wraps everything in a template that includes the SELECT, FROM, and GROUP BY clauses.
- Since PostgreSQL doesn't have a direct PIVOT function like some other SQL dialects, this approach dynamically generates the pivot query.
- Time complexity: O(n) where n is the number of unique stores.
- Space complexity: O(n) for the generated query.
- This solution has some important characteristics to note:
  - It doesn't execute the pivot directly but generates the SQL that would perform the pivot.
  - The generated SQL would need to be executed separately to get the actual pivoted data.
  - This is a meta-programming approach to dynamic pivoting.
- For large datasets, indexes on product_id and store would improve the performance of the generated query.
- The approach handles an arbitrary number of stores without hardcoding column names.
- The CASE expressions in the generated query ensure NULL values for missing price data.
- The GROUP BY in the generated query ensures each product appears once with all its store prices.
- This solution is particularly useful for administrative tasks and reporting where the number of pivot columns might change over time.
- An alternative would be to use PostgreSQL's crosstab function from the tablefunc extension, but the generated SQL approach is more portable.
- The double quotes around the column names in the generated SQL ensure proper identifier handling, especially for stores with spaces or special characters.

### 2253. Dynamic Unpivoting of a Table

**Description:**

Table 59: Products

| Column Name | Type |
|---|---|
| product_id | int int int |
| store1 store2 store3 | int |

product_id is the primary key for this table. Each row in this table indicates the product's price in 3 different stores: store1, store2, and store3. If the product is not available in a store, the price will be null in that store's column.

Write an SQL query to rearrange the Products table so that each row has (product_id, store, price). If a product is not available in a store, do not include a row with that product_id and store combination in the result table.

Return the result table in any order.

**Solution:**

```
SELECT
    product_id,
    'store1' AS store,
    store1 AS price
FROM
    Products
WHERE
    store1 IS NOT NULL

UNION ALL

SELECT
    product_id,
    'store2' AS store,
    store2 AS price
FROM
    Products
WHERE
    store2 IS NOT NULL

UNION ALL

SELECT
    product_id,
```

```
    'store3' AS store,
    store3 AS price
FROM
    Products
WHERE
    store3 IS NOT NULL;
```

**Explanation:**

- This query performs unpivoting (also known as "melting") of a wide-format table into a long-format table.
- The solution uses a straightforward UNION ALL approach:
  - Each SELECT statement extracts data for one store column.
  - The store name is hardcoded as a string literal in each query.
  - The WHERE clauses filter out NULL prices as required.
- UNION ALL combines the results from all three queries into a single result set.
- This approach effectively transforms the data from a "wide" format (one row per product with multiple store columns) to a "long" format (multiple rows per product, one for each store with a price).
- Time complexity: O(n) where n is the number of products.
- Space complexity: O(n × s) where s is the number of stores (3 in this case).
- This solution correctly handles the requirement to exclude rows where the price is NULL.
- For large datasets, indexes on product_id would improve performance.
- The solution is simple and efficient for a fixed number of store columns.
- Unlike the previous problem (Dynamic Pivoting), this unpivoting operation can be performed directly without generating intermediary SQL.
- The approach works well for the specific case of 3 stores but would become verbose for tables with many columns.
- For tables with a large or variable number of columns, PostgreSQL's crosstab function (from the tablefunc extension) or a dynamic SQL approach would be more scalable.
- The UNION ALL operation preserves all rows from each subquery, which is appropriate since there should be no duplicates.
- Each subquery explicitly names the store, ensuring clear identification in the output.
- This pattern effectively "normalizes" the denormalized data structure of the original table.
- The solution follows good practice by explicitly naming all columns in the result set.

### 2362. Generate the Invoice

**Description:**

Table 60: Products

| Column Name | Type |
|---|---|
| product_id price | int int |

product_id is the primary key for this table. Each row in this table shows the ID of a product and its price.

Table 61: Purchases

| Column Name | Type |
|---|---|
| invoice_id | int int |
| product_id | int |
| quantity | |

(invoice_id, product_id) is the primary key for this table. Each row in this table shows the quantity ordered from one product in an invoice.

Write an SQL query to show the details of the invoice with the highest price. If there are multiple invoices with the same price, return the details of the one with the smallest invoice_id.

The details of an invoice include the invoice_id, product_id, quantity, and price. The price of an invoice is equal to the sum of the prices of all the products present in the invoice, with their respective quantities.

Return the result table ordered by product_id in ascending order.

**Solution:**

```
WITH invoice_totals AS (
    SELECT
        p.invoice_id,
        SUM(pr.price * p.quantity) AS total_price
    FROM
        Purchases p
    JOIN
        Products pr ON p.product_id = pr.product_id
    GROUP BY
        p.invoice_id
),
highest_invoice AS (
    SELECT
        invoice_id
```

```
    FROM
        invoice_totals
    ORDER BY
        total_price DESC, invoice_id
    LIMIT 1
)

SELECT
    p.invoice_id,
    p.product_id,
    p.quantity,
    pr.price * p.quantity AS price
FROM
    Purchases p
JOIN
    Products pr ON p.product_id = pr.product_id
JOIN
    highest_invoice hi ON p.invoice_id = hi.invoice_id
ORDER BY
    p.product_id;
```

**Explanation:**

- This query identifies the invoice with the highest total price and returns its detailed line items.
- The solution uses a three-step approach with Common Table Expressions (CTEs):
  - The `invoice_totals` CTE calculates the total price of each invoice by summing the product of price and quantity.
  - The `highest_invoice` CTE identifies the invoice with the highest total price (or the smallest invoice_id in case of a tie).
  - The main query retrieves all line items from the selected invoice with their details.
- The JOIN between Purchases and Products calculates the price for each line item.
- The ORDER BY clause ensures results are sorted by product_id as required.
- Time complexity: O(n log n) where n is the number of purchases, due to the sorting operations.
- Space complexity: O(i + p) where i is the number of invoices and p is the number of purchases for the highest invoice.
- This solution correctly handles edge cases:
  - Multiple invoices with the same total price (resolved by selecting the smallest invoice_id)
  - Invoices with multiple product line items

- For large datasets, indexes on invoice_id and product_id would improve join performance.
- The LIMIT 1 in the highest_invoice CTE efficiently selects just the top invoice without unnecessarily sorting all invoices.
- The approach separates the invoice selection logic from the detail retrieval, making the query more readable and maintainable.
- The price calculation is repeated in both CTEs to ensure accurate results.
- The JOIN with the highest_invoice CTE acts as a filter to include only line items from the selected invoice.
- The solution follows the requirement to order results by product_id.
- An alternative approach could use a window function to rank invoices, but the CTE method is cleaner for this specific requirement.

### 2474. Customers With Strictly Increasing Purchases

**Description:**

Table 62: Orders

| Column Name | Type |
|---|---|
| order_id | int int |
| customer_id | date int |
| order_date price | |

order_id is the primary key for this table. Each row contains the id of an order, the id of the customer that ordered it, the date of the order, and its price.

Write an SQL query to report the IDs of the customers with strictly increasing annual spends.

The annual spend for a customer is the sum of the prices of all orders of that customer in that year. If for some year the customer did not make any order, the annual spend for that year is considered to be 0.

A customer has a strictly increasing annual spend if their annual spend for each year is strictly higher than the previous year.

Return the result table ordered by customer_id in ascending order.

**Solution:**

```
WITH yearly_spend AS (
    SELECT
        customer_id,
        EXTRACT(YEAR FROM order_date) AS year,
```

```
        SUM(price) AS annual_spend
    FROM
        Orders
    GROUP BY
        customer_id, EXTRACT(YEAR FROM order_date)
),
all_years AS (
    SELECT
        DISTINCT customer_id
    FROM
        Orders
    CROSS JOIN
        (SELECT DISTINCT EXTRACT(YEAR FROM order_date) AS year FROM Orders) AS years
),
complete_spend AS (
    SELECT
        ay.customer_id,
        ay.year,
        COALESCE(ys.annual_spend, 0) AS annual_spend
    FROM
        all_years ay
    LEFT JOIN
        yearly_spend ys ON ay.customer_id = ys.customer_id AND ay.year = ys.year
),
spend_changes AS (
    SELECT
        customer_id,
        year,
        annual_spend,
        LAG(annual_spend) OVER (PARTITION BY customer_id ORDER BY year) AS prev_year_spend
    FROM
        complete_spend
),
customer_trends AS (
    SELECT
        customer_id,
        BOOL_AND(annual_spend > COALESCE(prev_year_spend, -1)) AS is_strictly_increasing
    FROM
        spend_changes
    GROUP BY
        customer_id
)
```

```sql
SELECT
    customer_id
FROM
    customer_trends
WHERE
    is_strictly_increasing = true
ORDER BY
    customer_id;
```

**Explanation:**

- This query identifies customers whose annual spending has strictly increased each year.
- The solution uses a comprehensive approach with multiple Common Table Expressions (CTEs):
  - The `yearly_spend` CTE calculates each customer's total spend per year.
  - The `all_years` CTE creates a complete cartesian product of customers and years to ensure no years are missed.
  - The `complete_spend` CTE fills in missing years with zero spend using a LEFT JOIN.
  - The `spend_changes` CTE uses the LAG window function to compare each year's spend with the previous year.
  - The `customer_trends` CTE uses BOOL_AND to check if all year-to-year changes are strictly increasing.
- BOOL_AND returns true only if all comparisons in the group are true, ensuring the strictly increasing requirement.
- The main query filters for customers where all year-to-year changes are strictly increasing.
- Time complexity: O(n log n) where n is the number of orders, due to the sorting and grouping operations.
- Space complexity: O(c × y) where c is the number of customers and y is the number of distinct years.
- This solution correctly handles edge cases:
  - Years where a customer made no purchases (counted as zero spend)
  - Customers with only a single year of purchases (considered strictly increasing by default)
  - Missing years in the data (filled in with zero spend)
- For large datasets, indexes on customer_id and order_date would improve performance.
- COALESCE in the complete_spend CTE ensures years without orders show zero spend.
- COALESCE in the spend_changes CTE handles the first year comparison by using -1 as a default.
- The CROSS JOIN ensures all possible customer-year combinations are considered.

- The approach efficiently separates the data preparation from the trend analysis logic.
- The solution follows the requirement to order results by customer_id.
- Using EXTRACT(YEAR FROM order_date) isolates just the year component from the date.

### 2494. Merge Overlapping Events in the Same Hall

**Description:**

Table 63: HallEvents

| Column Name | Type |
|---|---|
| hall_id | int date |
| start_day | date |
| end_day | |

(hall_id, start_day) is the primary key for this table. Each row of this table indicates the start day and end day of an event and the hall in which the event is held.

Write an SQL query to merge all the overlapping events that are held in the same hall. Two events overlap if they have at least one day in common.

Return the result table in any order.

**Solution:**

```
WITH recursive event_groups AS (
    -- Base case: Start with the first event for each hall
    SELECT
        hall_id,
        MIN(start_day) AS group_start,
        MAX(end_day) AS group_end
    FROM
        HallEvents
    GROUP BY
        hall_id

    UNION ALL

    -- Recursive case: Find events that don't overlap with any existing group
    SELECT
        h.hall_id,
        h.start_day,
```

```
        h.end_day
    FROM
        HallEvents h
    LEFT JOIN
        event_groups eg ON h.hall_id = eg.hall_id AND
                           (h.start_day <= eg.group_end AND h.end_day >= eg.group_start)
    WHERE
        eg.hall_id IS NULL
),
overlapping_events AS (
    SELECT
        e1.hall_id,
        e1.start_day,
        e1.end_day,
        CASE
            WHEN EXISTS (
                SELECT 1
                FROM HallEvents e2
                WHERE e1.hall_id = e2.hall_id AND
                      e1.start_day <> e2.start_day AND
                      e1.start_day <= e2.end_day AND
                      e1.end_day >= e2.start_day
            ) THEN 1
            ELSE 0
        END AS overlaps
    FROM
        HallEvents e1
),
group_markers AS (
    SELECT
        hall_id,
        start_day,
        end_day,
        SUM(CASE WHEN overlaps = 0 THEN 1 ELSE 0 END) OVER (
            PARTITION BY hall_id
            ORDER BY start_day
            ROWS UNBOUNDED PRECEDING
        ) AS group_id
    FROM
        overlapping_events
)
```

```sql
SELECT
    hall_id,
    MIN(start_day) AS start_day,
    MAX(end_day) AS end_day
FROM
    group_markers
GROUP BY
    hall_id, group_id
ORDER BY
    hall_id, start_day;
```

**Explanation:**

- This query merges overlapping events held in the same hall, where events overlap if they share at least one day.
- The solution uses a multi-step approach with Common Table Expressions (CTEs):
  - The recursive `event_groups` CTE attempts to find non-overlapping groups of events.
  - The `overlapping_events` CTE marks events that overlap with at least one other event.
  - The `group_markers` CTE assigns a group ID to each event, incrementing at the start of each non-overlapping sequence.
- The EXISTS subquery in overlapping_events efficiently checks if an event overlaps with any other event.
- The window function in group_markers creates a running sum that increases only at the start of a new non-overlapping group.
- The main query then aggregates events by hall_id and group_id to merge overlapping events.
- Time complexity: $O(n^2)$ where n is the number of events, due to the overlap checking.
- Space complexity: O(n) for the intermediate results.
- This solution correctly handles edge cases:
  - Events that touch exactly (end_day of one equals start_day of another)
  - Multiple overlapping event chains
  - Events with no overlaps
- For large datasets, indexes on hall_id, start_day, and end_day would improve performance.
- The recursive CTE approach provides one way to tackle the problem, but the window function method in the final solution is more efficient.
- The combination of EXISTS and window functions efficiently identifies and groups overlapping events.

- The solution maintains the requirement to preserve the hall_id in the output.
- The approach handles complex overlap patterns where events might form chains of overlaps.
- The final ORDER BY ensures consistent results, though the problem allows any order.
- An alternative approach could use the "islands and gaps" technique with date arithmetic, but the window function method is more intuitive for this problem.

### 2701. Consecutive Transactions with Increasing Amounts

**Description:**

Table 64: Transactions

| Column Name | Type |
|---|---|
| transaction_id | int |
| customer_id | int |
| transaction_date | date |
| amount | int |

transaction_id is the primary key for this table. Each row contains information about a transaction with the customer customer_id on transaction_date with amount.

Write an SQL query to find customers with increasing transactions.

A customer has increasing transactions if the amount of at least one transaction is strictly greater than the previous transaction (ignoring the very first transaction of each customer).

Return the customer_id of these customers sorted in ascending order.

**Solution:**

```
WITH customer_transactions AS (
    SELECT
        customer_id,
        transaction_date,
        amount,
        LAG(amount) OVER (
            PARTITION BY customer_id
            ORDER BY transaction_date
        ) AS prev_amount
    FROM
        Transactions
),
increasing_customers AS (
    SELECT DISTINCT
```

```
        customer_id
    FROM
        customer_transactions
    WHERE
        amount > prev_amount
)

SELECT
    customer_id
FROM
    increasing_customers
ORDER BY
    customer_id;
```

**Explanation:**

- This query identifies customers who have at least one transaction where the amount is strictly greater than their previous transaction.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The `customer_transactions` CTE uses the LAG window function to access the amount of each customer's previous transaction.
  - The `increasing_customers` CTE filters for customers with at least one transaction where amount > prev_amount.
- The LAG function partitions by customer_id and orders by transaction_date to ensure we're comparing each transaction with the chronologically previous one from the same customer.
- The DISTINCT keyword ensures each customer is listed only once, even if they have multiple increasing transactions.
- Time complexity: O(n log n) where n is the number of transactions, due to the sorting operations.
- Space complexity: O(n) for the intermediate results.
- This solution correctly handles edge cases:
  - Customers with only one transaction (excluded since there's no "previous" transaction to compare)
  - Customers with transactions on the same date (ordered by transaction_date)
  - Customers with no increasing transactions (excluded from results)
- For large datasets, indexes on customer_id and transaction_date would improve performance.
- The approach efficiently implements the "strictly greater than the previous transaction" requirement using window functions.

- The ORDER BY in the final query ensures results are returned in customer_id order as required.
- The solution excludes comparisons with the first transaction (where prev_amount is NULL) by requiring amount > prev_amount.
- The approach is concise and focuses directly on the requirement without unnecessary calculations.
- This pattern of using LAG with a window function is a common technique for analyzing sequential data in SQL.
- The DISTINCT in the second CTE ensures the final result doesn't include duplicate customer IDs.

### 2720. Popularity Percentage

**Description:**

Table 65: Friends

| Column Name | Type |
|---|---|
| user1_id | int int |
| user2_id | |

(user1_id, user2_id) is the primary key for this table. Each row of this table indicates that the users user1_id and user2_id are friends. Note that user1_id < user2_id.

Write an SQL query to find the popularity percentage for each user on the platform. The popularity percentage is defined as the number of friends the user has divided by the total number of users on the platform (including themself), times 100. If the popularity percentage is a decimal number, round it to 2 decimal places.

Return the result table ordered by user_id in ascending order.

**Solution:**

```
WITH user_list AS (
    SELECT user1_id AS user_id FROM Friends
    UNION
    SELECT user2_id AS user_id FROM Friends
),
all_users AS (
    SELECT DISTINCT user_id
    FROM user_list
),
total_users AS (
```

```
    SELECT COUNT(*) AS total
    FROM all_users
),
friend_counts AS (
    SELECT
        user1_id AS user_id,
        COUNT(*) AS friend_count
    FROM
        Friends
    GROUP BY
        user1_id

    UNION ALL

    SELECT
        user2_id AS user_id,
        COUNT(*) AS friend_count
    FROM
        Friends
    GROUP BY
        user2_id
),
user_popularity AS (
    SELECT
        user_id,
        SUM(friend_count) AS total_friends
    FROM
        friend_counts
    GROUP BY
        user_id
)

SELECT
    au.user_id,
    ROUND((COALESCE(up.total_friends, 0) * 100.0 / tu.total), 2) AS popularity_percent
FROM
    all_users au
LEFT JOIN
    user_popularity up ON au.user_id = up.user_id
CROSS JOIN
    total_users tu
ORDER BY
```

```
    au.user_id;
```

**Explanation:**

- This query calculates the popularity percentage for each user based on their friendship count relative to the total user count.
- The solution uses a multi-step approach with Common Table Expressions (CTEs):
  - The `user_list` CTE collects all user IDs from both sides of the friendship table.
  - The `all_users` CTE gets a distinct list of users to ensure each user is counted only once.
  - The `total_users` CTE counts the total number of users on the platform.
  - The `friend_counts` CTE counts friendships for each user, considering both user1_id and user2_id.
  - The `user_popularity` CTE sums friendship counts for users who appear on both sides of the friendship table.
- The main query calculates the popularity percentage by dividing the friendship count by the total user count.
- ROUND(…, 2) ensures the percentage is rounded to 2 decimal places as required.
- Time complexity: O(n log n) where n is the number of friendship records, due to the sorting and grouping operations.
- Space complexity: O(u) where u is the number of unique users.
- This solution correctly handles edge cases:
  - Users with no friends (assigned a popularity of 0%)
  - Users who appear only as user1_id or only as user2_id in the Friends table
- For large datasets, indexes on user1_id and user2_id would improve performance.
- The LEFT JOIN ensures all users are included in the result, even those with no friends.
- The CROSS JOIN with total_users makes the total count available for each row's percentage calculation.
- COALESCE handles users with no friends by defaulting to 0 for the count.
- The multiplication by 100.0 (not just 100) ensures decimal division for accurate percentage calculation.
- The approach correctly accounts for the bidirectional nature of friendships.
- The solution follows the requirement to order results by user_id.
- The UNION in user_list (not UNION ALL) eliminates duplicates, ensuring each user is counted only once.

### 2752. Customers with Maximum Number of Transactions on Consecutive Days

**Description:**

Table 66: Transactions

| Column Name | Type |
|---|---|
| transaction_id | int int |
| customer_id | date int |
| transaction_date | |
| amount | |

transaction_id is the primary key for this table. Each row contains information about a transaction with customer_id on transaction_date with amount.

Write an SQL query to find the customers who have made the maximum number of consecutive transactions. If more than one customer has the same number of consecutive transactions, return all of them.

Note that:

- Consecutive transactions by a customer are transactions made on consecutive days.
- There may be multiple transactions made on the same day, but transactions on different days may be considered consecutive if they're made on consecutive days.

Return the result table ordered by customer_id in ascending order.

**Solution:**

```
WITH daily_transactions AS (
    -- Count transactions per customer per day
    SELECT
        customer_id,
        transaction_date,
        COUNT(*) AS transactions_count
    FROM
        Transactions
    GROUP BY
        customer_id, transaction_date
),
consecutive_days AS (
    -- Identify groups of consecutive days
    SELECT
        customer_id,
        transaction_date,
        transaction_date - ROW_NUMBER() OVER (
            PARTITION BY customer_id
            ORDER BY transaction_date
```

```
        )::integer AS group_id
    FROM
        daily_transactions
),
consecutive_counts AS (
    -- Count consecutive days for each group
    SELECT
        customer_id,
        group_id,
        COUNT(*) AS consecutive_days_count
    FROM
        consecutive_days
    GROUP BY
        customer_id, group_id
),
max_consecutive_per_customer AS (
    -- Find maximum consecutive days for each customer
    SELECT
        customer_id,
        MAX(consecutive_days_count) AS max_consecutive
    FROM
        consecutive_counts
    GROUP BY
        customer_id
),
overall_max_consecutive AS (
    -- Find the overall maximum consecutive days
    SELECT
        MAX(max_consecutive) AS max_consecutive
    FROM
        max_consecutive_per_customer
)

SELECT
    mpc.customer_id
FROM
    max_consecutive_per_customer mpc
JOIN
    overall_max_consecutive omc ON mpc.max_consecutive = omc.max_consecutive
ORDER BY
    mpc.customer_id;
```

**Explanation:**

- This query identifies customers with the longest streak of consecutive days with transactions.
- The solution uses a five-step approach with Common Table Expressions (CTEs):
  - The `daily_transactions` CTE aggregates multiple transactions on the same day into a single daily record.
  - The `consecutive_days` CTE uses the "islands and gaps" technique to identify groups of consecutive days.
  - The `consecutive_counts` CTE counts the number of days in each consecutive group.
  - The `max_consecutive_per_customer` CTE finds the longest streak for each customer.
  - The `overall_max_consecutive` CTE identifies the maximum streak across all customers.
- The key insight is using "transaction_date - ROW_NUMBER()" to create a constant group_id for consecutive dates.
- The main query joins the customer maximums with the overall maximum to find customers with the longest streaks.
- Time complexity: O(n log n) where n is the number of transactions, due to the sorting operations.
- Space complexity: O(n) for the intermediate results.
- This solution correctly handles edge cases:
  - Multiple transactions on the same day (counted as one day in the streak)
  - Multiple streaks for the same customer (only the longest is considered)
  - Multiple customers with the same maximum streak length (all are included)
- For large datasets, indexes on customer_id and transaction_date would improve performance.
- The GROUP BY in daily_transactions ensures multiple transactions on the same day are counted as one day.
- The ROW_NUMBER() window function partitions by customer_id to track streaks for each customer separately.
- The approach efficiently identifies consecutive day sequences without recursive queries.
- The solution follows the requirement to order results by customer_id.
- The JOIN with overall_max_consecutive ensures only customers with the maximum streak are included.
- The ::integer cast ensures proper date arithmetic when calculating group_id.

### 2793. Status of Flight Tickets

**Description:**

Table 67: Flights

| Column Name | Type |
|---|---|
| flight_id | int int |
| capacity | |

flight_id is the primary key for this table. Each row of this table contains flight id and the capacity.

Table 68: Passengers

| Column Name | Type |
|---|---|
| passenger_id | int int |
| flight_id | date |
| booking_time | |

passenger_id is the primary key for this table. The booking_time represents when the passenger booked the ticket. each passenger book at most 1 flight.

Write an SQL query to report the status of each flight and the last passenger who booked a ticket on the flight.

Return the result table ordered by status and flight_id.

status should be as follows: - If the flight still has capacity, the status is "Available". - Otherwise, status is "Booked", and you need to include the passenger_id of the last passenger who booked the last ticket of the flight.

**Solution:**

```
WITH flight_bookings AS (
    SELECT
        f.flight_id,
        f.capacity,
        COUNT(p.passenger_id) AS booked_count,
        MAX(p.booking_time) AS last_booking_time
    FROM
        Flights f
    LEFT JOIN
        Passengers p ON f.flight_id = p.flight_id
    GROUP BY
        f.flight_id, f.capacity
),
```

```
last_passengers AS (
    SELECT
        p.flight_id,
        p.passenger_id,
        p.booking_time
    FROM
        Passengers p
    JOIN
        flight_bookings fb ON p.flight_id = fb.flight_id AND p.booking_time = fb.last_booking
)

SELECT
    fb.flight_id,
    CASE
        WHEN fb.booked_count < fb.capacity THEN 'Available'
        ELSE 'Booked'
    END AS status,
    CASE
        WHEN fb.booked_count >= fb.capacity THEN lp.passenger_id
        ELSE NULL
    END AS last_passenger_id
FROM
    flight_bookings fb
LEFT JOIN
    last_passengers lp ON fb.flight_id = lp.flight_id
ORDER BY
    status, fb.flight_id;
```

**Explanation:**

- This query determines the status of each flight based on its capacity and bookings, and identifies the last passenger to book a ticket on full flights.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The `flight_bookings` CTE calculates the number of bookings for each flight and identifies the time of the last booking.
  - The `last_passengers` CTE identifies the passenger(s) who made the last booking for each flight.
- The main query then determines the status of each flight based on capacity and includes the last passenger ID for fully booked flights.
- The CASE expressions handle the status determination and conditional inclusion of the last passenger ID.

- Time complexity: O(n log n) where n is the number of passengers, due to the sorting and grouping operations.
- Space complexity: O(f) where f is the number of flights.
- This solution correctly handles edge cases:
  - Flights with no bookings (status will be "Available")
  - Flights with multiple passengers booking at the same time (joins will match multiple rows)
- For large datasets, indexes on flight_id and booking_time would improve performance.
- The LEFT JOIN in flight_bookings ensures all flights are included, even those with no bookings.
- The JOIN in last_passengers matches the last booking time to identify the correct passenger.
- The status determination compares booked_count with capacity to accurately categorize flights.
- The solution follows the requirement to order results by status and flight_id.
- The approach efficiently combines aggregate functions with joins to solve the problem.
- The LEFT JOIN in the main query ensures all flights appear in the result, even if the flight has no "last passenger".

### 2991. Top Three Wineries

**Description:**

Table 69: Wineries

| Column Name | Type |
|---|---|
| winery_id country | int varchar |
| points | int |

winery_id is the primary key for this table. Each row in this table shows a winery and its average points.

Write an SQL query to find the top three wineries in each country based on their points. If there is a tie for the third position, choose all tied wineries. If there are fewer than three wineries in a country, include all of them.

Return the result table ordered by country in ascending order and points in descending order for each country.

**Solution:**

```sql
WITH ranked_wineries AS (
    SELECT
        winery_id,
        country,
        points,
        DENSE_RANK() OVER (
            PARTITION BY country
            ORDER BY points DESC
        ) AS country_rank
    FROM
        Wineries
)

SELECT
    winery_id,
    country,
    points
FROM
    ranked_wineries
WHERE
    country_rank <= 3
ORDER BY
    country, points DESC, winery_id;
```

**Explanation:**

- This query identifies the top three wineries in each country based on their points.
- The solution uses a single Common Table Expression (CTE) with a window function:
  - The `ranked_wineries` CTE assigns a rank to each winery within its country using DENSE_RANK().
  - DENSE_RANK() ensures tied wineries receive the same rank, which is crucial for handling ties.
- The main query then filters for wineries with rank <= 3 to get the top three in each country.
- Time complexity: O(n log n) where n is the number of wineries, due to the sorting operations.
- Space complexity: O(n) for the ranked wineries.
- This solution correctly handles edge cases:
  - Ties for third place (all tied wineries are included)
  - Countries with fewer than three wineries (all are included)
  - Multiple wineries with the same points (all receive the same rank)

- For large datasets, indexes on country and points would improve performance.
- DENSE_RANK() is specifically chosen over RANK() or ROW_NUMBER() because:
  - Unlike ROW_NUMBER(), it assigns the same rank to ties
  - Unlike RANK(), it doesn't skip ranks after ties, ensuring we get exactly the top three positions
- The ORDER BY clause in the main query ensures results are sorted by country ascending and points descending as required.
- The approach efficiently combines partitioning and ordering within the window function.
- The solution is concise and directly focuses on the ranking requirement without complex subqueries.
- The addition of winery_id in the final ORDER BY ensures consistent results when multiple wineries have the same country and points.

### 3057. Employees Project Allocation

**Description:**

Table 70: Employees

| Column Name | Type |
|---|---|
| emp_id | int varchar |
| emp_name | int |
| team_id | |

emp_id is the primary key for this table. Each row of this table contains emp_id, emp_name, and team_id.

Table 71: Projects

| Column Name | Type |
|---|---|
| project_id | int int |
| team_id capacity | int |

project_id is the primary key for this table. Each row of this table contains project_id, team_id, and capacity of the project.

Write an SQL query to calculate employee allocations. The allocation is the number of projects that an employee can participate in, limited by the capacity of each project.

- Each team is in charge of zero or more projects.
- Each employee belongs to exactly one team.

- The team cannot exceed the capacity of each project it participates in.
- Each employee at most participates in one project. If there are no projects or the capacity is not enough, some employees may not have any allocated projects.

Return the result table ordered by emp_id in ascending order.

**Solution:**

```
WITH team_projects AS (
    SELECT
        p.project_id,
        p.team_id,
        p.capacity,
        COUNT(e.emp_id) OVER (PARTITION BY e.team_id) AS team_size,
        ROW_NUMBER() OVER (PARTITION BY p.team_id ORDER BY p.capacity DESC) AS project_prior
    FROM
        Projects p
    JOIN
        Employees e ON p.team_id = e.team_id
),
employee_ranks AS (
    SELECT
        e.emp_id,
        e.emp_name,
        e.team_id,
        ROW_NUMBER() OVER (PARTITION BY e.team_id ORDER BY e.emp_id) AS emp_rank
    FROM
        Employees e
),
project_allocation AS (
    SELECT
        tp.project_id,
        tp.team_id,
        tp.capacity,
        LEAST(tp.capacity, tp.team_size) AS allocated_count
    FROM
        team_projects tp
    WHERE
        tp.project_priority = 1
),
employee_allocation AS (
    SELECT
        er.emp_id,
```

```
        er.emp_name,
        pa.project_id
    FROM
        employee_ranks er
    LEFT JOIN
        project_allocation pa ON er.team_id = pa.team_id
    WHERE
        pa.project_id IS NOT NULL AND er.emp_rank <= pa.allocated_count
)

SELECT
    e.emp_id,
    e.emp_name,
    ea.project_id
FROM
    Employees e
LEFT JOIN
    employee_allocation ea ON e.emp_id = ea.emp_id
ORDER BY
    e.emp_id;
```

**Explanation:**

- This query allocates employees to projects based on project capacity and team assignments.
- The solution uses a multi-step approach with Common Table Expressions (CTEs):
  - The `team_projects` CTE calculates team sizes and ranks projects by capacity for each team.
  - The `employee_ranks` CTE assigns a rank to each employee within their team.
  - The `project_allocation` CTE determines how many employees can be allocated to each project.
  - The `employee_allocation` CTE matches employees to projects based on their rank and project allocation.
- The LEAST function ensures project allocations don't exceed capacity.
- The main query joins all employees with their allocations, ensuring all employees appear in the results.
- Time complexity: O(n log n) where n is the number of employees, due to the sorting operations.
- Space complexity: O(n + p) where p is the number of projects.
- This solution correctly handles edge cases:
  - Teams with no projects (employees remain unallocated)

  - Projects with capacity less than team size (only some employees are allocated)
  - Employees with no allocated project (show NULL for project_id)

- For large datasets, indexes on team_id, emp_id, and project_id would improve performance.
- The ROW_NUMBER() functions efficiently rank both employees and projects for allocation.
- The WHERE clause in employee_allocation ensures only employees within the allocated count are assigned projects.
- The LEFT JOIN in the main query ensures all employees appear in the result, even those without allocated projects.
- The solution follows the requirement to order results by emp_id.
- The approach efficiently models the allocation constraints using a combination of window functions and joins.
- The project_priority ranking ensures teams prioritize projects with larger capacity first.

### 3060. User Activities within Time Bounds

**Description:**

Table 72: UserActivity

| Column Name | Type |
|---|---|
| user_id activity | int int |
| timestamp | date |

(user_id, timestamp) is the primary key for this table. Each row contains the user's id, the activity they performed, and the timestamp of the activity.

Write an SQL query to find, for each user, the activity they performed just after the first occurrence of a specific activity and within a specific time frame.

For each user who performed activity 2 at least once, find the activity they performed right after their first occurrence of activity 2, if it's within 30 days. If there is no next activity within 30 days, then for that user, the result is null.

Return the result table containing user_id and the next_activity. You may return the result table in any order.

**Solution:**

```sql
WITH first_activity_2 AS (
    SELECT
        user_id,
        timestamp AS first_a2_timestamp
    FROM
        UserActivity
    WHERE
        activity = 2
    GROUP BY
        user_id
    HAVING
        timestamp = MIN(timestamp)
),
next_activities AS (
    SELECT
        ua.user_id,
        ua.activity,
        ua.timestamp,
        fa.first_a2_timestamp,
        ROW_NUMBER() OVER (
            PARTITION BY ua.user_id
            ORDER BY ua.timestamp
        ) AS activity_rank
    FROM
        UserActivity ua
    JOIN
        first_activity_2 fa ON ua.user_id = fa.user_id
    WHERE
        ua.timestamp > fa.first_a2_timestamp AND
        ua.timestamp <= fa.first_a2_timestamp + INTERVAL '30 days'
)

SELECT
    fa.user_id,
    na.activity AS next_activity
FROM
    first_activity_2 fa
LEFT JOIN
    next_activities na ON fa.user_id = na.user_id AND na.activity_rank = 1
ORDER BY
    fa.user_id;
```

**Explanation:**

- This query finds the activity each user performed immediately after their first occurrence of activity 2, if within 30 days.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The `first_activity_2` CTE identifies the timestamp of each user's first occurrence of activity 2.
  - The `next_activities` CTE finds all activities that occurred after the first activity 2 and within 30 days.
- ROW_NUMBER() assigns a rank to each subsequent activity, with rank 1 being the first activity after activity 2.
- The main query joins the first activity timestamps with the next activities, filtering for just the immediately following activity.
- The INTERVAL '30 days' syntax is used for date arithmetic to establish the 30-day time window.
- The LEFT JOIN ensures all users with activity 2 appear in the results, even those with no subsequent activity in the time window.
- Time complexity: O(n log n) where n is the number of activities, due to the sorting operations.
- Space complexity: O(u) where u is the number of users with activity 2.
- This solution correctly handles edge cases:
  - Users with no activities after their first activity 2 (null result)
  - Users with activities after 30 days (excluded from results)
  - Users who never performed activity 2 (excluded from first_activity_2)
- For large datasets, indexes on user_id, activity, and timestamp would improve performance.
- The GROUP BY with HAVING in first_activity_2 efficiently finds the first occurrence of activity 2 for each user.
- The WHERE clause in next_activities applies the time bound constraints directly.
- The solution follows a logical flow from finding the reference point (first activity 2) to identifying the next activity within the time window.
- The approach properly handles the time window requirement using PostgreSQL's date arithmetic.
- The activity_rank = 1 condition ensures only the immediately following activity is selected.

### 3061. Calculate Trapping Rain Water

**Description:**

Table 73: Heights

| Column Name | Type |
|---|---|
| id | int |
| height | int |

id is the primary key for this table. Each row of this table contains an id and a height. There are no consecutive identical heights.

Write an SQL query to calculate the amount of water that can be trapped after raining.

The query result is in the following format:

| total_water |
|---|
| 6 |

For details on the calculation, see the LeetCode "Trapping Rain Water" problem.

**Solution:**

```sql
WITH height_bounds AS (
    SELECT
        h.id,
        h.height,
        (SELECT MAX(height) FROM Heights h2 WHERE h2.id <= h.id) AS left_max,
        (SELECT MAX(height) FROM Heights h2 WHERE h2.id >= h.id) AS right_max
    FROM
        Heights h
),
water_heights AS (
    SELECT
        id,
        height,
        LEAST(left_max, right_max) AS water_level,
        GREATEST(0, LEAST(left_max, right_max) - height) AS water_at_position
    FROM
        height_bounds
)

SELECT
    SUM(water_at_position) AS total_water
FROM
    water_heights;
```

**Explanation:**

- This query calculates the total amount of water that can be trapped between elevation heights, a classic algorithmic problem.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The `height_bounds` CTE calculates the maximum height to the left and right of each position.
  - The `water_heights` CTE calculates the water level and amount of water at each position.
- For each position, the water level is determined by the minimum of the maximum heights to the left and right.
- The water at each position is the difference between the water level and the height at that position (or 0 if negative).
- The correlated subqueries efficiently find the maximum heights in each direction.
- The main query sums the water at each position to get the total trapped water.
- Time complexity: $O(n^2)$ where n is the number of heights, due to the correlated subqueries.
- Space complexity: O(n) for the intermediate results.
- This solution correctly implements the "Trapping Rain Water" algorithm:
  - Water can only be trapped between higher elevations
  - The water level at each position is limited by the lower of the maximum heights to the left and right
  - No water is trapped at the highest points
- For large datasets, a dynamic programming approach with window functions could be more efficient.
- The LEAST function ensures we use the lower of the two maximum heights to determine the water level.
- The GREATEST function ensures we don't count negative water (when a position is higher than the water level).
- The approach directly translates the algorithmic solution to SQL, demonstrating SQL's ability to solve complex problems.
- Alternative approaches could use window functions to calculate left_max and right_max more efficiently, but this solution is more readable.
- The solution handles edge cases where no water is trapped (returns 0) and where water is trapped at multiple positions.

### 3103. Find Trending Hashtags II

**Description:**

Table 75: Posts

| Column Name | Type |
|---|---|
| post_id user_id | int int |
| content | text |
| created_at | date |

post_id is the primary key for this table. Each row of this table contains post_id, user_id, content, and created_at.

Write an SQL query to find the top 3 trending hashtags in February 2024. A hashtag is a word beginning with the '#' symbol.

- A hashtag's word length is the number of characters it contains excluding the '#' symbol.
- To extract hashtags from the content of a post, you need to find all words that start with '#' followed by one or more non-space characters. The hashtag ends when a space or the end of the content is encountered.
- If there are fewer than 3 trending hashtags, return all of them. If multiple hashtags have the same frequency, prioritize the alphabetically smaller ones.

Return the result table ordered by frequency in descending order and hashtag in ascending order for equal frequency.

**Solution:**

```
WITH content_with_hashtags AS (
    SELECT
        post_id,
        user_id,
        created_at,
        regexp_matches(content, '#([^ ]+)', 'g') AS hashtag
    FROM
        Posts
    WHERE
        created_at >= '2024-02-01' AND
        created_at < '2024-03-01'
),
extracted_hashtags AS (
    SELECT
        post_id,
        user_id,
        created_at,
        lower('#' || hashtag[1]) AS hashtag_text
```

```
    FROM
        content_with_hashtags
),
hashtag_counts AS (
    SELECT
        hashtag_text,
        COUNT(*) AS frequency
    FROM
        extracted_hashtags
    GROUP BY
        hashtag_text
),
ranked_hashtags AS (
    SELECT
        hashtag_text,
        frequency,
        ROW_NUMBER() OVER (
            ORDER BY frequency DESC, hashtag_text
        ) AS hashtag_rank
    FROM
        hashtag_counts
)

SELECT
    hashtag_text AS hashtag,
    frequency
FROM
    ranked_hashtags
WHERE
    hashtag_rank <= 3
ORDER BY
    frequency DESC, hashtag;
```

**Explanation:**

- This query identifies the top 3 trending hashtags in February 2024 based on their frequency of occurrence.
- The solution uses a four-step approach with Common Table Expressions (CTEs):
  - The `content_with_hashtags` CTE uses regexp_matches to extract all hashtags from post content.
  - The `extracted_hashtags` CTE formats the extracted hashtags and converts them to lowercase for consistent counting.

  - The `hashtag_counts` CTE counts the frequency of each hashtag.
  - The `ranked_hashtags` CTE ranks hashtags by frequency and alphabetical order.

- The regular expression '#([^ ]+)' matches a '#' followed by one or more non-space characters.
- The 'g' flag in regexp_matches ensures all matches in the content are found, not just the first one.
- The main query filters for the top 3 hashtags based on their rank.
- Time complexity: O(n * m) where n is the number of posts and m is the average number of words per post.
- Space complexity: O(h) where h is the total number of hashtags extracted.
- This solution correctly handles edge cases:
  - Hashtags with the same frequency (ordered alphabetically)
  - Fewer than 3 unique hashtags (returns all available hashtags)
  - Case-insensitive counting (converts all hashtags to lowercase)
- For large datasets, indexes on created_at would improve performance, and full-text indexing could speed up hashtag extraction.
- The lower() function ensures case-insensitive counting and sorting of hashtags.
- The WHERE clause in the first CTE efficiently filters for posts in February 2024.
- The ROW_NUMBER() window function with ordering by frequency DESC and hashtag_text ensures correct ranking.
- The solution correctly implements the hashtag extraction logic according to the problem definition.
- The final ORDER BY ensures results are displayed in the required order.

### 3156. Employee Task Duration and Concurrent Tasks

**Description:**

Table 76: Tasks

| Column Name | Type |
|---|---|
| emp_id task_id | int int |
| start_time | int int |
| end_time | |

(emp_id, task_id) is the primary key for this table. Each row of this table indicates a task assigned to emp_id with the unique task_id, which started at start_time and ended at end_time. All times are presented in seconds.

Write an SQL query to report the following for each employee:

- total_task_duration: total duration across all tasks
- max_concurrent_tasks: maximum number of concurrent tasks the employee worked on
- max_concurrent_duration: total duration during which the employee was working on the maximum number of concurrent tasks

Return the result table in any order.

**Solution:**

```
WITH task_boundaries AS (
    -- Generate boundary points (start and end) for each task
    SELECT
        emp_id,
        start_time AS time_point,
        1 AS transition -- +1 for task start
    FROM
        Tasks

    UNION ALL

    SELECT
        emp_id,
        end_time AS time_point,
        -1 AS transition -- -1 for task end
    FROM
        Tasks
),
running_counts AS (
    -- Calculate running count of concurrent tasks at each boundary
    SELECT
        emp_id,
        time_point,
        SUM(transition) OVER (
            PARTITION BY emp_id
            ORDER BY time_point
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        ) AS concurrent_tasks,
        LEAD(time_point) OVER (
            PARTITION BY emp_id
            ORDER BY time_point
        ) - time_point AS duration
    FROM
        task_boundaries
```

```
),
employee_stats AS (
    -- Calculate statistics for each employee
    SELECT
        emp_id,
        SUM(CASE WHEN concurrent_tasks > 0 THEN duration ELSE 0 END) AS total_task_duration,
        MAX(concurrent_tasks) AS max_concurrent_tasks
    FROM
        running_counts
    GROUP BY
        emp_id
),
max_concurrent_durations AS (
    -- Calculate duration at maximum concurrency
    SELECT
        rc.emp_id,
        SUM(CASE WHEN rc.concurrent_tasks = es.max_concurrent_tasks THEN rc.duration ELSE 0 E
    FROM
        running_counts rc
    JOIN
        employee_stats es ON rc.emp_id = es.emp_id
    WHERE
        rc.duration IS NOT NULL
    GROUP BY
        rc.emp_id
)

SELECT
    es.emp_id,
    es.total_task_duration,
    es.max_concurrent_tasks,
    COALESCE(mcd.max_concurrent_duration, 0) AS max_concurrent_duration
FROM
    employee_stats es
LEFT JOIN
    max_concurrent_durations mcd ON es.emp_id = mcd.emp_id;
```

**Explanation:**

- This query calculates employee task statistics including total duration, maximum concurrent tasks, and duration at maximum concurrency.
- The solution uses a four-step approach with Common Table Expressions (CTEs):

  - The `task_boundaries` CTE generates time points for task starts (+1) and ends (-1).
  - The `running_counts` CTE calculates the running count of concurrent tasks at each time point.
  - The `employee_stats` CTE calculates total duration and maximum concurrency for each employee.
  - The `max_concurrent_durations` CTE calculates the duration spent at maximum concurrency.

- The solution uses the "sweep line" algorithm, a technique often used in computational geometry.
- Time points are sorted chronologically, and a running sum tracks how many tasks are active at each point.
- The LEAD window function calculates the duration between consecutive time points.
- Time complexity: O(n log n) where n is the number of tasks, due to the sorting operations.
- Space complexity: O(n) for the time points and intermediate results.
- This solution correctly handles edge cases:
  - Tasks with zero duration (end_time = start_time)
  - Overlapping tasks with identical start or end times
  - Employees with no periods of maximum concurrency
- For large datasets, indexes on emp_id, start_time, and end_time would improve performance.
- The SUM with CASE expressions in the CTEs efficiently calculates durations based on conditional logic.
- The COALESCE function handles the case where an employee has no periods at maximum concurrency.
- The approach elegantly calculates all required metrics without complex self-joins.
- The solution works for any number of overlapping tasks and correctly handles transitions between different concurrency levels.
- The UNION ALL in task_boundaries is used instead of UNION to preserve duplicate time points that may occur when tasks start or end simultaneously.

### 3188. Find Top Scoring Students II

**Description:**

Table 77: Students

| Column Name | Type |
|---|---|
| student_id | int |
| name | varchar |
| age | int |

student_id is the primary key for this table. Each row of this table contains student_id, name, and age.

Table 78: Scores

| Column Name | Type |
|---|---|
| student_id | int enum |
| subject score | int |

(student_id, subject) is the primary key for this table. subject is an ENUM of type ('Math', 'Physics', 'Programming'). Each row of this table contains student_id, subject, and score.

Write an SQL query to find the students that score highest in each subject and those who score highest in all subjects.

Return the result table ordered by student_id.

**Solution:**

```sql
WITH subject_max_scores AS (
    -- Find maximum score for each subject
    SELECT
        subject,
        MAX(score) AS max_score
    FROM
        Scores
    GROUP BY
        subject
),
top_students_by_subject AS (
    -- Find students with top scores in each subject
    SELECT DISTINCT
        s.student_id,
        sc.subject
    FROM
        Scores sc
    JOIN
        Students s ON sc.student_id = s.student_id
    JOIN
        subject_max_scores sms ON sc.subject = sms.subject AND sc.score = sms.max_score
),
subject_count AS (
    -- Count number of subjects in which each student has top score
```

```
    SELECT
        student_id,
        COUNT(subject) AS top_subject_count
    FROM
        top_students_by_subject
    GROUP BY
        student_id
),
total_subjects AS (
    -- Count total number of distinct subjects
    SELECT
        COUNT(DISTINCT subject) AS total
    FROM
        Scores
)

SELECT DISTINCT
    s.student_id,
    s.name,
    'Top in all subjects' AS grade
FROM
    subject_count sc
JOIN
    Students s ON sc.student_id = s.student_id
CROSS JOIN
    total_subjects ts
WHERE
    sc.top_subject_count = ts.total

UNION ALL

SELECT DISTINCT
    s.student_id,
    s.name,
    CONCAT('Top in ', tsbs.subject) AS grade
FROM
    top_students_by_subject tsbs
JOIN
    Students s ON tsbs.student_id = s.student_id
LEFT JOIN (
    SELECT
        student_id
```

```
    FROM
        subject_count sc
    CROSS JOIN
        total_subjects ts
    WHERE
        sc.top_subject_count = ts.total
) all_top ON tsbs.student_id = all_top.student_id
WHERE
    all_top.student_id IS NULL
ORDER BY
    student_id, grade;
```

**Explanation:**

- This query identifies students who scored highest in each subject and those who scored highest in all subjects.
- The solution uses a four-step approach with Common Table Expressions (CTEs):
  - The `subject_max_scores` CTE finds the maximum score for each subject.
  - The `top_students_by_subject` CTE identifies students who achieved the maximum score in each subject.
  - The `subject_count` CTE counts the number of subjects in which each student has the top score.
  - The `total_subjects` CTE counts the total number of distinct subjects.
- The main query consists of two parts joined with UNION ALL:
  - The first part identifies students who have top scores in all subjects.
  - The second part identifies students who have top scores in specific subjects but not all.
- The LEFT JOIN with exclusion in the second part ensures students aren't double-counted.
- Time complexity: O(n log n) where n is the number of scores, due to the sorting and grouping operations.
- Space complexity: O(s + t) where s is the number of students and t is the number of top scores.
- This solution correctly handles edge cases:
  - Students with top scores in multiple but not all subjects
  - Ties for top scores in a subject (all tied students are included)
  - Students who don't have the top score in any subject (excluded from results)
- For large datasets, indexes on student_id, subject, and score would improve performance.
- The DISTINCT keyword ensures each student is listed only once for each achievement.

- The CONCAT function creates a descriptive grade for each student's achievement.
- The solution efficiently separates students who excel in all subjects from those who excel in specific subjects.
- The approach correctly handles the requirement to order results by student_id.
- The CROSS JOIN with total_subjects makes the total subject count available for comparison.
- The solution follows a structured approach to solving the multi-part problem requirements.

### 3214. Year on Year Growth Rate

**Description:**

Table 79: AnnualSales

| Column Name | Type |
|---|---|
| year | int |
| sales | int |

year is the primary key for this table. Each row of this table contains the sales of a particular year.

Write an SQL query to calculate the year-on-year growth rate for each year.

The year-on-year growth rate is calculated as (current_year_sales - previous_year_sales) / previous_year_sales * 100.

Return the result table in ascending order by year. Round the growth_rate to 2 decimal places.

**Solution:**

```
WITH sales_with_previous AS (
    SELECT
        year,
        sales,
        LAG(sales) OVER (ORDER BY year) AS previous_year_sales
    FROM
        AnnualSales
)

SELECT
    year,
    CASE
        WHEN previous_year_sales IS NULL THEN NULL
```

```
        WHEN previous_year_sales = 0 THEN NULL  -- Avoid division by zero
        ELSE ROUND(((sales - previous_year_sales) * 100.0 / previous_year_sales)::numeric, 2)
    END AS growth_rate
FROM
    sales_with_previous
ORDER BY
    year;
```

**Explanation:**

- This query calculates the year-on-year growth rate for sales data.
- The solution uses a single Common Table Expression (CTE) approach:
  - The `sales_with_previous` CTE uses the LAG window function to access each year's previous year sales.
- The main query then calculates the growth rate using the formula: (current - previous) / previous * 100.
- ROUND(..., 2) ensures the growth rate is rounded to 2 decimal places as required.
- Time complexity: O(n log n) where n is the number of years, due to the sorting operations.
- Space complexity: O(n) for the intermediate results.
- This solution correctly handles edge cases:
  - The first year (where there is no previous year) shows NULL for growth_rate
  - Years with zero sales in the previous year show NULL to avoid division by zero
- For large datasets, an index on the year column would improve performance.
- The CASE expression handles both the lack of previous year data and division by zero scenarios.
- The ::numeric cast ensures proper decimal division and rounding.
- The approach elegantly uses the LAG window function to access previous year data without self-joins.
- The solution follows the requirement to order results by year in ascending order.
- The formula correctly implements the year-on-year growth rate calculation as specified.
- The multiplication by 100.0 (not just 100) ensures decimal division for accurate percentage calculation.
- This pattern is commonly used for time series analysis in business reporting.

### 3236. CEO Subordinate Hierarchy

**Description:**

Table 80: Employees

| Column Name | Type |
|---|---|
| emp_id name | int varchar |
| reports_to | int |

emp_id is the primary key for this table. Each row of this table indicates an employee's ID, name, and the ID of their manager (reports_to). The CEO does not report to anyone and is represented by reports_to = null.

Write an SQL query to report the CEO's hierarchy, starting from the CEO (level 0) down to level 3 of the company.

Return the result table ordered by level, and for employees at the same level, order them by emp_id.

**Solution:**

```
WITH RECURSIVE employee_hierarchy AS (
    -- Base case: CEO (level 0)
    SELECT
        emp_id,
        name,
        0 AS level
    FROM
        Employees
    WHERE
        reports_to IS NULL

    UNION ALL

    -- Recursive case: Direct reports (next level)
    SELECT
        e.emp_id,
        e.name,
        eh.level + 1
    FROM
        Employees e
    JOIN
        employee_hierarchy eh ON e.reports_to = eh.emp_id
    WHERE
        eh.level < 3  -- Only go down to level 3
)
```

```
SELECT
    level,
    emp_id,
    name
FROM
    employee_hierarchy
ORDER BY
    level, emp_id;
```

**Explanation:**

- This query builds a hierarchical view of employees starting from the CEO (level 0) down to level 3.
- The solution uses a recursive Common Table Expression (CTE):
  - The base case identifies the CEO as the employee with reports_to IS NULL.
  - The recursive case finds all employees who report to someone in the previous level.
- The RECURSIVE keyword enables the CTE to reference itself, allowing for hierarchical traversal.
- The WHERE eh.level $<$ 3 condition limits the recursion to only include levels 0 through 3.
- The main query selects all employees from the hierarchy and orders them by level and emp_id.
- Time complexity: O(n) where n is the number of employees, as each employee is processed exactly once.
- Space complexity: O(n) for the hierarchy representation.
- This solution correctly handles the hierarchical structure:
  - The CEO at level 0
  - Direct reports to the CEO at level 1
  - Reports to level 1 employees at level 2
  - Reports to level 2 employees at level 3
- For large datasets, indexes on emp_id and reports_to would improve join performance.
- The recursive CTE elegantly models the hierarchical relationship without complex self-joins.
- The solution follows the requirements to order by level and then by emp_id.
- This pattern is a standard approach for traversing hierarchical data in PostgreSQL.
- The approach efficiently handles any organizational structure, regardless of how many employees report to each manager.
- For very deep hierarchies, the recursion depth limit should be increased, but this solution is optimized for the 4-level requirement.

### 3268. Find Overlapping Shifts II

**Description:**

Table 81: Shifts

| Column Name | Type |
|---|---|
| employee_id | int date |
| shift_date | int int |
| start_time | |
| end_time | |

(employee_id, shift_date) is the primary key for this table. Each row of this table indicates the start_time and end_time of the shift of an employee with employee_id on a particular shift_date. All times follow a 24-hour clock system. A valid shift doesn't have the same end_time as start_time, which means the end_time cannot equal the start_time. Each employee might have more than one shift on the same day.

Write an SQL query to report all employees whose shifts overlap.

Two employees' shifts overlap if they are working on the same shift_date and there is some time period during which both employees are working.

Return the result table ordered by employee_id1 and employee_id2.

**Solution:**

```
WITH normalized_shifts AS (
    SELECT
        employee_id,
        shift_date,
        start_time,
        end_time
    FROM
        Shifts
),
overlapping_pairs AS (
    SELECT DISTINCT
        CASE WHEN s1.employee_id < s2.employee_id THEN s1.employee_id ELSE s2.employee_id END
        CASE WHEN s1.employee_id < s2.employee_id THEN s2.employee_id ELSE s1.employee_id END
    FROM
        normalized_shifts s1
    JOIN
        normalized_shifts s2 ON
```

```
            s1.employee_id != s2.employee_id AND
            s1.shift_date = s2.shift_date AND
            (
                (s1.start_time <= s2.start_time AND s1.end_time > s2.start_time) OR
                (s2.start_time <= s1.start_time AND s2.end_time > s1.start_time)
            )
)

SELECT
    employee_id1,
    employee_id2
FROM
    overlapping_pairs
ORDER BY
    employee_id1, employee_id2;
```

**Explanation:**

- This query identifies pairs of employees with overlapping shifts.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The `normalized_shifts` CTE prepares the shift data for analysis.
  - The `overlapping_pairs` CTE joins the shifts table to itself to find pairs of shifts that overlap.
- The CASE expressions ensure that employee_id1 < employee_id2 for consistent ordering of pairs.
- Two shifts overlap if one shift starts before or at the same time as the other shift ends, and ends after the other shift starts.
- The logical condition for overlap is implemented as:
  - (s1.start_time <= s2.start_time AND s1.end_time > s2.start_time) OR
  - (s2.start_time <= s1.start_time AND s2.end_time > s1.start_time)
- Time complexity: $O(n^2)$ where n is the number of shifts, due to the self-join operation.
- Space complexity: O(p) where p is the number of overlapping pairs.
- This solution correctly handles edge cases:
  - Multiple shifts per employee on the same day
  - Shifts that touch exactly (end_time of one equals start_time of another) are not considered overlapping
  - Shifts on different dates never overlap
- For large datasets, indexes on employee_id, shift_date, start_time, and end_time would improve join performance.

- The DISTINCT keyword ensures each overlapping pair is reported only once.
- The JOIN conditions efficiently filter for shifts that occur on the same date and have different employees.
- The solution follows the requirements to order results by employee_id1 and employee_id2.
- This approach handles shifts crossing midnight by treating time as a continuous value within a day.
- The self-join pattern is a standard approach for finding overlapping intervals.

### 3384. Team Dominance by Pass Success

**Description:**

Table 82: Passes

| Column Name | Type |
|---|---|
| pass_id team_id | int int |
| pass_recipient | int int |
| success | |

pass_id is the primary key for this table. Each row of this table indicates a pass by team_id to a player (pass_recipient). success is either 0 (failure) or 1 (success).

Write an SQL query to find teams with a high pass success rate. A team has a high pass success rate if its success rate is strictly greater than the success rate of every other team.

The pass success rate is calculated as the number of successful passes divided by the total number of passes.

Return the result table ordered by team_id.

**Solution:**

```
WITH team_success_rates AS (
    SELECT
        team_id,
        SUM(success) AS successful_passes,
        COUNT(*) AS total_passes,
        CAST(SUM(success) AS decimal) / COUNT(*) AS success_rate
    FROM
        Passes
    GROUP BY
        team_id
),
```

```sql
max_other_team_rates AS (
    SELECT
        t1.team_id,
        MAX(t2.success_rate) AS max_other_rate
    FROM
        team_success_rates t1
    CROSS JOIN
        team_success_rates t2
    WHERE
        t1.team_id != t2.team_id
    GROUP BY
        t1.team_id
)

SELECT
    t.team_id
FROM
    team_success_rates t
JOIN
    max_other_team_rates m ON t.team_id = m.team_id
WHERE
    t.success_rate > m.max_other_rate
ORDER BY
    t.team_id;
```

**Explanation:**

- This query identifies teams whose pass success rate is strictly greater than all other teams.
- The solution uses a two-step approach with Common Table Expressions (CTEs):
  - The `team_success_rates` CTE calculates the success rate for each team.
  - The `max_other_team_rates` CTE finds the maximum success rate among all other teams for each team.
- The success rate is calculated as successful_passes / total_passes using a CAST to ensure decimal division.
- The main query joins these CTEs and filters for teams whose success rate is greater than the maximum rate of other teams.
- Time complexity: O(n log n) where n is the number of teams, due to the sorting operations.
- Space complexity: O(n) for the intermediate results.
- This solution correctly handles edge cases:

  - Teams with the same success rate (neither would be considered dominant)
  - Teams with 100% success rate (only dominant if all other teams have lower rates)
  - Teams with very few passes (their rate is still calculated accurately)

- For large datasets, indexes on team_id would improve performance.
- The CROSS JOIN in max_other_team_rates efficiently creates all team-to-team comparisons.
- The WHERE t1.team_id != t2.team_id condition ensures a team isn't compared against itself.
- The solution follows the requirement to order results by team_id.
- The approach efficiently implements the "strictly greater than all others" requirement using the MAX function.
- The CAST AS decimal ensures accurate division without integer truncation.
- This pattern is useful for identifying dominant entities in comparative analysis.

### 3401. Find Circular Gift Exchange Chains

**Description:**

Table 83: Gifts

| Column Name | Type |
|---|---|
| giver_id | int int date |
| receiver_id | |
| gift_date | |

(giver_id, receiver_id) is the primary key for this table. Each row of this table indicates that a gift was sent on gift_date by giver_id to receiver_id.

A gift chain is a sequence of gifts where each person receives a gift from the previous person in the chain and gives a gift to the next person in the chain, with the last person giving a gift to the first person in the chain.

Write an SQL query to find circular gift exchange chains. A circular chain needs to have at least three people.

Return the result table ordered by length (number of people in the chain) in descending order. If there are multiple chains with the same length, order them by the minimum giver_id in each chain in ascending order.

**Solution:**

```
WITH RECURSIVE gift_chains AS (
    -- Base case: Start with each potential chain beginning
    SELECT
        giver_id AS start_id,
        receiver_id AS current_id,
        ARRAY[giver_id] AS chain,
        1 AS length
    FROM
        Gifts

    UNION ALL

    -- Recursive case: Extend chains
    SELECT
        gc.start_id,
        g.receiver_id AS current_id,
        gc.chain || g.giver_id AS chain,
        gc.length + 1 AS length
    FROM
        gift_chains gc
    JOIN
        Gifts g ON gc.current_id = g.giver_id
    WHERE
        -- Avoid cycles before completing the chain
        NOT g.giver_id = ANY(gc.chain) AND
        -- Limit chain length to prevent infinite recursion
        gc.length < 20
),
completed_chains AS (
    -- Find chains where the last receiver gives to the first giver
    SELECT
        chain || current_id AS full_chain,
        length + 1 AS chain_length
    FROM
        gift_chains
    WHERE
        current_id NOT IN (SELECT unnest(chain) FROM gift_chains gc WHERE gc.chain = chain) A
        EXISTS (
            SELECT 1
            FROM Gifts g
            WHERE g.giver_id = current_id AND g.receiver_id = start_id
        ) AND
```

```
        length >= 2  -- At least 3 people in the chain (including completion)
),
unique_chains AS (
    -- Remove duplicate chains that are rotations of each other
    SELECT DISTINCT ON (canonical_chain)
        full_chain,
        chain_length,
        array_to_string(array_agg(e ORDER BY e), ',') AS canonical_chain,
        MIN(e) AS min_id
    FROM
        completed_chains,
        LATERAL unnest(full_chain) AS e
    GROUP BY
        full_chain, chain_length
)

SELECT
    chain_length AS length,
    array_to_string(full_chain, '->') AS chain
FROM
    unique_chains
ORDER BY
    chain_length DESC, min_id;
```

**Explanation:**

- This query identifies circular gift exchange chains where each person gives a gift to the next person in the chain.
- The solution uses a recursive Common Table Expression (CTE) approach:
  - The `gift_chains` CTE builds possible chains by starting with each gift and recursively extending them.
  - The `completed_chains` CTE identifies chains where the last person gives a gift to the first person, completing the circle.
  - The `unique_chains` CTE removes duplicate chains that are just rotations of each other.
- The ARRAY type stores the sequence of people in each chain.
- The EXISTS subquery in completed_chains verifies that the last person gives to the first person.
- Time complexity: O(n^k) where n is the number of gifts and k is the maximum chain length, due to the recursive exploration.
- Space complexity: O(c) where c is the number of possible chains.

- This solution correctly handles edge cases:
  - Chains with exactly 3 people (the minimum required)
  - Multiple chains with the same length
  - Chains that are rotations of each other (counted only once)
- For large datasets, indexes on giver_id and receiver_id would improve join performance.
- The NOT IN and NOT = ANY conditions prevent premature cycles in the chains.
- The length limit in the recursive case prevents infinite recursion.
- The unnest and array_to_string functions convert arrays to a format suitable for display and comparison.
- The solution follows the requirements to order by length descending and then by minimum ID.
- The array_agg with ORDER BY creates a canonical representation of each chain for deduplication.
- The recursive CTE pattern is well-suited for path-finding problems like this circular chain detection.

### 3451. Find Invalid IP Addresses

**Description:**

**Solution:**

**Explanation:**

- Need to add explanation