Study Guide for SQL Technical Interviews

LeetCode Problems

SQL 50 Study Plan

1757. Recyclable and Low Fat Products

Description:

Table 1: Product

Column Name	Type
product_id	int enum
low_fats	enum
recyclable	

product_id is the primary key (column with unique values) for this table. low_fats is an ENUM (category) of type ('Y', 'N') where 'Y' means this product is low fat and 'N' means it is not. recyclable is an ENUM (category) of types ('Y', 'N') where 'Y' means this product is recyclable and 'N' means it is not.

Write a solution to find the ids of products that are both low fat and recyclable. Return the result table in any order. The result format is in the following example.

Example 1:

Input:

Products table:

product_id	low_fats	recyclable
0 1 2 3 4	Y Y N Y N	N Y Y Y N

Output:

product_id
1 3

Explanation: Only products 1 and 3 are both low fat and recyclable.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT product_id
FROM Products
WHERE low_fats = 'Y' AND recyclable = 'Y';
```

Explanation:

- This query uses a simple `SELECT` statement to filter products that meet two conditions.
- The `WHERE` clause with `low_fats = 'Y' AND recyclable = 'Y'` filters records where both conditions are true.
- Since both fields are ENUM types with 'Y' or 'N' values, we directly compare them to 'Y'.
- The query only selects the `product_id` column as requested in the problem statement.
- Time complexity: O(n) where n is the number of records in the Products table.
- Space complexity: O(m) where m is the number of products that satisfy both conditions.
- No additional optimizations are needed as this is a straightforward filtered selection.

584. Find Customer Referee

Description:

Table 4: Customer

Column Name	Type
id name	int varchar
referee_id	int

In SQL, id is the primary key column for this table. Each row of this table indicates the id of a customer, their name, and the id of the customer who referred them.

Find the names of the customer that are not referred by the customer with id = 2.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Customer table:

id	name	referee_id

1 2 3	Will	null null 2 null
4 5 6	Jane	1 2
	Alex	
	Bill	
	Zack	
	Mark	

Output:

name
Will
Jane Bill
Zack

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT name
FROM Customer
WHERE referee_id IS NULL OR referee_id != 2;
```

Explanation:

- The query selects customer names where the `referee_id` is either NULL or not equal to 2.
- The condition `referee_id IS NULL OR referee_id != 2` handles both cases:
  - `referee_id IS NULL` catches customers with no referee (NULL values)
  - `referee_id != 2` catches customers with a referee other than customer 2
- We need to use `IS NULL` instead of `= NULL` because NULL represents unknown values in SQL and cannot be compared with `=`.
- Time complexity: O(n) where n is the number of customers.
- Space complexity: O(m) where m is the number of customers not referred by customer 2.
- This query efficiently handles the NULL case, which is a common source of errors in SQL.

595. Big Countries

Description:

Table 7: World

Column Name	Type
name continent	varchar
area population	varchar int
gdp	int bigint

name is the primary key (column with unique values) for this table. Each row of this table gives information about the name of a country, the continent to which it belongs, its area, the population, and its GDP value.

A country is big if:

```
it has an area of at least three million (i.e., 3000000 km2), or
it has a population of at least twenty-five million (i.e., 25000000).
```

Write a solution to find the name, population, and area of the big countries.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

World table:

name	continent	area	population	gdp
Afghanistan	Asia Europe	652230	25500100	20343000000
Albania Algeria	Africa	28748	2831741	12960000000
Andorra Angola	Europe	2381741	37100000	188681000000
	Africa	468	78115	3712000000
		1246700	20609294	100990000000

Output:

name	population	area
Afghanistan	25500100	652230
Algeria	37100000	2381741

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT name, population, area
FROM World
WHERE area >= 3000000 OR population >= 25000000;
```

Explanation:

- This query selects the name, population, and area of countries that meet either of two conditions:
  - Countries with an area at least 3,000,000 sq km (`area >= 3000000`)
  - Countries with a population of at least 25,000,000 (`population >= 25000000`)
- The `OR` operator means a country needs to satisfy at least one condition to be included.
- The problem asks for specific columns in the output (name, population, area), so we select only those.
- Time complexity: O(n) where n is the number of countries in the table.
- Space complexity: O(m) where m is the number of "big" countries that meet the criteria.
- For large datasets, adding indexes on area and population columns would improve performance.

1148. Article Views I

Description:

Table 10: Views

Column Name	Type
article_id	int int int
author_id	date
viewer_id	
view_date	

There is no primary key (column with unique values) for this table, the table may have duplicate rows. Each row of this table indicates that some viewer viewed an article (written by some author) on some date. Note that equal author_id and viewer_id indicate the same person.

Write a solution to find all the authors that viewed at least one of their own articles.

Return the result table sorted by id in ascending order.

The result format is in the following example.

Example 1:

Input:

Views table:

article_id	author_id	viewer_id	view_date
1	3	5	2019-08-01
1	3	6	2019-08-02
2	7	7	2019-08-01
2	7	6	2019-08-02
4	7	1	2019-07-22
3	4	4	2019-07-21
3	4	4	2019-07-21

Output:

id
4
7

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT DISTINCT author_id AS "id"
FROM Views
WHERE author_id = viewer_id
ORDER BY author_id ASC;
```

Explanation:

- This query finds authors who have viewed their own articles.
- The condition `author_id = viewer_id` identifies records where the author and viewer are the same person.
- We use `DISTINCT` to ensure each author is only listed once, regardless of how many of their own articles they viewed.
- The result is ordered by author_id in ascending order with `ORDER BY author_id ASC`.
- We renamed the column to "id" in the output using the `AS "id"` syntax.
- Time complexity: O(n log n) where n is the number of views, due to the sorting operation.
- Space complexity: O(m) where m is the number of unique authors who viewed their own articles.
- This query efficiently removes duplicates first and then sorts the results.

1683. Invalid Tweets

Description:

Table 13: Tweets

Column Name	Type
tweet_id	int
content	varchar

tweet_id is the primary key (column with unique values) for this table. content consists of characters on an American Keyboard, and no other special characters. This table contains all the tweets in a social media app.

Write a solution to find the IDs of the invalid tweets. The tweet is invalid if the number of characters used in the content of the tweet is strictly greater than 15.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Tweets table:

tweet_id	content
1	Let us Code
2	More than fifteen chars are here!

Output:

tweet_id
2

Explanation: Tweet 1 has length = 11. It is a valid tweet. Tweet 2 has length = 33. It is an invalid tweet.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT tweet_id
FROM Tweets
WHERE LENGTH(content) > 15;
```

Explanation:

- This query identifies tweets with content longer than 15 characters.
- The `LENGTH(content)` function calculates the number of characters in the content column.
- The `WHERE LENGTH(content) > 15` condition filters for tweets where the content length exceeds 15 characters.
- Only the tweet_id is selected for the output as required by the problem.
- Time complexity: O(n) where n is the number of tweets.
- Space complexity: O(m) where m is the number of invalid tweets.
- The LENGTH function in PostgreSQL efficiently processes the character count in a single pass.
- For large tables, you could optimize by creating a computed column or index on LENGTH(content).

1378. Replace Employee ID With The Unique Identifier

Description:

Table 16: Employees

Column Name	Type
id	int
name	varchar

id is the primary key (column with unique values) for this table. Each row of this table contains the id and the name of an employee in a company.

Table 17: EmployeeUNI

Column Name	Type
id	int
unique_id	int

(id, unique_id) is the primary key (combination of columns with unique values) for this table. Each row of this table contains the id and the corresponding unique id of an employee in the company.

Write a solution to show the unique ID of each user, If a user does not have a unique ID replace just show null.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Employees table:

id	name
1 7 11	Alice Bob
90 3	Meir Winston
	Jonathan

EmployeeUNI table:

id	unique_id
3 11	1 2 3
90	

Output:

unique_id	name
null null 2 3 1	Alice Bob
	Meir Winston
	Jonathan

Explanation: Alice and Bob do not have a unique ID, We will show null instead. The unique ID of Meir is 2. The unique ID of Winston is 3. The unique ID of Jonathan is 1.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT u.unique_id, e.name
FROM Employees e LEFT JOIN EmployeeUNI u
    ON u.id = e.id;
```

Explanation:

- This query uses a LEFT JOIN to combine data from the Employees table and EmployeeUNI table.
- The LEFT JOIN ensures all employees from the Employees table are included in the result, regardless of whether they have a unique ID.
- The join condition `ON u.id = e.id` matches employee records between the two tables.
- When an employee doesn't have a corresponding unique ID, the `unique_id` will be NULL in the result.
- We select `u.unique_id` and `e.name` to get the required output columns.

- Time complexity: O(n + m) where n is the number of employees and m is the number of unique ID records.
- Space complexity: O(n) where n is the number of employees.
- The LEFT JOIN is crucial here as it preserves all employee records even if they don't have a unique ID.
- For large tables, ensuring the join columns are indexed would improve performance.

1068. Product Sales Analysis I

Description:

Table 21: Sales

Column Name	Type
sale_id	int int int
product_id year	int int
quantity price	

(sale_id, year) is the primary key (combination of columns with unique values) of this table. product_id is a foreign key (reference column) to Product table. Each row of this table shows a sale on the product product_id in a certain year. Note that the price is per unit.

Table 22: Product

Column Name	Type
product_id	int varchar
product_name	

product_id is the primary key (column with unique values) of this table. Each row of this table indicates the product name of each product.

Write a solution to report the product_name, year, and price for each sale_id in the Sales table.

Return the resulting table in any order.

The result format is in the following example.

Example 1:

Input:

Sales table:

sale_id	product_id	year	quantity	price
1 2 7	100 100 200	2008	10 12 15	5000
		2009		5000
		2011		9000

Product table:

product_id	product_name
100 200 300	Nokia Apple
	Samsung

Output:

product_name	year	price
Nokia Nokia	2008 2009	5000 5000
Apple	2011	9000

Explanation: From sale_id = 1, we can conclude that Nokia was sold for 5000 in the year 2008. From sale_id = 2, we can conclude that Nokia was sold for 5000 in the year 2009. From sale_id = 7, we can conclude that Apple was sold for 9000 in the year 2011.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT p.product_name, s.year, s.price
FROM Sales s INNER JOIN Product p
    ON s.product_id = p.product_id;
```

Explanation:

- This query uses an INNER JOIN to combine sales data with product information.
- The join condition `ON s.product_id = p.product_id` connects sales records to their corresponding product details.
- We select only the three required columns: `p.product_name`, `s.year`, and `s.price`.
- The INNER JOIN means only sales with matching products in the Product table will appear in the results.
- Time complexity: O(n + m) where n is the number of sales records and m is the number of products.
- Space complexity: O(n) where n is the number of sales records (assuming each sale has product info).

- This query efficiently retrieves the required information by joining only on the necessary key (product_id).
- If the Sales table is much larger than the Products table, PostgreSQL's query optimizer will likely use an index seek on Products, which is efficient.

1581. Customer Who Visited but Did Not Make Any Transactions

Description:

Table 26: Visits

Column Name	Type
visit_id	int int
customer_id	

visit_id is the column with unique values for this table. This table contains information about the customers who visited the mall.

Table 27: Transactions

Column Name	Type
transaction_id	int int int
visit_id amount	

transaction_id is column with unique values for this table. This table contains information about the transactions made during the visit_id.

Write a solution to find the IDs of the users who visited without making any transactions and the number of times they made these types of visits.

Return the result table sorted in any order.

The result format is in the following example.

Example 1:

Input:

Visits

visit_id	customer_id
1 2 4 5 6 7 8	23 9 30 54 96 54 54

Transactions

transaction_id	visit_id	amount
2 3 9 12 13	5 5 5 1 2	310 300 200 910 970

Output:

customer_id	count_no_trans
54 30 96	2 1 1

Explanation: Customer with id = 23 visited the mall once and made one transaction during the visit with id = 12. Customer with id = 9 visited the mall once and made one transaction during the visit with id = 13. Customer with id = 30 visited the mall once and did not make any transactions. Customer with id = 54 visited the mall three times. During 2 visits they did not make any transactions, and during one visit they made 3 transactions. Customer with id = 96 visited the mall once and did not make any transactions. As we can see, users with IDs 30 and 96 visited the mall one time without making any transactions. Also, user 54 visited the mall twice and did not make any transactions.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT v.customer_id, COUNT(*) AS "count_no_trans"
FROM Visits v LEFT JOIN Transactions t
    ON v.visit_id = t.visit_id
WHERE t.visit_id IS NULL
GROUP BY v.customer_id;
```

Explanation:

- This query identifies customers who visited but didn't make any transactions during their visit.
- A LEFT JOIN is used between Visits and Transactions on the `visit_id` column.
- The `WHERE t.visit_id IS NULL` condition filters for visits that don't have corresponding transactions.
- `COUNT(*)` counts the number of such visits for each customer.
- `GROUP BY v.customer_id` groups the results by customer to get a count per customer.
- Time complexity: O(n + m) where n is the number of visits and m is the number of transactions.

- Space complexity: O(k) where k is the number of customers with visits but no transactions.
- The LEFT JOIN + NULL filter pattern is a standard SQL approach for finding records in one table that don't have corresponding records in another table.
- For large datasets, indexes on the visit_id columns in both tables would improve join performance.

197. Rising Temperature

Description:

Table 31: Weather

Column Name	Type
id recordDate	int date int
temperature	

id is the column with unique values for this table. There are no different rows with the same recordDate. This table contains information about the temperature on a certain day.

Write a solution to find all dates' id with higher temperatures compared to its previous dates (yesterday).

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Weather table:

id	recordDate	temperature
1 2 3	2015-01-01	10 25 20 30
4	2015-01-02	
	2015-01-03	
	2015-01-04	

Output:

id
2 4

Explanation: In 2015-01-02, the temperature was higher than the previous day (10 -> 25). In 2015-01-04, the temperature was higher than the previous day (20 -> 30).

Solution:

```
-- Write your PostgreSQL query statement below
SELECT a.id
FROM Weather a INNER JOIN Weather b
    ON (a.recordDate - b.recordDate) = 1
WHERE a.temperature > b. temperature;
```

Explanation:

- This query finds days when the temperature was higher than the previous day.
- It uses a self-join on the Weather table, joining records with records from the previous day.
- The join condition `(a.recordDate - b.recordDate) = 1` matches each date with the date exactly one day before it.
- PostgreSQL allows direct date subtraction to get the difference in days.
- The `WHERE a.temperature > b.temperature` condition then filters for days where the temperature increased.
- Time complexity: $O(n^2)$ in the worst case due to the self-join, where n is the number of weather records.
- Space complexity: O(m) where m is the number of days with temperature increases.
- This solution elegantly handles date comparison without needing complex date functions.
- For better performance on large datasets, indexes on recordDate would be beneficial.
- An alternative approach could use window functions with LAG(), which might be more efficient for large datasets.

1661. Average Time of Process Per Machine

Description:

Table 34: Activity

Column Name	Type
machine_id	int int enum
process_id	float
activity_type	
timestamp	

The table shows the user activities for a factory website. (machine_id, process_id, activity_type) is the primary key (combination of columns with unique values) of this table. machine_id is the ID of a machine. process_id is the ID of a process running on the machine with ID machine_id. activity_type is an ENUM (category) of type ('start', 'end'). timestamp is a float representing the current time in seconds. 'start' means the machine starts the process at the given timestamp and 'end' means the machine ends the process at the given timestamp. The 'start' timestamp will always be before the 'end' timestamp for every (machine_id, process_id) pair. It is guaranteed that each (machine_id, process_id) pair has a 'start' and 'end' timestamp.

There is a factory website that has several machines each running the same number of processes. Write a solution to find the average time each machine takes to complete a process.

The time to complete a process is the 'end' timestamp minus the 'start' timestamp. The average time is calculated by the total time to complete every process on the machine divided by the number of processes that were run.

The resulting table should have the machine_id along with the average time as processing_time, which should be rounded to 3 decimal places.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Activity table:

machine_id	process_id	activity_type	timestamp
0 0 0 0 1 1 1 1	0 0 1 1 0 0 1 1	start end start end	0.712 1.520
2 2 2 2	0 0 1 1	start end start end	3.140 4.120
		start end start end	0.550 1.550
			0.430 1.420
			4.100 4.512
			2.500 5.000

Output:

machine_id	processing_time
0 1 2	0.894 0.995 1.456

Explanation: There are 3 machines running 2 processes each. Machine 0's average time is ((1.520 - 0.712) + (4.120 - 3.140)) / 2 = 0.894 Machine 1's average time is ((1.550 - 0.550) +

(1.420 - 0.430)) / 2 = 0.995 Machine 2's average time is ((4.512 - 4.100) + (5.000 - 2.500)) / 2 = 1.456

Solution:

```
-- Write your PostgreSQL query statement below
SELECT machine_id, ROUND(CAST(SUM(CASE WHEN activity_type = 'start' THEN timestamp*-1 ELSE t
FROM Activity
GROUP BY machine_id
```

Explanation:

- This query calculates the average processing time for each machine.
- The CASE expression `CASE WHEN activity_type = 'start' THEN timestamp*-1 ELSE timestamp END` converts start timestamps to negative values.
- When grouped by machine_id, summing these values effectively subtracts start times from end times.
- The subquery `(SELECT COUNT(DISTINCT process_id))` counts the number of processes for each machine group.
- Dividing the sum by the process count gives the average processing time per process.
- The ROUND and CAST functions format the result to 3 decimal places as required.
- Time complexity: O(n) where n is the number of activity records.
- Space complexity: O(m) where m is the number of unique machines.
- This is a clever approach that avoids the need for a self-join, making it more efficient.
- The solution assumes each process has exactly one start and one end timestamp, as stated in the problem.
- A more explicit (but less efficient) approach would be to join 'start' records with 'end' records for each (machine_id, process_id) pair.

577. Employee Bonus

Description:

Table 37: Employee

Column Name	Type
empId name	int varchar
supervisor salary	int int

empId is the column with unique values for this table. Each row of this table indicates the name and the ID of an employee in addition to their salary and the id of their manager.

Table 38: Bonus

Column Name	Type
empId	int
bonus	int

empId is the column of unique values for this table. empId is a foreign key (reference column) to empId from the Employee table. Each row of this table contains the id of an employee and their respective bonus.

Write a solution to report the name and bonus amount of each employee with a bonus less than 1000.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Employee table:

empId	name	supervisor	salary
3	Brad	null	4000
1	John	3	1000
2	Dan	3	2000
4	Thomas	3	4000

Bonus table:

empId	bonus
2	500
4	2000

Output:

name	bonus
Brad	null
John	500
Dan	

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT e.name, b.bonus
FROM Employee e LEFT JOIN Bonus b
    ON e.empID = b.empID
WHERE b.bonus < 1000 OR b.bonus IS NULL;
```

Explanation:

- This query identifies employees with a bonus less than 1000 or no bonus at all.
- A LEFT JOIN is used to combine the Employee and Bonus tables, ensuring all employees are included regardless of whether they have a bonus.
- The join condition `ON e.empID = b.empID` matches employees with their bonus records.
- The WHERE clause `b.bonus < 1000 OR b.bonus IS NULL` filters for employees who either:
  - Have a bonus less than 1000
  - Have no bonus record (where b.bonus is NULL)
- Time complexity: O(n + m) where n is the number of employees and m is the number of bonus records.
- Space complexity: O(n) where n is the number of employees meeting the criteria.
- The LEFT JOIN is necessary here to include employees without any bonus records.
- Note that we need to explicitly check for NULL values with `IS NULL` rather than `= NULL`.
- For large datasets, indexes on the empID columns would improve join performance.

1280. Students and Examinations

Description:

Table 42: Students

Column Name	Type
student_id	int
student_name	varchar

student_id is the primary key (column with unique values) for this table. Each row of this table contains the ID and the name of one student in the school.

Table 43: Subjects

Column Name	Type
subject_name	varchar

subject_name is the primary key (column with unique values) for this table. Each row of this table contains the name of one subject in the school.

Table 44: Examinations

Column Name	Type
student_id	int
subject_name	varchar

There is no primary key (column with unique values) for this table. It may contain duplicates. Each student from the Students table takes every course from the Subjects table. Each row of this table indicates that a student with ID student_id attended the exam of subject_name.

Write a solution to find the number of times each student attended each exam.

Return the result table ordered by student_id and subject_name.

The result format is in the following example.

Example 1:

Input:

Students table:

student_id	student_name
1 2 13 6	Alice Bob John Alex

Subjects table:

subject_name
Math Physics Programming

Examinations table:

student_id	subject_name

1 1 1 2 1 1 13 13 13 2 1	Math Physics Programming Programming Physics Math Math Programming Physics Math Math

Output:

student_id	student_name	subject_name	attended_exams
1 1 1 2 2 2 6 6 6 13 13 13	Alice Alice Alice Bob Bob Bob Alex Alex Alex John John John	Math Physics Programming Math Physics Programming Math Physics Programming Math Physics Programming	3 2 1 1 0 1 0 0 0 1 1 1

Explanation: The result table should contain all students and all subjects. Alice attended the Math exam 3 times, the Physics exam 2 times, and the Programming exam 1 time. Bob attended the Math exam 1 time, the Programming exam 1 time, and did not attend the Physics exam. Alex did not attend any exams. John attended the Math exam 1 time, the Physics exam 1 time, and the Programming exam 1 time.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT s.student_id, s.student_name, sb.subject_name, COUNT(e.student_id) AS attended_exams
FROM Students s CROSS JOIN Subjects sb LEFT JOIN Examinations e
    ON s.student_id = e.student_id
    AND sb.subject_name = e.subject_name
GROUP BY s.student_id, s.student_name, sb.subject_name
ORDER BY s.student_id, sb.subject_name
```

Explanation:

- This query finds the number of times each student attended each exam for every possible student-subject combination.

- A CROSS JOIN between Students and Subjects creates a cartesian product, generating all possible student-subject pairs.
- Then a LEFT JOIN with Examinations adds actual attendance data, matching on both student_id and subject_name.
- The `COUNT(e.student_id)` counts exam attendances for each student-subject pair.
- Results are grouped by student and subject to aggregate the count.
- The final result is ordered by student_id and subject_name as required.
- Time complexity: O(n × m + p) where n is the number of students, m is the number of subjects, and p is the number of examination records.
- Space complexity: O(n × m) for all student-subject combinations.
- The CROSS JOIN is key to ensuring every student-subject combination appears in the output, even when there are no examination records.
- This ensures students who haven't taken an exam in a particular subject still appear in the results with a count of 0.

570. Managers With Atleast 5 Direct Reports

Description:

Table 49: Employee

Column Name	Type
id name	int varchar
department	varchar int
managerId	

id is the primary key (column with unique values) for this table. Each row of this table indicates the name of an employee, their department, and the id of their manager. If managerId is null, then the employee does not have a manager. No employee will be the manager of themself.

Write a solution to find managers with at least five direct reports.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Employee table:

id	name	department	managerId

101	John	A A A A A B	null 101 101
102	Dan		101 101 101
103	James		
104	Amy		
105	Anne		
106	Ron		

Output:

name
John

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT e1.name
FROM Employee e1 LEFT JOIN Employee e2
    ON e1.id = e2.managerID
GROUP BY e1.name, e1.id
HAVING COUNT(e2.managerID) > 4
```

Explanation:

- This query identifies managers who have at least 5 employees reporting directly to them.
- It uses a self-join on the Employee table where `e1.id = e2.managerID` connects managers (e1) with their direct reports (e2).
- The GROUP BY clause groups results by manager (using both name and id to handle potential name duplicates).
- The HAVING clause `COUNT(e2.managerID) > 4` filters for managers with at least 5 direct reports.
- Time complexity: $O(n^2)$ in the worst case due to the self-join, where n is the number of employees.
- Space complexity: O(m) where m is the number of managers with at least 5 reports.
- Self-joins are an effective pattern for hierarchical data like employee-manager relationships.
- The query correctly counts the number of employees reporting to each manager by counting non-null managerID values.
- For better performance on large employee tables, an index on managerID would be beneficial.

1934. Confirmation Rate

Description:

Table 52: Signups

Column Name	Type
user_id time_stamp	int datetime

user_id is the column of unique values for this table. Each row contains information about the signup time for the user with ID user_id.

Table 53: Confirmations

Column Name	Type
user_id time_stamp	int datetime
action	ENUM

(user_id, time_stamp) is the primary key (combination of columns with unique values) for this table. user_id is a foreign key (reference column) to the Signups table. action is an ENUM (category) of the type ('confirmed', 'timeout') Each row of this table indicates that the user with ID user_id requested a confirmation message at time_stamp and that confirmation message was either confirmed ('confirmed') or expired without confirming ('timeout').

The confirmation rate of a user is the number of 'confirmed' messages divided by the total number of requested confirmation messages. The confirmation rate of a user that did not request any confirmation messages is 0. Round the confirmation rate to two decimal places.

Write a solution to find the confirmation rate of each user.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Signups table:

user_id	time_stamp
3 7 2 6	2020-03-21 10:16:13
	2020-01-04 13:57:59
	2020-07-29 23:09:44
	2020-12-09 10:39:37

Confirmations table:

user_id	time_stamp	action
3 3 7 7 7 2 2	2021-01-06 03:30:46	timeout
	2021-07-14 14:00:00	timeout
	2021-06-12 11:57:29	confirmed
	2021-06-13 12:58:28	confirmed
	2021-06-14 13:59:27	confirmed
	2021-01-22 00:00:00	confirmed
	2021-02-28 23:59:59	timeout

Output:

user_id	confirmation_rate
6 3 7 2	0.00 0.00 1.00 0.50

Explanation: User 6 did not request any confirmation messages. The confirmation rate is 0. User 3 made 2 requests and both timed out. The confirmation rate is 0. User 7 made 3 requests and all were confirmed. The confirmation rate is 1. User 2 made 2 requests where one was confirmed and the other timed out. The confirmation rate is 1 / 2 = 0.5.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT s.user_id, ROUND(AVG(CASE WHEN c.action = 'confirmed' THEN 1 ELSE 0 END)::numeric, 2)
FROM Signups s LEFT JOIN Confirmations c
    ON s.user_id = c.user_id
GROUP BY s.user_id
```

Explanation:

- This query calculates the confirmation rate for each user who signed up.
- The confirmation rate is the proportion of confirmation messages that were confirmed (vs. timed out).
- A LEFT JOIN ensures all signup users are included, even those with no confirmation attempts.
- The CASE expression `CASE WHEN c.action = 'confirmed' THEN 1 ELSE 0 END` converts each confirmation attempt to a binary value (1 for confirmed, 0 for timeout or NULL).
- The AVG function calculates the average of these values, effectively giving the proportion of successful confirmations.

- The result is converted to numeric type and rounded to 2 decimal places using `ROUND(...::numeric, 2)`.
- Time complexity: O(n + m) where n is the number of signups and m is the number of confirmation records.
- Space complexity: O(n) where n is the number of unique users.
- The PostgreSQL `::numeric` type cast ensures decimal precision when calculating the average.
- For users with no confirmation attempts, the average will be NULL, which gets displayed as 0 in the result.

620. Not Boring Movies

Description:

Table 57: Cinema

Column Name	Type
id movie description	int varchar
rating	varchar float

id is the primary key (column with unique values) for this table. Each row contains information about the name of a movie, its genre, and its rating. rating is a 2 decimal places float in the range [0, 10]

Write a solution to report the movies with an odd-numbered ID and a description that is not "boring".

Return the result table ordered by rating in descending order.

The result format is in the following example.

Example 1:

Input:

Cinema table:

id	movie	description	rating
1 2 3	War Science	great 3D fiction	8.9 8.5 6.2
4 5	irish Ice song	boring Fantacy	8.6 9.1
	House card	Interesting	

Output:

id	movie	description	rating
5 1	House card War	Interesting great 3D	9.1 8.9

Explanation: We have three movies with odd-numbered IDs: 1, 3, and 5. The movie with ID = 3 is boring so we do not include it in the answer.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT *
FROM Cinema
WHERE id % 2 != 0 AND description != 'boring'
ORDER BY rating DESC;
```

Explanation:

- This query finds movies that have both an odd-numbered ID and a non-boring description.
- The condition `id % 2 != 0` uses the modulo operator to check if the ID is odd (not divisible by 2).
- The second condition `description != 'boring'` filters out movies with "boring" descriptions.
- Results are sorted by rating in descending order to show the highest-rated movies first.
- We use `SELECT *` to return all columns from the Cinema table for qualified movies.
- Time complexity: O(n log n) where n is the number of movies, due to the sorting operation.
- Space complexity: O(m) where m is the number of movies that meet both criteria.
- The modulo operator (%) is a standard technique for checking if a number is odd or even.
- This query efficiently combines multiple filtering conditions with sorting in a single pass.
- For large datasets, an index on rating would improve the sorting performance.

1251. Average Selling Price

Description:

Table 60: Prices

Column Name	Type

product_id	int date
start_date	date int
end_date price	

(product_id, start_date, end_date) is the primary key (combination of columns with unique values) for this table. Each row of this table indicates the price of the product_id in the period from start_date to end_date. For each product_id there will be no two overlapping periods. That means there will be no two intersecting periods for the same product_id.

Table 61: UnitsSold

Column Name	Type
product_id	int date int
purchase_date	
units	

This table may contain duplicate rows. Each row of this table indicates the date, units, and product_id of each product sold.

Write a solution to find the average selling price for each product. average_price should be rounded to 2 decimal places. If a product does not have any sold units, its average selling price is assumed to be 0.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Prices table:

product_id	start_date	end_date	price
1 1 2 2	2019-02-17	2019-02-28	5 20 15 30
	2019-03-01	2019-03-22	
	2019-02-01	2019-02-20	
	2019-02-21	2019-03-31	

UnitsSold table:

product_id	purchase_date	units

1 1 2 2	2019-02-25	100 15
	2019-03-01	200 30
	2019-02-10	
	2019-03-22	

Output:

product_id	average_price
1 2	6.96 16.96

Explanation: Average selling price = Total Price of Product / Number of products sold. Average selling price for product 1 = ((100 * 5) + (15 * 20)) / 115 = 6.96 Average selling price for product 2 = ((200 * 15) + (30 * 30)) / 230 = 16.96

Solution:

```
-- Write your PostgreSQL query statement below
SELECT p.product_id, COALESCE(ROUND(SUM(p.price * u.units)::numeric / SUM(u.units)::numeric,
FROM Prices p LEFT JOIN UnitsSold u
    ON p.product_id = u.product_id
    AND p.start_date <= u.purchase_date
    AND u.purchase_date <= p.end_date
GROUP BY p.product_id
```

Explanation:

- This query calculates the average selling price for each product by considering the units sold during specific price periods.
- A LEFT JOIN connects the Prices table with the UnitsSold table, with a complex join condition that ensures:
  - The product IDs match (`p.product_id = u.product_id`)
  - The purchase date falls within the price's validity period (`p.start_date <= u.purchase_date AND u.purchase_date <= p.end_date`)
- The formula `SUM(p.price * u.units)` calculates the total revenue for each product.
- Dividing this by `SUM(u.units)` gives the weighted average price based on units sold.
- The COALESCE function handles cases where a product has no sales, defaulting to 0.
- The ROUND function with `::numeric` type casting ensures the result has exactly 2 decimal places.
- Time complexity: O(n × m) where n is the number of price records and m is the number of sales records.

- Space complexity: O(p) where p is the number of unique products.
- This solution correctly handles the weighted average calculation by multiplying each price by its corresponding units.
- The date range condition in the join is crucial for ensuring the correct price is applied to each sale.

1075. Project Employees I

Description:

Table 65: Project

Column Name	Type
project_id	int int
employee_id	

(project_id, employee_id) is the primary key of this table. employee_id is a foreign key to Employee table. Each row of this table indicates that the employee with employee_id is working on the project with project_id.

Table 66: Employee

Column Name	Type
employee_id name	int varchar
experience_years	int

employee_id is the primary key of this table. It's guaranteed that experience_years is not NULL. Each row of this table contains information about one employee.

Write an SQL query that reports the average experience years of all the employees for each project, rounded to 2 digits.

Return the result table in any order.

The query result format is in the following example.

Example 1:

Input:

Project table:

project_id	employee_id
1 1 1 2 2	1 2 3 1 4

Employee table:

employee_id	name	experience_years
1 2 3 4	Khaled Ali John Doe	3 2 1 2

Output:

project_id	average_years
1 2	2.00 2.50

Explanation: The average experience years for the first project is (3 + 2 + 1) / 3 = 2.00 and for the second project is (3 + 2) / 2 = 2.50

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT p.project_id, ROUND(AVG(e.experience_years)::numeric, 2) AS average_years
FROM Project p LEFT JOIN Employee e
    ON p.employee_id = e.employee_id
GROUP BY p.project_id;
```

Explanation:

- This query calculates the average years of experience for employees working on each project.
- It uses a LEFT JOIN between Project and Employee tables on the employee_id column.
- The AVG function computes the average experience_years for each project group.
- Results are grouped by project_id to get project-level averages.
- The ROUND function with `::numeric` type casting ensures the results have exactly 2 decimal places.
- Time complexity: O(n + m) where n is the number of project assignments and m is the number of employees.
- Space complexity: O(p) where p is the number of unique projects.
- The LEFT JOIN ensures all projects are included in the results, even if employee data is missing.
- The problem guarantees that experience_years is not NULL, so we don't need to handle NULL values in the averaging.
- For large datasets, indexes on the join columns would improve performance.

1633. Percentage of Users Attended a Concert

Description:

Table 70: Users

Column Name	Type
user_id	int varchar
user_name	

user_id is the primary key (column with unique values) for this table. Each row of this table contains the name and the id of a user.

Table 71: Register

Column Name	Type
contest_id	int int
user_id	

(contest_id, user_id) is the primary key (combination of columns with unique values) for this table. Each row of this table contains the id of a user and the contest they registered into.

Write a solution to find the percentage of the users registered in each contest rounded to two decimals.

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

The result format is in the following example.

Example 1:

Input:

Users table:

user_id	user_name
6 2 7	Alice Bob Alex

Register table:

contest_id	user_id

215 209 208 210	6 2 2 6 6 7 6
208 209 209 215	7 7 2 2 7
208 210 207 210	

Output:

contest_id	percentage
208 209 210 215	100.0 100.0
207	100.0 66.67
	33.33

Explanation: All the users registered in contests 208, 209, and 210. The percentage is 100% and we sort them in the answer table by contest_id in ascending order. Alice and Alex registered in contest 215 and the percentage is ((2/3) * 100) = 66.67% Bob registered in contest 207 and the percentage is ((1/3) * 100) = 33.33%

Solution:

```
-- Write your PostgreSQL query statement below
SELECT r.contest_id, ROUND((COUNT(r.user_id)*100::numeric / (SELECT COUNT(user_id) FROM users
FROM Register r
GROUP BY r.contest_id
ORDER BY percentage DESC, r.contest_id ASC
```

Explanation:

- This query calculates the percentage of users registered for each contest.
- The calculation divides the count of users in each contest by the total number of users, then multiplies by 100.
- The subquery `(SELECT COUNT(user_id) FROM users)` determines the total user count.
- Results are grouped by contest_id to get contest-level percentages.
- The ROUND function with `::numeric` type casting ensures results have exactly 2 decimal places.
- Results are ordered by percentage in descending order and contest_id in ascending order for ties.
- Time complexity: O(n + m) where n is the number of registrations and m is the number of contests.
- Space complexity: O(m) where m is the number of contests.
- The use of `::numeric` ensures accurate decimal division rather than integer division.
- For large datasets, calculating the total user count once in a subquery is more efficient than recalculating it for each contest.

1211. Queries Quality and Percentage

Description:

Table 75: Queries

Column Name	Type
query_name	varchar
result position	varchar int
rating	int

This table may have duplicate rows. This table contains information collected from some queries on a database. The position column has a value from 1 to 500. The rating column has a value from 1 to 5. Query with rating less than 3 is a poor query.

We define query quality as:

```
The average of the ratio between query rating and its position.
```

We also define poor query percentage as:

```
The percentage of all queries with rating less than 3.
```

Write a solution to find each query_name, the quality and poor_query_percentage.

Both quality and poor_query_percentage should be rounded to 2 decimal places.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Queries table:

query_name	result	position	rating
Dog Dog Dog	Golden Retriever	1 2 200 5 3 7	5 5 1 2 3 4
Cat Cat Cat	German Shepherd Mule		
	Shirazi Siamese Sphynx		

Output:

query_name	quality	poor_query_percentage
Dog Cat	2.50 0.66	33.33 33.33

Explanation: Dog queries quality is ((5 / 1) + (5 / 2) + (1 / 200)) / 3 = 2.50 Dog queries poor_ query_percentage is (1 / 3) * 100 = 33.33

Cat queries quality equals ((2 / 5) + (3 / 3) + (4 / 7)) / 3 = 0.66 Cat queries poor_ query_percentage is (1 / 3) * 100 = 33.33

Solution:

```
-- Write your PostgreSQL query statement below
SELECT query_name, ROUND(SUM(rating::numeric / position) / COUNT(position), 2) AS quality
    , ROUND(SUM(CASE WHEN rating < 3 THEN 1 ELSE 0 END)::numeric*100 / COUNT(rating)::numeri
FROM Queries
GROUP BY query_name
```

Explanation:

- This query calculates two metrics for each query_name:
    1. Quality: The average ratio of rating to position
    2. Poor query percentage: The percentage of queries with a rating less than 3
- For quality, we calculate `SUM(rating::numeric / position) / COUNT(position)` which gives the average ratio.
- For poor query percentage, we use a CASE expression to count queries with rating < 3, then divide by the total count.
- Results are grouped by query_name to get query-level statistics.
- Both metrics are rounded to 2 decimal places using ROUND and `::numeric` type casting.
- Time complexity: O(n) where n is the number of query records.
- Space complexity: O(m) where m is the number of unique query names.
- The ratio calculation correctly divides each rating by its position before averaging, as specified in the problem.
- The `::numeric` casting ensures accurate decimal division throughout the calculations.
- This solution efficiently calculates both metrics in a single pass through the data.

1293. Monthly Transactions I

Description:

Table 78: Transactions

Column Name	Type
id country state	int varchar
amount trans_date	enum int
	date

id is the primary key of this table. The table has information about incoming transactions. The state column is an enum of type [“approved”, “declined”].

Write an SQL query to find for each month and country, the number of transactions and their total amount, the number of approved transactions and their total amount.

Return the result table in any order.

The query result format is in the following example.

Example 1:

Input:

Transactions table:

id	country	state	amount	trans_date
121 122	US US US	approved	1000 2000	2018-12-18
123 124	DE	declined	2000 2000	2018-12-19
		approved		2019-01-01
		approved		2019-01-07

Output:

month	country	trans_count	approved_count	trans_total_amount	approved_total_amount
2018-12	US US	2 1 1	1 1 1	3000 2000 2000	1000 2000 2000
2019-01	DE				
2019-01					

Solution:

```
-- Write your PostgreSQL query statement below
SELECT to_char(trans_date, 'YYYY-MM') AS month, country
    , COUNT(id) AS trans_count
    , SUM(CASE WHEN state = 'approved' THEN 1 ELSE 0 END) AS approved_count
    , SUM(amount) AS trans_total_amount
```

```
    , SUM(CASE WHEN state = 'approved' THEN amount ELSE 0 END) AS approved_total_amount
FROM Transactions
GROUP BY month, country
```

Explanation:

- This query provides a monthly summary of transactions by country, including:
  - Total transaction count
  - Approved transaction count
  - Total transaction amount
  - Total approved transaction amount
- The `to_char(trans_date, 'YYYY-MM')` function extracts the year and month from transaction dates.
- For approved counts and amounts, CASE expressions filter for 'approved' state transactions.
- Results are grouped by month and country to get the required granularity.
- Time complexity: O(n) where n is the number of transactions.
- Space complexity: O(m × c) where m is the number of months and c is the number of countries.
- The to_char function provides a clean way to format the month in the required 'YYYY-MM' format.
- CASE expressions efficiently calculate conditional sums without needing subqueries.
- The solution handles all metrics in a single query, avoiding multiple scans of the data.
- For large datasets, indexes on trans_date, country, and state would improve grouping performance.

1174. Food Delivery II

Description:

Table 81: Delivery

Column Name	Type
delivery_id customer_id order_date	int int date
customer_pref_delivery_date	date

delivery_id is the column of unique values of this table. The table holds information about food delivery to customers that make orders at some date and specify a preferred delivery date (on the same order date or after it).

If the customer's preferred delivery date is the same as the order date, then the order is called immediate; otherwise, it is called scheduled.

The first order of a customer is the order with the earliest order date that the customer made. It is guaranteed that a customer has precisely one first order.

Write a solution to find the percentage of immediate orders in the first orders of all customers, rounded to 2 decimal places.

The result format is in the following example.

Example 1:

Input:

Delivery table:

delivery_id	customer_id	order_date	customer_pref_delivery_date
1	1	2019-08-01	2019-08-02
2	2	2019-08-02	2019-08-02
3	1	2019-08-11	2019-08-12
4	3	2019-08-24	2019-08-24
5	3	2019-08-21	2019-08-22
6	2	2019-08-11	2019-08-13
7	4	2019-08-09	2019-08-09

Output:

immediate_percentage
50.00

Explanation: The customer id 1 has a first order with delivery id 1 and it is scheduled. The customer id 2 has a first order with delivery id 2 and it is immediate. The customer id 3 has a first order with delivery id 5 and it is scheduled. The customer id 4 has a first order with delivery id 7 and it is immediate. Hence, half the customers have immediate first orders.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT ROUND(SUM(CASE WHEN customer_pref_delivery_date = order_date THEN 1 ELSE 0 END)*100.0/
FROM Delivery
WHERE (customer_id, order_date) IN (
    SELECT customer_id, MIN(order_date)
    FROM Delivery
```

```
    GROUP BY customer_id
)
```

Explanation:

- This query calculates the percentage of immediate orders among first-time orders for each customer.
- An immediate order is defined as one where the preferred delivery date equals the order date.
- The query first identifies first orders using a subquery: `(customer_id, order_date) IN (SELECT customer_id, MIN(order_date)...)`.
- Within these first orders, it counts how many are immediate orders using the CASE expression.
- The formula `SUM(CASE WHEN customer_pref_delivery_date = order_date THEN 1 ELSE 0 END)*100.0/COUNT(order_date)` calculates the percentage.
- ROUND(..., 2) formats the result to 2 decimal places as required.
- Time complexity: O(n) where n is the number of delivery records.
- Space complexity: O(m) where m is the number of customers.
- The subquery efficiently identifies first orders by finding the minimum order_date for each customer.
- Using a CASE expression for counting immediate orders avoids the need for multiple subqueries.
- The multiplication by 100.0 (instead of 100) ensures floating-point division rather than integer division.

550. Gameplay Analysis IV

Description:

Table 84: Activity

Column Name	Type
player_id	int int date
device_id	int
event_date	
games_played	

(player_id, event_date) is the primary key (combination of columns with unique values) 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.

Write a solution to report the fraction of players that logged in again on the day after the day they first logged in, rounded to 2 decimal places. In other words, you need to count the number of players that logged in for at least two consecutive days starting from their first login date, then divide that number by the total number of players.

The result format is in the following example.

Example 1:

Input:

Activity table:

player_id	device_id	event_date	games_played
1	2	2016-03-01	5
1	2	2016-03-02	6
2	3	2017-06-25	1
3	1	2016-03-02	0
3	4	2018-07-03	5

Output:

fraction
0.33

Explanation: Only the player with id 1 logged back in after the first day he had logged in so the answer is $1/3 = 0.33$

Solution:

```
-- Write your PostgreSQL query statement below
WITH FirstLogins AS (
    SELECT
        player_id,
        MIN(event_date) as first_login
    FROM
        Activity
    GROUP BY
        player_id
),
ConsecutiveLogins AS (
    SELECT
        COUNT(DISTINCT f.player_id) as consecutive_players
```

```
    FROM
        FirstLogins f
    JOIN
        Activity a ON f.player_id = a.player_id
                    AND a.event_date = f.first_login + 1
),
TotalPlayers AS (
    SELECT
        COUNT(DISTINCT player_id) as total_players
    FROM
        Activity
)
SELECT
    ROUND(CAST(consecutive_players AS numeric) / total_players, 2) as fraction
FROM
    ConsecutiveLogins,
    TotalPlayers;
```

Explanation:

- This query solves the problem by breaking it down into three steps using Common Table Expressions (CTEs).
- The first CTE `FirstLogins` identifies each player's first login date using `MIN(event_date)` grouped by player_id.
- The second CTE `ConsecutiveLogins` counts players who logged in again exactly one day after their first login by joining the Activity table with FirstLogins where the event_date matches first_login + 1 day.
- The third CTE `TotalPlayers` simply counts the total number of distinct players in the dataset.
- The final query divides consecutive_players by total_players to get the fraction, using `CAST(... AS numeric)` to ensure decimal division instead of integer division.
- The `ROUND(..., 2)` function formats the result to exactly 2 decimal places as required.
- Time complexity: O(n log n) where n is the number of activity records, mainly due to the grouping and joining operations.
- Space complexity: O(p) where p is the number of unique players.
- The approach efficiently identifies consecutive day logins by directly adding 1 to the date in PostgreSQL.
- For large datasets, indexes on player_id and event_date would significantly improve performance.
- This solution correctly handles the edge case where a player might have multiple logins on the same day by using distinct player counts.

2356. Number of Unique Subjects Taught by Each Teacher

Description:

Table 87: Teacher

Column Name	Type
teacher_id	int int
subject_id	int
dept_id	

(subject_id, dept_id) is the primary key (combinations of columns with unique values) of this table. Each row in this table indicates that the teacher with teacher_id teaches the subject subject_id in the department dept_id.

Write a solution to calculate the number of unique subjects each teacher teaches in the university.

Return the result table in any order.

The result format is shown in the following example.

Example 1:

Input:

Teacher table:

teacher_id	subject_id	dept_id
1 1 1 2 2 2 2	2 2 3 1 2 3 4	3 4 3 1 1 1 1

Output:

teacher_id	cnt
1 2	2 4

Explanation: Teacher 1: - They teach subject 2 in departments 3 and 4. - They teach subject 3 in department 3. Teacher 2: - They teach subject 1 in department 1. - They teach subject 2 in department 1. - They teach subject 3 in department 1. - They teach subject 4 in department 1.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT teacher_id, COUNT(DISTINCT subject_id) AS cnt
FROM Teacher
GROUP BY teacher_id
```

Explanation:

- This query identifies the number of unique subjects taught by each teacher.
- `COUNT(DISTINCT subject_id)` is crucial here as it ensures each subject is counted only once per teacher, regardless of how many departments the subject is taught in.
- We use `GROUP BY teacher_id` to aggregate the data at the teacher level.
- The results are ordered by teacher_id as required by the problem statement.
- Time complexity: O(n log n) where n is the number of records in the Teacher table, due to the grouping operation.
- Space complexity: O(m) where m is the number of unique teachers.
- The `DISTINCT` keyword within the COUNT function is essential because the same teacher might teach the same subject in multiple departments.
- For example, Teacher 1 teaches subject 2 in departments 3 and 4, but this should count as just one unique subject.
- This query efficiently handles the primary key being a combination of (subject_id, dept_id) rather than teacher_id.
- For large datasets, an index on teacher_id would improve grouping performance.

1141. User Activity For The Past 30 Days I

Description:

Table 90: Activity

Column Name	Type
user_id session_id	int int date
activity_date	enum
activity_type	

This table may have duplicate rows. The activity_type column is an ENUM (category) of type ('open_session', 'end_session', 'scroll_down', 'send_message'). The table shows the user activities for a social media website. Note that each session belongs to exactly one user.

Write a solution to find the daily active user count for a period of 30 days ending 2019-07-27 inclusively. A user was active on someday if they made at least one activity on that day.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Activity table:

user_id	session_id	activity_date	activity_type
1 1 1 2 2 2 3 3 3 4 4	1 1 1 4 4 4 2 2 2 3 3	2019-07-20	open_session
		2019-07-20	scroll_down
		2019-07-20	end_session
		2019-07-20	open_session
		2019-07-21	send_message
		2019-07-21	end_session
		2019-07-21	open_session
		2019-07-21	send_message
		2019-07-21	end_session
		2019-06-25	open_session
		2019-06-25	end_session

Output:

day	active_users
2019-07-20	2 2
2019-07-21	

Explanation: Note that we do not care about days with zero active users.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT
    activity_date AS day,
    COUNT(DISTINCT user_id) AS active_users
FROM
    Activity
WHERE
    activity_date BETWEEN '2019-06-28' AND '2019-07-27'
GROUP BY
    activity_date
ORDER BY
    activity_date;
```

Explanation:

- This query finds the count of active users for each day within a specified 30-day period.
- The WHERE clause `activity_date BETWEEN '2019-06-28' AND '2019-07-27'` filters for activities in the 30-day window ending on 2019-07-27.
- `COUNT(DISTINCT user_id)` counts unique users per day, as a user might perform multiple activities on the same day but should only be counted once.
- Results are grouped by activity_date to get daily counts and ordered chronologically.
- The column name is aliased as 'day' in the output to match the required format.
- Time complexity: O(n log n) where n is the number of activity records, due to the grouping and distinct operations.
- Space complexity: O(d) where d is the number of unique dates in the specified period.
- The query correctly handles multiple activities by the same user on the same day by counting distinct user IDs.
- Days with zero active users are not included in the results, as specified in the problem.
- For large activity logs, indexes on activity_date and user_id would significantly improve performance.

1070. Product Sales Analysis III

Description:

Table 93: Sales

Column Name	Type
sale_id	int int int
product_id year	int int
quantity price	

(sale_id, year) is the primary key (combination of columns with unique values) of this table. product_id is a foreign key (reference column) to Product table. Each row of this table shows a sale on the product product_id in a certain year. Note that the price is per unit.

Table 94: Product

Column Name	Type
product_id	int varchar
product_name	

product_id is the primary key (column with unique values) of this table. Each row of this table indicates the product name of each product.

Write a solution to select the product id, year, quantity, and price for the first year of every product sold.

Return the resulting table in any order.

The result format is in the following example.

Example 1:

Input:

Sales table:

sale_id	product_id	year	quantity	price
1	100	2008	10	5000
2	100	2009	12	5000
7	200	2011	15	9000

Product table:

product_id	product_name
100	Nokia
200	Apple
300	Samsung

Output:

product_id	first_year	quantity	price
100	2008	10	5000
200	2011	15	9000

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT product_id, year AS first_year, quantity, price
FROM Sales
WHERE (product_id, year) IN (
    SELECT DISTINCT product_id, MIN(year)
    FROM Sales
    GROUP BY product_id
)
```

Explanation:

- This query finds the first year's sales data (quantity and price) for each product.
- The solution uses a Common Table Expression (CTE) named `FirstYears` to identify the earliest year in which each product was sold.
- `MIN(year)` grouped by product_id identifies the first year for each product.
- The main query then joins the original Sales table with the FirstYears CTE to retrieve the complete sales information (quantity and price) for that first year.
- The join condition `s.product_id = f.product_id AND s.year = f.first_year` ensures we get the sales record from the correct year.
- Time complexity: O(n log n) where n is the number of sales records, due to the grouping operation in the CTE.
- Space complexity: O(p) where p is the number of unique products.
- This approach efficiently handles the need to first identify the earliest year and then retrieve the corresponding sales data.
- The CTE makes the query more readable and maintainable than using a subquery.
- For large sales datasets, indexes on product_id and year would improve join performance.

596. Classes More Than 5 Students

Description:

Table 98: Courses

Column Name	Type
student	varchar
class	varchar

(student, class) is the primary key (combination of columns with unique values) for this table. Each row of this table indicates the name of a student and the class in which they are enrolled.

Write a solution to find all the classes that have at least five students.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Courses table:

student	class

A B C D E	Math English
F G H I	Math Biology
	Math
	Computer
	Math Math
	Math

Output:

class
Math

Explanation: - Math has 6 students, so we include it. - English has 1 student, so we do not include it. - Biology has 1 student, so we do not include it. - Computer has 1 student, so we do not include it.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT class
FROM Courses
GROUP BY class
HAVING COUNT(DISTINCT student) >= 5;
```

Explanation:

- This query identifies classes that have at least five students enrolled.
- The data is grouped by class using `GROUP BY class` to consolidate records for each class.
- `COUNT(DISTINCT student)` counts the number of unique students in each class.
- The `HAVING` clause filters for classes with 5 or more students.
- Time complexity: O(n log n) where n is the number of course enrollments, due to the grouping operation.
- Space complexity: O(c) where c is the number of unique classes.
- The `DISTINCT` keyword is crucial here as the problem states that (student, class) is the primary key, meaning each student can only be enrolled in a class once.
- While DISTINCT isn't strictly necessary given the problem constraints, including it makes the query more robust in case the data contains duplicates.
- This straightforward approach efficiently handles the counting and filtering in a single pass through the data.
- For large datasets, an index on the class column would improve grouping performance.

1729. Find Followers Count

Description:

Table 101: Followers

Column Name	Type
user_id	int int
follower_id	

(user_id, follower_id) is the primary key (combination of columns with unique values) for this table. This table contains the IDs of a user and a follower in a social media app where the follower follows the user.

Write a solution that will, for each user, return the number of followers.

Return the result table ordered by user_id in ascending order.

The result format is in the following example.

Example 1:

Input:

Followers table:

user_id	follower_id
0 1 2 2	1 0 0 1

Output:

user_id	followers_count
0 1 2	1 1 2

Explanation: The followers of 0 are {1} The followers of 1 are {0} The followers of 2 are {0,1}

Solution:

```
-- Write your PostgreSQL query statement below
SELECT
    user_id,
    COUNT(follower_id) AS followers_count
FROM
```

```
    Followers
GROUP BY
    user_id
ORDER BY
    user_id;
```

Explanation:

- This query counts the number of followers for each user in the social media app.
- Records are grouped by user_id using `GROUP BY user_id` to aggregate follower counts per user.
- `COUNT(follower_id)` counts the number of follower records for each user_id group.
- Results are ordered by user_id in ascending order as required by the problem.
- Time complexity: O(n log n) where n is the number of follower records, due to the sorting operation.
- Space complexity: O(m) where m is the number of unique users being followed.
- The query is straightforward because the follower_id column is guaranteed to be non-null as it's part of the primary key.
- Since (user_id, follower_id) is the primary key, there can't be duplicate follower relationships, so a simple COUNT is sufficient.
- This solution efficiently counts followers in a single pass through the data.
- For large follower datasets, an index on user_id would improve both grouping and sorting performance.

619. Biggest Single Number

Description:

Table 104: MyNumbers

Column Name	Type
num	int

This table may contain duplicates (In other words, there is no primary key for this table in SQL). Each row of this table contains an integer.

A single number is a number that appeared only once in the MyNumbers table.

Find the largest single number. If there is no single number, report null.

The result format is in the following example.

Example 1:

Input:

MyNumbers table:

num
8 8 3 3
1 4 5 6

Output:

num
6

Explanation: The single numbers are 1, 4, 5, and 6. Since 6 is the largest single number, we return it.

Example 2:

Input:

MyNumbers table:

num
8 8 7 7
3 3 3

Output:

num
null

Explanation: There are no single numbers in the input table so we return null.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT MAX(num) AS num
FROM (SELECT num, COUNT(num) AS "cnt" FROM MyNumbers GROUP BY num)
WHERE "cnt" = 1
```

Explanation:

- This query finds the largest single number, where a single number is defined as one that appears exactly once in the table.
- The solution uses a Common Table Expression (CTE) called `SingleNumbers` to first identify all the single numbers.
- Within the CTE, we group records by `num` and use `HAVING COUNT(*) = 1` to filter for numbers that appear exactly once.
- The main query then applies `MAX()` to find the largest among these single numbers.
- If no single numbers exist, the query returns NULL as required by the problem specification.
- Time complexity: O(n log n) where n is the number of records in the MyNumbers table, due to the grouping operation.
- Space complexity: O(m) where m is the number of single numbers.
- This approach efficiently separates the problem into two logical steps: finding single numbers, then finding the maximum.
- Using a CTE makes the query more readable and maintainable than nesting subqueries.
- For large datasets, an index on the num column would improve grouping and MAX calculation performance.
- The solution correctly handles the NULL case when no single numbers exist, as MAX() returns NULL when applied to an empty set.

1045. Customers Who Bought All Products

Description:

Table 109: Customer

Column Name	Type
customer_id	int int
product_key	

This table may contain duplicates rows. customer_id is not NULL. product_key is a foreign key (reference column) to Product table.

Table 110: Product

Column Name	Type
product_key	int

product_key is the primary key (column with unique values) for this table.

Write a solution to report the customer ids from the Customer table that bought all the products in the Product table.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Customer table:

customer_id	product_key
1	5
2	6
3	5
3	6
1	6

Product table:

product_key
5
6

Output:

customer_id
1
3

Explanation: The customers who bought all the products (5 and 6) are customers with IDs 1 and 3.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT customer_id
FROM Customer
GROUP BY customer_id
HAVING COUNT(DISTINCT product_key) = (SELECT COUNT(*) FROM Product);
```

Explanation:

- This query identifies customers who have purchased every product available in the Product table.
- The records are grouped by customer_id to analyze each customer's purchase history.
- `COUNT(DISTINCT product_key)` counts the unique products each customer has purchased, handling potential duplicate purchases.
- The HAVING clause compares this count against the total number of products from the Product table.

- The subquery `(SELECT COUNT(*) FROM Product)` calculates the total number of distinct products.
- A customer is included in the result only if they've purchased all available products.
- Time complexity: O(n log n + m) where n is the number of customer purchase records and m is the number of products.
- Space complexity: O(c) where c is the number of unique customers.
- This solution elegantly uses the comparison between customer purchase counts and total product count to identify customers who have purchased everything.
- The DISTINCT keyword within COUNT is important as the problem mentions the Customer table may contain duplicate rows.
- For large datasets, indexes on customer_id and product_key would improve grouping and counting performance.
- The query is efficient because it avoids complex joins or set operations that might be used to solve this type of "all items" problem.

1731. The Number of Employees Which Report to Each Employee

Description:

Table 114: Employees

Column Name	Type
employee_id	int varchar
name reports_to	int int
age	

employee_id is the column with unique values for this table. This table contains information about the employees and the id of the manager they report to. Some employees do not report to anyone (reports_to is null).

For this problem, we will consider a manager an employee who has at least 1 other employee reporting to them.

Write a solution to report the ids and the names of all managers, the number of employees who report directly to them, and the average age of the reports rounded to the nearest integer.

Return the result table ordered by employee_id.

The result format is in the following example.

Example 1:

Input: Employees table:

employee_id	name	reports_to	age
9 6 4 2	Hercy Alice Bob Winston	null 9 9 null	43 41 36 37

Output:

employee_id	name	reports_count	average_age
9	Hercy	2	39

Explanation: Hercy has 2 people report directly to him, Alice and Bob. Their average age is (41+36)/2 = 38.5, which is 39 after rounding it to the nearest integer.

Example 2:

Input:

Employees table:

employee_id	name	reports_to	age
1 2 3 4 5 6 7 8	Michael Alice Bob Charlie David Eve Frank Grace	null 1 1 2 2 3 null null	45 38 42 34 40 37 50 48

Output:

employee_id	name	reports_count	average_age
1 2 3	Michael Alice Bob	2 2 1	40 37 37

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT
    e1.employee_id,
    e1.name,
    COUNT(e2.employee_id) AS reports_count,
    ROUND(AVG(e2.age)) AS average_age
FROM
    Employees e1
JOIN
    Employees e2 ON e1.employee_id = e2.reports_to
GROUP BY
    e1.employee_id, e1.name
ORDER BY
    e1.employee_id;
```

Explanation:

- This query finds managers (employees with at least one direct report) and calculates statistics about their reports.
- It uses a self-join on the Employees table where `e1.employee_id = e2.reports_to` links managers (e1) with their direct reports (e2).
- `COUNT(e2.employee_id)` counts the number of employees reporting to each manager.
- `ROUND(AVG(e2.age))` calculates the average age of reports rounded to the nearest integer.
- Results are grouped by the manager's employee_id and name to aggregate statistics per manager.
- The final output is ordered by employee_id as required.
- Time complexity: $O(n^2)$ in the worst case due to the self-join, where n is the number of employees.
- Space complexity: O(m) where m is the number of managers.
- The self-join approach effectively models the hierarchical manager-report relationship.
- The JOIN (instead of LEFT JOIN) naturally filters for employees who are managers, as employees without direct reports won't appear in the results.
- For large employee tables, an index on the reports_to column would significantly improve join performance.
- The ROUND function without decimal places parameter rounds to the nearest integer as required by the problem.

1789. Primary Department For Each Employee

Description:

Table 119: Employee

Column Name	Type
employee_id	int int
department_id	varchar
primary_flag	

(employee_id, department_id) is the primary key (combination of columns with unique values) for this table. employee_id is the id of the employee. department_id is the id of the department to which the employee belongs. primary_flag is an ENUM (category) of type ('Y', 'N'). If the flag is 'Y', the department is the primary department for the employee. If the flag is 'N', the department is not the primary.

Employees can belong to multiple departments. When the employee joins other departments, they need to decide which department is their primary department. Note that when an employee belongs to only one department, their primary column is 'N'.

Write a solution to report all the employees with their primary department. For employees who belong to one department, report their only department.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Employee table:

employee_id	department_id	primary_flag
1 2 2 3 4 4 4	1 1 2 3 2 3 4	N Y N N N Y N

Output:

employee_id	department_id
1 2 3 4	1 1 3 3

Explanation: - The Primary department for employee 1 is 1. - The Primary department for employee 2 is 1. - The Primary department for employee 3 is 3. - The Primary department for employee 4 is 3.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT
    employee_id,
    department_id
FROM
    Employee
WHERE
    primary_flag = 'Y'
UNION
SELECT
    employee_id,
    department_id
FROM
    Employee
WHERE
    employee_id IN (
        SELECT employee_id
        FROM Employee
        GROUP BY employee_id
        HAVING COUNT(department_id) = 1
    );
```

Explanation:

- This query finds each employee's primary department, handling two distinct cases:
    1. Employees with a designated primary department (primary_flag = 'Y')
    2. Employees who belong to only one department (where primary_flag would be 'N' by problem definition)
- The first part of the UNION selects employees with an explicitly marked primary department.
- The second part identifies employees who belong to only one department using a subquery that counts departments per employee.
- The UNION combines these two sets of employees, with duplicate elimination as needed.
- Time complexity: O(n log n) where n is the number of employee-department records, due to the grouping operation.
- Space complexity: O(e) where e is the number of employees.
- The solution correctly handles the special case where an employee belongs to only one department (their primary_flag is 'N' by definition).
- The UNION efficiently combines the two cases without requiring complex conditional logic.

- For large datasets, indexes on employee_id and primary_flag would improve filtering and grouping performance.
- The query structure clearly separates the two logical conditions, making it readable and maintainable.

610. Triangle Judgement

Description:

Table 122: Triangle

Column Name	Type
x y z	int int int

In SQL, (x, y, z) is the primary key column for this table. Each row of this table contains the lengths of three line segments.

Report for every three line segments whether they can form a triangle.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Triangle table:

x	y	z
13	10	15 20 30 15

Output:

x	y	z	triangle
13	10	15 20 30 15	No Yes

Solution:

```
SELECT x, y, z
    , CASE WHEN x + y > z AND x + z > y AND z + y > x THEN 'Yes'
    ELSE 'No' END AS triangle
FROM Triangle
```

Explanation:

- This query determines whether each set of three line segments can form a triangle.
- In geometry, three segments can form a triangle if the sum of the lengths of any two sides is greater than the length of the remaining side.
- The CASE expression implements this triangle inequality theorem, checking all three combinations:
  - x + y > z (sum of sides x and y must exceed side z)
  - x + z > y (sum of sides x and z must exceed side y)
  - y + z > x (sum of sides y and z must exceed side x)
- If all three conditions are true, 'Yes' is returned; otherwise, 'No'.
- Time complexity: O(n) where n is the number of triangle records.
- Space complexity: O(n) for the result set.
- This solution directly applies the mathematical definition of the triangle inequality theorem.
- The CASE expression provides a clean, readable implementation of the multi-condition logic.
- No grouping or joining is needed, making this an efficient single-pass solution.
- The query structure preserves the original columns (x, y, z) and adds the derived 'triangle' column as required.

180. Consecutive Numbers

Description:

Table 125: Logs

Column Name	Type
id num	int varchar

In SQL, id is the primary key for this table. id is an autoincrement column starting from 1.

Find all numbers that appear at least three times consecutively.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input: Logs table:

id	num

1 2 3	1 1 1 2
4 5 6	1 2 2
7	

Output:

ConsecutiveNums
1

Explanation: 1 is the only number that appears consecutively for at least three times.

Solution:

```
WITH base AS (
    SELECT id, num
        , LAG(num, 1) OVER (ORDER BY id)
        , LEAD(num, 1) OVER (ORDER BY id)
    FROM Logs
)

SELECT DISTINCT num AS ConsecutiveNums
FROM base
WHERE num = lag AND num = lead
```

Explanation:

- This query identifies numbers that appear at least three times consecutively in the Logs table.
- The solution uses a Common Table Expression (CTE) with window functions to look ahead at subsequent rows.
- `LEAD(num, 1) OVER (ORDER BY id)` retrieves the value from the next row, while `LEAD(num, 2)` gets the value from two rows ahead.
- The main query then filters for rows where the current number equals both the next number and the number after that, indicating three consecutive occurrences.
- `DISTINCT` ensures each consecutive number appears only once in the results, even if it occurs consecutively multiple times.
- Time complexity: O(n log n) where n is the number of log entries, due to the sorting operation implicit in the window functions.
- Space complexity: O(n) for the CTE results and O(m) for the final output where m is the number of distinct consecutive numbers.

- Window functions provide an elegant way to compare values across rows without requiring complex self-joins.
- This approach efficiently handles the "consecutive" requirement by leveraging PostgreSQL's LEAD function to look ahead.
- For larger datasets, an index on the id column would improve the performance of the window functions.
- The solution assumes that ids are sequential without gaps, as implied by the problem description stating id is an autoincrement column.

1164. Product Price at a Given Date

Description:

Table 128: Products

Column Name	Type
product_id	int int date
new_price	
change_date	

(product_id, change_date) is the primary key (combination of columns with unique values) of this table. Each row of this table indicates that the price of some product was changed to a new price at some date.

Write a solution to find the prices of all products on 2019-08-16. Assume the price of all products before any change is 10.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Products table:

product_id	new_price	change_date
1 2 1 1 2 3	20 50 30 35 65	2019-08-14
	20	2019-08-14
		2019-08-15
		2019-08-16
		2019-08-17
		2019-08-18

Output:

product_id	price
2 1 3	50 35 10

Solution:

```sql
-- Write your PostgreSQL query statement below
WITH LatestPriceBeforeDate AS (
    SELECT
        product_id,
        new_price,
        ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY change_date DESC) AS rn
    FROM
        Products
    WHERE
        change_date <= '2019-08-16'
)
SELECT
    DISTINCT p.product_id,
    COALESCE(lp.new_price, 10) AS price
FROM
    (SELECT DISTINCT product_id FROM Products) p
LEFT JOIN
    LatestPriceBeforeDate lp ON p.product_id = lp.product_id AND lp.rn = 1;
```

Explanation:

- This query finds the price of each product on a specific date (2019-08-16), with a default price of 10 for products without any price changes before that date.
- The solution uses a Common Table Expression (CTE) to identify the most recent price change for each product before or on the target date.
- Within the CTE, `ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY change_date DESC)` assigns row numbers to price changes for each product, with the most recent change getting row number 1.
- The main query performs a LEFT JOIN between all unique product IDs and the filtered latest price changes.
- `COALESCE(lp.new_price, 10)` handles products that have no price changes before the target date, defaulting to the specified price of 10.
- Time complexity: O(n log n) where n is the number of price change records, due to the window function and partitioning.
- Space complexity: O(p) where p is the number of unique products.

- The solution correctly handles all three cases:
  - Products with price changes on the exact target date
  - Products with price changes before the target date
  - Products with no price changes before the target date (default price of 10)
- The use of ROW_NUMBER() to identify the most recent price change is more efficient than using self-joins or subqueries with MAX().
- For large datasets, indexes on product_id and change_date would significantly improve performance.

1204. Last Person to Fit in the Bus

Description:

Table 131: Queue

Column Name	Type
person_id	int varchar
person_name	int int
weight turn	

person_id column contains unique values. This table has the information about all people waiting for a bus. The person_id and turn columns will contain all numbers from 1 to n, where n is the number of rows in the table. turn determines the order of which the people will board the bus, where turn=1 denotes the first person to board and turn=n denotes the last person to board. weight is the weight of the person in kilograms.

There is a queue of people waiting to board a bus. However, the bus has a weight limit of 1000 kilograms, so there may be some people who cannot board.

Write a solution to find the person_name of the last person that can fit on the bus without exceeding the weight limit. The test cases are generated such that the first person does not exceed the weight limit.

Note that only one person can board the bus at any given turn.

The result format is in the following example.

Example 1:

Input:

Queue table:

person_id	person_name	weight	turn

5 4 3 6 1 2	Alice Bob Alex John Cena Winston Marie	250 175 350 400 500 200	1 5 2 3 6 4

Output:

person_name
John Cena

Explanation: The folowing table is ordered by the turn for simplicity.

Turn	ID	Name	Weight	Total Weight
1 2 3 4	5 3 6	Alice Alex	250 350	250 600 1000
5 6	2 4 1	John Cena	400 200	1200
		Marie Bob	175 500	
		Winston		

Solution:

```sql
-- Write your PostgreSQL query statement below
WITH running AS (
    SELECT person_name, weight, turn
        , SUM(weight) OVER (ORDER BY turn)
    FROM queue
),

person AS (
    SELECT person_name, turn, sum, LEAD(sum, 1) OVER (ORDER BY turn)
    FROM running
),

winner AS (
    SELECT FIRST_VALUE(person_name) OVER (ORDER BY turn) AS person_name
    FROM person
    WHERE lead > 1000 OR lead IS NULL
)

SELECT DISTINCT person_name
FROM winner
```

Explanation:

- This query identifies the last person who can board the bus without exceeding the 1000 kg weight limit using a three-step approach with CTEs.
- The first CTE (`running`) calculates a running sum of weights as people board in order using `SUM(weight) OVER (ORDER BY turn)`.
- The second CTE (`person`) applies the `LEAD` function to look ahead at the next person's cumulative weight, which helps identify the threshold where the weight limit is exceeded.
- The third CTE (`winner`) uses `FIRST_VALUE` to find the first person where the next person in line would exceed the 1000kg limit or where there is no next person (last in queue).
- The `WHERE lead > 1000 OR lead IS NULL` condition handles two cases: when the next person would cause the total to exceed 1000kg, or when we've reached the last person.
- The final query selects the distinct person_name from the winner CTE.
- Time complexity: O(n log n) where n is the number of people in the queue, due to the window functions and sorting.
- Space complexity: O(n) for the multiple CTE results.
- This approach cleverly leverages the LEAD function to peek ahead at what the weight would be after the next person boards.
- The FIRST_VALUE window function efficiently identifies the critical threshold where adding one more person would exceed the limit.
- The solution handles the edge case where all people can fit on the bus (using the OR lead IS NULL condition).
- For large datasets, an index on the turn column would improve the performance of the window functions.
- The multi-step approach makes the logic very clear and separates the different aspects of the calculation.

1907. Count Salary Categories

Description:

Table 135: Accounts

Column Name	Type
account_id	int int
income	

account_id is the primary key (column with unique values) for this table. Each row contains information about the monthly income for one bank account.

Write a solution to calculate the number of bank accounts for each salary category. The salary categories are:

"Low Salary": All the salaries strictly less than $20000.
"Average Salary": All the salaries in the inclusive range [$20000, $50000].
"High Salary": All the salaries strictly greater than $50000.

The result table must contain all three categories. If there are no accounts in a category, return 0.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Accounts table:

account_id	income
3	108939
2	12747
8	87709
6	91796

Output:

category	accounts_count
Low Salary	1
Average Salary	0
High Salary	3

Explanation: Low Salary: Account 2. Average Salary: No accounts. High Salary: Accounts 3, 6, and 8.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT 'Low Salary' AS category, COUNT(*) AS accounts_count
FROM Accounts
WHERE income < 20000
UNION
SELECT 'Average Salary' AS category, COUNT(*) AS accounts_count
FROM Accounts
WHERE 20000 <= income AND income <= 50000
UNION
SELECT 'High Salary' AS category, COUNT(*) AS accounts_count
```

```
FROM Accounts
WHERE 50000 < income
```

Explanation:

- This query calculates the number of bank accounts in each of three salary categories: Low, Average, and High.
- The solution uses UNION ALL to combine the results of three separate COUNT queries, one for each salary category.
- For "Low Salary," it counts accounts with income strictly less than $20,000.
- For "Average Salary," it counts accounts with income between $20,000 and $50,000 (inclusive).
- For "High Salary," it counts accounts with income strictly greater than $50,000.
- Each query explicitly specifies the category name as a string literal.
- Time complexity: O(n) where n is the number of accounts, as each account needs to be evaluated against the income criteria.
- Space complexity: O(1) as the result set always has exactly 3 rows.
- The UNION ALL approach ensures that all three categories appear in the results, even if some have zero accounts.
- This is more efficient than using a GROUP BY with a CASE expression because we only need to scan the table once per category.
- For large account tables, indexes on the income column would significantly improve filtering performance.
- The solution meets the requirement to include all three categories in the output, with 0 counts for empty categories.

1978. Employees Whose Manager Left The Company

Description: Table: Employees

Column Name	Type
employee_id	int varchar
name	int int
manager_id	
salary	

In SQL, employee_id is the primary key for this table. This table contains information about the employees, their salary, and the ID of their manager. Some employees do not have a manager (manager_id is null).

Find the IDs of the employees whose salary is strictly less than $30000 and whose manager left the company. When a manager leaves the company, their information is deleted from the Employees table, but the reports still have their manager_id set to the manager that left.

Return the result table ordered by employee_id.

The result format is in the following example.

Example 1:

Input:

Employees table:

employee_id	name	manager_id	salary
3 12 13 1 9 11	Mila	9 null null 11	60301
	Antonella	null 6	31000
	Emery Kalel		67084
	Mikaela		21241
	Joziah		50937
			28485

Output:

employee_id
11

Explanation: The employees with a salary less than $30000 are 1 (Kalel) and 11 (Joziah). Kalel's manager is employee 11, who is still in the company (Joziah). Joziah's manager is employee 6, who left the company because there is no row for employee 6 as it was deleted.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT employee_id
FROM Employees
WHERE salary < 30000 AND manager_id NOT IN (
    SELECT employee_id FROM Employees
)
ORDER BY employee_id
```

Explanation:

- This query identifies employees who meet three conditions:

1. Their salary is strictly less than $30,000
2. Their manager has left the company (manager_id doesn't exist in the employee_id column)

- The `NOT IN` subquery efficiently checks whether the manager_id exists among current employees.
- Results are ordered by employee_id as required by the problem.
- Time complexity: O(n log n) where n is the number of employees, due to the sorting operation and the subquery.
- Space complexity: O(m) where m is the number of employees meeting the criteria.
- This solution correctly identifies employees whose managers left by checking for manager_id values that don't appear in the employee_id column.
- The NOT IN approach is straightforward and readable, though for very large datasets, an anti-join pattern might be more efficient.
- The multiple conditions in the WHERE clause are evaluated together to filter the results in a single pass through the data.
- For large employee tables, indexes on employee_id, manager_id, and salary would significantly improve filtering and sorting performance.

626. Exchange Seats

Description:

Table 141: Seat

Column Name	Type
id student	int varchar

id is the primary key (unique value) column for this table. Each row of this table indicates the name and the ID of a student. The ID sequence always starts from 1 and increments continuously.

Write a solution to swap the seat id of every two consecutive students. If the number of students is odd, the id of the last student is not swapped.

Return the result table ordered by id in ascending order.

The result format is in the following example.

Example 1:

Input:

Seat table:

id	student
1 2 3	Abbot Doris
4 5	Emerson
	Green
	Jeames

Output:

id	student
1 2 3	Doris Abbot
4 5	Green
	Emerson
	Jeames

Explanation:

Note that if the number of students is odd, there is no need to change the last one's seat.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT CASE WHEN id % 2 = 1 AND id = (SELECT MAX(id) FROM seat) THEN id
    WHEN id % 2 = 1 THEN id + 1 ELSE id - 1 END AS id
    , student
FROM Seat
ORDER BY id ASC
```

Explanation:

- This query swaps the ID of every two consecutive students using a CASE expression to determine the new ID for each student.
- The CASE expression handles three scenarios:
    1. Odd-numbered IDs: Changes to ID+1 (moving the student to the next seat)
    2. Even-numbered IDs: Changes to ID-1 (moving the student to the previous seat)
    3. Special case for the last student if total count is odd: Keeps the original ID
- The condition `id % 2 = 1 AND id = (SELECT MAX(id) FROM Seat)` identifies the last student with an odd ID number.
- The query preserves the student names while changing only their IDs.
- Results are ordered by the new ID to maintain the desired output sequence.
- Time complexity: O(n) where n is the number of students.

- Space complexity: O(n) for the result set.
- The solution elegantly handles the seat swapping without requiring complex joins or temporary tables.
- For large datasets, the subquery to find the maximum ID would benefit from an index on the id column.
- This approach maintains a single pass through the data while implementing the logical swapping operation.

1341. Movie Rating

Description:

Table 144: Movies

Column Name	Type
movie_id	int
title	varchar

movie_id is the primary key (column with unique values) for this table. title is the name of the movie.

Table 145: Users

Column Name	Type
user_id	int
name	varchar

user_id is the primary key (column with unique values) for this table. The column 'name' has unique values.

Table 146: MovieRating

Column Name	Type
movie_id	int
user_id	int
rating	int
created_at	date

(movie_id, user_id) is the primary key (column with unique values) for this table. This table contains the rating of a movie by a user in their review. created_at is the user's review date.

Write a solution to:

```
Find the name of the user who has rated the greatest number of movies. In case of a tie, retu
Find the movie name with the highest average rating in February 2020. In case of a tie, retur
```

The result format is in the following example.

Example 1:

Input:

Movies table:

movie_id	title
1 2 3	Avengers Frozen 2
	Joker

Users table:

user_id	name
1 2 3 4	Daniel Monica
	Maria James

MovieRating table:

movie_id	user_id	rating	created_at
1 1 1 1 2 2 2 3 3	1 2 3 4 1 2 3 1 2	3 4 2 1 5 2 2 3 4	2020-01-12
			2020-02-11
			2020-02-12
			2020-01-01
			2020-02-17
			2020-02-01
			2020-03-01
			2020-02-22
			2020-02-25

Output:

results
Daniel Frozen 2

Explanation: Daniel and Monica have rated 3 movies ("Avengers", "Frozen 2" and "Joker") but Daniel is smaller lexicographically. Frozen 2 and Joker have a rating average of 3.5 in February but Frozen 2 is smaller lexicographically.

Solution:

```sql
-- Write your PostgreSQL query statement below
(SELECT u.name AS results--, COUNT(mr.rating)
FROM MovieRating mr INNER JOIN Users u
    ON mr.user_id = u.user_id
GROUP BY u.name
ORDER BY COUNT(mr.rating) DESC, u.name ASC
LIMIT 1)
UNION ALL
(SELECT m.title AS results--, AVG(mr.rating)
FROM MovieRating mr INNER JOIN Movies m
    ON mr.movie_id = m.movie_id
WHERE mr.created_at BETWEEN '2020-02-01' AND '2020-02-29'
GROUP BY m.title
ORDER BY AVG(mr.rating) DESC, m.title ASC
LIMIT 1)
```

Explanation:

- This query solves two separate problems and combines their results using UNION ALL:
  1. Finding the user who has rated the most movies (with lexicographic tie-breaking)
  2. Finding the movie with the highest average rating in February 2020 (with lexicographic tie-breaking)
- For the first part:
  - JOIN the MovieRating and Users tables to get user information
  - GROUP BY user_id and name to count ratings per user
  - ORDER BY count (descending) and then by name (ascending) for tie-breaking
  - LIMIT 1 to get only the top result
- For the second part:
  - JOIN the MovieRating and Movies tables to get movie information
  - Filter for ratings created in February 2020 using the BETWEEN clause
  - GROUP BY movie_id and title to calculate average rating per movie
  - ORDER BY average rating (descending) and then by title (ascending) for tie-breaking
  - LIMIT 1 to get only the top result
- Time complexity: O(n log n) where n is the number of ratings, due to the sorting operations.
- Space complexity: O(n) for the intermediate joined results before grouping.
- UNION ALL is used instead of UNION since we know the results will be distinct and we want to preserve the order.

- The solution efficiently handles the tie-breaking requirements by incorporating the secondary sort criteria (name, title) in the ORDER BY clauses.

1321. Restaurant Growth

Description:

Table 151: Customer

Column Name	Type
customer_id name	int varchar
visited_on amount	date int

In SQL,(customer_id, visited_on) is the primary key for this table. This table contains data about customer transactions in a restaurant. visited_on is the date on which the customer with ID (customer_id) has visited the restaurant. amount is the total paid by a customer.

You are the restaurant owner and you want to analyze a possible expansion (there will be at least one customer every day).

Compute the moving average of how much the customer paid in a seven days window (i.e., current day + 6 days before). average_amount should be rounded to two decimal places.

Return the result table ordered by visited_on in ascending order.

The result format is in the following example.

Example 1:

Input: Customer table:

customer_id	name	visited_on	amount
1 2 3 4 5 6 7 8 9	Jhon Daniel Jade	2019-01-01	100 110 120 130
1 3	Khaled Winston	2019-01-02	110 140 150 80
	Elvis Anna Maria	2019-01-03	110 130 150
	Jaze Jhon Jade	2019-01-04	
		2019-01-05	
		2019-01-06	
		2019-01-07	
		2019-01-08	
		2019-01-09	
		2019-01-10	
		2019-01-10	

Output:

visited_on	amount	average_amount
2019-01-07	860 840 840 1000	122.86 120 120
2019-01-08		142.86
2019-01-09		
2019-01-10		

Explanation: 1st moving average from 2019-01-01 to 2019-01-07 has an average_amount of (100 + 110 + 120 + 130 + 110 + 140 + 150)/7 = 122.86 2nd moving average from 2019-01-02 to 2019-01-08 has an average_amount of (110 + 120 + 130 + 110 + 140 + 150 + 80)/7 = 120 3rd moving average from 2019-01-03 to 2019-01-09 has an average_amount of (120 + 130 + 110 + 140 + 150 + 80 + 110)/7 = 120 4th moving average from 2019-01-04 to 2019-01-10 has an average_amount of (130 + 110 + 140 + 150 + 80 + 110 + 130 + 150)/7 = 142.86

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT visited_on, SUM(cumount) OVER (ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS amount, RO
FROM (
    SELECT visited_on, SUM(amount) AS cumount
    FROM Customer
    GROUP BY visited_on
    ORDER BY visited_on
)
OFFSET 6;
```

Explanation:

- This query calculates a 7-day moving sum and average of restaurant revenue using window functions.
- The inner subquery first calculates the total daily revenue by grouping the Customer table by visited_on.
- The main query then applies window functions to implement the 7-day moving window with `ROWS BETWEEN 6 PRECEDING AND CURRENT ROW`.
- This window frame includes the current row and 6 rows before it, perfectly creating the 7-day window required.
- Two window calculations are performed:
  - `SUM(cumount) OVER (ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)` for the 7-day total amount
  - The same sum divided by 7 and rounded to 2 decimal places for the average amount

- The `OFFSET 6` clause skips the first 6 days in the result set, since these days don't have enough preceding days to form a complete 7-day window.
- Time complexity: O(n log n) where n is the number of customer visit records, primarily due to the sorting operation.
- Space complexity: O(d) where d is the number of unique visited dates.
- This solution elegantly uses PostgreSQL's window functions, which are optimized for these types of sliding window calculations.
- Window functions are typically more efficient than self-joins for moving window problems because they only require a single scan of the sorted data.
- The query handles multiple visits on the same day by pre-aggregating daily amounts in the subquery.
- For large datasets, an index on visited_on would improve the performance of both grouping and sorting operations.

602. Friend Requests II: Who Has The Most Friends

Description:

Table 154: RequestAccepted

Column Name	Type
requester_id	int int date
accepter_id	
accept_date	

(requester_id, accepter_id) is the primary key (combination of columns with unique values) for this table. This table contains the ID of the user who sent the request, the ID of the user who received the request, and the date when the request was accepted.

Write a solution to find the people who have the most friends and the most friends number.

The test cases are generated so that only one person has the most friends.

The result format is in the following example.

Example 1:

Input: RequestAccepted table:

requester_id	accepter_id	accept_date
1 1 2 3	2 3 3 4	2016/06/03
		2016/06/08
		2016/06/08
		2016/06/09

Output:

id	num
3	3

Explanation: The person with id 3 is a friend of people 1, 2, and 4, so he has three friends in total, which is the most number than any others.

Follow up: In the real world, multiple people could have the same most number of friends. Could you find all these people in this case?

Solution:

```sql
-- Write your PostgreSQL query statement below
WITH full_count AS (
    SELECT requester_id AS id, COUNT(*)
    FROM RequestAccepted
    GROUP BY requester_id
    UNION ALL
    SELECT accepter_id AS id, COUNT(*)
    FROM RequestAccepted
    GROUP BY accepter_id
)

SELECT id, SUM(count) AS num
FROM full_count
GROUP BY id
ORDER BY SUM(count) DESC
LIMIT 1;
```

Explanation:

- This query identifies the person with the most friends in the system.
- The solution uses a Common Table Expression (CTE) called `full_count` that combines both requesters/counts and accepters/counts with UNION ALL.
- This approach counts each person once for each friendship they're involved in, whether as a requester or accepter.
- After grouping by id, SUM(count) gives the total number of friendships for each person.
- Results are ordered by friend count in descending order, and LIMIT 1 retrieves only the person with the most friends.
- Time complexity: O(n log n) where n is the number of accepted friend requests, due to the sorting operation.
- Space complexity: O(n) for the CTE and intermediate results.

- The UNION ALL approach efficiently combines both sides of each friendship relation.
- Unlike UNION, UNION ALL preserves duplicates, which is necessary to count each friendship relation correctly.
- The solution correctly handles the problem's definition of friendship count, which includes both outgoing and incoming accepted requests.
- For large datasets, indexes on requester_id and accepter_id would improve the performance of the UNION ALL operation.

585. Investments in 2016

Description:

Table 157: Insurance

Column Name	Type
pid tiv_2015	int float
tiv_2016 lat lon	float float float

pid is the primary key (column with unique values) for this table. Each row of this table contains information about one policy where: pid is the policyholder's policy ID. tiv_2015 is the total investment value in 2015 and tiv_2016 is the total investment value in 2016. lat is the latitude of the policy holder's city. It's guaranteed that lat is not NULL. lon is the longitude of the policy holder's city. It's guaranteed that lon is not NULL.

Write a solution to report the sum of all total investment values in 2016 tiv_2016, for all policyholders who:

```
have the same tiv_2015 value as one or more other policyholders, and
are not located in the same city as any other policyholder (i.e., the (lat, lon) attribute pa
```

Round tiv_2016 to two decimal places.

The result format is in the following example.

Example 1:

Input:

Insurance table:

pid	tiv_2015	tiv_2016	lat	lon
1 2 3 4	10 20 10 10	5 20 30 40	10 20	10 20
			20 40	20 40

Output:

tiv_2016
45.00

Explanation: The first record in the table, like the last record, meets both of the two criteria. The tiv_2015 value 10 is the same as the third and fourth records, and its location is unique.

The second record does not meet any of the two criteria. Its tiv_2015 is not like any other policyholders and its location is the same as the third record, which makes the third record fail, too. So, the result is the sum of tiv_2016 of the first and last record, which is 45.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT ROUND(SUM(tiv_2016)::numeric, 2) AS tiv_2016
FROM (
   SELECT *,
       COUNT(*)OVER(PARTITION BY tiv_2015) AS tiv_2015_cnt,
       COUNT(*)OVER(PARTITION BY lat, lon) AS loc_cnt
   FROM Insurance
   )t0
WHERE tiv_2015_cnt > 1
AND loc_cnt = 1
```

Explanation:

- This query calculates the sum of insurance investments in 2016 for policyholders who meet two specific criteria.
- The solution uses one subquery with two window functions to generate the condition test cases:
    1. `tiv_2015_cnt` gets the count of the tiv_2015 values for policyholders.
    2. `loc_cnt` get the count of the exact same location (lat, lon).
- The main query then sums the tiv_2016 values for policies meeting both criteria.
- The result is rounded to 2 decimal places using ROUND and the ::numeric type cast.
- Time complexity: $O(n^2)$ due to the correlated subqueries, where n is the number of insurance policies.
- Space complexity: O(1) as we're only calculating a single aggregate value.
- The window function approach is easier to understand, but often less efficient than using solutions that leverage `EXISTS` for these types of conditions.
    - For larger datasets, `EXISTS` is great because it stops the subquery execution once the condition is met

- The solution correctly handles the combination of conditions where we need both a match (same tiv_2015) and non-match (unique location).
- For large datasets, indexes on tiv_2015, lat, and lon would significantly improve the performance of the subqueries.

185. Department Top Three Salaries

Description:

Table 160: 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 161: 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.

The result format is in the following example.

Example 1:

Input: Employee table:

id	name	salary	departmentId

1 2 3	Joe	85000	1 2 2 1 1 1 1
4 5 6	Henry	80000	
7	Sam Max	60000	
	Janet	90000	
	Randy	69000	
	Will	85000	
		70000	

Department table:

id	name
1 2	IT Sales

Output:

Department	Employee	Salary
IT IT IT IT	Max Joe	90000
Sales Sales	Randy Will	85000
	Henry Sam	85000
		70000
		80000
		60000

Explanation:

In the IT department:

- Max earns the highest unique salary
- Both Randy and Joe earn the second-highest unique salary
- Will earns the third-highest unique salary

In the Sales department:

- Henry earns the highest salary
- Sam earns the second-highest salary
- There is no third-highest salary as there are only two employees

Constraints:

- `There are no employees with the exact same name, salary and department.`

Solution:

```
-- Write your PostgreSQL query statement below
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.

1667. Fix Names in a Table

Description: Table: Users

Column Name	Type
user_id	int
name	varchar

user_id is the primary key (column with unique values) for this table. This table contains the ID and the name of the user. The name consists of only lowercase and uppercase characters.

Write a solution to fix the names so that only the first character is uppercase and the rest are lowercase.

Return the result table ordered by user_id.

The result format is in the following example.

Example 1:

Input:

Users table:

user_id	name
1	aLice
2	bOB

Output:

user_id	name
1	Alice
2	Bob

Solution:

```
-- Write your PostgreSQL query statement below
SELECT user_id, CONCAT(UPPER(LEFT(name, 1)), LOWER(SUBSTRING(name, 2))) AS name
FROM Users
ORDER BY user_id
```

Explanation:

- This query modifies each name to have only the first character uppercase and the rest lowercase.
- The solution uses several string manipulation functions to achieve this:
  - `SUBSTRING(name, 1, 1)` extracts the first character of the name
  - `UPPER()` converts this first character to uppercase

  - `SUBSTRING(name, 2)` extracts the rest of the name (all characters from position 2 onward)
  - `LOWER()` converts these remaining characters to lowercase
  - `CONCAT()` combines the uppercase first character with the lowercase remainder

- Results are ordered by user_id as required by the problem.
- Time complexity: O(n log n) where n is the number of users, due to the sorting operation.
- Space complexity: O(n) for the result set.
- The string manipulation approach efficiently handles names of any length in a single pass.
- PostgreSQL's rich string functions make this transformation straightforward, requiring just one SQL statement.
- The solution correctly handles edge cases such as empty names or names that are already in the desired format.
- For large user tables, an index on user_id would improve sorting performance.
- This string manipulation pattern is very common in database applications for standardizing text formats.

1527. Patients With a Condition

Description:

Table 168: Patients

Column Name	Type
patient_id	int varchar
patient_name	varchar
conditions	

patient_id is the primary key (column with unique values) for this table. 'conditions' contains 0 or more code separated by spaces. This table contains information of the patients in the hospital.

Write a solution to find the patient_id, patient_name, and conditions of the patients who have Type I Diabetes. Type I Diabetes always starts with DIAB1 prefix.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Patients table:

patient_id	patient_name	conditions
1 2 3 4 5	Daniel Alice Bob George Alain	YFEV COUGH DIAB100 MYOP ACNE DIAB100 DIAB201

Output:

patient_id	patient_name	conditions
3 4	Bob George	DIAB100 MYOP ACNE DIAB100

Explanation: Bob and George both have a condition that starts with DIAB1.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT patient_id, patient_name, conditions
FROM Patients
WHERE conditions LIKE 'DIAB1%' OR conditions LIKE '% DIAB1%'
```

Explanation:

- This query identifies patients who have Type I Diabetes, which is indicated by the 'DIAB1' prefix in their conditions.
- The solution uses the LIKE operator with pattern matching to find the 'DIAB1' prefix in two possible positions:
    1. `conditions LIKE 'DIAB1%'` checks if 'DIAB1' appears at the start of the conditions string
    2. `conditions LIKE '% DIAB1%'` checks if 'DIAB1' appears after a space elsewhere in the string
- These two patterns together cover all cases where 'DIAB1' appears as a separate code in the conditions list.
- Time complexity: O(n) where n is the number of patients, as each patient's conditions need to be checked against the patterns.
- Space complexity: O(m) where m is the number of patients with Type I Diabetes.
- The two-part pattern matching is crucial to correctly handle the problem's requirement that DIAB1 must appear as a distinct code.
- For example, this approach correctly identifies 'DIAB100' as a Type I Diabetes code, but would not match 'ADIAB1' since it's not a separate code.

- The solution efficiently filters patients in a single pass through the data.
- For large patient databases, a full-text search index on the conditions column would improve pattern matching performance.

196. Delete Duplicate Emails

Description: Table: Person

Column Name	Type
id email	int varchar

id is the primary key (column with unique values) for this table. Each row of this table contains an email. The emails will not contain uppercase letters.

Write a solution to delete all duplicate emails, keeping only one unique email with the smallest id.

For SQL users, please note that you are supposed to write a DELETE statement and not a SELECT one.

For Pandas users, please note that you are supposed to modify Person in place.

After running your script, the answer shown is the Person table. The driver will first compile and run your piece of code and then show the Person table. The final order of the Person table does not matter.

The result format is in the following example.

Example 1:

Input: Person table:

id	email
1 2 3	john@example.com
	bob@example.com
	john@example.com

Output:

id	email
1 2	john@example.com
	bob@example.com

Explanation: john@example.com is repeated two times. We keep the row with the smallest Id = 1.

Solution:

```sql
-- Write your PostgreSQL query statement below
DELETE FROM Person
WHERE id IN (
    SELECT p1.id
    FROM Person p1
    JOIN Person p2 ON p1.email = p2.email AND p1.id > p2.id
);
```

Explanation:

- This query deletes duplicate emails from the Person table, keeping only the one with the smallest id for each email.
- The solution uses a self-join to identify pairs of rows with the same email.
- The condition `p1.id > p2.id` ensures we're only selecting the rows with higher IDs for each email (the duplicates we want to delete).
- The DELETE statement removes all rows whose IDs are in the result set of the subquery.
- Time complexity: $O(n^2)$ in the worst case due to the self-join, where n is the number of person records.
- Space complexity: O(n) for the intermediate result set from the join.
- This approach correctly preserves the row with the smallest ID for each email while removing all duplicates.
- The self-join pattern is a common technique for identifying duplicate values in SQL.
- Note that PostgreSQL supports using DELETE with a FROM clause, but the syntax above is more standard across different SQL implementations.
- For large tables, an index on the email column would significantly improve the performance of the join operation.
- Some database systems might not allow selecting from the same table being modified in a subquery, but PostgreSQL handles this correctly.

176. Second Highest Salary

Description:

Table 174: Employee

Column Name	Type
id	int
salary	int

id is the primary key (column with unique values) for this table. Each row of this table contains information about the salary of an employee.

Write a solution to find the second highest distinct salary from the Employee table. If there is no second highest salary, return null (return None in Pandas).

The result format is in the following example.

Example 1:

Input:

Employee table:

id	salary
1 2 3	100 200 300

Output:

SecondHighestSalary
200

Example 2:

Input:

Employee table:

id	salary
1	100

Output:

SecondHighestSalary
null

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT
    (SELECT DISTINCT salary
     FROM Employee
     ORDER BY salary DESC
     LIMIT 1 OFFSET 1) AS SecondHighestSalary;
```

Explanation:

- This query finds the second highest distinct salary from the Employee table, returning NULL if no second highest exists.
- The solution uses a subquery with DISTINCT to eliminate duplicate salaries.
- The ORDER BY clause sorts salaries in descending order (highest first).
- LIMIT 1 OFFSET 1 skips the first result (highest salary) and takes the next one (second highest).
- The outer SELECT ensures that NULL is returned when no second highest salary exists, rather than an empty result set.
- Time complexity: O(n log n) where n is the number of employees, due to the sorting operation.
- Space complexity: O(1) as we're only retrieving a single value.
- This approach handles the NULL case elegantly - when there are fewer than 2 distinct salaries, the subquery returns no rows, and the outer SELECT converts this to NULL.
- The use of DISTINCT ensures we're considering unique salary values rather than employee records, as required by the problem.
- For large employee tables, an index on the salary column would improve sorting performance.
- This pattern of "LIMIT 1 OFFSET n" is a standard technique for finding the nth highest value in SQL.
- An alternative implementation could use window functions like DENSE_RANK(), but this approach is more concise for the specific case of finding just the second highest value.

1484. Group Products Sold by The Date

Description:

Table Activities:

Column Name	Type
sell_date	date varchar
product	

There is no primary key (column with unique values) for this table. It may contain duplicates. Each row of this table contains the product name and the date it was sold in a market.

Write a solution to find for each date the number of different products sold and their names.

The sold products names for each date should be sorted lexicographically.

Return the result table ordered by sell_date.

The result format is in the following example.

Example 1:

Input:

Activities table:

sell_date	product
2020-05-30	Headphone
2020-06-01	Pencil Mask
2020-06-02	Basketball
2020-05-30	Bible Mask
2020-06-01	T-Shirt
2020-06-02	
2020-05-30	

Output:

sell_date	num_sold	products
2020-05-30	3 2 1	Basketball,Headphone,T-shirt
2020-06-01		Bible,Pencil Mask
2020-06-02		

Explanation:

For 2020-05-30, Sold items were (Headphone, Basketball, T-shirt), we sort them lexicographically and separate them by a comma. For 2020-06-01, Sold items were (Pencil, Bible), we sort them lexicographically and separate them by a comma. For 2020-06-02, the Sold item is (Mask), we just return it.

Solution:

```sql
-- Write your PostgreSQL query statement below
SELECT sell_date, COUNT(DISTINCT product) AS num_sold, STRING_AGG(DISTINCT product, ',') AS p
FROM Activities
GROUP BY sell_date
ORDER BY sell_date ASC
```

Explanation:

- This query groups products sold by date and formats the output as required.
- For each date, it calculates:
  - The number of unique products sold using `COUNT(DISTINCT product)`
  - A comma-separated list of product names, sorted lexicographically using `STRING_AGG(DISTINCT product, ',' ORDER BY product)`
- Results are grouped by sell_date and ordered chronologically.
- PostgreSQL's STRING_AGG function is ideal for this task, combining distinct product names into a single string with specified delimiter and ordering.
- Time complexity: O(n log n) where n is the number of activity records, due to the sorting operations.
- Space complexity: O(d) where d is the number of unique dates.
- The DISTINCT keyword within both aggregate functions ensures each product is counted and listed only once per date.
- This elegantly handles the problem's requirement for lexicographically sorted product names.
- For large activity tables, indexes on sell_date and product would improve grouping and sorting performance.
- STRING_AGG is PostgreSQL's equivalent to GROUP_CONCAT in MySQL or LISTAGG in Oracle, perfect for creating delimited lists from grouped data.

1327. List The Products Ordered in a Period

Description:

Table 182: Products

Column Name	Type
product_id	int varchar
product_name	varchar
product_category	

product_id is the primary key (column with unique values) for this table. This table contains data about the company's products.

Table 183: Orders

Column Name	Type
product_id	int date int
order_date unit	

This table may have duplicate rows. product_id is a foreign key (reference column) to the Products table. unit is the number of products ordered in order_date.

Write a solution to get the names of products that have at least 100 units ordered in February 2020 and their amount.

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Products table:

product_id	product_name	product_category
1 2 3 4 5	Leetcode Solutions Jewels of Stringology HP Lenovo Leetcode Kit	Book Book Laptop Laptop T-shirt

Orders table:

product_id	order_date	unit
1 1 2 2 3 3 4 4 4 5 5 5	2020-02-05	60 70 30 80 2
	2020-02-10	3 20 30 60 50
	2020-01-18	50 50
	2020-02-11	
	2020-02-17	
	2020-02-24	
	2020-03-01	
	2020-03-04	
	2020-03-04	
	2020-02-25	
	2020-02-27	
	2020-03-01	

Output:

product_name	unit
Leetcode Solutions	130 100
Leetcode Kit	

Explanation: Products with product_id = 1 is ordered in February a total of (60 + 70) = 130. Products with product_id = 2 is ordered in February a total of 80. Products with product_id = 3 is ordered in February a total of (2 + 3) = 5. Products with product_id = 4 was not ordered in February 2020. Products with product_id = 5 is ordered in February a total of (50 + 50) = 100.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT p.product_name, SUM(o.unit) AS unit
FROM Products p INNER JOIN Orders o
    ON p.product_id = o.product_id
WHERE o.order_date BETWEEN '2020-02-01' AND '2020-02-29'
GROUP BY p.product_name
HAVING 100 <= SUM(o.unit)
ORDER BY SUM(o.unit) DESC
```

Explanation:

- This query identifies products with at least 100 units ordered in February 2020.
- It joins the Products and Orders tables to connect order information with product names.
- The WHERE clause filters for orders placed within February 2020 using the BETWEEN operator.
- For each product, SUM(o.unit) calculates the total units ordered during the period.
- The HAVING clause filters for products with at least 100 total units.
- Results are grouped by product_id and product_name, then ordered by total units in descending order.
- Time complexity: O(n log n) where n is the number of orders, due to the grouping and sorting operations.
- Space complexity: O(p) where p is the number of products meeting the criteria.
- The combination of JOIN, GROUP BY, and HAVING efficiently implements the multi-step filtering process.
- Including product_id in the GROUP BY clause prevents grouping errors if products with different IDs have the same name.
- For large order datasets, indexes on product_id and order_date would significantly improve filtering and join performance.
- The BETWEEN operator provides a clean way to specify the date range, automatically including both boundary dates.

1527. Find Users With Valid E-Mails

Description:

Table 187: Users

Column Name	Type
user_id name mail	int varchar varchar

user_id is the primary key (column with unique values) for this table. This table contains information of the users signed up in a website. Some e-mails are invalid.

Write a solution to find the users who have valid emails.

A valid e-mail has a prefix name and a domain where:

```
The prefix name is a string that may contain letters (upper or lower case), digits, underscor
The domain is '@leetcode.com'.
```

Return the result table in any order.

The result format is in the following example.

Example 1:

Input:

Users table:

user_id	name	mail
1 2 3 4 5 6 7	Winston	winston@leetcode.com
	Jonathan	jonathanisgreat
	Annabelle	bella-@leetcode.com
	Sally Marwan	sally.come@leetcode.com
	David Shapiro	quarz#2020@leetcode.com
		david69@gmail.com
		.shapo@leetcode.com

Output:

user_id	name	mail

1 3 4	Winston	winston@leetcode.com
	Annabelle	bella-@leetcode.com
	Sally	sally.come@leetcode.com

Explanation: The mail of user 2 does not have a domain. The mail of user 5 has the # sign which is not allowed. The mail of user 6 does not have the leetcode domain. The mail of user 7 starts with a period.

Solution:

```
-- Write your PostgreSQL query statement below
SELECT *
FROM Users
WHERE mail ~ '^[a-zA-Z][a-zA-Z0-9_.-]*@leetcode\.com$';
```

Explanation:

- This query identifies users with valid email addresses according to the specified rules.
- It uses PostgreSQL's `~` operator for regular expression matching to validate email addresses.
- The regular expression pattern has several components:
  - `^[a-zA-Z]` ensures the email starts with a letter (upper or lower case)
  - `[a-zA-Z0-9_.-]*` allows the prefix to contain letters, digits, underscores, periods, and/or dashes
  - `@leetcode\.com$` requires the exact domain "@leetcode.com" at the end (with the period escaped)
- Time complexity: O(n) where n is the number of users, as each email needs to be checked against the pattern.
- Space complexity: O(m) where m is the number of users with valid emails.
- Regular expressions provide a powerful and concise way to validate complex string patterns like email addresses.
- The `^` and `$` anchors ensure the entire string is matched, preventing partial matches.
- PostgreSQL's regex engine efficiently evaluates the pattern against each email address.
- For large user databases, this validation could potentially be pre-computed and stored as a boolean flag for faster querying.
- The solution correctly handles all the edge cases mentioned in the problem, such as emails starting with non-letters or having invalid characters.