1. **Identify Departments with Sufficient Staffing**
   Write a query to identify departments that have more than three employees. The departments table includes the department_id and department_name, while the employees table includes details about each employee, such as employee_id and department_id.

2. **Retrieve High-Earning Employees**
   List the names of employees who earn more than $5000 per month. The employees table includes columns like employee_id and name, and is joined with the salaries table, which contains employee_id and amount.

3. **Find Projects with Sufficient Team Size**
   List the project names that have at least two members assigned. The projects table includes project_id and project_name, and is linked to the project_members table, which tracks the project_id and member_id of each member.

4. **Identify Products without Sales**
   Retrieve the names of products that have no recorded sales. The products table includes product_id and product_name, while the sales table logs each sale with product_id and sale_date.

5. **List Customers from Specific Regions**
   List the names of customers from the "USA". The customers table includes customer_id, customer_name, and country, and can be joined with the orders table to find related sales data.

6. **Count Recently Active Users**
   Find the number of unique users who logged in during the last month. The users table includes user_id, and the user_logins table logs user_id and login_date.
7. **List High-Expenditure Departments**
   List the department names where the total salary expenditure exceeds $100,000. The departments table includes department_id and department_name, while the employees and salaries tables provide salary information.

8. **Identify Products Sold in Specific Months**
   Write a query to find products sold in June. The products table includes product_id and product_name, and the sales table logs each sale with product_id and sale_date.

9. **Count Employees in Different Roles**
   Count the number of employees in each role. The roles table includes role_id and role_name, and is linked to the employees table by the role_id.
10. **List Recently Hired Employees**
   Retrieve the names of employees hired after the year 2020. The employees table includes employee_id, name, and hire_date, and can be joined with additional details in the hire_dates table.

Solutions

sql

Copy code

```sql
-- 1. Identify Departments with Sufficient Staffing
SELECT d.department_name, COUNT(e.employee_id) AS employee_count
FROM departments d
JOIN employees e ON d.department_id = e.department_id
GROUP BY d.department_id, d.department_name
HAVING COUNT(e.employee_id) > 3;

-- 2. Retrieve High-Earning Employees
SELECT e.name
FROM employees e
JOIN salaries s ON e.employee_id = s.employee_id
WHERE s.amount > 5000;

-- 3. Find Projects with Sufficient Team Size
SELECT p.project_name
FROM projects p
JOIN project_members pm ON p.project_id = pm.project_id
GROUP BY p.project_id, p.project_name
HAVING COUNT(pm.member_id) >= 2;

-- 4. Identify Products without Sales
SELECT p.product_name
FROM products p
LEFT JOIN sales s ON p.product_id = s.product_id
WHERE s.product_id IS NULL;

-- 5. List Customers from Specific Regions
SELECT c.customer_name
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE c.country = 'USA';

-- 6. Count Recently Active Users
SELECT COUNT(DISTINCT u.user_id) AS active_users
```

```
FROM users u
JOIN user_logins ul ON u.user_id = ul.user_id
WHERE ul.login_date BETWEEN '2023-06-01' AND '2023-06-30';

-- 7. List High-Expenditure Departments
SELECT d.department_name, SUM(s.salary) AS total_salary
FROM departments d
JOIN employees e ON d.department_id = e.department_id
JOIN salaries s ON e.employee_id = s.employee_id
GROUP BY d.department_name
HAVING SUM(s.salary) > 100000;

-- 8. Identify Products Sold in Specific Months
SELECT p.product_name
FROM products p
JOIN sales s ON p.product_id = s.product_id
WHERE MONTH(s.sale_date) = 6;

-- 9. Count Employees in Different Roles
SELECT r.role_name, COUNT(e.employee_id) AS employee_count
FROM roles r
JOIN employees e ON r.role_id = e.role_id
GROUP BY r.role_name;

-- 10. List Recently Hired Employees
SELECT e.name
FROM employees e
JOIN hire_dates h ON e.employee_id = h.employee_id
WHERE h.hire_date > '2020-12-31';
```

1. **Find Products Priced Above Average**
   Table Description:
   - products table:
     - product_id (unique identifier for each product)
     - product_name (name of the product)
     - price (price of the product)

   Question:
   Write a query to find the names of products whose price is above the average price.

2. **List Employees Not Assigned to Any Project**
   Table Description:
   - employees table:
     - employee_id (unique identifier for each employee)
     - employee_name (name of the employee)

- project_assignments table:
  - employee_id (identifier for the employee assigned to the project)
  - project_id (identifier for the project)

Question:
Write a query to find the names of employees who are not assigned to any project.

3. **Find the Most Recent Sale for Each Product**
   Table Description:
   - sales table:
     - product_id (identifier for the product sold)
     - sale_date (date of the sale)

   Question:
   Write a query to find the most recent sale date for each product.

4. **List Customers with Orders Above a Certain Amount**
   Table Description:
   - customers table:
     - customer_id (unique identifier for each customer)
     - customer_name (name of the customer)
   - orders table:
     - order_id (unique identifier for each order)
     - customer_id (identifier for the customer who placed the order)
     - order_total (total amount of the order)

   Question:
   Write a query to find the names of customers who have placed orders with a total amount greater than $500.

5. **Find Departments with No Managers**
   Table Description:
   - departments table:
     - department_id (unique identifier for each department)
     - department_name (name of the department)
   - employees table:
     - employee_id (unique identifier for each employee)
     - employee_name (name of the employee)
     - department_id (identifier for the department the employee belongs to)
     - role_id (identifier for the role of the employee)
   - roles table:
     - role_id (unique identifier for each role)
     - role_name (name of the role)

   Question:
   Write a query to find the names of departments that do not have a manager assigned. The manager of a department has a specific role_id in the employees table.

Solutions

1. **Find Products Priced Above Average**
   SQL Query:
   sql
   Copy code

```
SELECT product_name
FROM products
WHERE price > (SELECT AVG(price) FROM products);
```

2. **List Employees Not Assigned to Any Project**
   SQL Query:
   sql
   Copy code

```
SELECT employee_name
FROM employees
WHERE employee_id NOT IN (SELECT employee_id FROM project_assignments);
```

3. **Find the Most Recent Sale for Each Product**
   SQL Query:
   sql
   Copy code

```
SELECT product_id, MAX(sale_date) AS last_sale_date
FROM sales
GROUP BY product_id;
```

4.

List Customers with Orders Above a Certain Amount
SQL Query:
sql
Copy code

```
SELECT customer_name
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders WHERE order_total > 500);
```

5.

Find Departments with No Managers
SQL Query:
sql
Copy code

```
SELECT department_name
FROM departments
WHERE department_id NOT IN (
  SELECT department_id
  FROM employees
```

```
    WHERE role_id = (SELECT role_id FROM roles WHERE role_name = 'Manager')
);
```

1. **Find Products Priced Above Average and Recently Added**
   Table Description:
   - products table:
     - product_id (unique identifier for each product)
     - product_name (name of the product)
     - price (price of the product)
     - added_date (date when the product was added)

   Question:
   Write a query to find the names of products that were added in the last month and whose price is above the average price of all products.

2. **List Employees Not Assigned to Any Project in the Last Year**
   Table Description:
   - employees table:
     - employee_id (unique identifier for each employee)
     - employee_name (name of the employee)
   - project_assignments table:
     - employee_id (identifier for the employee assigned to the project)
     - project_id (identifier for the project)
     - assignment_date (date when the employee was assigned to the project)

   Question:
   Write a query to find the names of employees who have not been assigned to any project in the last year.

3. **Find Products with the Highest Number of Sales in the Last Quarter**
   Table Description:
   - products table:
     - product_id (identifier for the product)
     - product_name (name of the product)
   - sales table:
     - product_id (identifier for the product sold)
     - sale_date (date of the sale)
     - quantity (quantity of product sold in each sale)

   Question:
   Write a query to find the names of products that have the highest total quantity sold in the last quarter.

```
SELECT top 1 FROM

(SELECT * FROM

(SELECT SUM(quantity) as tot_quant, product_id

FROM sales

WHERE sale_date BETWEEN ‘2024-04-01’ AND ‘2024-06-30’

GROUP BY 2

)  a

ORDER BY 1 DESC) b
```

```
SELECT * FROM a WHERE tot_quant IN

(SELECT MAX(tot_quant) as max_tot_quant FROM a)
```

```
SELECT * FROM a

 JOIN

(SELECT MAX(tot_quant) as max_tot_quant FROM a) b

ON a.tot_quant=b.max_tot_quant
```

```
SELECT * FROM

(SELECT *, RANK() OVER(ORDER BY tot_quant DESC) AS Rank_NUM

FROM a) b

WHERE rank_num=1
```

4. **List Customers with No Orders in the Past Year but Previously Active**
   Table Description:
   - customers table:
     - customer_id (unique identifier for each customer)

        - customer_name (name of the customer)
    - orders table:
        - order_id (unique identifier for each order)
        - customer_id (identifier for the customer who placed the order)
        - order_date (date of the order)

    Question:
    Write a query to find the names of customers who have not placed any orders in the past year but had placed at least one order before that.

5. **Find Departments with Employees in Multiple Roles**
    Table Description:
    - departments table:
        - department_id (unique identifier for each department)
        - department_name (name of the department)
    - employees table:
        - employee_id (unique identifier for each employee)
        - employee_name (name of the employee)
        - department_id (identifier for the department the employee belongs to)
        - role_id (identifier for the role of the employee)
    - roles table:
        - role_id (unique identifier for each role)
        - role_name (name of the role)

    Question:
    Write a query to find the names of departments where at least one employee holds multiple roles.

Solutions

Find Products Priced Above Average and Recently Added
SQL Query:
sql
Copy code

```
SELECT product_name
FROM products
WHERE  price       > (SELECT AVG(price) FROM products)
                     AND added_date >= DATEADD(month, -1, GETDATE());
```

The SQL expression added_date >= DATEADD(month, -1, GETDATE()); is used to filter records where the added_date is within the last month from the current date.

Explanation:

added_date: This is the date column in your table that you want to filter on.

DATEADD(month, -1, GETDATE()):

GETDATE(): Returns the current date and time.

DATEADD(month, -1, GETDATE()): Subtracts one month from the current date. This gives the date exactly one month before the current date.

added_date >= DATEADD(month, -1, GETDATE()): This condition checks if the added_date is greater than or equal to the date one month before the current date. Essentially, it filters for records where added_date falls within the last month.

List Employees Not Assigned to Any Project in the Last Year
SQL Query:
sql

```
SELECT employee_name
FROM employees
WHERE employee_id NOT IN (
        SELECT employee_id
        FROM project_assignments
        WHERE assignment_date >= DATEADD(year, -1, GETDATE())
       );
```

Find Products with the Highest Number of Sales in the Last Quarter
SQL Query:
sql

```
SELECT product_name
FROM products
WHERE product_id IN (
        SELECT product_id
        FROM sales
        WHERE sale_date >= DATEADD(quarter, -1, GETDATE())
        GROUP BY product_id
        HAVING SUM(quantity) = (
                SELECT MAX(SUM(quantity))
                FROM sales
                WHERE sale_date >= DATEADD(quarter, -1, GETDATE())
                GROUP BY product_id
  )
);
```

List Customers with No Orders in the Past Year but Previously Active
SQL Query:
sql
Copy code

```
SELECT customer_name
```

```
FROM customers
WHERE customer_id NOT IN (
    SELECT customer_id
    FROM orders
    WHERE order_date >= DATEADD(year, -1, GETDATE())
)
AND customer_id IN (
    SELECT customer_id
    FROM orders
    WHERE order_date < DATEADD(year, -1, GETDATE())
);
```

Find Departments with Employees in Multiple Roles
SQL Query:
sql
Copy code

```
SELECT department_name
FROM departments
WHERE department_id IN (
    SELECT department_id
    FROM employees
    GROUP BY department_id, employee_id
    HAVING COUNT(DISTINCT role_id) > 1
);
```

---

1. **Subquery Calculation:**
   - The subquery calculates the difference (Diff) between the current timestamp and the previous timestamp for each machine_id and process_id combination, ordered by timestamp.
   - LAG(timestamp) is used to get the previous timestamp.
2. **Filtering 'end' Activities:**
   - The WHERE activity_type = 'end' clause ensures only the differences for 'end' activities are considered in the average calculation.
3. **Grouping by Machine ID:**
   - The GROUP BY machine_id groups the results by each machine_id to calculate the average Diff for each machine.

Example with a Sample Table:

Assume we have a table activity with the following columns:

- machine_id (ID of the machine)
- process_id (ID of the process)

- activity_type (Type of activity: 'start' or 'end')
- timestamp (Timestamp of the activity)

```
SELECT
   machine_id,
   ROUND(AVG(Diff), 3) AS processing_time
FROM
   (SELECT
      machine_id,
      process_id,
      activity_type,
      timestamp - LAG(timestamp) OVER (PARTITION BY machine_id, process_id ORDER BY
timestamp) AS Diff
    FROM
      activity
   ) AS a
WHERE
   activity_type = 'end'
GROUP BY
   machine_id;
```

Calculate Cumulative Sales Quantity

- **Table:** sales
  - **Columns:**
    - date: The date of the sale.
    - product_id: The ID of the product sold.
    - quantity: The quantity of the product sold.
- **Question:** Write a query to calculate the cumulative quantity sold for each product by date.

2. Track Running Average of Sales

- **Table:** sales
  - **Columns:**
    - date: The date of the sale.
    - revenue: The revenue generated on that date.
- **Question:** Write a query to calculate the running average revenue up to each date.

3. Calculate a Running Difference in Revenue

- **Table:** sales
  - **Columns:**
    - date: The date of the sale.
    - revenue: The revenue generated on that date.
- **Question:** Write a query to calculate the difference in revenue from the previous date to the current date.

4. Calculate Cumulative Number of Orders

- **Table:** orders
  - **Columns:**
    - date: The date the order was placed.
    - order_id: The unique ID of the order.
- **Question:** Write a query to calculate the cumulative number of orders placed up to each date.

5. Find Cumulative Daily Expenses by Category

- **Table:** expenses
  - **Columns:**
    - date: The date the expense was recorded.
    - category: The category of the expense (e.g., food, travel).
    - expense: The amount spent on that date.
- **Question:** Write a query to find the cumulative expenses for each category up to each date.

SQL Solutions and Code

1. Calculate Cumulative Sales Quantity

sql

Copy code

```
SELECT date, product_id, quantity,
  SUM(quantity) OVER (
    PARTITION BY product_id
    ORDER BY date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cumulative_quantity
FROM sales
ORDER BY product_id, date;
```

2. Track Running Average of Sales

sql

Copy code

```
SELECT date, revenue,
  AVG(revenue) OVER (
    ORDER BY date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS running_avg
FROM sales
ORDER BY date;
```

3. Calculate a Running Difference in Revenue

sql

Copy code

```
SELECT date, revenue,
    revenue - LAG(revenue) OVER (ORDER BY date) AS running_difference
FROM sales
ORDER BY date;
```

4. Calculate Cumulative Number of Orders

sql

Copy code

```
SELECT date, order_id,
    COUNT(order_id) OVER (
      ORDER BY date
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cumulative_orders
FROM orders
ORDER BY date;
```

5. Find Cumulative Daily Expenses by Category

sql

Copy code

```
SELECT date, category, expense,
    SUM(expense) OVER (
      PARTITION BY category
      ORDER BY date
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cumulative_expense
FROM expenses
ORDER BY category, date;
```

Table: Sales

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| sale_id     | int   |
| product_id  | int   |
| year        | int   |
| quantity    | int   |
| price       | int   |
+-------------+-------+
```

(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: Product

Column Name	Type
product_id	int
product_name	varchar

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  |
+------------+------------+----------+-------+
```

Table: Logs

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| num         | varchar |
+-------------+---------+
```

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

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  | 1   |
| 2  | 1   |
| 3  | 1   |
| 4  | 2   |
| 5  | 1   |
| 6  | 2   |
| 7  | 2   |
+----+-----+
```

Output:

```
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
```

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

Grouping by `order_date`:

- By grouping the results by both `product_name` and `order_date`, you're likely to get multiple rows for the same product if it was ordered on different dates. This might not be what you intended if you want a single sum of units sold per product during the specified date range.

`WHERE o.unit >= 100`:

- The `WHERE` clause filters out any orders with fewer than 100 units, meaning only orders with 100 or more units will be included in the sum. If the intention was to sum all units sold for products that have at least one order with 100 units, this condition should be adjusted.

LEFT JOIN:

- Since you're filtering on `o.unit` and `order_date`, it effectively turns the `LEFT JOIN` into an `INNER JOIN`, because rows in `products` that don't have matching rows in `orders` would be excluded by the `WHERE` clause.

```
SELECT p.product_name,
       SUM(o.unit) AS total_units_sold
FROM products p
LEFT 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 SUM(o.unit) >= 100;
```

To retrieve the top 5 clinics with the highest count of patient feedback records that have a satisfaction score of 7 or higher

You are provided with two tables: `clinics` and `feedback`.

1. Table `clinics`:
   - `clinic_id` (INT): A unique identifier for each clinic.
   - `clinic_name` (VARCHAR): The name of the clinic.
2. Table `feedback`:
   - `clinic_id` (INT): A foreign key linking the feedback to a specific clinic.
   - `satisfaction_score` (INT): A score given by patients, ranging from 1 to 10, indicating their satisfaction level with the clinic.

Write a SQL query to find the names of clinics and the count of high satisfaction scores (where the satisfaction score is 7 or above) for each clinic. The query should return the top 5 clinics based on the number of high satisfaction scores. If a clinic has no high satisfaction scores, it should still be included in the result with a count of 0.

The result should include:

- The name of the clinic (`clinic_name`).
- The count of high satisfaction scores (`high_satisfaction_count`).

Expected Output:

The query should display the clinic name and the corresponding count of high satisfaction scores, limited to the top 5 clinics.

   - The CTE named `high_satisfaction` is defined using the `WITH` clause.
   - It selects `clinic_id` and `satisfaction_score` from the `feedback` table where the `satisfaction_score` is 7 or higher. This creates a temporary result set that only contains high-satisfaction feedback records.

```
WITH high_satisfaction AS (
    SELECT
        clinic_id,
        satisfaction_score
    FROM feedback
    WHERE satisfaction_score >= 7
)
SELECT
    mh.clinic_name,
    COUNT(high_satisfaction.satisfaction_score) AS
high_satisfaction_count
FROM clinics AS mh
LEFT JOIN high_satisfaction ON mh.clinic_id =
high_satisfaction.clinic_id
GROUP BY mh.clinic_name
LIMIT 5;
```

2. **Main Query:**
    - The main query selects the clinic name and the count of high satisfaction scores for each clinic.
    - The `clinics` table is aliased as `mh`, and it is left joined with the `high_satisfaction` CTE on the `clinic_id`.
    - The `COUNT` function is used to count the number of high satisfaction feedback records for each clinic.
    - The query groups the results by `clinic_name` to aggregate the count for each clinic.
    - Finally, it limits the output to the top 5 clinics.

Key Points:

- **CTE (`high_satisfaction`)**: Filters the feedback to include only records with a satisfaction score of 7 or higher.
- **LEFT JOIN**: Ensures that all clinics are included in the result, even those that might not have any feedback with a satisfaction score of 7 or higher.
- **COUNT**: Counts the number of high satisfaction records per clinic.
- **GROUP BY**: Groups the results by clinic name to get aggregated counts.
- **LIMIT 5**: Limits the result to the top 5 clinics based on the number of high satisfaction records.

This query is useful for identifying which clinics have received the most positive feedback, which can be an important metric for performance evaluation.

You are provided with a table named `grades` which contains the grades of students for various exams.

1. **Table `grades`:**
   - `student_id` (INT): A unique identifier for each student.
   - `grade` (DECIMAL): The grade a student received in an exam.

Write a SQL query to find the highest grade achieved by each student.

1. First, create a Common Table Expression (CTE) to calculate the highest grade for each student.
2. Then, retrieve the `student_id` and their corresponding `highest_grade` from the CTE.

The query should return:

- `student_id`: The unique identifier for each student.
- `highest_grade`: The highest grade that the student has received.

Expected Output:

The query should display each student's ID along with their highest grade from the `grades` table.

- The CTE named `MaxGrades` is defined using the `WITH` clause.
- It selects `student_id` and calculates the maximum grade (`MAX(grade)`) for each student from the `grades` table.

The `GROUP BY` clause groups the results by `student_id` to ensure that the maximum grade is calculated for each individual student.

```
WITH MaxGrades AS (
    SELECT
        student_id,
        MAX(grade) AS highest_grade
    FROM grades
    GROUP BY student_id
)
SELECT
    MaxGrades.student_id,
    MaxGrades.highest_grade
FROM MaxGrades;
```

2. **Main Query:**
   - The main query simply selects the `student_id` and `highest_grade` from the `MaxGrades` CTE.
   - This returns a list of students along with their highest grade.

Key Points:

- **CTE (`MaxGrades`)**: Calculates the maximum grade for each student.
- **GROUP BY**: Ensures that the maximum grade is calculated for each student individually.
- **Main Query**: Retrieves the `student_id` and the corresponding highest grade for each student from the CTE.

This query efficiently identifies the top grade for every student in the dataset, which can be useful for academic performance analysis.

You have two tables: `hotels` and `hotel_stays`.

1. **Table `hotels`:**
   - **`hotel_id`** (INT): A unique identifier for each hotel.
   - **`hotel_name`** (VARCHAR): The name of the hotel.
2. **Table `hotel_stays`:**
   - **`hotel_id`** (INT): A foreign key linking a stay to a specific hotel.
   - **`length_of_stay`** (INT): The number of days a guest stayed at the hotel.
   - **`year`** (INT): The year when the stay occurred.

Write a SQL query to calculate the average length of stay for each hotel in the year 2022. Then, rank the hotels based on their average stay length, with the longest average stay receiving the highest rank.

WITH A AS (SELECT hotel_id, AVG(length_of_stay) AS avg_stay
FROM hotel_stays
WHERE year=2022
GROUP BY hotel_id)

WITH B AS (SELECT a.*, h.hotel_name, ROW_NUMBER() OVER (ORDER BY avg_stay DESC)
AS hotel_rank

FROM A
LEFT JOIN hotels h

ON a.hotel_id=h.hotel_id

ORDER BY hotel_rank)

The query should return:

- `hotel`: The name of the hotel.
- `avg_stay_length`: The average number of days guests stayed at the hotel in 2022.
- `hotel_rank`: The rank of the hotel based on the average stay length, with 1 being the highest.

The results should be ordered by the rank, from highest to lowest average stay length.

Expected Output:

The query should display each hotel's name, their average stay length in 2022, and their rank based on this average stay length. The hotels should be ordered by their rank.

1. **Select Clause**:
   - `h.hotel_name AS hotel`: Selects the hotel name.
   - `AVG(hs.length_of_stay) AS avg_stay_length`: Calculates the average length of stay for each hotel.
   - `ROW_NUMBER() OVER(ORDER BY AVG(hs.length_of_stay) DESC) AS hotel_rank`: Assigns a rank to each hotel based on the average stay length in descending order (the longer the stay, the higher the rank).
2. **From Clause**:
   - `FROM hotels AS h`: Selects data from the `hotels` table.
   - `LEFT JOIN hotel_stays AS hs ON h.hotel_id = hs.hotel_id`: Joins the `hotels` table with the `hotel_stays` table on the `hotel_id` field, ensuring that all hotels are included, even if they have no stays.
3. **Where Clause**:
   - `WHERE hs.year = 2022`: Filters the data to include only stays from the year 2022.
4. **Group By Clause**:
   - `GROUP BY h.hotel_name`: Groups the results by hotel name to calculate the average stay length for each hotel.
5. **Order By Clause**:
   - `ORDER BY hotel_rank`: Orders the final results by the hotel rank.

Final Query:

sql

Copy code

```
SELECT
    h.hotel_name AS hotel,
    AVG(hs.length_of_stay) AS avg_stay_length,
    ROW_NUMBER() OVER(ORDER BY AVG(hs.length_of_stay) DESC) AS
hotel_rank
FROM hotels AS h
LEFT JOIN hotel_stays AS hs
```

```
ON h.hotel_id = hs.hotel_id
WHERE hs.year = 2022
GROUP BY h.hotel_name
ORDER BY hotel_rank;
```

This query provides a ranking of hotels based on the average length of stay in 2022, where hotels with longer average stays receive a higher rank.

CTE:

sql
Copy code

```
WITH AverageStay AS (
    SELECT
        h.hotel_name AS hotel,
        AVG(hs.length_of_stay) AS avg_stay_length
    FROM hotels AS h
    LEFT JOIN hotel_stays AS hs
    ON h.hotel_id = hs.hotel_id
    WHERE hs.year = 2022
    GROUP BY h.hotel_name
),
RankedHotels AS (
    SELECT
        hotel,
        avg_stay_length,
        ROW_NUMBER() OVER (ORDER BY avg_stay_length DESC)   AS
hotel_rank
    FROM AverageStay
)
SELECT
    hotel,
    avg_stay_length,
    hotel_rank
FROM RankedHotels
ORDER BY hotel_rank;
```

Explanation:

1. **`AverageStay` CTE:** Calculates the average length of stay for each hotel.
2. **`RankedHotels` CTE:** Uses the `ROW_NUMBER()` window function to assign a rank to each hotel based on the average stay length.
3. **Final `SELECT`:** Retrieves the hotel name, average stay length, and rank, and orders the results by rank.

This way, the CTEs help to structure your query in a more readable and modular fashion.

sql

Copy code

```
WITH A AS ( SELECT hotel_id, AVG(length_of_stay) AS avg_stay

FROM hotel_stays

WHERE year = 2022

GROUP BY hotel_id ),

B AS ( SELECT hotel_id, avg_stay,

RANK() OVER (ORDER BY avg_stay DESC) AS hotel_rank

FROM A )

SELECT h.hotel_name, b.avg_stay, b.hotel_rank

FROM B b

JOIN hotels h

ON h.hotel_id = b.hotel_id;
```

You are given a table named `payments` that records transactions made by customers using their credit cards at various merchants. The table structure is as follows:

1. **Table `payments`:**
   - **`payment_id`** (INT): A unique identifier for each payment.
   - **`credit_card_number`** (VARCHAR): The credit card number used for the payment.
   - **`merchant_id`** (INT): A unique identifier for the merchant where the payment was made.
   - **`amount`** (DECIMAL): The amount paid.
   - **`payment_time`** (DATETIME): The date and time when the payment was made.

Write a SQL query to identify cases where a payment was made at the same merchant using the same credit card for the same amount within 10 minutes of another payment. For each group of such payments, report the count of repeated payments.

The query should return:

- **credit_card_number**: The credit card number used in the repeated payments.
- **merchant_id**: The merchant ID where the repeated payments were made.
- **amount**: The amount paid in these repeated transactions.
- **repeated_payment_count**: The number of repeated payments that meet the criteria.

```sql
sql
Copy code
WITH a AS (
    SELECT
        *,
        LAG(amount) OVER (PARTITION BY credit_card_number,
merchant_id ORDER BY payment_time) AS prev_amt,
        LAG(payment_time) OVER (PARTITION BY credit_card_number,
merchant_id ORDER BY payment_time) AS prev_datetime
    FROM
        payments
)
SELECT
    *,
    amount - prev_amt AS amt_diff,
    ABS(TIMESTAMPDIFF(MINUTE, payment_time, prev_datetime)) AS
time_diff
FROM
    a
WHERE
    amount - prev_amt = 0
    AND ABS(TIMESTAMPDIFF(MINUTE, payment_time, prev_datetime)) <=
10;
```

Explanation:

1. **CTE a**:
   - Uses the LAG() function to get the previous amount (prev_amt) and payment_time (prev_datetime) for each combination of credit_card_number and merchant_id, ordered by payment_time.

    ◦ This allows you to compare the current transaction with the previous one for the same credit card and merchant.
2. **Main Query**:
    ◦ Calculates amt_diff as the difference between the current amount and the previous amount.
    ◦ Calculates time_diff as the absolute difference in minutes between the current payment_time and the previous payment_time.
    ◦ Filters the results where amt_diff is 0 (indicating identical transaction amounts) and time_diff is <= 10 minutes (indicating the transactions happened close to each other).

Result:

This query will return all rows where two consecutive transactions for the same credit card and merchant have the same amount and occurred within 10 minutes of each other. This could help in detecting potential duplicate transactions or suspicious activity.

Expected Approach:

1. **Window Function:** Use a window function to compare each payment with previous payments made using the same credit card at the same merchant for the same amount.
2. **Time Difference:** Calculate the time difference between the current payment and previous payments to identify those made within 10 minutes.
3. **Group & Filter:** Group the results by `credit_card_number`, `merchant_id`, and `amount`, and filter to count only those groups with repeated payments.

SQL Solution Example:

sql
Copy code

```
WITH payment_duplicates AS (
    SELECT
        p1.credit_card_number,
        p1.merchant_id,
        p1.amount,
        COUNT(*) AS repeated_payment_count
    FROM
        payments p1
    JOIN
```

```
        payments p2
    ON
        p1.credit_card_number = p2.credit_card_number
        AND p1.merchant_id = p2.merchant_id
        AND p1.amount = p2.amount
        AND p1.payment_id != p2.payment_id
        AND ABS(TIMESTAMPDIFF(MINUTE, p1.payment_time,
p2.payment_time)) <= 10
    GROUP BY
        p1.credit_card_number,
        p1.merchant_id,
        p1.amount
    HAVING
        repeated_payment_count > 1
)
SELECT
    credit_card_number,
    merchant_id,
    amount,
    repeated_payment_count
FROM
    payment_duplicates;
```

Explanation:

- **Self-Join:** The table `payments` is joined with itself to compare each payment (`p1`) against every other payment (`p2`).
- **Conditions in JOIN:**
  - The credit card number, merchant ID, and amount must be the same.
  - The time difference between the two payments should be 10 minutes or less (`TIMESTAMPDIFF(MINUTE, p1.payment_time, p2.payment_time) <= 10`).
  - The payments should not be identical (`p1.payment_id != p2.payment_id`).
- **Grouping:** The results are grouped by `credit_card_number`, `merchant_id`, and `amount`.
- **Filtering:** The `HAVING` clause ensures that only those groups with more than one payment are counted, indicating repeated payments.

LEET CODE QUESTIONS
https://leetcode.com/problems/group-sold-products-by-the-date/description/?envType=study-plan-v2&envId=top-sql-50

```
SELECT sell_date,
       COUNT(DISTINCT product) AS num_sold,
       GROUP_CONCAT(DISTINCT product ORDER BY product SEPARATOR ', ') AS
products
FROM Activities
GROUP BY sell_date
ORDER BY sell_date;
```

https://leetcode.com/problems/product-price-at-a-given-date/description/?envType=study-plan-v2&envId=top-sql-50

```
SELECT
    p1.product_id,
    COALESCE(MAX(p2.new_price), 10) AS price
FROM
    (SELECT DISTINCT product_id FROM Products) p1
LEFT JOIN
    Products p2
ON
    p1.product_id = p2.product_id
    AND p2.change_date <= '2019-08-16'
GROUP BY
    p1.product_id;
```

This query is designed to find the most recent price of each product up to a specific date (`2019-08-16`). If there is no price change recorded before or on that date, the query defaults the price to `10` using the `COALESCE` function. Here's a breakdown of the query:

1. Subquery `a`:

sql
Copy code

```
(SELECT DISTINCT product_id FROM products) a
```

- This subquery extracts a list of all distinct `product_id`s from the `products` table.
- It ensures that each product appears only once in the result, regardless of how many price changes it has.

2. Subquery `b`:

sql
Copy code
```
(SELECT
     *,
     ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY change_date
DESC) AS rnm1
 FROM
     Products
 WHERE change_date <= '2019-08-16') b
```

- This subquery retrieves all product price changes (`new_price`) up to the date `2019-08-16`.
- The `ROW_NUMBER()` function creates a unique number (`rnm1`) for each price change per `product_id`, ordered by `change_date` in descending order (most recent price change first).
- The partitioning by `product_id` ensures that the row numbers reset for each product, meaning we get a rank of price changes per product based on date.

3. Join Condition:

sql
Copy code
```
LEFT JOIN
ON a.product_id=b.product_id
AND rnm1=1
```

- The query performs a `LEFT JOIN` between subquery `a` (all distinct products) and subquery `b` (price changes up to the target date).
- The condition `rnm1=1` ensures that only the most recent price change per product (the row with the highest row number) is joined.

4. COALESCE Function:

sql
Copy code
```
COALESCE(new_price, 10) AS price
```

- `COALESCE` is used to handle cases where a product has no price change before `2019-08-16`. In such cases, the `LEFT JOIN` results in `NULL` for `new_price`, and `COALESCE` assigns a default price of `10`.
- So, if there is no price found, the product is assumed to have a default price of 10.

5. Final Output:

sql
Copy code

```
SELECT a.product_id, COALESCE(new_price, 10) as price
```

- This selects each `product_id` and its corresponding price.
- If the product had a price change before or on `2019-08-16`, the latest price is used.
- If not, the default price of `10` is returned.

Summary:

This query retrieves the most recent price for each product as of `2019-08-16`. If no price change exists for a product before this date, it defaults the price to `10`.

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.

```
SELECT
    employee_id,
    department_id
FROM
    employee
WHERE
    primary_flag = 'Y'

UNION ALL

SELECT
    e.employee_id,
    e.department_id
FROM
    employee e
LEFT JOIN
    employee e2
ON
    e.employee_id = e2.employee_id
    AND e.department_id <> e2.department_id
```

```
WHERE
    e2.employee_id IS NULL
    AND e.primary_flag = 'N'
ORDER BY employee_id;
```

Breakdown of the Query:

1. **Main Table (`employee e`):**
   - The query starts with the `employee` table aliased as `e`. This is the primary dataset from which you are selecting records.
2. **LEFT JOIN Operation:**
   - The `LEFT JOIN` is performed between the table `e` (aliased as `e`) and itself (aliased as `e2`). This join is designed to find other departments for the same employee.
   - The join condition `e.employee_id = e2.employee_id` ensures that the comparison is made within the same employee.
   - The additional condition `e.department_id <> e2.department_id` ensures that only records where the department is different are considered. Essentially, this helps in finding if the employee belongs to more than one department.
3. **Filtering with `WHERE e2.employee_id IS NULL`:**
   - After the `LEFT JOIN`, if an employee is associated with only one department, there will be no corresponding record in `e2` with a different department for that employee. Thus, `e2.employee_id` will be `NULL` for such records.
   - The `WHERE e2.employee_id IS NULL` condition filters out employees who have only one department. Employees with multiple departments would have non-null values in `e2.employee_id` for at least one of the joined rows.
4. **Additional Condition `AND e.primary_flag = 'N'`:**
   - This condition ensures that only employees with `primary_flag` set to 'N' are included in the results. According to your scenario, this flag 'N' indicates employees with only one department.

Summary:

- The query finds employees who have exactly one department (`e2.employee_id IS NULL`), and whose `primary_flag` is 'N'.
- It will not include employees who have multiple departments, even if their `primary_flag` is 'N', because `e2.employee_id` would not be `NULL` for those employees.

This approach helps you identify employees who exclusively belong to a single department, and their primary department is recorded as such.

sql
Copy code
```
SELECT
```

```
    e.employee_id,
    e.department_id AS primary_department
FROM
    employee e
LEFT JOIN
    employee e2
ON
    e.employee_id = e2.employee_id
    AND e.department_id <> e2.department_id
WHERE
    e2.employee_id IS NULL
    AND e.primary_flag = 'N';
```

Breakdown of the Query:

1. **Main Table (`employee e`):**
   - The query starts with the `employee` table aliased as `e`. This is the primary dataset from which you are selecting records.
2. **LEFT JOIN Operation:**
   - The `LEFT JOIN` is performed between the table `e` (aliased as `e`) and itself (aliased as `e2`). This join is designed to find other departments for the same employee.
   - The join condition `e.employee_id = e2.employee_id` ensures that the comparison is made within the same employee.
   - The additional condition `e.department_id <> e2.department_id` ensures that only records where the department is different are considered. Essentially, this helps in finding if the employee belongs to more than one department.
3. **Filtering with `WHERE e2.employee_id IS NULL`:**
   - After the `LEFT JOIN`, if an employee is associated with only one department, there will be no corresponding record in `e2` with a different department for that employee. Thus, `e2.employee_id` will be `NULL` for such records.
   - The `WHERE e2.employee_id IS NULL` condition filters out employees who have only one department. Employees with multiple departments would have non-null values in `e2.employee_id` for at least one of the joined rows.
4. **Additional Condition `AND e.primary_flag = 'N'`:**
   - This condition ensures that only employees with `primary_flag` set to 'N' are included in the results. According to your scenario, this flag 'N' indicates employees with only one department.

Summary:

- The query finds employees who have exactly one department (`e2.employee_id IS NULL`), and whose `primary_flag` is 'N'.

- It will not include employees who have multiple departments, even if their `primary_flag` is 'N', because `e2.employee_id` would not be `NULL` for those employees.

This approach helps you identify employees who exclusively belong to a single department, and their primary department is recorded as such.

```
# Write your MySQL query statement below
SELECT
    visited_on,
    (
        SELECT SUM(amount)
        FROM customer
        WHERE visited_on BETWEEN DATE_SUB(c.visited_on, INTERVAL 6 DAY) AND
c.visited_on
    ) AS amount,
    ROUND(
        (
            SELECT SUM(amount) / 7
            FROM customer
            WHERE visited_on BETWEEN DATE_SUB(c.visited_on, INTERVAL 6 DAY)
AND c.visited_on
        ),
        2
    ) AS average_amount
FROM customer c
WHERE visited_on >= (
        SELECT DATE_ADD(MIN(visited_on), INTERVAL 6 DAY)
        FROM customer
    )
GROUP BY visited_on;
```

Explanation

1. **`SELECT visited_on`**: This specifies that you want to retrieve the `visited_on` date for each row in the output.
2. **`amount`**:

The subquery:
sql
Copy code

```
(
    SELECT SUM(amount)
    FROM customer
    WHERE visited_on BETWEEN DATE_SUB(c.visited_on, INTERVAL 6 DAY)
AND c.visited_on
)
```

-
    - This subquery calculates the total amount paid in the 7-day window ending on `c.visited_on`.
    - `DATE_SUB(c.visited_on, INTERVAL 6 DAY)` calculates the start of the 7-day window.
    - `BETWEEN DATE_SUB(c.visited_on, INTERVAL 6 DAY) AND c.visited_on` specifies the date range for the window.
    - `SUM(amount)` aggregates the total amount paid within this 7-day period.

3. **`average_amount`**:

The subquery:
sql
Copy code

```
ROUND(
    (
        SELECT SUM(amount) / 7
        FROM customer
        WHERE visited_on BETWEEN DATE_SUB(c.visited_on, INTERVAL 6
DAY) AND c.visited_on
    ),
    2
)
```

-
    - This calculates the average amount over the same 7-day window.
    - `SUM(amount) / 7` computes the average of the total amount paid in the window.
    - `ROUND(..., 2)` rounds this average to two decimal places.

4. **`FROM customer c`**: Specifies that the main table used in the query is `customer`, aliased as `c`.
5. **`WHERE visited_on >= (SELECT DATE_ADD(MIN(visited_on), INTERVAL 6 DAY) FROM customer)`**:
    - This condition ensures that only dates with at least a full 7-day window of previous data are included.

- `DATE_ADD(MIN(visited_on), INTERVAL 6 DAY)` calculates the earliest date that has enough preceding days for a 7-day window.

6. **`GROUP BY visited_on`**: Groups the results by the `visited_on` date. This ensures that you get a single row for each date with the aggregated `amount` and the computed `average_amount`.

Summary

This query calculates two things for each date (`visited_on`):

- **`amount`**: The total amount paid over the 7-day window ending on that date.
- **`average_amount`**: The average amount paid per day over the same 7-day window, rounded to two decimal places.

The `WHERE` clause ensures that only dates with a complete 7-day window are considered, and `GROUP BY` groups the results by date to provide the correct aggregated values.

Detailed Explanation

sql
Copy code

```sql
WHERE visited_on >= (
    SELECT DATE_ADD(MIN(visited_on), INTERVAL 6 DAY)
    FROM customer
)
```

Purpose

This part of the query filters out dates to ensure that only those with a full 7-day window of data are included. Here's how it works:

Breakdown

1. **`SELECT DATE_ADD(MIN(visited_on), INTERVAL 6 DAY) FROM customer`**:
    - **`MIN(visited_on)`**:
        - Finds the earliest (`minimum`) date in the `customer` table.
    - **`DATE_ADD(MIN(visited_on), INTERVAL 6 DAY)`**:
        - Adds 6 days to the earliest date.
        - This calculation gives the earliest possible date where a 7-day window could be complete, including the day itself.
2. **`WHERE visited_on >= (...)`**:
    - **`visited_on`**:
        - This is the date of each row in the `customer` table.
    - **`>= (...)`**:

- Ensures that only dates that are on or after the computed earliest possible date for a full 7-day window are included.
- This effectively filters out any dates before this computed start date because those dates do not have enough preceding data to form a complete 7-day window.

Example

Assume you have the following data in the `customer` table:

visited_on
2019-01-01
2019-01-02
2019-01-03
2019-01-04
2019-01-05
2019-01-06
2019-01-07
2019-01-08
2019-01-09
2019-01-10

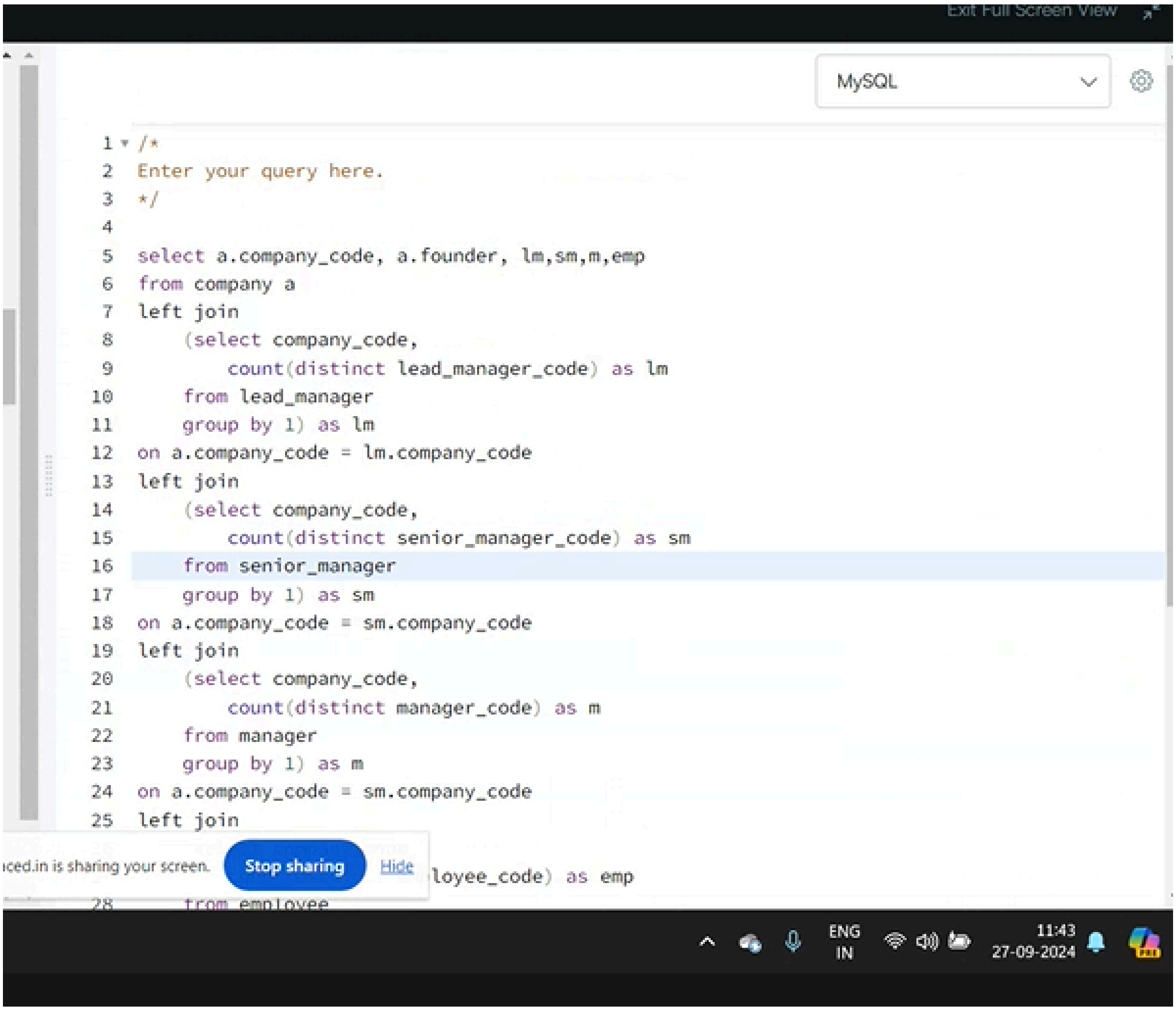
- **`MIN(visited_on)`**:
  - This would return `2019-01-01`.
- **`DATE_ADD(MIN(visited_on), INTERVAL 6 DAY)`**:
  - Adding 6 days to `2019-01-01` results in `2019-01-07`.
- **`WHERE visited_on >= '2019-01-07'`**:
  - This condition filters the table to only include dates from `2019-01-07` onward.

Why This Is Important

1. **Ensures Complete Data**:
   - By filtering out dates before `2019-01-07`, the query ensures that each selected date has a full 7 days of preceding data to calculate the moving average.
2. **Avoids Incomplete Windows**:
   - Dates before the computed start date (`2019-01-07` in this case) cannot have a complete 7-day window of data, so including them would result in inaccurate or incomplete averages.

Summary

This part of the query is crucial for ensuring that the moving average calculation only includes dates where there is enough historical data (7 days) to compute a valid average. It effectively ensures data completeness for accurate analysis.



Finding Duplicate Rows with Complex Conditions

You have a table with millions of records, and some of them are duplicate entries based on certain columns. How would you find all the duplicate rows but keep only the most recent entry based on a `timestamp` column?

Key Concepts: `GROUP BY`, `HAVING`, `ROW_NUMBER()`, `RANK()`, `DISTINCT`

---

2. Recursive Query for Hierarchical Data

You are given a table representing an employee hierarchy with `employee_id` and `manager_id`. Write a query to find all employees reporting directly or indirectly to a specific manager.

Key Concepts: Recursive CTE, `WITH RECURSIVE`, hierarchical queries, self-joins

---

3. Running Total with Date Gaps

You have a table with `transaction_date` and `transaction_amount`. Write a query to calculate the running total of transaction amounts **per day**, even if some days are missing (fill in the missing dates with the previous day's running total).

Key Concepts: `WINDOW` functions, `LEAD()`, `LAG()`, date handling, `COALESCE()`

---

4. Gaps and Islands Problem

You have a table of logins with `user_id` and `login_time`. Write a query to identify **"islands"** of consecutive logins (i.e., where the difference between two consecutive login times is no more than 1 day) and "gaps" (where the difference is greater than 1 day). Return the start and end time of each island.

Key Concepts: `LAG()`, `LEAD()`, date arithmetic, window functions

---

5. Moving Averages

You are given a table of stock prices with columns `stock_id`, `date`, and `price`. Write a query to calculate the **7-day moving average** of the stock prices for each stock.

Key Concepts: `WINDOW` functions, `ROWS BETWEEN`, aggregate functions

---

6. Top N Per Category

You have a table `sales` with `product_id`, `salesperson_id`, and `sales_amount`. Write a query to find the **top 3 salespersons** by sales amount for each product.

Key Concepts: `ROW_NUMBER()`, `DENSE_RANK()`, partitions, sorting, `LIMIT`

---

7. Finding Active Users

You have a table of user activities with `user_id` and `activity_timestamp`. Write a query to find users who were active for **at least 15 days** in the last 30 days.

Key Concepts: Date functions, `COUNT()`, `DISTINCT`, window functions

---

8. Detecting Overlapping Ranges

You have a table of bookings with `booking_id`, `start_time`, and `end_time`. Write a query to find all pairs of bookings where the time ranges overlap.

Key Concepts: Self-joins, range conditions, `INTERSECT`, `EXISTS`

---

9. Percentage Distribution Across Categories

You have a table with product sales data, including `category_id`, `product_id`, and `sale_amount`. Write a query to calculate the **percentage** of total sales for each category and each product within the category.

Key Concepts: Aggregation, `SUM()`, `GROUP BY`, `WINDOW` functions

---

10. Detecting Data Skew

You have a table with user ratings for different products (`user_id`, `product_id`, `rating`). Write a query to find the products with the **most skewed** user ratings, where the standard deviation of ratings is the highest.

Key Concepts: Statistical functions, aggregation, variance, standard deviation (`STDDEV()`), `GROUP BY`

---

11. Finding Missing Data

Given a table of `employees` and a table of `salaries`, write a query to find employees who are **missing salary entries** for any of the past 12 months.

Key Concepts: Date generation, `LEFT JOIN`, `GROUP BY`, filtering null values

---

12. Cohort Analysis

You have user registration data (`user_id`, `registration_date`) and purchase data (`user_id`, `purchase_date`). Write a query to perform **cohort analysis**, where you group users by the month they registered and then track how many users made their first purchase in the following months.

Key Concepts: Time-based grouping, `JOIN`, window functions, cohort grouping

---

13. Sales Forecasting

You have a table of daily sales data (`sale_date`, `product_id`, `sales_amount`). Write a query to forecast the next day's sales based on the **average daily sales for the past 7 days**.

Key Concepts: Window functions, moving averages, time series, date manipulation

---

14. Event Sequences

Given a table of user events (`user_id`, `event_type`, `event_time`), write a query to find users who performed the sequence of events: `login -> add_to_cart -> purchase`, in that specific order, within 1 hour.

Key Concepts: `LAG()`, `LEAD()`, window functions, event sequences, `PARTITION BY`

---

15. Data Anomalies

You have a table of sales transactions with columns `transaction_id`, `store_id`, `transaction_amount`. Write a query to identify **anomalous transactions** where the `transaction_amount` is **significantly higher or lower** than the average for that store.

Key Concepts: `AVG()`, `STDDEV()`, z-scores, statistical detection, window functions

---

16. Nth Highest Salary

Given a table of employees (`employee_id`, `salary`), write a query to find the **Nth highest salary** without using `TOP`, `LIMIT`, or subqueries.

Key Concepts: Window functions, ranking functions like `DENSE_RANK()`, `RANK()`

---

17. Pivoting Data

You have a table of sales data (`product_id`, `sale_date`, `amount`). Write a query to **pivot** the sales data, so that each row represents a `product_id`, and each column represents the total sales for each month.

Key Concepts: Conditional aggregation, `CASE WHEN`, `GROUP BY`, `COALESCE()`

---

18. Detecting Unused Items

You have a table of items (`item_id`, `created_at`) and a table of purchases (`purchase_id`, `item_id`, `purchase_date`). Write a query to find all items that have **never been purchased**.

Key Concepts: `LEFT JOIN`, `IS NULL`, anti-joins

---

19. Efficiently Filtering Recent Data

You are working with a large dataset that tracks daily user activity. How would you efficiently filter and retrieve the most recent 7 days' worth of activity, ensuring the query scales well for millions of rows?

Key Concepts: Indexing, date filtering, `WHERE` clause optimization, `LIMIT`

---

20. Normalization and Denormalization

You have a table with duplicate and redundant data across multiple columns. Explain how you would normalize the data into separate tables to reduce redundancy, and how you might **denormalize** the data for fast reporting purposes.

Key Concepts: Database design, normalization (1NF, 2NF, 3NF), denormalization, performance tuning

Finding Duplicate Rows with Complex Conditions

sql
Copy code
```
WITH RankedDuplicates AS (
  SELECT *,
         ROW_NUMBER() OVER (PARTITION BY column1, column2, column3
ORDER BY timestamp_column DESC) AS rn
  FROM table_name
)
```

```sql
SELECT *
FROM RankedDuplicates
WHERE rn > 1;
```

This query finds all duplicate rows based on `column1`, `column2`, and `column3` and keeps the most recent one based on the `timestamp_column`.

---

2. Recursive Query for Hierarchical Data

sql
Copy code

```sql
WITH RECURSIVE EmployeeHierarchy AS (
  SELECT employee_id, manager_id
  FROM employees
  WHERE manager_id = <specific_manager_id> -- Start with the
specific manager

  UNION ALL

  SELECT e.employee_id, e.manager_id
  FROM employees e
  INNER JOIN EmployeeHierarchy eh ON e.manager_id = eh.employee_id
)
SELECT *
FROM EmployeeHierarchy;
```

This query recursively finds all employees reporting directly or indirectly to the specific manager.

---

3. Running Total with Date Gaps

sql
Copy code

```sql
WITH AllDates AS (
  SELECT generate_series(
    (SELECT MIN(transaction_date) FROM transactions),
    (SELECT MAX(transaction_date) FROM transactions),
    '1 day'::interval
  )::date AS transaction_date
)
SELECT d.transaction_date,
```

```
       COALESCE(SUM(t.transaction_amount) OVER (ORDER BY
d.transaction_date), 0) AS running_total
FROM AllDates d
LEFT JOIN transactions t ON d.transaction_date = t.transaction_date
ORDER BY d.transaction_date;
```

This query fills in missing dates and calculates a running total of transactions per day, even when days have no transactions.

---

4. Gaps and Islands Problem

sql
Copy code

```
WITH NumberedLogins AS (
  SELECT user_id, login_time,
         ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY
login_time) -
         ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY login_time
+ INTERVAL '1 day') AS island_group
  FROM logins
)
SELECT user_id, MIN(login_time) AS island_start, MAX(login_time) AS
island_end
FROM NumberedLogins
GROUP BY user_id, island_group;
```

This query identifies islands of consecutive logins where the difference between consecutive logins is no more than 1 day.

---

5. Moving Averages

sql
Copy code

```
SELECT stock_id, date, price,
       AVG(price) OVER (
         PARTITION BY stock_id
         ORDER BY date
         ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
       ) AS moving_avg
FROM stock_prices;
```

This calculates the 7-day moving average of stock prices for each stock.

---

6. Top N Per Category

sql
Copy code
```
WITH RankedSales AS (
  SELECT product_id, salesperson_id, sales_amount,
         ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY
sales_amount DESC) AS rn
  FROM sales
)
SELECT product_id, salesperson_id, sales_amount
FROM RankedSales
WHERE rn <= 3;
```

This query returns the top 3 salespersons by sales amount for each product.

---

7. Finding Active Users

sql
Copy code
```
SELECT user_id
FROM user_activities
WHERE activity_timestamp >= CURRENT_DATE - INTERVAL '30 days'
GROUP BY user_id
HAVING COUNT(DISTINCT DATE(activity_timestamp)) >= 15;
```

This query finds users who were active for at least 15 distinct days in the last 30 days.

---

8. Detecting Overlapping Ranges

sql
Copy code
```
SELECT b1.booking_id, b2.booking_id
FROM bookings b1
JOIN bookings b2
  ON b1.booking_id <> b2.booking_id
```

```
  AND b1.start_time < b2.end_time
  AND b1.end_time > b2.start_time;
```

This query finds pairs of bookings where the time ranges overlap.

---

9. Percentage Distribution Across Categories

sql
Copy code

```
WITH CategoryTotals AS (
  SELECT category_id, SUM(sale_amount) AS total_sales
  FROM product_sales
  GROUP BY category_id
)
SELECT p.category_id, p.product_id, p.sale_amount,
       p.sale_amount / ct.total_sales * 100 AS
percentage_of_category
FROM product_sales p
JOIN CategoryTotals ct ON p.category_id = ct.category_id;
```

This query calculates the percentage of total sales for each product within its category.

---

10. Detecting Data Skew

sql
Copy code

```
SELECT product_id, STDDEV(rating) AS rating_stddev
FROM user_ratings
GROUP BY product_id
ORDER BY rating_stddev DESC
LIMIT 1;
```

This query identifies the product with the most skewed user ratings by finding the one with the highest standard deviation of ratings.

---

11. Finding Missing Data

sql
Copy code

```
WITH AllMonths AS (
  SELECT generate_series(
    date_trunc('month', CURRENT_DATE) - INTERVAL '11 months',
    date_trunc('month', CURRENT_DATE),
    '1 month'::interval
  ) AS month
)
SELECT e.employee_id, am.month
FROM employees e
CROSS JOIN AllMonths am
LEFT JOIN salaries s ON e.employee_id = s.employee_id AND
date_trunc('month', s.salary_date) = am.month
WHERE s.salary_id IS NULL;
```

This query finds employees missing salary entries for any of the past 12 months.

---

12. Cohort Analysis

sql
Copy code
```
WITH Cohorts AS (
  SELECT user_id, date_trunc('month', registration_date) AS
cohort_month
  FROM user_registrations
)
SELECT cohort_month, date_trunc('month', p.purchase_date) AS
purchase_month, COUNT(DISTINCT p.user_id) AS user_count
FROM Cohorts c
JOIN purchases p ON c.user_id = p.user_id
GROUP BY cohort_month, purchase_month
ORDER BY cohort_month, purchase_month;
```

This query performs cohort analysis by grouping users by registration month and tracking their purchases in subsequent months.

---

13. Sales Forecasting

sql
Copy code
```
SELECT sale_date, product_id,
```

```
        AVG(sales_amount) OVER (
          PARTITION BY product_id
          ORDER BY sale_date
          ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
        ) AS forecasted_sales
FROM sales;
```

This query forecasts the next day's sales based on the average daily sales for the past 7 days.

---

14. Event Sequences

sql
Copy code

```
WITH SequencedEvents AS (
  SELECT user_id, event_type, event_time,
         LAG(event_type) OVER (PARTITION BY user_id ORDER BY
event_time) AS prev_event,
         LAG(event_time) OVER (PARTITION BY user_id ORDER BY
event_time) AS prev_time
  FROM user_events
)
SELECT user_id
FROM SequencedEvents
WHERE event_type = 'purchase'
  AND prev_event = 'add_to_cart'
  AND event_time - prev_time <= INTERVAL '1 hour'
  AND LAG(prev_event) OVER (PARTITION BY user_id ORDER BY
event_time) = 'login';
```

This query finds users who performed the sequence of events: `login -> add_to_cart -> purchase`, in that specific order within 1 hour.

---

15. Data Anomalies

sql
Copy code

```
WITH StoreStats AS (
  SELECT store_id, AVG(transaction_amount) AS avg_amount,
STDDEV(transaction_amount) AS stddev_amount
  FROM transactions
```

```
  GROUP BY store_id
)
SELECT t.transaction_id, t.store_id, t.transaction_amount
FROM transactions t
JOIN StoreStats ss ON t.store_id = ss.store_id
WHERE ABS(t.transaction_amount - ss.avg_amount) > 2 *
ss.stddev_amount;
```

This query identifies transactions where the amount is significantly higher or lower than the store average (outside 2 standard deviations).

---

16. Nth Highest Salary

sql
Copy code

```
SELECT salary
FROM (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rank
  FROM employees
) ranked_salaries
WHERE rank = <N>;
```

This query finds the Nth highest salary using `DENSE_RANK()` without `LIMIT` or subqueries.

---

17. Pivoting Data

sql
Copy code

```
SELECT product_id,
       SUM(CASE WHEN EXTRACT(MONTH FROM sale_date) = 1 THEN amount
ELSE 0 END) AS jan_sales,
       SUM(CASE WHEN EXTRACT(MONTH FROM sale_date) = 2 THEN amount
ELSE 0 END) AS feb_sales,
       ...
       SUM(CASE WHEN EXTRACT(MONTH FROM sale_date) = 12 THEN amount
ELSE 0 END) AS dec_sales
FROM sales
GROUP BY product_id;
```

This query pivots the sales data so that each month has its own column.

---

18. Detecting Unused Items

sql
Copy code
```
SELECT i.item_id
FROM items i
LEFT JOIN purchases p ON i.item_id = p.item_id
WHERE p.item_id IS NULL;
```

This query finds all items that have never been purchased using a `LEFT JOIN`.

---

19. Efficiently Filtering Recent Data

sql
Copy code
```
SELECT *
FROM user_activity
WHERE activity_date >= CURRENT_DATE - INTERVAL '7 days'
ORDER BY activity_date DESC;
```

This query efficiently retrieves the most recent 7 days' worth of user activity, assuming proper indexing on `activity_date`.

Normalization:

Normalization is the process of structuring a relational database to minimize redundancy and improve data integrity. It typically involves dividing large tables into smaller ones and establishing relationships between them to reduce duplication. Below is an example of a table in **2nd Normal Form** (2NF):

Example:

Suppose you have a table like this:

plaintext
Copy code
```
| OrderID | CustomerID | CustomerName | ProductID | ProductName |
Quantity |
```

In 2NF, this can be broken into two normalized tables:

1. **Customers Table**:

plaintext
Copy code
```
| CustomerID | CustomerName |
```

2. **Orders Table**:

plaintext
Copy code
```
| OrderID | CustomerID | ProductID | Quantity |
```

3. **Products Table**:

plaintext
Copy code
```
| ProductID | ProductName |
```

This eliminates redundancy by separating customer information from orders and product details.

Denormalization:

Denormalization is the process of combining tables to improve read performance by reducing the number of joins, though at the cost of some redundancy. It is often done in OLAP systems for fast query retrieval.

Example:

To improve query speed, you might denormalize the above schema into a single table:

plaintext
Copy code
```
| OrderID | CustomerID | CustomerName | ProductID | ProductName |
Quantity |
```

This denormalized structure is optimized for querying but increases redundancy since customer and product information is repeated for every order.

In practical terms:

- **Normalization** is used for write-heavy systems where data integrity and avoiding redundancy are priorities.
- **Denormalization** is preferred in read-heavy systems where fast querying is more critical than storage efficiency.

Select depts with no managers:

```
SELECT dept_name
FROM depts
WHERE dept_id NOT IN (
    SELECT dept_id
    FROM employees
    WHERE role_id = (
        SELECT role_id
        FROM roles
        WHERE role_name =
            'Manager')
);
```

ain query:
- Retrieves dept names from 'depts' where dept_id is not in the list of dept_ids returned by subquery.

If you change the second inner query from `WHERE role_id = (SELECT role_id FROM roles WHERE role_name = 'Manager')` to `WHERE role_id NOT IN (SELECT role_id FROM roles WHERE role_name = 'Manager')`, it will instead retrieve departments where no one holds any role other than the role of 'Manager'.

This would essentially reverse the logic, and instead of finding departments with no managers, you would be attempting to exclude departments where employees hold only the role of manager, but allow any other roles.

Here's the original intended logic breakdown:
- The subquery `SELECT role_id FROM roles WHERE role_name = 'Manager'` fetches the `role_id` of the manager role.
- The outer query retrieves the `dept_name` of departments where the `dept_id` is not present in the list of `dept_id`s of employees whose role is a manager.

If you switch to `role_id NOT IN`, it alters the focus from identifying "manager-less" departments to filtering out those where someone has a manager role but might also have other roles. So, you should be cautious about making this change if you intend to keep the original logic intact.

Let me know if you need more detailed clarification!

```
WITH highest_sales AS (
    SELECT product_id, SUM(quantity) AS tot_quantity
    FROM sales
    WHERE sale_date >= DATEADD(QUARTER, -1, GETDATE())
    GROUP BY product_id
)
SELECT p.product_name, hs.tot_quantity
FROM product p
JOIN highest_sales hs ON p.product_id = hs.product_id
WHERE hs.tot_quantity = (SELECT MAX(tot_quantity) FROM highest_sales);
```

1. Count of Exams Attended by Students for Each Subject

Code:

```
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;
```

Sample Data:

Sample Data:

- `students`:

student_id	student_name
1	Alice
2	Bob

- `subjects`:

subject_name
Math
Science

- `examinations`:

student_id	subject_name
1	Math
2	Science

Final Output:

student_id	student_name	subject_name	attended_exams
1	Alice	Math	1
2	Bob	Science	1

Explanation:
This query counts how many exams each student has attended per subject by using a `CROSS JOIN` to combine all students with subjects and a `LEFT JOIN` to count exams attended.

2. Second Query: Sellers with No Orders in 2020

Code:

```
SELECT s.seller_name
FROM (SELECT * FROM orders WHERE YEAR(sale_date)=2020) AS o
LEFT JOIN sellers s ON o.seller_id = s.seller_id
WHERE o.order_id IS NULL
ORDER BY seller_name;
```

Sample Data:

Sample Data:

- `sellers`:

seller_id	seller_name
1	Alice
2	Bob

- `orders`:

order_id	seller_id	sale_date
1	1	2020-05-10
2	2	2020-06-15

Final Output:

seller_name
(empty)

Explanation:
This query finds sellers who have no orders in 2020 by filtering the `orders` table for 2020 and checking for `NULL` in the `order_id` after a `LEFT JOIN`.

3. Top Travellers by Total Distance Travelled

Code:

```
WITH top_travellers AS (
  SELECT u.name, SUM(r.distance) AS distance_travelled
```

```
  FROM users u
  LEFT JOIN rides r ON u.id = r.user_id
  GROUP BY u.id, u.name)
SELECT name, distance_travelled
FROM top_travellers
ORDER BY distance_travelled DESC;
```

Sample Data:

- `users`:

id	name
1	Alice
2	Bob

- `rides`:

id	user_id	distance
1	1	100
2	2	200

Final Output:

name	distance_travelled
Bob	200
Alice	100

Explanation:

This query calculates the total distance each user has travelled by summing the `distance` field, then ranks users in descending order of total distance using a `LEFT JOIN`.