SQL Practice

SQL 50

- *case syntax:* SUM(CASE WHEN rating < 3 THEN 1 ELSE 0 END)

Select

- select product_id from Products where low_fats = 'Y' and recyclable = 'Y'
- select name from Customer where referee_id is null or referee_id != 2
- select name, population, area from World where area >= 3000000 or population >= 25000000
- select distinct author_id as id from Views where author_id = viewer_id order by author_id asc
- select tweet_id from Tweets where length(content) > 15

Basic Joins

- **select unique_id, name from Employees left join EmployeeUNI on Employees.id = EmployeeUNI.id**
- **select product_name, year, price from Sales left join Product on Sales.product_id = Product.product_id**
- **select V.customer_id, count(V.visit_id) 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**
- **select W2.id from Weather W1 inner join Weather W2 on W2.recordDate = W1.recordDate + interval 1 day and W1.temperature<W2.temperature**
- **select machine_id, round(avg(end_time-start_time),3) as processing_time from (select machine_id, process_id, max(case when activity_type = 'start' then timestamp end) as start_time, max(case when activity_type = 'end' then timestamp end) as end_time from Activity group by machine_id, process_id) as subquery group by machine_id**
    1. The inner subquery groups the data by `machine_id` and `process_id`, and then finds the maximum `timestamp` for each `activity_type` ('start' and 'end'). This gives us the start and end timestamps for each process.
        - `select` is used to specify the columns to be retrieved.
        - `machine_id` and `process_id` are selected directly.

        - `max(case when activity_type = 'start' then timestamp end)` is a function that returns the maximum `timestamp` for rows where `activity_type` is `'start'`.
        - The second part `as subquery` is likely referring to the entire subquery being aliased as `subquery`. This is a common practice in SQL to give an alias to a subquery for easier reference in the outer query.
    2. The outer query then calculates the average processing time for each machine by:
        - Subtracting the start time from the end time to get the processing time for each process.
        - Averaging the processing times for all processes on each machine.
        - Rounding the result to 3 decimal places using the `ROUND()` function.
- **select name, bonus from Employee left join Bonus on Employee.empId = Bonus.empId where Bonus.bonus is null or Bonus.bonus < 1000**
- **select s.student_id, s.student_name, sub.subject_name, count(e.subject_name) as attended_exams from Students s cross join Subjects sub left join Examinations e on s.student_id = e.student_id and e.subject_name = sub.subject_name group by s.student_id, s.student_name, sub.subject_name order by s.student_id, sub.subject_name**
- **select e1.name from Employee e1 inner join employee e2 on e1.id = e2.managerId group by e1.id having count(e2.id) >= 5**
- **select s.user_id, round(avg(if (c.action = 'confirmed', 1, 0)), 2) as confirmation_rate from Signups s left join Confirmations c on s.user_id = c.user_id group by s.user_id**

Basic Aggregate Functions

- **select * from Cinema where id%2!=0 and description != 'boring' order by rating desc**
- **select p.product_id, ifnull(round(sum(p.price * u.units)/sum(u.units), 2), 0) as average_price from Prices p left join UnitsSold u on p.product_id = u.product_id and u.purchase_date between p.start_date and p.end_date group by p.product_id**
    - `IFNULL(expression, replacement_value)
        - expression: This is the value you're checking. If it's NULL, then IFNULL will return the replacement_value.

        - replacement_value: This is the value that gets returned if the expression is NULL.`

- **select project_id, round(avg(e.experience_years),2) as average_years from Project p left join Employee e on p.employee_id = e.employee_id group by p.project_id**
- **select contest_id, round(count(distinct user_id) * 100 / (select count(user_id) from Users), 2) as percentage from Register group by contest_id order by percentage desc, contest_id asc**
- **select query_name, round(sum(rating/position) / count(*), 2) as quality, round(sum(case when rating < 3 then 1 else 0 end)* 100 /(count(*)), 2) as poor_query_percentage from Queries group by query_name**
    - `COUNT(*)` gives the **denominator** (all rows in the group).
- **select date_format(trans_date, '%Y-%m') as month, country, count(*) 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 date_format(trans_date, '%Y-%m'), country**
    - `count(case when state = 'approved' then 1 else 0 end) as approved_count`: `COUNT(...)` counts *all rows*, not just the approved ones. That's why this `approved_count` is wrong.
    - Correct approach: Use `SUM` with a boolean or conditional instead
- **select round(sum(case when min_order_date = min_customer_pref_delivery_date then 1 else 0 end) *100/ count(*), 2) as immediate_percentage from (select delivery_id, customer_id, min(order_date) as min_order_date, min(customer_pref_delivery_date) as min_customer_pref_delivery_date from Delivery group by customer_id) as new_table**
- **select round(sum(player_login) / count(distinct player_id), 2) as fraction from (select player_id, datediff(event_date, min(event_date) over (partition by player_id)) = 1 as player_login from Activity) as new_table**
    - A `PARTITION BY` clause is used to partition rows of table into groups. It is always used inside `OVER()` clause.

Sorting and Grouping

- **select teacher_id, count(distinct subject_id) as cnt from Teacher group by teacher_id**

- **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**
- **select product_id, year as first_year, quantity, price from Sales where (product_id, year) in (select product_id, min(year) from sales group by product_id)**
- **select class from Courses group by class having count(student) >=5**
- **select user_id, count(follower_id) as followers_count from Followers group by user_id order by user_id asc**
- **select max(num) as num from (select num from MyNumbers group by num having count(num) = 1) t**
- **select customer_id from Customer group by customer_id having count(distinct product_key) = (select count(*) from Product)**

Advanced Select and Joins

- **select e1.employee_id, e1.name, count(e2.reports_to) as reports_count, round(avg(e2.age)) as average_age from Employees e1 inner join Employees e2 on e1.employee_id = e2.reports_to group by e1.employee_id order by e1.employee_id**
- **select employee_id, department_id from Employee where primary_flag = 'Y' group by employee_id union select employee_id, department_id from Employee group by employee_id having count(employee_id) = 1**
- **select x, y, z, (case when (x+y > z and y+z > x and x+z > y) then 'Yes' else 'No' end) as triangle from Triangle**
- **select l1.num as ConsecutiveNums from Logs l1 inner join Logs l2 on l1.id = l2.id + 1 inner join Logs l3 on l2.id = l3.id + 1 where l1.id - l2.id = 1 and l2.id - l3.id = 1 and l1.num = l2.num and l2.num = l3.num group by l1.num**
- **select product_id, new_price as price from Products where (product_id, change_date) in (select product_id, max(change_date) from Products where change_date <= '2019-08-16' group by product_id) union select product_id, 10 as price from Products where (product_id) not in (select product_id from Products where change_date <= '2019-08-16' group by product_id)**
- **select q1.person_name from Queue q1 inner join Queue q2 on q1.turn >= q2.turn group by q1.turn having sum(q2.weight) <= 1000 order by sum(q2.weight) desc limit 1**

- **select "Low Salary" as category, count(income) as accounts_count from Accounts where income < 20000 union select "Average Salary" as category, count(income) as accounts_count from Accounts where income between 20000 and 50000 union select "High Salary" as category, count(income) as accounts_count from Accounts where income > 50000**

Subqueries

- **select employee_id from Employees where salary < 30000 and manager_id not in (select employee_id from Employees) order by employee_id**

-
  ```
    select
    case
      when id = (select max(id) from Seat) and mod(id,2) = 1
      then id
      when mod(id, 2) = 1
      then id + 1
      else id - 1
    end as id, student
  from Seat order by id asc
  ```
- `(select name as results from Users u inner join MovieRating mr on u.user_id = mr.user_id group by u.user_id order by count(mr.rating) desc, name asc limit 1) union all (select title as results from Movies m inner join MovieRating mr on m.movie_id = mr.movie_id where mr.created_at between '2020-02-01' and '2020-02-29' group by m.movie_id order by avg(mr.rating) desc, title asc limit 1)`
- `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 order by visited_on asc`
- `select requester_id as id, count(*) as num from (select requester_id from RequestAccepted union all select accepter_id from RequestAccepted) as friend_count group by id order by num desc limit 1`
- `select round(sum(tiv_2016), 2) as tiv_2016 from Insurance where tiv_2015 in (select tiv_2015 from Insurance group by tiv_2015`

`having count(*) > 1) and (lat, lon) in (select lat, lon from insurance group by lat, lon having count(*) = 1)`

- `select d.name as Department, e.name as Employee, e.salary as Salary from Employee e inner join Department d on e.departmentId = d.id where 3 > ( select count(distinct e2.salary) from Employee e2 where e2.salary > e.salary and e.departmentId = e2.departmentId)`

Advanced String Functions/Regex/Clause

- `select user_id, concat(upper(left(name, 1)), lower(right(name, length(name)-1))) as name from Users order by user_id`
- `select * from patients where conditions like "DIAB1%" or conditions like "% DIAB1%"`
- `delete p1 from Person p1 inner join Person p2 on p1.email = p2.email and p1.id > p2.id`
- `select max(salary) as secondHighestSalary from Employee where salary != (select max(salary) from Employee)`
- `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 asc`
- `select product_name, sum(unit) as unit from Products p inner join Orders o on p.product_id = o.product_id where month(order_date) = 2 and year(order_date) = 2020 group by product_name having unit >= 100`
- `select * from Users where regexp_like(mail, '^[A-Za-z][A-Za-z0-9_.-]*@leetcode\\.com$', 'c')`
    - regex always starts with `^` and ends with `$` even if nothing is present in between
    - `*` means “repeat this group any number of times.
    -
        - `.` normally means “any character” in regex, but here we want a literal dot. So `\\.` is used: the first backslash escapes the second (MySQL string rule), leaving `\.` which regex understands as “a real dot.”
    - Third parameter `'c'` = **case-sensitive** matching.

Database for Instagram

- Create Database:
    - create database if not exists instagramDb;

- Use the database to create tables:
  - use instagramDb;
- Create tables into the Db:
  - create table if not exists users(
    userId INT PRIMARY KEY,
    userName VARCHAR(50),
    email VARCHAR(100)
)
  - create table if not exists posts(
    postId INT PRIMARY KEY,
    userId INT,
    caption VARCHAR(100)
)
- Insert values into the tables:
  - insert into users (userId, userName, email) VALUES
    (1, "riti", "abc@gmail.com"),
    (2, "ankit", "bdc@gmail.com")
  - insert into posts (postId, userId, caption) VALUES
    (101, 561, "light")
    (103, 563, "water")
- select * from users
- select * from posts

Constraints

- Constraints define rules or conditions that must be satisfied by the data in the table. Common constraints include uniqueness, nullability, default values, etc.
  - Unique constraint: Ensures values in a column are unique across the table.
  - Not null constraint: Ensures a column cannot have a null value.
  - Check constraint: Enforces a condition to be true for each row.
  - Default constraint: Provides a default value for a column if no value is specified.
  - Primary key : Enforces the uniqueness of values in one or more columns
  - Foreign key: it helps to perform operations related to the parent table, such as joining tables or ensuring referential integrity