Blinkit and Zepto SQL questions .

(YoE 2+ years)

CTC= 16 LPA

---

1. Write a query to calculate the 7-day Moving Average for Sales Data

A **moving average** helps smooth out short-term fluctuations and highlight trends over time.

Assuming a table sales_data with columns:

- sale_date (DATE)
- sales_amount (INT)

SQL Query (Using AVG() with OVER() Clause):

```
SELECT
  sale_date,
  sales_amount,
  AVG(sales_amount) OVER (
    ORDER BY sale_date
    ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  ) AS moving_avg_7d
FROM sales_data;
```

✅ This calculates the 7-day moving average, including the current day and the past 6 days.

---

2. Given a table with employee in and out times, Calculate Total Hours Worked per Employee per Day

Given a table attendance:

- employee_id (INT)
- in_time (DATETIME)
- out_time (DATETIME)

SQL Query:

```
SELECT
  employee_id,
  DATE(in_time) AS work_date,
  SUM(TIMESTAMPDIFF(HOUR, in_time, out_time)) AS total_hours
FROM attendance
GROUP BY employee_id, work_date;
```

✅ This calculates **total working hours** per employee, per day.

---

3. Write a Query to Find Top N Highest-Grossing Products per Category

Given a table sales:

- product_id (INT)
- category_id (INT)
- revenue (DECIMAL)

SQL Query (Using DENSE_RANK() for Top N Products):

```
WITH ranked_products AS (
```

```
  SELECT
    category_id,
    product_id,
    SUM(revenue) AS total_revenue,
    DENSE_RANK() OVER (PARTITION BY category_id ORDER BY SUM(revenue) DESC) AS rnk
  FROM sales
  GROUP BY category_id, product_id
)
SELECT category_id, product_id, total_revenue
FROM ranked_products
WHERE rnk <= N;
```

✅ Replace N with the desired number of top products.

---

4. Write a query to Identify First and Last Transaction for Each Customer Within a Specific Time Range

Assuming a table transactions with columns:

- customer_id (INT)
- transaction_date (DATETIME)
- amount (DECIMAL)

SQL Query (Using FIRST_VALUE() and LAST_VALUE()):

```
SELECT DISTINCT customer_id,
  FIRST_VALUE(transaction_date) OVER (
    PARTITION BY customer_id
    ORDER BY transaction_date
```

```
    ) AS first_transaction,
    LAST_VALUE(transaction_date) OVER (
        PARTITION BY customer_id
        ORDER BY transaction_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS last_transaction
FROM transactions
WHERE transaction_date BETWEEN '2024-01-01' AND '2024-12-31';
```

✅ Retrieves the **first and last transaction dates** within a given time range.

---

5. Implement a SQL query to Rank Products by Sales Within Each Region & Break Ties Using Additional Criteria

Assuming a table sales_data with columns:

- product_id (INT)
- region (VARCHAR)
- sales_amount (DECIMAL)
- order_count (INT)

SQL Query (Using RANK() with Tie Breaker):

```
SELECT
    region,
    product_id,
    sales_amount,
    order_count,
    RANK() OVER (
        PARTITION BY region
        ORDER BY sales_amount DESC, order_count DESC
```

```
) AS sales_rank
FROM sales_data;
```

✅ **Ranks products by sales amount** within each region and breaks ties using order_count.

---

6. Retrieve the Second Highest Salary Without LIMIT or OFFSET

Assuming a table employees with column:

- salary (DECIMAL)

SQL Query (Using DISTINCT and MAX() in a Subquery):

```
SELECT MAX(salary) AS second_highest_salary
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

✅ Finds the **second highest salary** efficiently.

Alternative (Using DENSE_RANK()):

```
WITH salary_ranks AS (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
  FROM employees
)
SELECT salary AS second_highest_salary
FROM salary_ranks
WHERE rnk = 2;
```

✅ Handles **duplicate salaries** properly.

---

7. Identify the Percentage Contribution of Each Product to Total Revenue (Grouped by Category)

Assuming a table sales with columns:

- product_id (INT)
- category_id (INT)
- revenue (DECIMAL)

SQL Query (Using SUM() and OVER()):

```
SELECT
  category_id,
  product_id,
  revenue,
  (revenue * 100.0) / SUM(revenue) OVER (PARTITION BY category_id) AS revenue_percentage
FROM sales;
```

✅ Calculates each product's **percentage contribution** within its category.

---

8. Write a Query to Find Gaps in Sequential Data (Missing Invoice Numbers)

Assuming a table invoices with columns:

- invoice_id (INT)
- invoice_date (DATE)

SQL Query (Using LAG() and GAPS Calculation):

```
WITH missing_invoices AS (
  SELECT
    invoice_id,
    LAG(invoice_id) OVER (ORDER BY invoice_id) AS prev_invoice
  FROM invoices
)
SELECT invoice_id, prev_invoice,
```

```
    (invoice_id - prev_invoice) AS gap_size
FROM missing_invoices
WHERE (invoice_id - prev_invoice) > 1;
```

✅ Finds missing invoice numbers by detecting **gaps in sequence**.

---

9. Write a Query to Find Calculate Month-over-Month Growth Percentage for Each Product

Assuming a table sales_data with columns:

- product_id (INT)
- sale_month (DATE)
- revenue (DECIMAL)

SQL Query (Using LAG() to Compare Previous Month's Revenue):

```
WITH monthly_sales AS (
  SELECT
    product_id,
    sale_month,
    revenue,
    LAG(revenue) OVER (PARTITION BY product_id ORDER BY sale_month) AS prev_month_revenue
  FROM sales_data
)
SELECT
  product_id,
  sale_month,
  revenue,
  prev_month_revenue,
```

ROUND(((revenue - prev_month_revenue) / prev_month_revenue) * 100, 2) AS mom_growth_percentage

FROM monthly_sales

WHERE prev_month_revenue IS NOT NULL;

✅ Computes **Month-over-Month Growth (%)** for each product.

---

10. Write a Query Find Customers Who Made Purchases in Every Quarter of the Year

Assuming a table sales with columns:

- customer_id (INT)
- purchase_date (DATE)

SQL Query (Using EXTRACT() and HAVING COUNT(DISTINCT)):

SELECT customer_id

FROM sales

WHERE EXTRACT(YEAR FROM purchase_date) = 2024

GROUP BY customer_id

HAVING COUNT(DISTINCT EXTRACT(QUARTER FROM purchase_date)) = 4;

✅ Identifies customers who made at least **one purchase in every quarter** of 2024.

---

11. Calculate Cumulative Sales for Each Product Over Time & Compare with Average Sales

Assuming a table sales_data with columns:

- product_id (INT)
- sale_date (DATE)
- sales_amount (DECIMAL)

SQL Query (Using SUM() and AVG() Over Time):

```
WITH cumulative_sales AS (
  SELECT
    product_id,
    sale_date,
    sales_amount,
    SUM(sales_amount) OVER (
      PARTITION BY product_id ORDER BY sale_date
    ) AS cumulative_sales,
    AVG(sales_amount) OVER (PARTITION BY product_id) AS avg_sales
  FROM sales_data
)
SELECT product_id, sale_date, sales_amount, cumulative_sales, avg_sales
FROM cumulative_sales;
```

✅ Computes **cumulative sales** per product over time and compares it with its **average sales**.

---

12. Retrieve All Customers Who Made More Purchases This Month Compared to the Previous Month

Assuming a table sales with columns:

- customer_id (INT)
- purchase_date (DATE)

SQL Query (Using LAG() to Compare Monthly Purchases):

```
WITH monthly_purchases AS (
  SELECT
    customer_id,
    EXTRACT(YEAR FROM purchase_date) AS year,
```

```
    EXTRACT(MONTH FROM purchase_date) AS month,
    COUNT(*) AS purchase_count
  FROM sales
  GROUP BY customer_id, year, month
),
comparison AS (
  SELECT
    customer_id,
    year,
    month,
    purchase_count,
    LAG(purchase_count) OVER (PARTITION BY customer_id ORDER BY year, month) AS prev_month_purchases
  FROM monthly_purchases
)
SELECT customer_id, month, purchase_count, prev_month_purchases
FROM comparison
WHERE purchase_count > prev_month_purchases;
```

✅ Identifies **customers with increased purchases** compared to the previous month.

---

13. Identify Products That Have Never Been Sold Along with Their Categories

Assuming two tables:

- products (**product_id**, category_id, product_name)
- sales (sale_id, product_id, sale_date, quantity, revenue)

SQL Query (Using LEFT JOIN and IS NULL):

```
SELECT p.product_id, p.product_name, p.category_id
FROM products p
LEFT JOIN sales s ON p.product_id = s.product_id
WHERE s.product_id IS NULL;
```

✅ Finds products **never sold** by checking NULL sales records.

---

14. Find the Top 3 Customers Who Contributed the Most to Revenue in the Last Year

Assuming a table sales with columns:

- customer_id (INT)
- sale_date (DATE)
- revenue (DECIMAL)

SQL Query (Using SUM() and RANK()):

```
WITH customer_revenue AS (
  SELECT
    customer_id,
    SUM(revenue) AS total_revenue
  FROM sales
  WHERE sale_date >= DATE_SUB(CURDATE(), INTERVAL 1 YEAR)
  GROUP BY customer_id
)
SELECT customer_id, total_revenue
FROM customer_revenue
ORDER BY total_revenue DESC
LIMIT 3;
```

✅ Retrieves **top 3 customers** with the highest revenue in the last year.

---

15. Write a Query to Compare Sales of the Same Product Across Two Different Time Periods

Assuming a table sales_data with columns:

- product_id (INT)
- sale_date (DATE)
- revenue (DECIMAL)

SQL Query (Using Conditional Aggregation for Two Time Periods):

```
SELECT
  product_id,
  SUM(CASE WHEN sale_date BETWEEN '2023-01-01' AND '2023-12-31' THEN revenue ELSE 0 END) AS sales_2023,
  SUM(CASE WHEN sale_date BETWEEN '2024-01-01' AND '2024-12-31' THEN revenue ELSE 0 END) AS sales_2024
FROM sales_data
GROUP BY product_id;
```

✅ Compares sales of **the same product** across two different years.

---

16. Identify the Longest Streak of Consecutive Days a Customer Has Made a Purchase

Assuming a table sales with columns:

- customer_id (INT)
- purchase_date (DATE)

SQL Query (Using LAG() and DENSE_RANK() to Find Consecutive Streaks):

```
WITH purchase_streaks AS (
  SELECT
```

```
    customer_id,
    purchase_date,
    purchase_date - INTERVAL DENSE_RANK() OVER (
      PARTITION BY customer_id ORDER BY purchase_date
    ) DAY AS streak_group
  FROM sales
)
SELECT
  customer_id,
  COUNT(*) AS longest_streak
FROM purchase_streaks
GROUP BY customer_id, streak_group
ORDER BY longest_streak DESC
LIMIT 1;
```

✅ Finds the **longest consecutive purchase streak** per customer.

---

17. Calculate the Rolling Retention Rate for an App Based on Daily User Activity

Assuming a table user_activity with columns:

- user_id (INT)
- activity_date (DATE)
- signup_date (DATE)

SQL Query (Using COUNT() and LAG()):

```
WITH daily_retention AS (
  SELECT
    activity_date,
```

```
    COUNT(DISTINCT user_id) AS active_users,
    LAG(COUNT(DISTINCT user_id)) OVER (ORDER BY activity_date) AS prev_day_users
  FROM user_activity
  GROUP BY activity_date
)
SELECT
  activity_date,
  active_users,
  prev_day_users,
  ROUND((active_users * 100.0) / NULLIF(prev_day_users, 0), 2) AS rolling_retention_rate
FROM daily_retention;
```

✅ Computes **daily rolling retention** by comparing active users with the previous day.

---

18. Find Duplicate Rows in a Table and Group Them by Their Count

Assuming a table data_table with multiple columns (e.g., col1, col2, etc.)

SQL Query (Using GROUP BY and HAVING COUNT(*) > 1):

```
SELECT col1, col2, col3, COUNT(*) AS duplicate_count
FROM data_table
GROUP BY col1, col2, col3
HAVING COUNT(*) > 1;
```

✅ Identifies **duplicate rows** and groups them by their count.

---

19. Delete Duplicate Rows but Keep the Row with the Earliest Timestamp

Assuming a table data_table with a created_at timestamp column:

SQL Query (Using ROW_NUMBER() to Keep the Earliest Record):

```
WITH duplicates_cte AS (
  SELECT
    *,
    ROW_NUMBER() OVER (PARTITION BY col1, col2, col3 ORDER BY created_at) AS row_num
  FROM data_table
)
DELETE FROM data_table
WHERE (col1, col2, col3, created_at) IN (
  SELECT col1, col2, col3, created_at FROM duplicates_cte WHERE row_num > 1
);
```

✅ Removes duplicates **while keeping the earliest timestamp record**.

---

20. Calculate the Median Salary for Employees Grouped by Department

Assuming a table employees with columns:

- department_id (INT)
- salary (DECIMAL)

SQL Query (Using PERCENTILE_CONT() in PostgreSQL/SQL Server)

```
SELECT
  department_id,
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary) AS median_salary
FROM employees
GROUP BY department_id;
```

✅ Finds the **median salary per department** using **PERCENTILE_CONT()**, which works in PostgreSQL and SQL Server.

Alternative (For MySQL Using ROW_NUMBER() in a CTE)

```
WITH ranked_salaries AS (
  SELECT
    department_id,
    salary,
    ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary) AS row_num,
    COUNT(*) OVER (PARTITION BY department_id) AS total_count
  FROM employees
)
SELECT department_id, AVG(salary) AS median_salary
FROM ranked_salaries
WHERE row_num IN (FLOOR((total_count + 1) / 2), CEIL((total_count + 1) / 2))
GROUP BY department_id;
```

✅ Works in MySQL by **finding middle-ranked salaries** for odd/even counts.

---

21. Identify Products Sold for 3 Consecutive Months but Not in the 4th Month

Assuming a table sales with columns:

- product_id (INT)
- sale_date (DATE)

SQL Query (Using LAG() to Compare Consecutive Months)

```
WITH monthly_sales AS (
  SELECT
    product_id,
```

```
    DATE_FORMAT(sale_date, '%Y-%m') AS sale_month
  FROM sales
  GROUP BY product_id, sale_month
),
sales_streaks AS (
  SELECT
    product_id,
    sale_month,
    LAG(sale_month, 1) OVER (PARTITION BY product_id ORDER BY sale_month) AS prev_month_1,
    LAG(sale_month, 2) OVER (PARTITION BY product_id ORDER BY sale_month) AS prev_month_2,
    LEAD(sale_month, 1) OVER (PARTITION BY product_id ORDER BY sale_month) AS next_month
  FROM monthly_sales
)
SELECT product_id
FROM sales_streaks
WHERE prev_month_1 IS NOT NULL
AND prev_month_2 IS NOT NULL
AND next_month IS NULL;
```

✅ Finds **products sold for 3 consecutive months but missing in the 4th month**.

---

22. Segment Customers into Cohorts Based on First Purchase Date and Calculate Retention Rates

Assuming a table sales with columns:

- customer_id (INT)

- purchase_date (DATE)

Step 1: Assign Each Customer to a Cohort (Based on First Purchase Date)

```
WITH cohort_assignment AS (
  SELECT
    customer_id,
    MIN(DATE_FORMAT(purchase_date, '%Y-%m')) AS cohort_month
  FROM sales
  GROUP BY customer_id
),
cohort_analysis AS (
  SELECT
    ca.cohort_month,
    DATE_FORMAT(s.purchase_date, '%Y-%m') AS active_month,
    COUNT(DISTINCT s.customer_id) AS active_users
  FROM sales s
  JOIN cohort_assignment ca ON s.customer_id = ca.customer_id
  GROUP BY ca.cohort_month, active_month
)
SELECT
  cohort_month,
  active_month,
  active_users,
  ROUND(100.0 * active_users / FIRST_VALUE(active_users) OVER (PARTITION BY cohort_month ORDER BY active_month), 2) AS retention_rate
FROM cohort_analysis
ORDER BY cohort_month, active_month;
```

✅ Segments customers into **cohorts based on first purchase date** and calculates **monthly retention rates**.

🚀 **Key SQL Techniques Used**

✅ **Percentile Functions** for **Median Salary Calculation**
✅ **Window Functions (LAG(), LEAD())** for **Sequential Analysis**
✅ **Cohort Analysis** for **Customer Retention Tracking**

---

23. Implement a query to Calculate the Total Time Spent by Each User, Excluding Overlapping Time Intervals

Assuming a table user_sessions with columns:

- user_id (INT)
- start_time (DATETIME)
- end_time (DATETIME)

SQL Query (Using LAG(), GREATEST(), and SUM())

```
WITH ordered_sessions AS (
  SELECT
    user_id,
    start_time,
    end_time,
    LAG(end_time) OVER (PARTITION BY user_id ORDER BY start_time) AS prev_end_time
  FROM user_sessions
)
SELECT
  user_id,
  SUM(TIMESTAMPDIFF(SECOND, GREATEST(start_time, COALESCE(prev_end_time, start_time)), end_time)) AS total_time_spent
```

```
FROM ordered_sessions
GROUP BY user_id;
```

✅ **Handles overlapping sessions** by ensuring that only non-overlapping time is counted.

---

24. Find All Products That Were Never Purchased Together in the Same Transaction

Assuming a table sales with columns:

- transaction_id (INT)
- product_id (INT)

SQL Query (Using SELF JOIN and NOT EXISTS)

```
SELECT DISTINCT p1.product_id, p2.product_id
FROM sales p1
CROSS JOIN sales p2
WHERE p1.product_id < p2.product_id
AND NOT EXISTS (
  SELECT 1
  FROM sales s1
  JOIN sales s2 ON s1.transaction_id = s2.transaction_id
  WHERE s1.product_id = p1.product_id AND s2.product_id = p2.product_id
);
```

✅ **Finds product pairs that were never purchased together** in the same transaction.

---

25. Generate a Time-Series Report for Sales Data, Filling Missing Dates with Zero Sales

Assuming a table sales with columns:

- sale_date (DATE)
- product_id (INT)
- sales_amount (DECIMAL)

SQL Query (Using Recursive CTE for Date Generation and LEFT JOIN)

```
WITH date_series AS (
   SELECT MIN(sale_date) AS sale_date FROM sales
   UNION ALL
   SELECT DATE_ADD(sale_date, INTERVAL 1 DAY)
   FROM date_series
   WHERE sale_date < (SELECT MAX(sale_date) FROM sales)
)
SELECT d.sale_date,
     s.product_id,
     COALESCE(SUM(s.sales_amount), 0) AS total_sales
FROM date_series d
LEFT JOIN sales s ON d.sale_date = s.sale_date
GROUP BY d.sale_date, s.product_id
ORDER BY d.sale_date, s.product_id;
```

✅ **Generates missing dates** and fills in **zero sales** where necessary.

🚀 **Key SQL Techniques Used**

✅ **Window Functions (LAG(), GREATEST())** for **Overlapping Time Intervals**
✅ **SELF JOIN + NOT EXISTS** for **Finding Product Pair Constraints**
✅ **Recursive CTE (WITH RECURSIVE)** for **Generating Missing Dates in Time-Series Reports**