

MYNTRA DATA ANALYST INTERVIEW QUESTIONS

YOE: 2-5

CTC: 25+



Myntra



-
- ◆ Q1. Find the second highest salary without using LIMIT, OFFSET, or TOP

💡 Input Table: Employees

EmpID	Name	Salary
1	Alice	60000
2	Bob	80000
3	Charlie	75000
4	David	80000
5	Eve	90000

💡 SQL Query:

```
SELECT MAX(Salary) AS Second_Highest_Salary  
FROM Employees  
WHERE Salary < (  
    SELECT MAX(Salary) FROM Employees  
);
```

💡 Expected Output:

Second_Highest_Salary
80000

-
- ◆ Q2. Given a table of orders, write a query to find the running total (cumulative sum) of revenue for each day

💡 Input Table: Orders

OrderID	OrderDate	Revenue
101	2024-01-01	100
102	2024-01-01	200
103	2024-01-02	150
104	2024-01-03	300
105	2024-01-03	100

🧠 SQL Query:

```
SELECT  
    OrderDate,  
    SUM(Revenue) AS Daily_Revenue,  
    SUM(SUM(Revenue)) OVER (ORDER BY OrderDate) AS Running_Total_Revenue  
FROM Orders  
GROUP BY OrderDate  
ORDER BY OrderDate;
```

💡 Expected Output:

OrderDate	Daily_Revenue	Running_Total_Revenue
2024-01-01	300	300
2024-01-02	150	450
2024-01-03	400	850

- ◆ Q3. Write an SQL query to identify employees who earn more than their managers

💡 Input Table: Employees

EmpID	Name	Salary	ManagerID
1	Alice	70000	NULL
2	Bob	80000	1
3	Charlie	60000	1
4	David	90000	2
5	Eve	85000	2

💡 SQL Query:

```
SELECT e.EmpID, e.Name, e.Salary, e.ManagerID
FROM Employees e
JOIN Employees m ON e.ManagerID = m.EmpID
WHERE e.Salary > m.Salary;
```

💡 Expected Output:

EmpID	Name	Salary	ManagerID
2	Bob	80000	1
4	David	90000	2
5	Eve	85000	2

✅ Q4. Find the top N customers who made the highest purchases, ensuring no duplicates if customers have the same purchase amount

💡 Input Table: Purchases

CustomerID	CustomerName	PurchaseAmount
1	Alice	5000

CustomerID	CustomerName	PurchaseAmount
2	Bob	7000
3	Charlie	7000
4	David	6500
5	Eve	6000

🎯 Goal:

Get the Top N distinct purchase amounts and the corresponding customers (no duplicates for same amount).

Let's say N = 2, i.e., top 2 distinct purchase amounts.

🧠 SQL Query:

```
WITH RankedPurchases AS (
    SELECT
        CustomerID,
        CustomerName,
        PurchaseAmount,
        DENSE_RANK() OVER (ORDER BY PurchaseAmount DESC) AS Rank
    FROM Purchases
)
SELECT
    CustomerID,
    CustomerName,
    PurchaseAmount
FROM RankedPurchases
```

WHERE Rank <= 2;

📍 Expected Output:

CustomerID	CustomerName	PurchaseAmount
2	Bob	7000
3	Charlie	7000
4	David	6500

✓ DENSE_RANK() ensures that equal amounts get the same rank, so you don't exclude tied top purchasers.

✓ Q5. Identify consecutive login streaks for users where they logged in for at least three consecutive days

📍 Input Table: UserLogins

UserID	LoginDate
101	2024-05-01
101	2024-05-02
101	2024-05-03
101	2024-05-05
102	2024-05-10
102	2024-05-11
102	2024-05-12
102	2024-05-13
103	2024-05-15

UserID	LoginDate
103	2024-05-17

🎯 Goal:

Find users who logged in for at least 3 consecutive days.

🧠 SQL Query:

```
WITH RankedLogins AS (
    SELECT
        UserID,
        LoginDate,
        ROW_NUMBER() OVER (PARTITION BY UserID ORDER BY LoginDate) AS rn
    FROM UserLogins
),
Streaks AS (
    SELECT
        UserID,
        LoginDate,
        DATE_SUB(LoginDate, INTERVAL rn DAY) AS StreakGroup
    FROM RankedLogins
),
GroupedStreaks AS (
    SELECT
        UserID,
        MIN(LoginDate) AS StartDate,
```

```

    MAX(LoginDate) AS EndDate,
    COUNT(*) AS StreakLength
  FROM Streaks
 GROUP BY UserID, StreakGroup
)
SELECT *
FROM GroupedStreaks
WHERE StreakLength >= 3;

```

✍️ Expected Output:

UserID	StartDate	EndDate	StreakLength
101	2024-05-01	2024-05-03	3
102	2024-05-10	2024-05-13	4

✓ The idea is to use ROW_NUMBER() to find gaps between actual login date and sequential row number. Same gap means consecutive logins.

✓ Q6. Write an efficient query to detect duplicate records in a table and delete only the extra duplicates, keeping one copy

✍️ Input Table: Customers

CustomerID	Name	Email
1	Alice	alice@example.com
2	Bob	bob@example.com
3	Alice	alice@example.com

CustomerID	Name	Email
4	Charlie	charlie@email.com
5	Alice	alice@example.com

🎯 Goal:

- Identify duplicate rows based on Name + Email
- Keep only 1 copy, delete the rest

📌 SQL Query (MySQL / PostgreSQL style):

-- Step 1: Delete duplicates but keep one row

DELETE FROM Customers

WHERE CustomerID NOT IN (

SELECT MIN(CustomerID)

FROM Customers

GROUP BY Name, Email

);

✓ **MIN(CustomerID)** helps keep one copy, and deletes others having the same Name + Email.

📌 Output Table After Deletion:

CustomerID	Name	Email
1	Alice	alice@example.com
2	Bob	bob@example.com
4	Charlie	charlie@email.com

 SQL Query (If using CTE in PostgreSQL):

```
WITH Duplicates AS (
    SELECT *,
        ROW_NUMBER() OVER (PARTITION BY Name, Email ORDER BY CustomerID) AS rn
    FROM Customers
)
DELETE FROM Customers
WHERE CustomerID IN (
    SELECT CustomerID FROM Duplicates WHERE rn > 1
);
```

 Q7. You have a table with millions of records. How would you optimize a slow-performing query that involves multiple joins and aggregations?

 Goal: Optimize a slow query on large data involving:

- Multiple joins
 - Aggregations (SUM, COUNT, GROUP BY, etc.)
-

 Optimization Techniques:

 1. Add Proper Indexes

- Add indexes on JOIN keys and WHERE clause columns:

```
CREATE INDEX idx_customer_id ON Orders(customer_id);
```

```
CREATE INDEX idx_order_date ON Orders(order_date);
```

 2. Use EXPLAIN Plan

- Analyze the query execution using:

EXPLAIN ANALYZE

SELECT ...

FROM ...

JOIN ...

WHERE ...

- Identify full scans, slow joins, missing indexes

✓ 3. Avoid SELECT *

- Always project only required columns.

-- Instead of this

SELECT * FROM Orders

-- Use this

SELECT OrderID, CustomerID, Revenue FROM Orders

✓ 4. Use CTEs or Temp Tables for Intermediate Results

WITH RevenuePerCustomer AS (

```
    SELECT CustomerID, SUM(Revenue) AS TotalRev  
    FROM Orders  
    GROUP BY CustomerID  
)
```

SELECT c.CustomerName, r.TotalRev

FROM RevenuePerCustomer r

JOIN Customers c ON c.CustomerID = r.CustomerID;

✓ Avoids redundant computation and improves readability.

✓ 5. Join Order Matters

- Join smaller filtered datasets first to reduce row multiplication.

6. Partitioning and Batching

- If processing huge date-wise data, use table partitioning
- For maintenance scripts: batch delete or batch insert

7. Materialized Views or Summary Tables

- Precompute aggregations and store them:

CREATE MATERIALIZED VIEW MonthlyRevenue AS

```
SELECT CustomerID, MONTH(OrderDate) AS Month, SUM(Revenue) AS TotalRevenue  
FROM Orders  
GROUP BY CustomerID, MONTH(OrderDate);
```

8. Use Analytical Functions Instead of Subqueries (where applicable)

Final Tip:

Combine EXPLAIN plans with index strategy, column selection, and CTEs to dramatically improve performance.

 **Q8. Retrieve the first order for each customer, ensuring that ties (multiple first orders on the same date) are handled correctly**

 Input Table: Orders

OrderID	CustomerID	OrderDate	Amount
1	101	2024-05-01	500
2	101	2024-05-01	450
3	101	2024-05-03	600
4	102	2024-05-02	700

	OrderID	CustomerID	OrderDate	Amount
5	103		2024-05-01	300
6	103		2024-05-01	350

🎯 Goal:

- Get each customer's first order(s)
- Handle ties (same earliest date → return all matching orders)

📌 SQL Query using RANK():

```
WITH RankedOrders AS (
    SELECT *,
        RANK() OVER (PARTITION BY CustomerID ORDER BY OrderDate ASC) AS rnk
    FROM Orders
)
SELECT OrderID, CustomerID, OrderDate, Amount
FROM RankedOrders
WHERE rnk = 1;
```

👉 Output:

	OrderID	CustomerID	OrderDate	Amount
1	101		2024-05-01	500
2	101		2024-05-01	450
4	102		2024-05-02	700
5	103		2024-05-01	300

OrderID	CustomerID	OrderDate	Amount
6	103	2024-05-01	350

RANK() handles ties. If you only want one row, use ROW_NUMBER() instead.

Q9. Find products that were never purchased by any customer

 Input Tables:

 Products

ProductID	ProductName
1	Shoes
2	Bag
3	Watch
4	Hat

 OrderItems

OrderID	ProductID	Quantity
101	1	2
102	3	1
103	1	1

 Goal:

List all products from Products table that do not exist in OrderItems.

 SQL Query using LEFT JOIN:

```
SELECT p.ProductID, p.ProductName  
FROM Products p  
LEFT JOIN OrderItems o ON p.ProductID = o.ProductID  
WHERE o.ProductID IS NULL;
```

 You can also use NOT IN or NOT EXISTS depending on performance and data distribution.

 Output:

ProductID	ProductName
2	Bag
4	Hat

 Q10. Given a table with customer transactions, find customers who made transactions in every month of the year

 Input Table: Transactions

TransactionID	CustomerID	TransactionDate
1	101	2024-01-15
2	101	2024-02-10
3	101	2024-03-20
...
13	101	2024-12-01
14	102	2024-01-05

TransactionID CustomerID TransactionDate

15 102 2024-04-17

📝 (Assume full data includes all 12 months for Customer 101 but not for 102)

🎯 Goal:

- Find customers who transacted in all 12 months of a given year.

🧠 SQL Query (PostgreSQL / MySQL):

```
SELECT CustomerID  
FROM (  
    SELECT CustomerID,  
        EXTRACT(MONTH FROM TransactionDate) AS txn_month  
    FROM Transactions  
    WHERE EXTRACT(YEAR FROM TransactionDate) = 2024  
    GROUP BY CustomerID, EXTRACT(MONTH FROM TransactionDate)  
) AS monthly_txns  
GROUP BY CustomerID  
HAVING COUNT(DISTINCT txn_month) = 12;
```

📤 Output:

CustomerID

101

✓ Ensures the customer has one or more transactions in all 12 distinct months

 **Q11. Find employees who have the same salary as another employee in the same department**

 **Input Table: Employees**

EmpID	EmpName	Department	Salary
1	Alice	Sales	50000
2	Bob	Sales	60000
3	Carol	Sales	50000
4	David	HR	45000
5	Eva	HR	45000
6	Frank	HR	47000

 **Goal:**

- List employees whose salary matches someone else's in the same department
-

 **SQL Query (Self Join):**

```
SELECT e1.EmpID, e1.EmpName, e1.Department, e1.Salary  
FROM Employees e1  
JOIN Employees e2  
ON e1.Department = e2.Department  
AND e1.Salary = e2.Salary  
AND e1.EmpID <> e2.EmpID;
```

 **Output:**

EmpID	EmpName	Department	Salary
1	Alice	Sales	50000
3	Carol	Sales	50000
4	David	HR	45000
5	Eva	HR	45000

Self join filters for other employees in the same department with same salary but different IDs

Q12. Retrieve the department with the highest total salary paid to employees

👉 Input Table: Employees

EmpID	EmpName	Department	Salary
1	Alice	Sales	50000
2	Bob	Sales	60000
3	Carol	HR	45000
4	David	HR	47000
5	Eva	Tech	90000
6	Frank	Tech	80000

🎯 Goal:

Return the department with the maximum total salary paid.

🧠 SQL Query (Using ORDER BY + LIMIT or RANK()):

```
SELECT Department, SUM(Salary) AS TotalSalary  
FROM Employees  
GROUP BY Department  
ORDER BY TotalSalary DESC  
LIMIT 1;
```

⌚ Alternate using Window Function (if LIMIT is restricted):

```
WITH DeptSalaries AS (  
    SELECT Department, SUM(Salary) AS TotalSalary  
    FROM Employees  
    GROUP BY Department  
,  
RankedDepts AS (  
    SELECT *, RANK() OVER (ORDER BY TotalSalary DESC) AS rnk  
    FROM DeptSalaries  
)  
SELECT Department, TotalSalary  
FROM RankedDepts  
WHERE rnk = 1;
```

👉 Output:

Department	TotalSalary
Tech	170000

 **Q13. Rank orders based on order value for each customer and return only the top 3 orders per customer**

 **Input Table: Orders**

OrderID	CustomerID	OrderValue
1	101	1000
2	101	1500
3	101	1200
4	101	800
5	102	700
6	102	1000
7	102	1200
8	102	1300

 **Goal:**

- Rank orders for each customer by value (high to low)
 - Return only top 3 orders per customer
-

 **SQL Query using RANK() or ROW_NUMBER():**

```
WITH RankedOrders AS (
    SELECT *,
        RANK() OVER (PARTITION BY CustomerID ORDER BY OrderValue DESC) AS rnk
    FROM Orders
)
```

```
SELECT OrderID, CustomerID, OrderValue  
FROM RankedOrders  
WHERE rnk <= 3;
```

 Use ROW_NUMBER() if you want to break ties.

 Output:

OrderID	CustomerID	OrderValue
2	101	1500
3	101	1200
1	101	1000
8	102	1300
7	102	1200
6	102	1000

 Q14. Find the median salary of employees using SQL (without built-in median functions)

 Input Table: Employees

EmpID	EmpName	Salary
1	Alice	50000
2	Bob	60000
3	Carol	55000

EmpID	EmpName	Salary
4	David	70000
5	Eva	75000

🎯 Goal:

Find the median salary, which is:

- Middle value (for odd count)
- Average of two middle values (for even count)

🧠 SQL Query (Generic & DBMS-agnostic using Window Functions):

```
WITH RankedSalaries AS (
    SELECT Salary,
        ROW_NUMBER() OVER (ORDER BY Salary) AS rn,
        COUNT(*) OVER () AS total_count
    FROM Employees
),
MedianCalc AS (
    SELECT Salary, total_count, rn
    FROM RankedSalaries
    WHERE rn = (total_count + 1) / 2      -- for odd
        OR rn = total_count / 2          -- for even
        OR rn = total_count / 2 + 1      -- for even
)
SELECT ROUND(AVG(Salary), 2) AS MedianSalary
FROM MedianCalc;
```

 Output:

MedianSalary
60000.00

 Works for both odd and even numbers of rows.

 **Q15. Pivot a table where each row represents a sales transaction and transform it into a summary format where each column represents a month**

 Input Table: Sales

TransactionID	CustomerID	SaleAmount	SaleDate
1	101	1000	2024-01-15
2	101	1500	2024-02-12
3	101	700	2024-03-10
4	102	1200	2024-01-10
5	102	800	2024-03-21

 Goal:

Pivot the table to show:

- Each CustomerID in one row
 - Monthly sales in separate columns like Jan, Feb, Mar, etc.
-

 SQL Query using CASE + SUM (Standard SQL approach):

```
SELECT  
CustomerID,  
SUM(CASE WHEN MONTH(SaleDate) = 1 THEN SaleAmount ELSE 0 END) AS Jan,  
SUM(CASE WHEN MONTH(SaleDate) = 2 THEN SaleAmount ELSE 0 END) AS Feb,  
SUM(CASE WHEN MONTH(SaleDate) = 3 THEN SaleAmount ELSE 0 END) AS Mar  
-- Extend up to Dec if needed  
FROM Sales  
GROUP BY CustomerID;
```

Use TO_CHAR(SaleDate, 'MM') for PostgreSQL or EXTRACT(MONTH FROM SaleDate) as needed.

 Output:

CustomerID	Jan	Feb	Mar
101	1000	1500	700
102	1200	0	800

Q16. Find the moving average of sales for the last 7 days for each product

 Input Table: Sales

ProductID	SaleDate	SaleAmount
P1	2024-06-01	100
P1	2024-06-02	200
P1	2024-06-03	150

ProductID	SaleDate	SaleAmount
P1	2024-06-04	250
P1	2024-06-05	100
P1	2024-06-06	300
P1	2024-06-07	200
P1	2024-06-08	500
P2	2024-06-01	120
P2	2024-06-02	180

🎯 Goal:

For each ProductID and SaleDate, calculate the 7-day moving average of SaleAmount.

🧠 SQL Query using RANGE BETWEEN INTERVAL (if supported):

```
SELECT  
    ProductID,  
    SaleDate,  
    ROUND(AVG(SaleAmount) OVER (  
        PARTITION BY ProductID  
        ORDER BY SaleDate  
        RANGE BETWEEN INTERVAL 6 DAY PRECEDING AND CURRENT ROW  
    ), 2) AS MovingAvg7Days
```

FROM Sales;

⌚ Alternate using ROWS if RANGE with intervals is unsupported:

```
SELECT  
    ProductID,
```

SaleDate,
ROUND(AVG(SaleAmount) OVER (
PARTITION BY ProductID
ORDER BY SaleDate
ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
), 2) AS MovingAvg7Days
FROM Sales;

Output (sample for P1):

ProductID	SaleDate	MovingAvg7Days
P1	2024-06-01	100.00
P1	2024-06-02	150.00
P1	2024-06-03	150.00
P1	2024-06-04	175.00
P1	2024-06-05	160.00
P1	2024-06-06	183.33
P1	2024-06-07	185.71
P1	2024-06-08	228.57

Q17. Find the first and last occurrence of each event per user

Input Table: Events

UserID	EventName	EventTime
1	login	2024-05-01 10:00:00
1	login	2024-05-01 15:00:00
1	logout	2024-05-01 18:00:00
2	login	2024-05-02 09:00:00
2	login	2024-05-02 10:30:00
2	logout	2024-05-02 11:30:00
2	logout	2024-05-02 12:00:00

🎯 Goal:

Return each UserID, EventName, the first and last time that event occurred.

🧠 SQL Query using MIN() and MAX():

```
SELECT  
    UserID,  
    EventName,  
    MIN(EventTime) AS FirstOccurrence,  
    MAX(EventTime) AS LastOccurrence  
FROM Events  
GROUP BY UserID, EventName;
```

👉 Output:

UserID	EventName	FirstOccurrence	LastOccurrence
1	login	2024-05-01 10:00:00	2024-05-01 15:00:00

UserID	EventName	FirstOccurrence	LastOccurrence
1	logout	2024-05-01 18:00:00	2024-05-01 18:00:00
2	login	2024-05-02 09:00:00	2024-05-02 10:30:00
2	logout	2024-05-02 11:30:00	2024-05-02 12:00:00

 **Q18. Identify users who have placed an order in two consecutive months but not in the third month**

 **Input Table: Orders**

UserID	OrderDate
101	2023-01-15
101	2023-02-10
101	2023-03-05
102	2023-04-12
102	2023-05-18
103	2023-06-20
103	2023-07-25
103	2023-09-03

 **Goal:**

Find users who ordered in 2 consecutive months, but not in the next (third) month.

 **Solution Logic:**

1. Extract Year-Month from OrderDate.
 2. Rank months per user.
 3. Self-join or lag-lead to compare 3-month streaks.
-

🧠 SQL Query (Using LAG + LEAD):

```
WITH MonthData AS (
    SELECT
        UserID,
        DATE_TRUNC('month', OrderDate) AS MonthStart
    FROM Orders
    GROUP BY UserID, DATE_TRUNC('month', OrderDate)
),
WithLagLead AS (
    SELECT
        UserID,
        MonthStart,
        LAG(MonthStart, 1) OVER (PARTITION BY UserID ORDER BY MonthStart) AS PrevMonth,
        LEAD(MonthStart, 1) OVER (PARTITION BY UserID ORDER BY MonthStart) AS NextMonth
    FROM MonthData
)
SELECT DISTINCT UserID
FROM WithLagLead
WHERE
    DATE_PART('month', MonthStart) - DATE_PART('month', PrevMonth) = 1
```

AND (

NextMonth IS NULL OR DATE_PART('month', NextMonth) - DATE_PART('month', MonthStart) > 1

);

This logic checks for a consecutive previous month, but gap or missing third month.

 Output:

UserID
102
103

Q19. Find the most frequently purchased product category by each user over the past year

 Input Table: Orders

UserID	OrderDate	Category
101	2024-07-01	Electronics
101	2024-07-15	Fashion
101	2024-08-01	Electronics
102	2024-07-10	Grocery
102	2024-08-20	Grocery
102	2024-09-05	Fashion
103	2024-07-01	Fashion

UserID	OrderDate	Category
103	2024-07-10	Fashion
103	2024-08-01	Fashion

🎯 Goal:

Find the top product category (by frequency) per user in the last 12 months.

🧠 SQL Query (Using RANK()):

```
WITH LastYearOrders AS (
    SELECT *
    FROM Orders
    WHERE OrderDate >= CURRENT_DATE - INTERVAL '12 months'
),
CategoryCounts AS (
    SELECT
        UserID,
        Category,
        COUNT(*) AS CategoryCount
    FROM LastYearOrders
    GROUP BY UserID, Category
),
Ranked AS (
    SELECT *,
        RANK() OVER (PARTITION BY UserID ORDER BY CategoryCount DESC) AS rnk
    FROM CategoryCounts
)
```

)

```
SELECT UserID, Category, CategoryCount  
FROM Ranked  
WHERE rnk = 1;
```

Output:

UserID	Category	CategoryCount
101	Electronics	2
102	Grocery	2
103	Fashion	3

Q20. Write a query to generate a sequential ranking of products based on total sales, but reset the ranking for each year

Input Table: Sales

ProductID	SaleAmount	SaleDate
P1	500	2022-01-15
P2	700	2022-03-22
P1	300	2022-05-20
P3	800	2022-07-01
P1	1000	2023-02-10
P2	600	2023-03-15

ProductID	SaleAmount	SaleDate
P3	900	2023-04-12

🎯 Goal:

- Rank products by total sales per year.
- Rankings should restart each year.
- Use RANK() or DENSE_RANK() for ties.

🧠 Step-by-Step Logic:

1. Extract the year from SaleDate.
2. Aggregate total sales per product per year.
3. Use a window function (RANK() or DENSE_RANK()) to assign rank within each year.

🧠 SQL Query:

```
WITH YearlySales AS (
    SELECT
        EXTRACT(YEAR FROM SaleDate) AS SalesYear,
        ProductID,
        SUM(SaleAmount) AS TotalSales
    FROM Sales
    GROUP BY EXTRACT(YEAR FROM SaleDate), ProductID
),
RankedProducts AS (
    SELECT *,
        RANK() OVER (PARTITION BY SalesYear ORDER BY TotalSales DESC) AS SalesRank
)
```

```
FROM YearlySales  
)  
SELECT *  
FROM RankedProducts  
ORDER BY SalesYear, SalesRank;
```

Output:

SalesYear	ProductID	TotalSales	SalesRank
2022	P3	800	1
2022	P2	700	2
2022	P1	800	1
2023	P1	1000	1
2023	P3	900	2
2023	P2	600	3

⚠ If two products have the same total sales, RANK() assigns the same rank and skips the next number. Use DENSE_RANK() if you want continuous ranking without gaps.