# Data Analyst Interview Questions (0-3 Years) 17-19 lpa

# **SQL Questions**

# 1. Write a query to find duplicate rows in a table.

To detect duplicates, identify columns that should be unique and group by them.

#### **Example:**

SELECT column1, column2, COUNT(\*) AS count FROM your\_table GROUP BY column1, column2 HAVING COUNT(\*) > 1;

#### **Explanation:**

- GROUP BY combines rows with the same values in the specified columns.
- HAVING COUNT(\*) > 1 filters those combinations that occur more than once, indicating duplicates.

Tip: Add ROW\_NUMBER() or RANK() with CTE to highlight or delete duplicates if needed.

# 2. Explain the difference between INNER JOIN and OUTER JOIN with examples.

#### **◆ INNER JOIN:**

Returns only **matching** records from both tables.

SELECT e.name, d.department\_name

FROM employees e

INNER JOIN departments d ON e.department\_id = d.department\_id;

Output: Only employees who belong to a department.

#### LEFT OUTER JOIN:

Returns **all records from the left** table, and matching records from the right table. If no match, NULL is returned.

SELECT e.name, d.department name

FROM employees e

LEFT JOIN departments d ON e.department id = d.department id;

Output: All employees, with department info where available.

#### RIGHT OUTER JOIN:

Returns all records from the right table, and matching records from the left.

#### FULL OUTER JOIN:

Returns all records from both tables, matching where possible.

#### **Key Difference:**

- INNER JOIN = intersection (matched data only)
- OUTER JOIN = union + NULLs (matched + unmatched data)

# 3. Write a query to fetch the second-highest salary from an employee table.

Option 1: Using DISTINCT, ORDER BY, and LIMIT (MySQL/PostgreSQL)

SELECT DISTINCT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 1;

## **Option 2: Using subquery (Generic SQL)**

SELECT MAX(salary)

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

#### **Explanation:**

- The subquery fetches the highest salary.
- The outer query finds the maximum salary less than the highest giving the second-highest.

# 4. How do you use GROUP BY and HAVING together? Provide an example.

Use GROUP BY to group data and HAVING to filter **aggregated results** (unlike WHERE, which filters raw rows).

SELECT department\_id, COUNT(\*) AS emp\_count

FROM employees

GROUP BY department id

HAVING COUNT(\*) > 5;

## **Explanation:**

- Groups employees by department.
- Filters groups where the count of employees is **more than 5**.

# 5. Write a query to find employees earning more than their managers.

Assume the table employees has:

emp\_id, name, salary, manager\_id

SELECT e.name AS employee\_name, e.salary, m.name AS manager\_name, m.salary AS manager\_salary

FROM employees e

JOIN employees m ON e.manager id = m.emp id

WHERE e.salary > m.salary;

#### **Explanation:**

- Self-join: matches employees (e) with their managers (m).
- Filters those where employee's salary > manager's salary.

# 6. What is a window function in SQL? Provide examples of ROW\_NUMBER and RANK.

#### Definition:

A **window function** performs calculations **across a set of table rows** related to the current row — without collapsing rows like GROUP BY.

#### Syntax:

FUNCTION\_NAME() OVER (PARTITION BY column ORDER BY column)

## Example: ROW\_NUMBER()

Assigns a unique sequential number to each row within a partition.

SELECT name, department, salary,

ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row\_num FROM employees;

• Each employee within the same department gets a row number based on salary rank (highest first).

## Example: RANK()

Assigns **the same rank** to rows with **equal values**, but skips the next rank(s).

SELECT name, department, salary,

RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank\_num FROM employees;

• If 2 employees have the same salary, both get rank 1, and the next gets rank 3.

# 7. Write a query to fetch the top 3 performing products based on sales.

Assume table sales\_data has: product\_id, product\_name, total\_sales

SELECT product\_id, product\_name, total\_sales FROM sales\_data ORDER BY total\_sales DESC LIMIT 3;

# Alternate using RANK() (if ties matter):

```
SELECT product_id, product_name, total_sales
FROM (
    SELECT *, RANK() OVER (ORDER BY total_sales DESC) AS rank_num
    FROM sales_data
) ranked_sales
WHERE rank_num <= 3;
```

# 8. Explain the difference between UNION and UNION ALL.

Feature	UNION	UNION ALL
Duplicates	Removes duplicates	Keeps all rows, including duplicates
Performance	Slower (because of sorting)	Faster (no de-duplication)
Use case	When you want distinct rows	When duplicates are meaningful

#### **Example:**

SELECT city FROM customers

UNION

SELECT city FROM vendors;

→ Returns a unique list of cities.

SELECT city FROM customers

UNION ALL

SELECT city FROM vendors;

 $\rightarrow$  Returns **all cities**, including duplicates.

# 9. How do you use a CASE statement in SQL? Provide an example.

◆ CASE lets you write conditional logic in SQL (similar to IF/ELSE).

SELECT name, salary,

**CASE** 

WHEN salary >= 100000 THEN 'High'

WHEN salary >= 50000 THEN 'Medium'

ELSE 'Low'

END AS salary\_category

FROM employees;

#### **Explanation:**

- Assigns a category based on salary value.
- Works inside SELECT, WHERE, ORDER BY, etc.

# 10. Write a query to calculate the cumulative sum of sales.

Assume table sales has:

order\_date, product\_id, sales\_amount

SELECT order date, product id, sales amount,

SUM(sales\_amount) OVER (PARTITION BY product\_id ORDER BY order\_date) AS cumulative\_sales FROM sales;

#### **Explanation:**

- SUM(...) OVER (...) calculates a **running total** per product based on order date.
- PARTITION BY groups by product, and ORDER BY ensures the accumulation follows chronological order.

# 11. What is a CTE (Common Table Expression), and how is it used?

#### Definition:

A CTE (Common Table Expression) is a temporary, named result set that you can reference within a SQL query.

It improves readability and simplifies complex subqueries or recursive logic.

Syntax:

```
WITH cte_name AS (
 SELECT ...
SELECT * FROM cte_name;
```

## Example – Filter top-paid employees using CTE:

```
WITH HighEarners AS (
 SELECT emp id, name, salary
 FROM employees
 WHERE salary > 100000
SELECT * FROM HighEarners;
```

#### **Benefits:**

- Reusable and readable
- Allows recursion (e.g., hierarchical data)
- Avoids repeating subqueries

# 12. Write a query to identify customers who have made transactions above \$5,000 multiple times.

Assume transactions table has: customer id, transaction amount

```
SELECT customer_id, COUNT(*) AS high_value_txns
FROM transactions
WHERE transaction amount > 5000
GROUP BY customer id
HAVING COUNT(*) > 1;
```

#### **Explanation:**

- Filters high-value transactions (> \$5000).
- Groups them by customer.
- Returns customers who've done this **more than once**.

# 13. Explain the difference between DELETE and TRUNCATE commands.

Feature	DELETE	TRUNCATE
Removes rows	Yes (can use WHERE condition)	Yes (removes all rows)
WHERE supported?	■ Yes	+ No
Logging	Logs each deleted row (slower)	Minimal logging (faster)
Rollback	Can be rolled back (if within transaction)	Can be rolled back (in some RDBMS)
Identity reset	+ Retains identity	Resets identity (in most DBs)
Use case	Partial deletion or audit trail needed	Full data wipe without audit needed

# 14. How do you optimize SQL queries for better performance?

Here are **key SQL optimization techniques**:

◆ 1. Use SELECT only required columns

-- Bad SELECT \* FROM orders;

-- Good

SELECT order\_id, customer\_id FROM orders;

- 2. Create proper indexes
  - Index frequently used columns in JOIN, WHERE, ORDER BY.
- 3. Avoid functions on indexed columns

-- Slower (cannot use index) WHERE YEAR(order\_date) = 2024

-- Better

WHERE order\_date BETWEEN '2024-01-01' AND '2024-12-31'

- 4. Use EXISTS instead of IN (for subqueries)
- -- Prefer EXISTS (better for large datasets)
  SELECT name FROM customers c
  WHERE EXISTS (
   SELECT 1 FROM orders o WHERE o.customer\_id = c.customer\_id
  );
- 5. Avoid unnecessary joins or nested subqueries
- ◆ 6. Use appropriate data types and avoid implicit conversions
- ◆ 7. Analyze execution plans (EXPLAIN or EXPLAIN ANALYZE)

# 15. Write a query to find all customers who have not made

# any purchases in the last 6 months.

#### Assume:

- customers(customer\_id, name)
- transactions(customer\_id, transaction\_date)

SELECT c.customer\_id, c.name

FROM customers c

LEFT JOIN transactions t

ON c.customer id = t.customer id

AND t.transaction\_date >= CURRENT\_DATE - INTERVAL '6 months'

WHERE t.customer\_id IS NULL;

#### **Explanation:**

- LEFT JOIN includes all customers.
- WHERE t.customer\_id IS NULL ensures the customer had no purchase in the last 6 months.

# 16. How do you handle NULL values in SQL? Provide examples.

NULL represents missing or unknown data.

## 1. Using IS NULL / IS NOT NULL:

SELECT \* FROM employees WHERE manager\_id IS NULL;

# 2. Replace NULL using COALESCE() or IFNULL() (MySQL):

SELECT name, COALESCE(phone\_number, 'Not Provided') AS contact FROM customers;

# 3. Handling NULLs in aggregation (e.g., AVG, SUM):

• These functions ignore NULLs by default.

SELECT AVG(salary) FROM employees;

#### 4. Conditional checks:

SELECT name,

**CASE** 

WHEN salary IS NULL THEN 'Unknown'

ELSE 'Known'

END AS salary status

FROM employees;

# 17. Write a query to transpose rows into columns.

Assume a table sales with:

region, month, sales amount

We want to **pivot month values** into columns.

## Using CASE:

```
SELECT region,

SUM(CASE WHEN month = 'Jan' THEN sales_amount ELSE 0 END) AS Jan,

SUM(CASE WHEN month = 'Feb' THEN sales_amount ELSE 0 END) AS Feb,

SUM(CASE WHEN month = 'Mar' THEN sales_amount ELSE 0 END) AS Mar

FROM sales

GROUP BY region;
```

## Using PIVOT (SQL Server or Oracle syntax):

```
SELECT region, [Jan], [Feb], [Mar]
FROM (
    SELECT region, month, sales_amount
    FROM sales
) AS src
PIVOT (
    SUM(sales_amount)
    FOR month IN ([Jan], [Feb], [Mar])
) AS p;
```

# 18. Explain indexing and how it improves query performance.

#### What is an index?

An **index** is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional space and write-time performance.

# How indexing helps:

Feature	With Index	Without Index
Search performance	e Fast (uses binary/tree search)	Slow (scans every row — full scan)
Used in	WHERE, JOIN, ORDER BY, GROUP BY	' Inefficient for large datasets
Types	B-tree (default), Bitmap, Hash, etc.	-

# Example:

-- Creating index

CREATE INDEX idx\_customer\_id ON transactions(customer\_id);

• This helps queries like:

SELECT \* FROM transactions WHERE customer\_id = 101;

# Important notes:

- Too many indexes can slow down INSERT/UPDATE.
- Avoid indexing columns with low cardinality (e.g., gender).
- Use **composite indexes** when querying multiple columns together.

# 19. Write a query to fetch the maximum transaction amount for each customer.

Assume a transactions table:

Column	Description
customer_id	ID of the customer
transaction_id	Unique transaction ID
amount	Transaction amount

#### Query:

SELECT customer\_id, MAX(amount) AS max\_transaction FROM transactions GROUP BY customer\_id;

#### **Explanation:**

- GROUP BY groups all transactions by customer.
- MAX(amount) returns the highest transaction for each group (customer).

# 20. What is a self-join, and how is it used?

#### Definition:

A **self-join** is a regular join where a table is joined with itself.

It is useful when rows in a table are related to other rows in the same table.

## Example Use Case – Employees and Managers:

Assume:

emp_id	name	manager_id
1	Alice	NULL
2	Bob	1
3	Carol	1
4	David	2

Here, manager id refers to emp id of another employee.

# Query: Get employee names along with their manager names

SELECT e.name AS employee\_name, m.name AS manager\_name FROM employees e

LEFT JOIN employees m

ON e.manager\_id = m.emp\_id;

# Explanation:

- e is an alias for employees (as employee).
- m is another alias for the same table (as manager).
- The join links an employee to their manager using manager\_id = emp\_id.

# **Data Analysis/Scenario-Based Questions**

# 21. How would you design a database to store credit card

# transaction data?

To store credit card transaction data, we need to normalize the structure while keeping it scalable, secure, and query-efficient.

#### Suggested Schema Design:

#### 1. Customers Table

customer id (PK), name, email, phone, address

#### 2. Cards Table

card\_id (PK), customer\_id (FK), card\_number (masked), card\_type, status, issued\_date

#### 3. Merchants Table

merchant\_id (PK), name, category, location

#### 4. Transactions Table

transaction\_id (PK), card\_id (FK), merchant\_id (FK), transaction\_date, amount, currency, status, location

#### **Best Practices:**

- Mask sensitive fields (like card numbers).
- Store card number as encrypted or tokenized.
- Use partitioning on date fields for faster querying.
- Add indexes on card\_id, merchant\_id, transaction\_date.

# 22. Write a query to identify the most profitable regions based on transaction data.

Assume a transactions table:

(transaction\_id, customer\_id, amount, region, transaction\_date)

# Query to find top 3 profitable regions:

SELECT region, SUM(amount) AS total revenue FROM transactions

**GROUP BY region** 

ORDER BY total\_revenue DESC

LIMIT 3;

# Explanation:

- Aggregates transaction amounts per region.
- Orders regions by total revenue.
- Retrieves top 3 using LIMIT.

Optional: You could also calculate profit by subtracting costs (if a cost column is present).

# 23. How would you analyze customer churn using SQL?

# Step-by-step SQL approach:

# Step 1: Define churn

Let's say a churned customer is one who hasn't transacted in the **last 6 months**.

# Step 2: Sample schema

- customers(customer\_id, name, signup\_date)
- transactions(customer\_id, transaction\_date, amount)

## Step 3: Query to identify churned customers

SELECT c.customer\_id, c.name
FROM customers c
LEFT JOIN transactions t
ON c.customer\_id = t.customer\_id
AND t.transaction\_date >= CURRENT\_DATE - INTERVAL '6 months'
WHERE t.transaction\_id IS NULL;

## Step 4: Analyze churn metrics

You could extend this analysis by calculating:

- Churn rate = (Churned Customers / Total Customers) \* 100
- Monthly churn trend
- Compare churned vs. active customers in terms of average spend

# 24. Explain the difference between OLAP and OLTP databases.

Feature	OLTP (Online Transaction Processing)	OLAP (Online Analytical Processing)
Purpose	Handles real-time transactional queries	Used for analytical/reporting queries
Operations		SELECT (aggregate, group, slice, dice)
Data Structure	HIGNIV NORMALIZEG ( SINE)	De-normalized (star/snowflake schema)
Speed	Fast for read/write of single rows	Fast for complex analytical queries
Examples	Banking systems, e-commerce order processing	Business intelligence, dashboards, sales trends
Users	Clerks, DBAs	Analysts, Data Scientists
Backup/Recovery	Essential and frequent	Less frequent

- In short:
  - **OLTP** = operational, fast, real-time transactions.
  - **OLAP** = analytical, slow-changing, historical data.

# 25. How would you determine the Average Revenue Per User (ARPU) from transaction data?

- ◆ ARPU = Total Revenue / Total Number of Users
- Assume a transactions table:

(transaction\_id, customer\_id, amount, transaction\_date)

SQL Query:

**SELECT** 

SUM(amount) \* 1.0 / COUNT(DISTINCT customer\_id) AS ARPU

#### FROM transactions;

## **Explanation:**

- SUM(amount) gets total revenue.
- COUNT(DISTINCT customer\_id) counts unique users.
- Multiply by 1.0 to ensure float division.

You can also compute monthly ARPU by grouping by month.

**SELECT** 

DATE\_TRUNC('month', transaction\_date) AS month, SUM(amount) \* 1.0 / COUNT(DISTINCT customer\_id) AS monthly\_arpu

FROM transactions

**GROUP BY month** 

ORDER BY month;

# 26. Describe a scenario where you would use a LEFT JOIN instead of an INNER JOIN.

#### Use LEFT JOIN when:

You want **all records from the left table**, even if there's **no matching record** in the right table.

#### Real-life Scenario:

**Question**: List all customers and their transactions — even if they haven't made any.

Query:

SELECT c.customer\_id, c.name, t.transaction\_id, t.amount

FROM customers c

LEFT JOIN transactions t

ON c.customer\_id = t.customer\_id;

# Why LEFT JOIN?

- Shows **all customers**, including those with **no transactions** (returns NULLs for those).
- Using INNER JOIN would exclude customers with zero activity.

# 27. Write a query to calculate YoY (Year-over-Year) growth for a set of transactions.

Assume a table named transactions with: (customer\_id, transaction\_date, amount)

## Step 1: Extract year-wise revenue

**SELECT** 

EXTRACT(YEAR FROM transaction\_date) AS year,

SUM(amount) AS total\_revenue

**FROM transactions** 

GROUP BY EXTRACT(YEAR FROM transaction\_date);

■ Step 2: Calculate YoY Growth using a CTE and Self-Join

WITH yearly\_revenue AS (

```
SELECT
EXTRACT(YEAR FROM transaction_date) AS year,
SUM(amount) AS total_revenue
FROM transactions
GROUP BY EXTRACT(YEAR FROM transaction_date)
)
SELECT
curr.year AS current_year,
curr.total_revenue,
prev.total_revenue AS previous_year_revenue,
ROUND(((curr.total_revenue - prev.total_revenue) / prev.total_revenue) * 100, 2) AS
yoy_growth_percent
FROM yearly_revenue curr
LEFT JOIN yearly_revenue prev
ON curr.year = prev.year + 1;
```

# Explanation:

- Joins each year to its previous year.
- Computes YoY growth as a percentage.

# 28. How would you implement fraud detection using transactional data?

Fraud detection typically involves pattern recognition, anomaly detection, and rule-based filtering.

# Possible SQL-Based Checks:

# Type Rule ◆ Unusual Amounts Flag transactions > 3x average amount of that user ◆ Rapid Repeats Detect multiple transactions from same user within seconds ◆ Location Mismatch Transactions from different countries within a short time ◆ Card Sharing Same card used by different customers or IPs

# Example Query – Unusual high amount per user:

```
WITH avg_txn AS (
    SELECT customer_id, AVG(amount) AS avg_amount
    FROM transactions
    GROUP BY customer_id
)
SELECT t.*
FROM transactions t
JOIN avg_txn a
    ON t.customer_id = a.customer_id
WHERE t.amount > 3 * a.avg_amount;
```

# 29. Write a query to find customers who have used more than 2 credit cards for transactions in a given month.

Assume a transactions table:

(customer\_id, card\_id, transaction\_date)

#### Query:

SELECT customer\_id,

TO\_CHAR(transaction\_date, 'YYYY-MM') AS txn\_month, COUNT(DISTINCT card id) AS cards used

**FROM transactions** 

GROUP BY customer\_id, TO\_CHAR(transaction\_date, 'YYYY-MM')
HAVING COUNT(DISTINCT card id) > 2;

#### Explanation:

- Groups by customer\_id and month.
- Counts distinct card\_id used.
- Filters where more than 2 cards were used in a month.

# 30. How would you approach a business problem where you need to analyze the spending patterns of premium customers?

## **■ Step-by-Step Structured Approach:**

#### Step 1: Understand the Objective

- Clarify with stakeholders what **"spending pattern"** means.
  - o Is it frequency, amount, category, channel, or timing?
- Define "premium customer":
  - Based on credit score, card tier (e.g., Platinum, Centurion), monthly spend threshold, etc.

# Step 2: Data Collection

- Gather relevant datasets:
  - Customer table (ID, tier, demographics)
  - Transactions table (amount, date, category, location)
  - Cards table (card\_type, limits, activation)

# Step 3: Data Cleaning & Preparation

- Handle missing values and outliers.
- Filter only **premium customers** using defined criteria.
- Enrich data (e.g., categorize merchant types or locations).

# Step 4: Exploratory Data Analysis (EDA)

Use SQL/Python/Power BI to derive insights like:

Focus Area	Example Analysis
x Spend Amount	Average monthly/yearly spend
Time Trends	Seasonality or weekly spending behavior
Categories	Where they spend most (Travel, Dining, Shopping)
Geography	City or region-wise behavior

Focus Area	Example Analysis
Trends	Is their spend increasing/decreasing YoY?

## Step 5: Segmentation

- Use clustering or thresholds to group premium customers into:
  - High spenders
  - Frequent spenders
  - Category loyalists (e.g., only travel)
- Identify anomalies or subgroups with unique patterns.

### Step 6: Business Recommendations

- Personalize rewards or offers based on their dominant categories.
- Enhance retention strategies for segments showing decline.
- Promote premium card upgrades based on usage patterns.

## Bonus: Sample SQL Query

```
Get top 3 spending categories of premium customers monthly:

SELECT customer_id,

DATE_TRUNC('month', transaction_date) AS txn_month,

category,

SUM(amount) AS total_spend

FROM transactions

WHERE customer_id IN (

SELECT customer_id FROM customers WHERE tier = 'Premium'
)

GROUP BY customer_id, txn_month, category

ORDER BY customer_id, txn_month, total_spend DESC;
```