🚀 PwC

Data Analyst Interview Experience

💰 CTC: ₹18 LPA

💾 SQL Questions

💾 Table Creation

We'll create a table named sales to store transaction details.

```
CREATE TABLE sales (
    sale_id INT AUTO_INCREMENT PRIMARY KEY,
    product_id INT,
    product_name VARCHAR(100),
    category VARCHAR(50),
    sale_date DATE,
    quantity INT,
```

```
  revenue DECIMAL(10,2)
);
```

📥 Insert Sample Data

```
INSERT INTO sales (product_id, product_name, category, sale_date, quantity, revenue)
VALUES
(1, 'Laptop A', 'Electronics', '2024-01-15', 10, 50000),
(1, 'Laptop A', 'Electronics', '2024-02-15', 12, 60000),
(1, 'Laptop A', 'Electronics', '2024-03-10', 8, 40000),
(2, 'Mobile B', 'Electronics', '2024-01-20', 15, 30000),
(2, 'Mobile B', 'Electronics', '2024-02-22', 20, 40000),
(2, 'Mobile B', 'Electronics', '2024-03-25', 5, 10000),
(3, 'Shirt C', 'Clothing', '2024-01-12', 25, 25000),
(3, 'Shirt C', 'Clothing', '2024-02-18', 30, 30000),
(3, 'Shirt C', 'Clothing', '2024-03-21', 15, 15000),
(4, 'Shoes D', 'Footwear', '2024-01-11', 5, 10000),
(4, 'Shoes D', 'Footwear', '2024-02-15', 7, 14000),
(5, 'Watch E', 'Accessories', '2024-03-05', 10, 20000);
```

🧩 Q1️⃣ : Find the cumulative revenue by month for each product category

✅ Solution:

```
SELECT
  category,
  DATE_FORMAT(sale_date, '%Y-%m') AS sale_month,
  SUM(revenue) AS monthly_revenue,
```

SUM(SUM(revenue)) OVER (PARTITION BY category ORDER BY DATE_FORMAT(sale_date, '%Y-%m')) AS cumulative_revenue

FROM sales

GROUP BY category, sale_month

ORDER BY category, sale_month;

💡 **Explanation:**

- DATE_FORMAT(sale_date, '%Y-%m') extracts the month and year.
- The first SUM(revenue) gives monthly revenue.
- The **window function** SUM(SUM(revenue)) OVER (...) computes cumulative revenue over time for each category.

🧩 Q 2️⃣ : Retrieve the top 5 products by sales volume, excluding products with zero sales in the past 3 months

✅ **Solution:**

SELECT

product_name,

SUM(quantity) AS total_sales

FROM sales

WHERE sale_date >= DATE_SUB(CURDATE(), INTERVAL 3 MONTH)

GROUP BY product_name

HAVING total_sales > 0

ORDER BY total_sales DESC

LIMIT 5;

💡 **Explanation:**

- DATE_SUB(CURDATE(), INTERVAL 3 MONTH) filters the last 3 months' data.
- SUM(quantity) computes total sales volume.

- HAVING total_sales > 0 removes products with zero recent sales.
- ORDER BY total_sales DESC LIMIT 5 picks the **top 5 products**.

🧩 Q2️⃣ Using Window Functions

💾 Table Reminder

We're using the same sales table and data as before 👇

```
CREATE TABLE sales (
  sale_id INT AUTO_INCREMENT PRIMARY KEY,
  product_id INT,
  product_name VARCHAR(100),
  category VARCHAR(50),
  sale_date DATE,
  quantity INT,
  revenue DECIMAL(10,2)
);
```

✅ Solution Using Window Functions

```
WITH recent_sales AS (
  SELECT
    product_name,
    SUM(quantity) AS total_sales
  FROM sales
  WHERE sale_date >= DATE_SUB(CURDATE(), INTERVAL 3 MONTH)
  GROUP BY product_name
),
```

```
ranked_sales AS (
  SELECT
    product_name,
    total_sales,
    RANK() OVER (ORDER BY total_sales DESC) AS sales_rank
  FROM recent_sales
  WHERE total_sales > 0
)
SELECT
  product_name,
  total_sales,
  sales_rank
FROM ranked_sales
WHERE sales_rank <= 5;
```

🔍 **Explanation:**

Step	Description
1. recent_sales CTE	Filters only data from the last 3 months and calculates total quantity sold per product.
2. ranked_sales CTE	Applies a window function RANK() to rank products based on their total sales volume.
3. Final SELECT	Retrieves the **top 5 ranked** products, ensuring only those with **positive sales** are included.

🧠 **Alternative Using SUM() OVER() Directly (Single Query)**

If you want to use a **pure window-function approach** (no separate grouping), here's another valid version 👇

```
SELECT
  product_name,
  SUM(quantity) OVER (PARTITION BY product_name) AS total_sales,
  RANK() OVER (ORDER BY SUM(quantity) OVER (PARTITION BY product_name) DESC) AS sales_rank
FROM sales
WHERE sale_date >= DATE_SUB(CURDATE(), INTERVAL 3 MONTH)
GROUP BY product_name
HAVING total_sales > 0
ORDER BY sales_rank
LIMIT 5;
```

📊 **Key Window Functions Used**

- **SUM() OVER(PARTITION BY ...)** → calculates total sales per product.
- **RANK() OVER(ORDER BY ...)** → assigns ranking based on total sales volume.

🧩 Q3️⃣ Identify customers who made purchases in two or more consecutive months

---

💾 **Table Creation**

```
CREATE TABLE transactions (
  transaction_id INT AUTO_INCREMENT PRIMARY KEY,
  customer_id INT,
  purchase_date DATE,
```

```
  amount DECIMAL(10,2)
);
```

📥 Sample Data

```
INSERT INTO transactions (customer_id, purchase_date, amount) VALUES
(101, '2024-01-10', 500),
(101, '2024-02-12', 700),
(101, '2024-04-05', 200),
(102, '2024-01-25', 1000),
(102, '2024-03-15', 1500),
(103, '2024-02-11', 600),
(103, '2024-03-09', 800),
(103, '2024-04-08', 900),
(104, '2024-03-22', 300);
```

✅ Solution (Using Window Functions)

```
WITH month_data AS (
  SELECT
    customer_id,
    DATE_FORMAT(purchase_date, '%Y-%m') AS purchase_month
  FROM transactions
  GROUP BY customer_id, purchase_month
),
month_diff AS (
  SELECT
    customer_id,
```

```
    purchase_month,
    LAG(purchase_month) OVER (PARTITION BY customer_id ORDER BY purchase_month) AS prev_month
  FROM month_data
)
SELECT DISTINCT customer_id
FROM month_diff
WHERE TIMESTAMPDIFF(MONTH, STR_TO_DATE(CONCAT(prev_month, '-01'), '%Y-%m-%d'),
                   STR_TO_DATE(CONCAT(purchase_month, '-01'), '%Y-%m-%d')) = 1;
```

💡 **Explanation**

Step	Description
month_data	Extracts each customer's unique purchase months.
LAG()	Fetches the previous month for each customer.
TIMESTAMPDIFF(MONTH, ...)	Checks if the difference between consecutive months = 1.
DISTINCT	Returns customers with **two or more consecutive month purchases.**

🧩 Q4️⃣ Calculate the Monthly User Retention Rate

💾 **Table Creation**

```
CREATE TABLE user_logins (
  user_id INT,
  login_date DATE
```

);

📥 **Sample Data**

```
INSERT INTO user_logins (user_id, login_date) VALUES
(1, '2024-01-10'),
(1, '2024-02-05'),
(2, '2024-01-15'),
(2, '2024-03-01'),
(3, '2024-02-10'),
(3, '2024-03-12'),
(4, '2024-01-22'),
(5, '2024-03-15'),
(6, '2024-02-20'),
(6, '2024-03-25');
```

✅ **Solution (Using Window Functions)**

```
WITH month_users AS (
  SELECT
    DATE_FORMAT(login_date, '%Y-%m') AS month,
    user_id
  FROM user_logins
  GROUP BY month, user_id
),
retention AS (
  SELECT
    m1.month AS current_month,
```

```
    COUNT(DISTINCT m1.user_id) AS active_users,
    COUNT(DISTINCT m2.user_id) AS retained_users
  FROM month_users m1
  LEFT JOIN month_users m2
    ON m1.user_id = m2.user_id
   AND DATE_ADD(STR_TO_DATE(CONCAT(m2.month, '-01'), '%Y-%m-%d'), INTERVAL 1 MONTH) = STR_TO_DATE(CONCAT(m1.month, '-01'), '%Y-%m-%d')
  GROUP BY m1.month
)
SELECT
  current_month,
  retained_users,
  active_users,
  ROUND(retained_users / active_users * 100, 2) AS retention_rate
FROM retention
ORDER BY current_month;
```

💡 **Explanation**

Step	Description
month_users	Extracts unique users for each month.
LEFT JOIN	Checks which users logged in again next month.
retention	Counts current and retained users per month.
Final Output	Shows **monthly user retention %** = retained / active × 100.

🧩 Q5️⃣ Find the Nth Highest Salary (Dynamic N)

💾 Table Creation

```
CREATE TABLE employees (
  emp_id INT,
  emp_name VARCHAR(50),
  salary DECIMAL(10,2)
);
```

📥 Sample Data

```
INSERT INTO employees (emp_id, emp_name, salary) VALUES
(1, 'Amit', 90000),
(2, 'Ravi', 85000),
(3, 'Neha', 95000),
(4, 'Sonal', 70000),
(5, 'Raj', 85000),
(6, 'Deepak', 60000);
```

✅ Solution (Dynamic N)

Suppose you pass a variable @N = 3 for the **3rd highest salary**.

```
SET @N := 3;

SELECT emp_name, salary
FROM (
  SELECT
    emp_name,
    salary,
```

```
    DENSE_RANK() OVER (ORDER BY salary DESC) AS salary_rank
  FROM employees
) ranked
WHERE salary_rank = @N;
```

💡 **Explanation**

Step	Description
DENSE_RANK()	Assigns rank to each unique salary (ties handled properly).
@N	Parameter to dynamically specify which rank you want (e.g., 2nd, 3rd, etc.).
Final WHERE	Filters only employees with that rank.

✅ **Example Output (for @N = 3)**

emp_name	salary
Ravi	85000
Raj	85000

🧠 Q6️⃣ Explain how indexing works in SQL and how to choose the right columns for optimal performance

---

📘 **What is an Index?**

An **index** in SQL is a **data structure (like a book's index)** that allows the database engine to quickly locate and retrieve rows without scanning the entire table.

It's usually implemented as a **B-Tree** or **Hash** structure depending on the database engine.

⚙️ How Indexing Works

When a query runs with a WHERE, JOIN, or ORDER BY clause —
SQL looks up the index instead of scanning all rows sequentially (known as a *full table scan*).

Example:

CREATE INDEX idx_employee_deptid ON employees(department_id);

When you run:

SELECT *

FROM employees

WHERE department_id = 10;

➡️ Instead of reading every row, the database goes directly to department_id = 10 using the index — drastically improving performance.

⚡ When Index Helps

- WHERE filters on **highly selective** columns (many unique values).
- Columns used in **JOIN conditions** (to match keys faster).
- Columns frequently used in **ORDER BY**, **GROUP BY**, or **DISTINCT** clauses.

⚠️ When Not to Use Indexes

- Columns with **low selectivity** (e.g., gender, status flags like Active/Inactive).
- **Small tables**, where full scan is already fast.
- Columns that are **frequently updated**, since each update must also update the index (adds overhead).

✅ Best Practices for Choosing Index Columns

Scenario	Recommended Index Type
Searching with WHERE	Index the filtered columns
Joining tables	Index the joining keys (foreign & primary)
Sorting/grouping	Index the columns used in ORDER BY or GROUP BY
Composite queries	Use **composite index** (e.g., (country, city)) — order matters
Frequent updates	Avoid indexing frequently modified columns

💡 **Analogy**

Think of an **index** as a *table of contents* in a book — it doesn't hold the actual data, but points you directly to where the data lives.

🧩 Q7️⃣ Describe the differences between LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN — and when to use each

---

🧱 **Setup Example**

Two tables:

Customers

customer_id	customer_name
1	Rahul
2	Sneha
3	Arjun

Orders

order_id	customer_id	amount
101	1	500
102	2	700
103	4	900

🔹 LEFT JOIN

Returns **all rows from the left table** and matching rows from the right table.
If there's no match, the right side shows **NULL**.

```
SELECT c.customer_name, o.amount

FROM Customers c

LEFT JOIN Orders o

ON c.customer_id = o.customer_id;
```

Result:

customer_name	amount
Rahul	500
Sneha	700
Arjun	NULL

Use When:
You want **all customers**, even if they haven't placed an order.

🔸 RIGHT JOIN

Returns **all rows from the right table** and matching rows from the left table.
If there's no match, the left side shows **NULL**.

```
SELECT c.customer_name, o.amount

FROM Customers c
```

RIGHT JOIN Orders o

ON c.customer_id = o.customer_id;

Result:

customer_name	amount
Rahul	500
Sneha	700
NULL	900

Use When:
You want **all orders**, even if they don't have a corresponding customer.

- **FULL OUTER JOIN**

Returns **all rows from both tables**, matching where possible.
If there's no match, missing side shows **NULL**.

(MySQL doesn't natively support FULL OUTER JOIN; you can simulate it with UNION.)

SELECT c.customer_name, o.amount

FROM Customers c

LEFT JOIN Orders o ON c.customer_id = o.customer_id

UNION

SELECT c.customer_name, o.amount

FROM Customers c

RIGHT JOIN Orders o ON c.customer_id = o.customer_id;

Result:

customer_name	amount
Rahul	500
Sneha	700

customer_name	amount
Arjun	NULL
NULL	900

Use When:
You need a **complete view** — showing both customers without orders and orders without customers.

🧠 **Summary Table**

JOIN Type	Returns	When to Use
LEFT JOIN	All rows from left + matching right	Retain all from left table
RIGHT JOIN	All rows from right + matching left	Retain all from right table
FULL OUTER JOIN	All rows from both	Need complete overview of both tables

---

🧩 Q8️⃣ What's the difference between a Temporary Table and a View, and when should each be used?

🧱 1️⃣ What is a Temporary Table?

A **Temporary Table** is a table that stores **data physically** for a short duration — usually within a **single session or transaction**.
Once the session ends, the table and its data are automatically deleted.

🧩 Example:

```
CREATE TEMPORARY TABLE temp_sales AS
```

```
SELECT product_id, SUM(revenue) AS total_revenue
FROM sales
GROUP BY product_id;
```

```
SELECT * FROM temp_sales;
```

⚙️ **Key Properties**

Property	Description
Physical Storage	Stored temporarily in the database.
Lifetime	Exists only for the current session.
Performance	Useful for heavy intermediate computations.
Modification	You can INSERT, UPDATE, or DELETE data.
Isolation	Accessible only to the session that created it.

✅ **When to Use**

- When you need to **store intermediate results** for reuse in the same query/session.
- When performing **complex ETL operations** with multiple steps.
- When you need **better performance** in multi-step analytical queries.

🧠 2️⃣ **What is a View?**

A **View** is a **virtual table** that stores only the **SQL query**, not the data itself.
When you query a view, SQL executes the underlying query dynamically.

🧩 **Example:**

```
CREATE VIEW high_value_customers AS
SELECT customer_id, SUM(amount) AS total_spent
FROM transactions
GROUP BY customer_id
```

HAVING total_spent > 10000;

SELECT * FROM high_value_customers;

⚙️ **Key Properties**

Property	Description
Physical Storage	No physical data storage (except for indexed/materialized views).
Lifetime	Permanent (until explicitly dropped).
Performance	Depends on the base query — not stored data.
Modification	Usually read-only (some can be updatable).
Accessibility	Accessible by multiple users.

✅ **When to Use**

- To **simplify complex SQL queries** for reusability.
- To **enhance data security** (restrict access to certain columns).
- To **create consistent business logic** used by multiple analysts.
- To **abstract underlying schema changes** from end users.

⚖️ 3️⃣ **Comparison Table**

Feature	Temporary Table	View
Storage Type	Physically stores data (temporarily)	Virtual – stores only query
Persistence	Exists during session	Permanent until dropped
Data Modification	Supports DML (INSERT, UPDATE, DELETE)	Usually read-only
Performance	Faster for large data manipulations	Depends on base tables
Use Case	Intermediate computations	Reusable SQL logic

Feature	Temporary Table	View
Accessibility	Session-specific	Global (accessible to all with permission)

🧭 🔽 When to Choose What

Situation	Choose
Need to hold intermediate query results temporarily	🟢 **Temporary Table**
Need a reusable, logical view of data	🟣 **View**
Need to simplify reporting queries for others	🟣 **View**
Need to speed up complex transformations in multi-step ETL	🟢 **Temporary Table**

💡 Analogy

- **Temporary Table** → Like a *scratchpad*: You write intermediate notes and throw it away after use.
- **View** → Like a *saved formula*: You can use it again and again without retyping it.

🐍 Python Questions

🐍 Q1️⃣ Find All Unique Pairs in a List That Sum to a Given Target

---

✅ Problem Statement

Given a list of integers, find all **unique pairs** that sum up to a specific **target value**.

💻 Code

```
def find_pairs(nums, target):
```

```
    seen = set()
    pairs = set()

    for num in nums:
        complement = target - num
        if complement in seen:
            pairs.add(tuple(sorted((num, complement))))
        seen.add(num)

    return list(pairs)

# Example
nums = [2, 4, 3, 5, 7, 8, 9, -1]
target = 10
print(find_pairs(nums, target))
```

📃 **Output**

[(2, 8), (3, 7), (5, 5), (1, 9)]

(depends on data; unique and order-independent)

💡 **Explanation**

- We use a **set** to store seen numbers.
- For each number, check if its **complement (target - num)** was already seen.
- sorted() ensures (2, 8) and (8, 2) are treated as the same pair.
- set() avoids duplicate pairs.

🔁 Q2️⃣ Check if a String Is a Palindrome (Ignoring Spaces, Punctuation & Case)

💻 Code

```
import re


def is_palindrome(s):
    cleaned = re.sub(r'[^A-Za-z0-9]', '', s).lower()
    return cleaned == cleaned[::-1]


# Example
text = "A man, a plan, a canal: Panama"
print(is_palindrome(text))
```

📜 Output

```
True
```

💡 Explanation

Step	Description
re.sub(r'[^A-Za-z0-9]', '', s)	Removes all non-alphanumeric characters
.lower()	Ensures case-insensitivity
[::-1]	Reverses the string for comparison

✅ Works even with spaces, punctuation, or mixed cases.

🧠 Q3️⃣ Deep Copy vs Shallow Copy in Python

🧩 Concept Overview

Type	Definition	Effect
Shallow Copy	Creates a new object, but **references** the inner objects	Changes in nested elements affect both copies
Deep Copy	Creates a completely **independent clone** of all objects recursively	Changes do **not** reflect in the original

---

💻 Example

```
import copy

original = [[1, 2], [3, 4]]

shallow_copy = copy.copy(original)
deep_copy = copy.deepcopy(original)

shallow_copy[0][0] = 99

print("Original:", original)
print("Shallow Copy:", shallow_copy)
print("Deep Copy:", deep_copy)
```

📜 Output

```
Original: [[99, 2], [3, 4]]
```

Shallow Copy: [[99, 2], [3, 4]]

Deep Copy: [[1, 2], [3, 4]]

💡 When to Use

Situation	Use
When dealing with **immutable objects** (e.g., tuples, ints, strings)	Shallow Copy
When working with **nested mutable structures** (lists, dicts, etc.)	Deep Copy

---

⚙️ Q4️⃣ What Are Decorators in Python? Show an Example

🧩 Definition

A **decorator** is a function that takes another function as input, **adds extra functionality**, and returns it — **without modifying its code directly.**

💻 Example

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Function '{func.__name__}' started...")
        result = func(*args, **kwargs)
        print(f"Function '{func.__name__}' ended.")
        return result
    return wrapper


@log_decorator
def add(a, b):
```

```
    return a + b


print(add(5, 3))
```

📃 **Output**

```
Function 'add' started...
Function 'add' ended.
8
```

💡 **Explanation**

Component	Description
@log_decorator	Applies the decorator to the add() function
wrapper()	Defines additional behavior before and after the function call
func(*args, **kwargs)	Executes the original function
Useful for	Logging, authentication, timing, debugging, access control

---

🧠 **Real-World Use Cases**

- Logging function calls in large codebases.
- Measuring execution time of functions.
- Checking permissions before executing secure functions.
- Caching results of expensive computations.

CASE STUDY

Goal

Estimate **annual smartphone units sold in India** (new retail sales in a year).

Approach (3-step, BA style)

1. **Top-down population → users → replacements** (primary estimate).
2. **Bottom-up households → devices per household → replacements** (sanity check).
3. **Triangulate + sensitivity range** and business implications.

I'll show every numeric step.

Assumptions (explicit)

- India population (2025, rounded estimate): **1,420,000,000** (1.42 billion).
- Share of population aged 15+ (likely phone-owning cohort): **65%**.
- Smartphone penetration among 15+ (people who already own a smartphone): **70%**.
- Average replacement / upgrade cycle: **3 years** (i.e., each owned smartphone is replaced once every 3 years on average).
- Annual net **new users** (first-time smartphone adopters, churn offset): **20,000,000** (20M).
- Average household size: **4.5** persons.
- Average smartphones per household (installed base): **2**.

(These are assumptions you should state in an interview; we'll test sensitivity later.)

---

1) Top-down calculation (digit-by-digit)

1. Total adults (15+):

- 1,420,000,000 × 0.65
  = 1,420,000,000 × 65 / 100
  = (1,420,000,000 × 65) / 100
  = 92,300,000,000 / 100 → **923,000,000** (923 million adults)

2. Current smartphone users (15+):

- 923,000,000 × 0.70
  = 923,000,000 × 70 / 100
  = (923,000,000 × 70) / 100
  = 64,610,000,000 / 100 → **646,100,000** (646.1 million smartphone users)

3. Annual replacement demand (assuming 3-year cycle):

- 646,100,000 ÷ 3
  = 215,366,666.666… → round → **215,370,000** (~215.4 million)

4. Add net new users (first-time buyers):

- 215,370,000 + 20,000,000 = **235,370,000**

Top-down estimate ≈ 235 million units/year.

---

2) Bottom-up (household) calculation (digit-by-digit)

1. Number of households:

- 1,420,000,000 ÷ 4.5
  = 315,555,555.555… → round → **315,555,556** households

2. Installed smartphones (avg 2 per household):

- 315,555,556 × 2 = **631,111,112** devices (installed base)

3. Annual replacement (3-year cycle):

- 631,111,112 ÷ 3 = 210,370,370.666… → round → **210,370,371**

4. Add household-level new devices (household formation, extra devices), approximate +10,000,000:

- 210,370,371 + 10,000,000 = **220,370,371**

Bottom-up estimate ≈ 220 million units/year.

---

3) Triangulation & Sensitivity

- Top-down → **~235M**
- Bottom-up → **~220M**

They're close. Now vary key levers:

- If replacement cycle = **2.5 years**:
    - Using top-down base users 646,100,000 ÷ 2.5 = 258,440,000 → +20M → **≈278M**
- If replacement cycle = **3.5 years**:
    - 646,100,000 ÷ 3.5 = 184,600,000 → +20M → **≈205M**

So plausible **range ≈ 200M – 280M** depending on upgrade frequency and growth. A conservative 90% confidence band: **~180M – 260M**.

Final quick answer (interview soundbite)

Estimated annual smartphone sales in India ≈ 220–240 million units per year.
A single point estimate I'd present: **~235 million units/year**, with a plausible range **180–260M** depending on replacement cycle and adoption dynamics.

Business implications (for a BA / product / sales team)

- **Large, mature replacement market**: most volume comes from replacements, not first-time buyers. Focus on upgrade incentives and trade-in programs.
- **Segment strategy**: low/mid-tier volumes dominate (price sensitive); premium segment smaller but high ARPU — tailor marketing and finance offers.
- **Channel mix & seasonality**: expect spikes during festival seasons and new model launches — optimize inventory and campaigns around these.
- **Used/Refurbished market**: sizeable — impacts new sales; consider certified-refurbished programs or partnerships.
- **Aftermarket & services**: warranties, insurance, accessories, and payment/EMI are growth levers.

Recommended next steps / data to refine estimate

1. Pull **official population** and age-breakdown (Census / UN / government projections).

2. Get **smartphone penetration by age** and urban/rural split (GSMA, TRAI, industry reports).
3. Find **replacement cycle estimates** and upgrade intent surveys (Counterpoint, IDC, Canalys, Gartner).
4. Analyze **channel sales data** (retailer POS, e-commerce partners) for seasonality and tier split.
5. Incorporate **used device flows** (trade-in, refurb) if you want net-new vs total sold.

Goal

Estimate **daily revenue generated by roadside tea stalls across India** (total cup-sales revenue per day).

Key assumptions (explicit — state these in an interview)

- **Number of roadside tea stalls:** base = **12,000,000** (useful benchmark; we'll show low/high alternatives).
- **Average customers per stall per day:** base = **80** customers.
- **Average spend per customer (mostly a cup of tea, sometimes snack):** base = **₹15**.

(Why these? roadside stalls are extremely numerous and small; average customers vary by location/time — so we'll run low/base/high scenarios.)

Step-by-step arithmetic (digit-by-digit)

1) Revenue per stall (base)

- customers × price = 80 × ₹15
    - 80 × 10 = 800
    - 80 × 5 = 400
    - 800 + 400 = **₹1,200** per stall per day

2) Total daily revenue (base)

- stalls × revenue_per_stall = 12,000,000 × 1,200
    - 12,000,000 × 1,000 = 12,000,000,000
    - 12,000,000 × 200 = 2,400,000,000
    - 12,000,000,000 + 2,400,000,000 = **₹14,400,000,000** per day

Convert formats:

- = **₹14.4 billion / day**
- = **₹1,440 crore / day**

Base estimate → ₹14.4 billion (₹1,440 crore) per day.

Sensitivity (low / high scenarios)

Low scenario (conservative)

- stalls = **8,000,000**
- customers = **50**
- price = **₹12**

Per-stall revenue: 50 × 12 = (50×10=500) + (50×2=100) = **₹600**
Total: 8,000,000 × 600 = (8,000,000×600) = **₹4,800,000,000** = **₹4.8 billion / day** = **₹480 crore/day**

High scenario (optimistic)

- stalls = **16,000,000**
- customers = **120**
- price = **₹20**

Per-stall revenue: 120 × 20 = (120×2×10) = 2400 = **₹2,400**
Total: 16,000,000 × 2,400

- 16,000,000 × 2,000 = 32,000,000,000
- 16,000,000 × 400 = 6,400,000,000
- Total = 38,400,000,000 → **₹38.4 billion / day** = **₹3,840 crore/day**

Final answer (range + single-point)

- **Plausible daily revenue range: ₹4.8 billion — ₹38.4 billion** per day.
- **Best single point (base): ≈ ₹14.4 billion per day** (≈ **₹1,440 crore/day**).

Business implications (for a BA / product / investor)

- **Large informal market:** big aggregated spending despite tiny tickets per customer — attractive for FMCG suppliers (tea, milk, sugar), last-mile logistics, and micro-loans for vendors.
- **High fragmentation:** distribution & brand penetration require localized strategies (bulk resale, sachet pricing, vendor financing).
- **Monetization opportunities:** packaged snacks, hygienic/ready-mix tea blends, digital payments adoption, loyalty programs, micro-franchising, and B2B supplies.
- **Seasonality & location:** revenue concentrated in urban/commute locations and breakfast/evening peaks — optimize inventory/time-based promos.

How to refine this estimate (data to gather)

1. **Authoritative counts** of street vendors / tea stalls by government labor surveys or municipal records.
2. **POS / e-commerce / aggregator** data (local delivery apps that list tea vendors).
3. **Field surveys** or sample audits in representative cities/rural areas for avg customers and price.
4. **Industry reports** from FMCG/tea associations (monthly supply to vendors, tea-leaf sales).
5. **Seasonal adjustments** (monsoon, festivals, school/commute patterns).