KPMG Data Analyst Interview Questions (0-3 Years)

SQL Questions

1. Write a SQL query to find the second-highest salary from an employee table.

Given an employee table with columns:

id | name | salary

---+-------+--------

1 | John | 50000

2 | Alice | 60000

3 | Bob | 70000

4 | David | 70000

5 | Emma | 80000

Solution 1: Using LIMIT with OFFSET (MySQL, PostgreSQL)

```
SELECT DISTINCT salary
FROM employee
ORDER BY salary DESC
```

LIMIT 1 OFFSET 1;

- ORDER BY salary DESC: Sorts the salaries in descending order.
- LIMIT 1 OFFSET 1: Skips the highest salary and returns the second-highest salary.

Solution 2: Using MAX() with WHERE

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

- The inner query finds the highest salary.
- The outer query finds the highest salary *below* the maximum, which is the second-highest.

Solution 3: Using DENSE_RANK() (Works in SQL Server, PostgreSQL, MySQL 8+)

```
SELECT salary
FROM (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS salary_rank
  FROM employee
) ranked_salaries
WHERE salary_rank = 2;
```

- DENSE_RANK() assigns ranks to unique salaries.
- The outer query filters the row where rank is 2 (second-highest salary).

---

2. How do you optimize a slow-running SQL query?

Here are key optimization techniques:

✅ **1. Use Indexing Efficiently**

- Index the columns used in WHERE, JOIN, and ORDER BY.
- Example:

```
CREATE INDEX idx_salary ON employee(salary);
```

✅ **2. Avoid SELECT *, Use Specific Columns**

- Instead of:

```
SELECT * FROM employee;
```

- Use:

```
SELECT id, name FROM employee;
```

✅ **3. Optimize Joins Using Proper Indexing**

- Ensure indexed columns are used in JOIN conditions.

✅ **4. Use EXPLAIN or EXPLAIN ANALYZE**

- Helps analyze query execution.

```
EXPLAIN SELECT * FROM employee WHERE salary > 50000;
```

✅ **5. Avoid Using Functions in WHERE Clause**

- **Bad:**

```
SELECT * FROM employee WHERE YEAR(hire_date) = 2023;
```

- **Good:**

```
SELECT * FROM employee WHERE hire_date BETWEEN '2023-01-01' AND '2023-12-31';
```

✅ **6. Optimize Subqueries and Use Joins Instead**

- **Bad:**

```
SELECT name FROM employee WHERE salary = (SELECT MAX(salary) FROM employee);
```

- **Good:**

```
SELECT name FROM employee e JOIN (SELECT MAX(salary) AS max_salary FROM employee) m ON e.salary = m.max_salary;
```

✅ **7. Use Proper Data Types & Partitioning for Large Tables**

- Use **partitioning** for large datasets to improve query performance.

---

3. Explain the difference between JOIN and UNION.

Feature	JOIN	UNION
Purpose	Combines data from multiple tables based on a relationship.	Combines results of two or more queries into a single result set.
Output	Returns columns from multiple tables.	Returns rows from multiple queries (same column structure).
Condition	Uses ON or USING to define relationships.	Requires same number of columns in both queries.
Duplicate Rows	Can return duplicates if INNER JOIN, LEFT JOIN, etc.	UNION removes duplicates (UNION ALL keeps them).
Example	SELECT e.id, e.name, d.department FROM employee e JOIN department d ON e.dept_id = d.id;	SELECT name FROM employee UNION SELECT name FROM contractor;

---

4. Write a query to find duplicate records in a table.

Given a table employees:

id | name | department

---+-------+-----------

1 | John | HR

2 | Alice | IT

3 | John | HR

4 | Bob | IT

5 | John | HR

Solution: Using GROUP BY with HAVING

To find duplicates based on the name and department:

```
SELECT name, department, COUNT(*) as count
FROM employees
```

```
GROUP BY name, department
HAVING COUNT(*) > 1;
```

- **GROUP BY** groups records with the same values.
- **HAVING COUNT(*) > 1** filters groups with more than one record, indicating duplicates.

To Find Complete Duplicate Rows

```
SELECT *, COUNT(*) as count
FROM employees
GROUP BY id, name, department
HAVING COUNT(*) > 1;
```

This returns:

```
name  | department | count
------+------------+------
John  | HR         | 3
```

---

5. What are window functions in SQL? Give an example.

Definition

Window functions perform calculations across a set of table rows related to the current row. Unlike aggregate functions, they do not collapse rows into a single output.

Common Window Functions

1. ROW_NUMBER(): Assigns a unique number to each row.
2. RANK(): Assigns a rank, with gaps for ties.
3. DENSE_RANK(): Assigns a rank without gaps.
4. SUM(), AVG(), COUNT(): Aggregate functions used over a window.

Example: Using ROW_NUMBER()

Given a sales table:

id | employee | sales_amount

---+----------+-------------

1 | John | 500

2 | Alice | 700

3 | Bob | 600

4 | John | 800

5 | Alice | 900

Find the top sales for each employee:

```
SELECT
  employee,
  sales_amount,
  ROW_NUMBER() OVER (PARTITION BY employee ORDER BY sales_amount DESC) as rank
FROM
  sales;
```

Output:

employee | sales_amount | rank

---------+--------------+-----

Alice | 900 | 1

Alice | 700 | 2

Bob | 600 | 1

John | 800 | 1

John | 500 | 2

Explanation:

- **PARTITION BY employee**: Resets the row numbering for each employee.
- **ORDER BY sales_amount DESC**: Orders sales in descending order for each partition.

6. How would you handle missing or null values in SQL?

Common Approaches:

1. **Use COALESCE()**:
    - Returns the first non-null value in a list.
    - Example:

```
SELECT COALESCE(phone_number, 'N/A') AS contact_number FROM employees;
```

2. **Use IS NULL / IS NOT NULL**:
    - Filters records with null or non-null values.
    - Example:

```
SELECT * FROM employees WHERE phone_number IS NULL;
```

3. **Use IFNULL() (MySQL) or NVL() (Oracle)**:
    - Similar to COALESCE() but for two values only.
    - Example:

```
SELECT IFNULL(salary, 0) AS salary FROM employees;
```

4. **Replace Nulls with Aggregate Values**:
    - Fill nulls with average, sum, or other aggregate values.
    - Example:

```
UPDATE employees
SET salary = (SELECT AVG(salary) FROM employees)
WHERE salary IS NULL;
```

5. **Use CASE Statement**:
    - Provides more control over how to handle nulls.
    - Example:

```
SELECT
  name,
```

```
  CASE
    WHEN phone_number IS NULL THEN 'No Contact'
    ELSE phone_number
  END AS contact_info
FROM employees;
```

6. **Delete Rows with Null Values**:
    - Remove incomplete records if necessary.
    - Example:

```
DELETE FROM employees WHERE phone_number IS NULL;
```

---

7. Explain the difference between HAVING and WHERE clauses.

Feature	WHERE Clause	HAVING Clause
Purpose	Filters rows before grouping.	Filters groups after GROUP BY.
Used With	SELECT, FROM, JOIN.	GROUP BY, aggregate functions (SUM(), COUNT()).
Can Use Aggregates?	❌ No (Cannot use SUM(), AVG(), etc.).	✅ Yes (Used for filtering aggregated values).
Example	SELECT * FROM sales WHERE amount > 500;	SELECT customer_id, SUM(amount) FROM sales GROUP BY customer_id HAVING SUM(amount) > 500;

✅ **Example Demonstration:**

Given a sales table:

```
id | customer_id | amount
---+------------+-------
1  | 101        | 200
2  | 102        | 400
```

3 | 101 | 300

4 | 103 | 800

5 | 102 | 600

📌 **Using WHERE (Filters before grouping)**:

```
SELECT customer_id, amount
FROM sales
WHERE amount > 300;
```

🔹 Filters out rows where amount is less than 300 before aggregation.

📌 **Using HAVING (Filters after grouping)**:

```
SELECT customer_id, SUM(amount) AS total_spent
FROM sales
GROUP BY customer_id
HAVING SUM(amount) > 500;
```

🔹 Groups data first and then filters out customers with SUM(amount) ≤ 500.

---

8. What is a Common Table Expression (CTE)? How is it different from a subquery?

Common Table Expression (CTE)

- A temporary result set defined using WITH.
- Improves query readability and reuse.
- Can be used multiple times within the same query.

✅ **Example of CTE**:

```
WITH total_sales AS (
  SELECT customer_id, SUM(amount) AS total_spent
  FROM sales
```

```
    GROUP BY customer_id
)
SELECT customer_id, total_spent
FROM total_sales
WHERE total_spent > 500;
```

- The WITH clause creates a temporary named result set (total_sales).
- The main query then filters customers who spent more than 500.

Difference Between CTE and Subquery

Feature	CTE (WITH Clause)	Subquery
Readability	✅ More readable, reusable	❌ Harder to read, especially with nesting
Reusability	✅ Can be referenced multiple times	❌ Defined once and cannot be reused
Performance	✅ Optimized for recursion & complex queries	❌ Can be slower in complex cases
Recursion Support	✅ Yes (RECURSIVE CTE)	❌ No recursion

✅ **Example of Subquery (Alternative to CTE)**:

```
SELECT customer_id, total_spent
FROM (
    SELECT customer_id, SUM(amount) AS total_spent
    FROM sales
    GROUP BY customer_id
) AS total_sales
WHERE total_spent > 500;
```

- Works but is harder to read compared to a CTE.

9. Write a SQL query to calculate the Customer Lifetime Value (CLV).

What is CLV?

Customer Lifetime Value (CLV) estimates the total revenue a business can expect from a customer over their lifetime.

Formula:

CLV=Average Purchase Value×Purchase Frequency×Customer LifespanCLV = \text{Average Purchase Value} \times \text{Purchase Frequency} \times \text{Customer Lifespan}CLV=Average Purchase Value×Purchase Frequency×Customer Lifespan

✅ **SQL Query to Calculate CLV:**

```
WITH customer_data AS (
  SELECT
    customer_id,
    SUM(amount) AS total_revenue,
    COUNT(DISTINCT order_id) AS total_orders,
    COUNT(DISTINCT YEAR(order_date)) AS years_active
  FROM sales
  GROUP BY customer_id
)
SELECT
  customer_id,
  (total_revenue / total_orders) * (total_orders / years_active) * 5 AS estimated_CLV
FROM customer_data;
```

Explanation:

- **total_revenue / total_orders** → Average purchase value.
- **total_orders / years_active** → Purchase frequency per year.

- **5** → Assumed customer lifespan (adjust based on business model).

📌 **Example Output:**

customer_id | estimated_CLV

----------+--------------

101 | 1500.00

102 | 2000.00

103 | 2500.00

---

10. What are indexes, and how do they improve query performance?

What is an Index?

- An index is a **data structure** that improves the speed of data retrieval operations on a table.
- Similar to a book index—it helps locate information quickly.

Types of Indexes in SQL

Type	Description
Primary Index	Automatically created on PRIMARY KEY.
Unique Index	Ensures column values are unique.
Composite Index	Index on multiple columns.
Full-text Index	Used for text searches (MySQL, PostgreSQL).
Clustered Index	Data is stored in sorted order (SQL Server).
Non-clustered Index	Stores pointers to data (MySQL, PostgreSQL).

✅ **Creating an Index**

CREATE INDEX idx_customer ON sales(customer_id);

✅ **Using EXPLAIN to Check Index Usage**

EXPLAIN SELECT * FROM sales WHERE customer_id = 101;

- If an index is used, it significantly reduces search time.

Performance Improvement:

Query	Without Index	With Index
SELECT * FROM sales WHERE customer_id = 101;	**Full table scan** (Slow)	**Index lookup** (Fast)
SELECT * FROM sales ORDER BY amount;	**Sorting required**	**Faster sorting**

When to Use Indexes?

✅ **Use Indexes When:**

- Columns are frequently used in WHERE, JOIN, ORDER BY, GROUP BY.
- Large tables need faster lookups.

❌ **Avoid Indexes When:**

- Table is small (overhead is unnecessary).
- Columns have low uniqueness (e.g., gender).
- Frequent INSERT, UPDATE, DELETE (Indexes slow down writes).

PYTHON Questions

11. How do you handle missing values in Pandas?

Missing values occur in datasets due to various reasons like data entry errors, sensor failures, or missing records. Pandas provides several ways to handle missing values.

Checking for Missing Values

import pandas as pd

```
df = pd.DataFrame({
   'Name': ['Alice', 'Bob', 'Charlie', None],
   'Age': [25, None, 30, 22],
   'Salary': [50000, 60000, None, 45000]
})


print(df.isnull())  # Checks for missing values (True if missing)
print(df.isnull().sum())  # Count of missing values per column
```

Output:

```
mathematica
CopyEdit
   Name   Age  Salary
0  False  False   False
1  False   True   False
2  False  False    True
3   True  False   False

Name     1
Age      1
Salary   1
dtype: int64
```

Methods to Handle Missing Values

1 .Removing Missing Values (dropna())

- Removes rows or columns with missing values.

```
df.dropna(inplace=True)  # Removes rows with NaN values
df.dropna(axis=1)  # Removes columns with NaN values
```

- 🔹 **Use When:** Data loss is acceptable.

---

2 .Filling Missing Values (fillna())

- Replace missing values with a specific value or strategy.

```
df['Age'].fillna(df['Age'].mean(), inplace=True)  # Fill with mean
df['Salary'].fillna(df['Salary'].median(), inplace=True)  # Fill with median
df['Name'].fillna("Unknown", inplace=True)  # Fill categorical column
```

- 🔹 **Use When:** Data should be retained, and an estimate is acceptable.

---

3 .Forward & Backward Fill (ffill(), bfill())

- **Forward Fill (ffill)** → Replaces missing values with the previous row value.
- **Backward Fill (bfill)** → Replaces missing values with the next row value.

```
df.fillna(method='ffill', inplace=True)  # Uses previous row value
df.fillna(method='bfill', inplace=True)  # Uses next row value
```

- 🔹 **Use When:** Time-series data needs continuity.

---

4 .Interpolation (interpolate())

- Estimates missing values using interpolation methods.

```
df.interpolate(method='linear', inplace=True)
```

- 🔹 **Use When:** Missing values follow a trend.

---

12. Explain the difference between a list and a tuple.

Feature	List (list)	Tuple (tuple)
Definition	Ordered, mutable sequence.	Ordered, immutable sequence.
Syntax	lst = [1, 2, 3]	tup = (1, 2, 3)
Mutability	✅ Can be modified (append(), remove()).	❌ Cannot be modified once created.
Performance	Slower (modification overhead).	Faster (fixed size).
Memory Usage	Uses more memory.	Uses less memory.
Usage	When modifications are required.	When data should remain constant.

✅ **Example:**

```
# List Example
my_list = [1, 2, 3]
my_list.append(4)  # ✅ Allowed
print(my_list)  # [1, 2, 3, 4]

# Tuple Example
my_tuple = (1, 2, 3)
# my_tuple.append(4) ❌ Not Allowed (Throws Error)
print(my_tuple)  # (1, 2, 3)
```

🔹 **When to Use?**

- **Lists:** When the data needs to be changed dynamically.
- **Tuples:** When the data should remain unchanged (e.g., coordinates, database records).

---

13. What are lambda functions in Python? Give an example.

What is a Lambda Function?

A lambda function in Python is an **anonymous function** (without a name) that is used for short, simple operations. It is defined using the lambda keyword.

✅ **Syntax:**

lambda arguments: expression

✅ **Example:**

```
square = lambda x: x ** 2

print(square(5))  # Output: 25
```

Equivalent to:

```
def square(x):
    return x ** 2
```

Why Use Lambda Functions?

- **Concise:** Reduces the need for defining separate functions.
- **Useful for One-time Operations:** Often used in functions like map(), filter(), and sorted().
- **Improves Code Readability** for short functions.

Common Use Cases of Lambda Functions

1 .Using lambda with map()

Applies a function to every element in an iterable.

```
numbers = [1, 2, 3, 4, 5]

squared_numbers = list(map(lambda x: x ** 2, numbers))

print(squared_numbers)  # [1, 4, 9, 16, 25]
```

- Equivalent to using a loop but shorter.

2 .Using lambda with filter()

Filters elements based on a condition.

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4, 6]
```

- Filters only even numbers.

---

3 .Using lambda with sorted() (Custom Sorting)

```
students = [('Alice', 25), ('Bob', 20), ('Charlie', 23)]
students_sorted = sorted(students, key=lambda x: x[1])  # Sort by age
print(students_sorted)
```

- Sorts students by age.

Summary Table

Feature	Lists	Tuples	Lambda Functions
Definition	Mutable sequence	Immutable sequence	Anonymous function
Syntax	[1, 2, 3]	(1, 2, 3)	lambda x: x + 1
Mutability	✅ Yes	❌ No	✅ Yes (but for one expression)
Performance	Slower	Faster	Faster (Short functions)
Common Use	When data needs modification	Fixed data	map(), filter(), sorted()

---

14. How do you read a CSV file using Pandas?

Pandas provides the read_csv() function to load CSV files into a **DataFrame**.

✅ **Basic Syntax:**

```
import pandas as pd


df = pd.read_csv('data.csv')  # Load CSV file into a DataFrame
print(df.head())  # Display first 5 rows
```

Handling Different Scenarios While Reading a CSV

Scenario	Solution
CSV has no headers	df = pd.read_csv('data.csv', header=None)
Use custom column names	df = pd.read_csv('data.csv', names=['A', 'B', 'C'])
Specify delimiter (e.g., ; instead of ,)	df = pd.read_csv('data.csv', delimiter=';')
Skip rows while reading	df = pd.read_csv('data.csv', skiprows=2) (Skips first 2 rows)
Read only specific columns	df = pd.read_csv('data.csv', usecols=['Name', 'Salary'])
Handle missing values	df = pd.read_csv('data.csv', na_values=['NA', '?', '-'])
Read a large file in chunks	df_chunk = pd.read_csv('data.csv', chunksize=1000) (Processes in batches)

---

15. Explain the difference between apply(), map(), and vectorization in Pandas.

Feature	apply()	map()	Vectorization (NumPy/Pandas)
Works on	DataFrame & Series	Series only	Entire column/array
Function Applied	Row-wise (axis=1) or column-wise (axis=0)	Element-wise	Element-wise using built-in functions
Performance	Slower than vectorization	Faster than apply()	Fastest (uses NumPy)
Use Case	Complex operations	Element-wise transformation	Mathematical operations

✅ **Example for apply() (Row-wise or Column-wise Transformation)**

```
df['Salary_After_Tax'] = df['Salary'].apply(lambda x: x * 0.9)  # Apply a function
```

✅ **Example for map() (Only for Series, Single Value Change)**

```
df['Category'] = df['Category'].map({'A': 'Excellent', 'B': 'Good', 'C': 'Average'})
```

✅ **Example for Vectorization (Fastest Method)**

```
df['New_Salary'] = df['Salary'] * 1.1  # Direct operation on the column
```

🔹 **Best Practice:** Prefer **vectorization** when possible for better performance.

---

16. How would you perform data transformation using Python?

Data transformation involves **modifying, aggregating, or restructuring data** for better analysis.

1 .Handling Missing Values

```
df.fillna(df.mean(), inplace=True)  # Replace NaNs with column mean
```

2 .Data Type Conversion

```
df['Age'] = df['Age'].astype(int)  # Convert float to integer
```

3 .String Transformations

```
df['Name'] = df['Name'].str.upper()  # Convert names to uppercase
```

4 .Feature Scaling (Normalization)

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df[['Salary']] = scaler.fit_transform(df[['Salary']])

5 .One-Hot Encoding (Categorical to Numeric)

df = pd.get_dummies(df, columns=['Gender'], drop_first=True) # Converts 'Male'/'Female' to 0/1

6 .Aggregation (Grouping Data)

df.groupby('Department')['Salary'].mean() # Get average salary per department

---

17. What is the difference between NumPy and Pandas?

Both **NumPy** and **Pandas** are Python libraries for data manipulation, but they serve different purposes.

Feature	NumPy	Pandas
Definition	Numerical computing library	Data analysis & manipulation library
Data Structure	ndarray (N-dimensional array)	DataFrame (Tabular) & Series (1D)
Performance	Faster for numerical operations	Slightly slower due to additional features
Use Case	Mathematical operations, ML preprocessing	Data wrangling, analysis, visualization
Indexing	Uses numerical indexing (0-based)	Labeled indexing (row/column names)
Built-in Functions	Mathematical functions (np.mean(), np.sum())	Data manipulation (df.groupby(), df.merge())

✅ **Example in NumPy (Array Operations)**

```
import numpy as np


arr = np.array([1, 2, 3, 4, 5])

print(arr * 2)  # [2 4 6 8 10]
```

✅ **Example in Pandas (Tabular Operations)**

```
import pandas as pd


df = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Salary': [50000, 60000]})

print(df['Salary'].mean())  # 55000
```

🔹 **Best Practice:**

- Use **NumPy** for numerical computations.
- Use **Pandas** for working with structured data (CSV, Excel, databases).

POWER BI Questions

21. What are Calculated Columns and Measures in Power BI?

Power BI allows users to create **Calculated Columns** and **Measures** using **DAX (Data Analysis Expressions)**, but they serve different purposes.

Feature	Calculated Column	Measure
Definition	A new column added to a table based on a DAX formula	A dynamic calculation applied to aggregated data
Stored in	The table (each row gets a value)	Computed dynamically in visuals
Calculation Type	Row-level (computed during data load)	Aggregate-level (computed at runtime)

Feature	Calculated Column	Measure
Performance	Can slow down large datasets (uses memory)	Optimized for performance
Example	TotalPrice = Sales[Quantity] * Sales[UnitPrice]	Total Sales = SUM(Sales[Amount])

✅ **Example of a Calculated Column:**

- Used when a **new field** is needed in the data model.

TotalPrice = Sales[Quantity] * Sales[UnitPrice]

✅ **Example of a Measure:**

- Used when calculations need to be **aggregated dynamically** in reports.

Total Sales = SUM(Sales[Amount])

---

22. Explain Power Query and How It Is Used for Data Transformation.

Power Query is a **data transformation tool** in Power BI used to clean, reshape, and load data from multiple sources.

🔹 **Key Functions of Power Query:**

✓ Import data from multiple sources (Excel, SQL, APIs, etc.)
✓ Clean and transform data (remove duplicates, change data types)
✓ Merge & append datasets
✓ Apply custom formulas

✅ **Example Workflow in Power Query:**

1. **Load Data**: Import an Excel file or SQL database.
2. **Transform Data**:
    - Remove duplicates
    - Fill missing values
    - Convert data types

3. **Apply Custom Columns**:
    - Add a column to calculate **Profit = Sales - Cost**
4. **Load to Power BI**:
    - Apply changes and load the transformed data into Power BI reports.

🔹 **Power Query Formula Language (M Language)**

Power Query uses the **M Language** for advanced transformations. Example:

```
= Table.TransformColumns(Sales, {{"Price", each _ * 1.1, type number}})
```

---

23. What Are the Different Types of Filters in Power BI?

Power BI provides **multiple filter types** to refine reports dynamically.

Filter Type	Description
Visual Filters	Applied to a specific chart or table
Page Filters	Applied to all visuals on a single report page
Report Filters	Applied to the entire report (all pages)
Drillthrough Filters	Allows navigation from a summary report to a detailed report
Cross-Filtering & Cross-Highlighting	Interaction between visuals (clicking on one affects another)
Top N Filters	Displays only the top **N** values based on a metric
Relative Date Filters	Filters data dynamically based on a time period (e.g., last 30 days)

✅ **Example of a Page Filter:**

- If a report has **sales data from multiple countries**, you can filter a single page to show only **India's sales**.

✅ **Example of a Top N Filter:**

- Show **Top 5 Products** by sales using the Top N filter.

💡 **Best Practice:**
Use **filters efficiently** to improve report performance instead of complex DAX calculations.

---

24. How Do You Optimize a Power BI Report for Performance?

Optimizing a Power BI report improves **speed, efficiency, and user experience**. Below are key optimization techniques:

✅ **Data Model Optimization**

- Use **Star Schema** instead of a flat table.
- **Reduce the number of columns** by removing unnecessary fields.
- Avoid high-cardinality columns (e.g., using **Month Name** instead of **Date**).

✅ **Query Optimization**

- **Filter data at the source** before importing it into Power BI.
- Use **Power Query transformations** instead of complex DAX expressions.
- Disable **"Auto Date/Time"** in Power BI settings to reduce unnecessary tables.

✅ **DAX Performance Optimization**

- Use **SUMX, AVERAGEX** carefully to avoid row-by-row calculations.
- Replace IF conditions with SWITCH for better performance.
- Use **variables (VAR)** to store intermediate calculations.

✅ **Report-Level Optimization**

- Reduce the number of visuals per page (ideally **under 8-10 visuals**).
- Turn off unnecessary **interactions between visuals**.
- Use **Aggregations and Summary Tables** instead of detailed data.

Example of an Optimized DAX Measure

Bad:

```
Total Sales = SUMX(Sales, Sales[Quantity] * Sales[Price])
```

Optimized:

```
Total Sales = SUM(Sales[Quantity] * Sales[Price])
```

🚀 **Result:** Less computation, better performance!

---

25. What is the Difference Between a Star Schema and a Snowflake Schema?

Both **Star Schema** and **Snowflake Schema** are used in Power BI and data warehousing.

Feature	Star Schema ⭐	Snowflake Schema ❄️
Structure	Fact table with directly linked dimension tables	Dimension tables are normalized into sub-tables
Complexity	Simple structure (faster joins)	More complex due to normalization
Performance	Faster query performance	Slightly slower due to more joins
Storage	Requires more storage (denormalized)	Optimized storage (normalized)
Example	FactSales → DimCustomer, DimProduct	FactSales → DimCustomer → CustomerRegion

✅ **Example in Power BI:**

- A **Star Schema** has a **Fact Table (Sales)** connected directly to **Customers, Products, and Dates**.
- A **Snowflake Schema** further normalizes **Customers into Regions** and **Products into Categories**.

Which One to Use?
✓ **Use Star Schema** when performance matters (recommended for Power BI).
✓ **Use Snowflake Schema** when **data consistency** is a priority.

---

26. Explain Row-Level Security (RLS) in Power BI.

Row-Level Security (RLS) in Power BI **restricts data access** based on user roles.

✅ **Why Use RLS?**

✓ Prevent users from seeing unauthorized data.
✓ Improve data security without creating multiple reports.
✓ Efficient data access control for different user roles.

✅ **Types of RLS in Power BI**

Type	Description
Static RLS	Filters are manually assigned (e.g., Sales Manager sees only their region).
Dynamic RLS	Uses a **USERPRINCIPALNAME()** function to filter data based on login credentials.

✅ **Example: Implementing RLS in Power BI**

📌 Suppose we have a **Sales Table** and a **Users Table** with Region info.

1 .**Create a Role in Power BI:**
Go to **Modeling → Manage Roles → New Role**

2 .**Apply a DAX Filter:**

[Region] = LOOKUPVALUE(Users[Region], Users[Email], USERPRINCIPALNAME())

3 .**Test the Role:**
Use **View As Role** to verify if users only see their assigned region.

🚀 **Result:** Users see only **their specific data** without modifying the report!

---

27. What is the Purpose of DAX? Give an Example of a DAX Function.

DAX (Data Analysis Expressions) is a formula language used in Power BI, Power Pivot, and Analysis Services for performing calculations and aggregations on data.

✅ **Purpose of DAX:**

✓ Perform **custom calculations** on data.
✓ Create **calculated columns and measures** for deeper insights.
✓ Optimize **data modeling and business logic**.
✓ Enable **time-based calculations** like YTD, MTD, and QoQ.

✅ **Example of a DAX Function:**

📌 **Calculate Total Sales**:

Total Sales = SUM(Sales[Quantity] * Sales[Price])

📌 **Calculate Year-to-Date (YTD) Sales**:

YTD Sales = TOTALYTD(SUM(Sales[TotalAmount]), Sales[OrderDate])

🚀 **Result:** Enables **dynamic calculations** in Power BI dashboards!

---

28. How Do You Create a Relationship Between Tables in Power BI?

To connect tables in Power BI, we create **relationships** using primary and foreign keys.

✅ **Steps to Create a Relationship:**

1 .Go to **Model View** in Power BI.
2 .Drag and drop the **common field** between two tables.
3 .Set the **relationship type** (e.g., **One-to-Many (1:M)**).
4 .Choose **Cross-filter direction** (Single or Both).
5 .Click **OK** and verify.

✅ **Example: Relationship Between Sales and Customer Tables**

- **Sales Table** (CustomerID, OrderID, Amount)
- **Customers Table** (CustomerID, Name, Region)

🚀 **Result:** Allows combining **Sales Data with Customer Info** in reports!

---

29. Explain How to Handle Large Datasets in Power BI Efficiently.

Handling large datasets efficiently in Power BI improves performance and **reduces report load time**.

✅ **Best Practices for Large Datasets**

- 🔹 **Use Import Mode Instead of DirectQuery** (if feasible).
- 🔹 **Reduce the Number of Columns** (Remove unused fields).
- 🔹 **Aggregate Data Before Loading** (Use summarized tables).
- 🔹 **Use Star Schema Instead of Snowflake Schema**.
- 🔹 **Enable Query Folding in Power Query** (Push transformations to the source).
- 🔹 **Optimize DAX Measures** (Use variables and avoid iterators).
- 🔹 **Use Composite Models** (Import + DirectQuery for hybrid performance).

📌 **Example:** Instead of storing **millions of transaction records**, create a **summary table**:

```
Sales Summary = SUMMARIZE(Sales, Sales[Year], Sales[Product], "Total Sales", SUM(Sales[Amount]))
```

🚀 **Result: Faster queries** and better dashboard performance!

---

30. How Do You Create a Dynamic Dashboard in Power BI?

A **Dynamic Dashboard** allows users to **interact with visuals, filter data, and drill down for deeper insights**.

✅ **Steps to Create a Dynamic Dashboard**

1 .**Use Slicers and Filters** for interactivity.
2 .**Enable Drill-through and Tooltips** for deeper insights.
3 .**Create Dynamic Measures** using SELECTEDVALUE() and SWITCH().
4 .**Use Bookmarks and Buttons** to toggle views.
5 .**Implement Row-Level Security (RLS)** for personalized access.

✅ **Example: Dynamic Measure Based on User Selection**

📌 Suppose a user selects **"Sales" or "Profit"** from a slicer:

```
Dynamic Measure = SWITCH(
    SELECTEDVALUE(Metrics[Metric]),
    "Sales", SUM(Sales[Amount]),
    "Profit", SUM(Sales[Profit])
)
```

🚀 **Result:** Users can switch between **Sales and Profit dynamically**!