1. How would you implement Slowly Changing Dimensions (SCD Type 2 vs Type 3)? Explain the differences and when to use each.

✅ **Answer:**

SCD Type 2 (Full History Tracking):

- Maintains historical data by inserting a **new record** each time an attribute changes.
- Requires tracking **effective_date**, **end_date**, and a **current_flag** or **version number**.

📌 **When to use:**
When you need a **complete audit trail** of changes, such as address changes over time.

sql

CopyEdit

```
-- Example Structure:
customer_id | name | address | start_date | end_date   | is_current
------------|------|---------|------------|------------|------------
123         | John | NY      | 2023-01-01 | 2024-01-10 | 0
123         | John | LA      | 2024-01-11 | NULL       | 1
```

SCD Type 3 (Limited History Tracking):

- Stores **previous values** in **additional columns** of the same row.
- Typically supports **only one level of historical change**.

📌 **When to use:**
When only the **current and previous values** are required for reference, like last and current department.

sql

CopyEdit

```
-- Example Columns:
employee_id | name | current_department | previous_department
```

---

2. Describe the differences between transient tables and temporary tables in data warehousing.

✅ **Answer:**

Feature	**Temporary Table**	**Transient Table**
Lifetime	Session-level (auto-drops on session end)	Persistent (until manually dropped)
Storage	Cache / Memory	Disk
Fail-safe & Time Travel	Not supported	Not supported
Use Case	For temporary/adhoc logic during pipelines	Intermediate staging without long-term retention

📌 **Summary:**

- Use **temporary tables** for lightweight, in-session operations.
- Use **transient tables** when you want persistence without incurring fail-safe storage costs (e.g., in Snowflake).

---

3. Write an SQL query using window functions to find the most recent designation of each employee.

✅ **Answer:**

sql

CopyEdit

```
SELECT *
FROM (
  SELECT *,
       ROW_NUMBER() OVER (PARTITION BY employee_id ORDER BY update_date DESC) AS rn
  FROM employee_designation
) t
WHERE rn = 1;
```

📌 **Explanation:**

- The ROW_NUMBER() window function ranks rows within each employee_id group by update_date.
- We filter for the latest (rn = 1) to get the most recent designation.

---

4. Explain how you'd model a “swipe payment” API in a relational schema.

✅ **Answer:**

To represent a **swipe payment system**:

Tables:

- **Users**: user_id, name, email, created_at
- **Cards**: card_id, user_id, card_type, status
- **Transactions**: txn_id, card_id, user_id, amount, location, swipe_time, status, device_id

Relationships:

- One-to-Many from Users → Cards
- One-to-Many from Cards → Transactions

📌 **Notes:**

- Add indexing on user_id, swipe_time for querying performance.
- Add foreign keys to maintain data integrity.

---

5. What’s the difference between Change Data Capture (CDC) and Change Data Tracking (CDT)?

✅ **Answer:**

Feature	**CDC (Change Data Capture)**	**CDT (Change Data Tracking)**
Granularity	Captures actual **INSERT/UPDATE/DELETE** data	Tracks only **that a change occurred**
Before/After Values	Supported (can track what changed)	Usually **not supported**
Use Case	Replication, streaming pipelines	Audit, triggers for lightweight change logic

Feature	CDC (Change Data Capture)	CDT (Change Data Tracking)
Complexity	Requires change log / version tracking	Simpler implementation (e.g., change flags)

📌 **CDC** is used for **data replication** and streaming (e.g., Kafka, Debezium), while **CDT** is suitable for systems that just need to react to changes without knowing the actual data differences.

6. Compare data lake storage vs blob storage. When would you use one over the other?

✅ **Answer:**

Feature	Blob Storage	Data Lake Storage (ADLS Gen2)
Hierarchical Namespace	❌ No	✅ Yes (folders, directories)
Optimized for	Unstructured object storage	Big data analytics & high throughput workloads
Security	Basic (limited to container-level)	Advanced (POSIX permissions, RBAC, ACLs)
Performance	Suitable for general-purpose use	Optimized for analytics engines (like Spark, Hive)
Integration	General storage (images, docs, backups)	Tight integration with Azure analytics tools

📌 **When to use:**

- Use **Blob** for general file storage.
- Use **ADLS Gen2** for analytics workloads, hierarchical file systems, and scenarios requiring fine-grained access control and scalable parallel reads/writes.

---

7. Walk me through how you'd design an end-to-end ETL pipeline from an on-premise database to Azure Databricks.

✅ **Answer:**

🔹 **ETL Pipeline Design (Step-by-Step):**

1. **Extract:**

    - Use **Self-hosted IR** in **Azure Data Factory (ADF)** to connect to on-premise SQL Server, Oracle, or other DB.
    - Use ADF **Copy Activity** to pull data.
2. **Land (Stage):**
    - Store extracted data as CSV/Parquet in **ADLS Gen2** (Bronze Layer of Data Lake).
3. **Transform:**
    - Trigger an **ADF pipeline** to run a **Databricks notebook** using Databricks activity.
    - In the notebook, use **PySpark** to clean, transform, deduplicate, join data.
4. **Load:**
    - Write transformed data to curated zones in **Delta format** (Silver/Gold Layer) in ADLS.
    - Optionally, load aggregated data into **Synapse or Power BI** datasets for visualization.

📌 Add **monitoring** via ADF alerts or logs.

---

8. In Python, how do you extract unique values from a dictionary? Provide a code snippet.

✅ **Answer:**

If you want to extract unique values from a dictionary:

python

CopyEdit

```
my_dict = {
    'a': 10,
    'b': 20,
    'c': 10,
    'd': 30
}
```

unique_values = set(my_dict.values())

print(unique_values) # Output: {10, 20, 30}

📌 set() removes duplicates from dictionary .values().

---

9. How would you execute one notebook from another in Databricks?

✅ **Answer:**

You can use dbutils.notebook.run():

python

CopyEdit

```
# In Notebook A
dbutils.notebook.run("/path/to/NotebookB", timeout_seconds=60, arguments={"input": "value"})
```

📌 Parameters:

- **Path**: Relative or absolute path to target notebook
- **Timeout**: In seconds
- **Arguments**: Passed as dictionary, accessed in the target notebook using dbutils.widgets.get("key")

📌 This is helpful for orchestrating modular pipelines.

---

10. Write Python code to sum amount per customer_id given a DataFrame or list of records.

✅ **Answer:**

python

CopyEdit

```
import pandas as pd


data = [
    {'customer_id': 1, 'amount': 100},
```

```
    {'customer_id': 2, 'amount': 200},
    {'customer_id': 1, 'amount': 50},
    {'customer_id': 2, 'amount': 150}
]

df = pd.DataFrame(data)

result = df.groupby('customer_id')['amount'].sum().reset_index()
print(result)
```

📌 **Output:**

nginx

CopyEdit

```
   customer_id  amount
0            1     150
1            2     350
```

Alternatively, in PySpark:

python

CopyEdit

```
from pyspark.sql.functions import sum

df.groupBy("customer_id").agg(sum("amount").alias("total_amount")).show()
```

---

11. Given a Boolean matrix, count the number of islands (connected components). How would you solve it?

✅ **Answer:**

Use **Depth-First Search (DFS)** or **Breadth-First Search (BFS)**.

python

CopyEdit

```
def num_islands(grid):
    if not grid:
        return 0

    def dfs(r, c):
        if r < 0 or r >= len(grid) or c < 0 or c >= len(grid[0]) or grid[r][c] != 1:
            return
        grid[r][c] = -1  # mark as visited
        dfs(r+1, c)
        dfs(r-1, c)
        dfs(r, c+1)
        dfs(r, c-1)

    count = 0
    for r in range(len(grid)):
        for c in range(len(grid[0])):
            if grid[r][c] == 1:
                dfs(r, c)
                count += 1
    return count

# Example usage:
matrix = [
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1]
]
print(num_islands(matrix))  # Output: 2
```

📌 An "island" is a group of adjacent 1s (connected horizontally or vertically). Mark each visited cell to avoid counting it again.

12. How would you create a database schema for a university: colleges, students, professors? What tables, keys, and relationships?

✅ **Answer:**

You can model this using a **relational schema** with the following tables:

📘 **Tables and Keys:**

1. **College**
   - college_id (PK)
   - college_name
   - location
2. **Department**
   - department_id (PK)
   - department_name
   - college_id (FK to College)
3. **Professor**
   - professor_id (PK)
   - name
   - email
   - department_id (FK to Department)
4. **Student**
   - student_id (PK)
   - name
   - email
   - enrollment_year
   - college_id (FK to College)
5. **Course**

    - course_id (PK)
    - course_name
    - department_id (FK to Department)
6. **Enrollment**
    - student_id (FK to Student)
    - course_id (FK to Course)
    - semester
    - grade
    - *(Composite PK: student_id + course_id + semester)*
7. **Teaching**
    - professor_id (FK to Professor)
    - course_id (FK to Course)
    - semester

📌 This ensures **1:N** (College → Students), **M:N** (Student ↔ Courses), and **M:N** (Professor ↔ Courses) relationships.

---

13. Design an optimal schema to store event logs (like clicks/swipes) for high-velocity web traffic.

✅ **Answer:**

For a high-volume, write-heavy log system:

🗃️ **EventLog Table (Wide, Denormalized, Partitioned)**

- event_id (UUID or BIGINT, PK)
- user_id (FK)
- event_type (e.g., click, swipe)
- event_time (timestamp)
- device_type (mobile, desktop)
- location (optional - city/country)
- session_id

- page_url
- metadata (JSON or MAP<string, string> — for extensibility)

💡 **Schema Design Tips:**

- Store in **Delta Lake / Parquet** for compression and performance.
- Partition by **event_date** or **event_type** for efficient querying.
- Add **Z-ordering** on frequently filtered columns (e.g., user_id or session_id).

📌 Avoid joins during ingestion; keep schema flat.

---

14. Given messy sales data, walk us through cleaning, transformation, and how you'd design reporting tables for business analytics.

✅ **Answer:**

🧹 **Cleaning & Transformation Steps:**

1. **Remove Duplicates:**
    - Use dropDuplicates() in PySpark or ROW_NUMBER() in SQL to keep the latest record.
2. **Handle Nulls:**
    - Fill missing region or salesperson using business logic or default values.
    - Drop rows with missing order_id or amount.
3. **Data Type Fixing:**
    - Ensure dates are parsed correctly.
    - Convert amount to Decimal or Double.
4. **Standardize Values:**
    - Convert categories to lowercase/title case.
    - Format phone numbers or addresses.

🗄️ **Reporting Table Design:**

1. **Fact_Sales:**
    - order_id (PK)
    - customer_id

- product_id
- sales_date
- amount
- discount
- region_id
- salesperson_id

2. **Dim_Date, Dim_Customer, Dim_Product, Dim_Region, Dim_Salesperson**

📌 Use **star schema** for BI tools like Power BI or Tableau.

---

15. Describe how you'd optimize a slow-performing JOIN query joining large tables. What indexing or partitioning strategies would you use?

✅ **Answer:**

🛠️ **Optimization Strategies:**

1. **Broadcast Join:**
   - If one table is small (<10MB), use broadcast() to avoid shuffle.
2. **Partition Pruning:**
   - Ensure tables are partitioned on join/filter columns.
   - Example: sales_data partitioned by region or sales_date.
3. **Clustered Index / Z-ordering:**
   - Use Z-ordering (Databricks) or clustered index on frequent filter/join columns.
4. **Avoid Data Skew:**
   - If one key dominates (e.g., region_id=1 in 90% rows), use **salting**.
5. **Repartitioning:**
   - Repartition data before joining to ensure even distribution across tasks.
6. **Filter Early:**
   - Apply WHERE conditions before joining to reduce row count.
7. **Statistics & Caching:**
   - Update table stats so the optimizer chooses the right plan.

- Cache common lookups if reused.

📌 Always start with .explain() to check if joins are shuffled or optimized.