Cloud Data Engineering Lab Solutions (Questions 11-20)

November 14, 2025

Q11: Data Flow - Left Outer Join

Lab Task: Join Customer and Order datasets in a Left Outer Join and compute order counts per customer.

Conceptual Basis

A **Left Outer Join** is used when you want to keep *all* records from the "left" stream (e.g., `Customer`) and only include matching records from the "right" stream (e.g., `Order`). If a customer has no orders, they will still appear in the results, but their order-related fields will be `NULL`. This is perfect for "find all customers and *if* they have orders" scenarios.

Step-by-Step Solution

1. **Add Sources:** In a new Data Flow, add `Source_Customer` and `Source_Order`.
2. **Add Join:** Add a **Join** transformation, with `Source_Customer` as the **Left stream** and `Source_Order` as the **Right stream**.
3. **Configure Join:**
   - **Join type:** Select **Left Outer**.
   - **Join conditions:** Match on `CustomerID == CustomerID`.
4. **Add Aggregate:** Add an **Aggregate** transformation after the join.
   - **Group by:** Select the customer's unique key (e.g., `CustomerID` from the Customer stream).
   - **Aggregates:** Create a new column `OrderCount`.
   - **Expression:** Use `count(OrderID)`. The `count()` function only counts non-NULL values, so customers with no orders (NULL `OrderID` from the left join) will correctly get a count of 0.
5. **Add Sink:** Add a **Sink** to output the results.

Q12: Data Flow - Lookup Transformation

Lab Task: Use Lookup transformation to fetch the category name from a reference dataset for each ProductID.

Conceptual Basis

A **Lookup** transformation is an optimization for a common join scenario. It's used to "enrich" a primary data stream by "looking up" a single value from a smaller *reference* or *dimension* dataset. It's more efficient than a full Join if the reference data is small enough to be broadcast (cached in memory).

Step-by-Step Solution

1. **Add Sources:** In your Data Flow, add the main `Source` (e.g., `Products` or `Sales`) and the reference `Source` (e.g., `Categories`, which contains `CategoryID` and `CategoryName`).
2. **Add Lookup:** Click the + icon after your main `Products` source and select **Lookup**.
3. **Configure Lookup:**
   - **Lookup stream:** Select your `Categories` source.
   - **Lookup conditions:** Match on `CategoryID == CategoryID`.
4. **Result:** The Lookup transformation adds all columns from the `Categories` stream to the `Products` stream for each matching row. You can use a `Select` transformation afterward to drop the duplicate `CategoryID` column, keeping only the `CategoryName`.

Q13: Data Flow - Exists Transformation

Lab Task: Use Exists transformation to filter emails from List A that are not in List B.

Conceptual Basis

An **Exists** transformation is a row-filtering operation, not a column-joining one. It checks for the *existence* of a row in another stream based on a condition. Unlike a Join, it does *not* combine columns. This is the direct ADF equivalent of SQL's `EXISTS` or `NOT EXISTS` clauses.

Step-by-Step Solution

1. **Add Sources:** In your Data Flow, add `Source_ListA` and `Source_ListB`.
2. **Add Exists:** Click the + icon after `Source_ListA` and select **Exists**.
3. **Configure Exists:**
   - **Exists stream:** Select `Source_ListB`.
   - **Exist type:** Select **Doesn't exist**.
   - **Exists conditions:** Match on `Email == Email`.
4. **Add Sink:** Add a **Sink**. The output will be only the rows from `Source_ListA` whose emails do not have a match in `Source_ListB`.

Q14: Data Flow - Conditional Split

Lab Task: Apply Conditional Split to classify orders into three categories: High (>10000), Medium (5000-10000), and Low (<5000).

Conceptual Basis

A **Conditional Split** transformation acts as a traffic router for your data. It evaluates a series of conditions (in order) and directs each row down the *first* stream whose condition it meets. It also includes a default stream for rows that match no conditions.

Step-by-Step Solution

1. **Add Source:** In your Data Flow, add your `Orders` source.
2. **Add Conditional Split:** Click the + icon and select **Conditional Split**.

3. **Configure Streams (in order):**
    - **Output Stream 1 (High):** `Amount > 10000`
    - **Output Stream 2 (Medium):** `Amount >= 5000  Amount <= 10000`
    - **Default Stream (Low):** The third stream will be the default, named `OtherStream` (or similar). This stream automatically catches all remaining rows (i.e., `Amount < 5000`).
4. **Add Sinks:** You can now add three separate **Sink** transformations, one connected to each of the three output streams (`High`, `Medium`, `Low`) to write them to different files or tables.

Q15: Data Flow - Union Transformation

Lab Task: Merge data from three regional CSV files using Union transformation for consolidated reporting.

Conceptual Basis

A **Union** transformation "stacks" rows from multiple data streams on top of each other. It is the direct equivalent of SQL's `UNION ALL` operator. This requires that all input streams have the *same* columns and data types (or compatible schemas).

Step-by-Step Solution

1. **Add Sources:** In your Data Flow, add three separate **Source** transformations (e.g., `Source_Region1`, `Source_Region2`, `Source_Region3`).
2. **Add Union:** Click the + icon after `Source_Region1` and select **Union**.
3. **Configure Union:**
    - **Incoming stream:** `Source_Region1` is already the primary.
    - **Union with:** Select `Source_Region2`.
    - **Union settings:** You can add more streams. Click the + icon and add `Source_Region3`.
    - **Union by:** Select **Name**. This will stack the rows by matching column names (e.g., `Region1.SaleAmount` will be stacked with `Region2.SaleAmount`).
4. **Add Sink:** Add a **Sink**. The output file will contain all rows from all three regions in a single dataset.

Q16: Data Flow - Pivot Transformation

Lab Task: Use Pivot transformation to convert monthly sales rows into columns (Jan-Dec).

Conceptual Basis

A **Pivot** transformation de-normalizes your data. It turns unique *values* from a single row-level column (e.g., "Jan", "Feb", "Mar") into multiple *columns* in the output, and then fills those columns with an aggregated value.

Step-by-Step Solution

1. **Input Data:** Assume your source has columns: `Region`, `MonthName`, `Sales`.
2. **Add Source:** Add your source to the Data Flow.
3. **Add Pivot:** Click the + icon and select **Pivot**.

4. **Configure Pivot:**
    - **Group by:** Select the column that should remain a row, e.g., `Region`.
    - **Pivot key:** Select the column whose *values* will become new columns, e.g., `MonthName`.
    - **Pivoted columns:** This is where you define the aggregate.
    - **Expression:** `sum(Sales)`
    - **Column name prefix:** (Optional) You can leave this blank.
5. **Add Sink:** Add a **Sink**. The output will have columns like: `Region`, `Jan`, `Feb`, `Mar`, etc.

Q17: Synapse Serverless SQL Script

Lab Task: Write a SQL script in Synapse serverless pool to read CSV data from ADLS Gen2, filter it, and join it with another dataset.

Conceptual Basis

A **Synapse Serverless SQL Pool** does not store data. It is a query engine that can read data *directly* from files in your data lake (e.g., ADLS Gen2) using the `OPENROWSET` T-SQL command. You pay per query (per data processed).

Step-by-Step Solution (SQL Script)

```
-- Use 'WITH' clauses to treat the files as tables
WITH
Sales AS (
    SELECT *
    FROM OPENROWSET(
        BULK 'https://<storage_acct>.dfs.core.windows.net/<container>/path/to/sales/*.
    csv',
        FORMAT = 'CSV',
        PARSER_VERSION = '2.0',
        HEADER_ROW = TRUE
    )
    -- Explicitly define schema for performance and to avoid errors
    WITH (
        TransactionID int,
        ProductID int,
        Amount float,
        TransactionDate date
    ) AS s
),
Products AS (
    SELECT *
    FROM OPENROWSET(
        BULK 'https://<storage_acct>.dfs.core.windows.net/<container>/path/to/products.
    csv',
        FORMAT = 'CSV',
        PARSER_VERSION = '2.0',
        HEADER_ROW = TRUE
    )
    WITH (
        ProductID int,
        ProductName varchar(100),
        Category varchar(50)
    ) AS p
)
-- Final SELECT performs the filter and join
SELECT
    s.TransactionID,
    p.ProductName,
    p.Category,
    s.Amount
FROM Sales s
```

```
JOIN Products p ON s.ProductID = p.ProductID
WHERE
    s.Amount > 1000
    AND p.Category = 'Electronics';
```

Listing 1: Synapse Serverless SQL Script

Q18: Synapse Dedicated Pool (PolyBase)

Lab Task: Create a table in a dedicated Synapse SQL pool and load data using PolyBase. Compare query performance with serverless SQL pool.

Conceptual Basis

- **Dedicated Pool:** A pre-provisioned, massively parallel processing (MPP) data warehouse. Data is **imported** and stored in a highly optimized columnar format. Queries are extremely fast.
- **PolyBase:** A technology (used via the `COPY INTO` command) that efficiently loads data in parallel from the data lake *into* the dedicated pool's tables.

Step-by-Step Solution

1. **Create Table:** In your *Dedicated* SQL pool, create the destination table.

```
CREATE TABLE dbo.Sales (
    TransactionID int NOT NULL,
    ProductID int,
    Amount float,
    TransactionDate date
)
WITH (
    DISTRIBUTION = HASH(TransactionID),
    CLUSTERED COLUMNSTORE INDEX
);
```

2. **Load with PolyBase (COPY):** Run this T-SQL to load from your ADLS file.

```
COPY INTO dbo.Sales
(TransactionID, ProductID, Amount, TransactionDate)
FROM 'https://<storage_acct>.dfs.core.windows.net/<container>/path/to/sales/*.csv'
WITH (
    FILE_TYPE = 'CSV',
    FIRSTROW = 2, -- If it has a header
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '0x0a'
);
```

Performance Comparison

- **Serverless Pool:** The query in Q17 will have *higher latency*. It has to read the raw CSV files, infer/apply schema, parse text, and perform the join *every time it runs*. It's ideal for ad-hoc analysis.
- **Dedicated Pool:** After the one-time `COPY` load, the data is stored in a optimized, indexed, and distributed format. A query like `SELECT ... FROM dbo.Sales WHERE Amount > 1000` will be **significantly faster** (orders of magnitude) because it's reading pre-processed, columnar data. It's built for high-performance, repetitive reporting.

Q19: PySpark - Flatten Nested JSON

Lab Task: Develop a PySpark script in Synapse Spark pool to read JSON data, flatten nested columns, and store results as CSV.

Conceptual Basis

Spark natively understands JSON structure. When you read a nested JSON, Spark creates a `StructType` (a struct, or object) column. To flatten this, you select the sub-fields using "dot notation" (e.g., `customer.address.city`).

Step-by-Step Solution (PySpark Script)

```
# Assume input JSON looks like:
# {"id": 1, "customer": {"name": "Alice", "email": "a@b.com"}}

# 1. Read the nested JSON data
df = spark.read.json("abfss://<container>@<storage_acct>.dfs.core.windows.net/path/to/
    data.json")
df.printSchema()
# root
#  |-- id: long (nullable = true)
#  |-- customer: struct (nullable = true)
#  |    |-- name: string (nullable = true)
#  |    |-- email: string (nullable = true)

# 2. Flatten the nested columns
# We select all top-level columns, and "explode" the struct
df_flattened = df.select(
    "id",
    "customer.name",
    "customer.email"
)

# 3. Store results as CSV
df_flattened.write.mode("overwrite").csv("abfss://<container>@<storage_acct>.dfs.core.
    windows.net/path/to/output.csv", header=True)
```

Listing 2: PySpark JSON Flattening

Q20: PySpark - DataFrame Filter GroupBy

Lab Task: Use Spark DataFrame APIs to read a CSV, filter rows, and perform groupBy by department.

Conceptual Basis

The Spark **DataFrame API** is a declarative, type-safe interface for data manipulation. `.filter()` is the transformation for removing rows (like SQL `WHERE`), and `.groupBy()` combined with `.agg()` is the transformation for aggregation (like SQL `GROUP BY`).

Step-by-Step Solution (PySpark Script)

```
from pyspark.sql.functions import avg, count

# 1. Read the CSV data
df_employees = spark.read.csv(
    "abfss://<container>@<storage_acct>.dfs.core.windows.net/path/to/employees.csv",
    header=True,
    inferSchema=True  # Automatically detect data types (e.g., int, string)
)

# 2. Filter rows (Transformation)
df_filtered = df_employees.filter(col("Salary") > 50000)

# 3. Perform groupBy and aggregate (Transformation)
df_agg = df_filtered.groupBy("Department") \
    .agg(
        avg("Salary").alias("AverageSalary"),
        count("*").alias("EmployeeCount")
    )
```

```
# 4. Display results (Action)
df_agg.show()
```

Listing 3: PySpark DataFrame API