LEARN

AZURE DATA FACTORY (ADF)

PART 4- Mapping DataFlows

C .R. Anil Kumar Reddy

www.linkedin.com/in/chenchuanil

Mapping DataFlows

Your Mapping Data Flows in Azure Data Factory (ADF) allows you to transform and move data in a no-code environment. Data flows support a variety of transformation activities to achieve tasks like aggregations, filtering, joins, and more.

Here’s a detailed overview of the common mapping data flow transformations, along with examples and use cases:

1 Source Transformation

The Source transformation is where data flows begin. It reads data from a specified dataset, such as Azure Blob Storage, Azure SQL Database, or Data Lake, into the data flow for further processing.

- **Use Case:** Ingest data from different sources, such as importing customer data from a CSV file in Azure Blob Storage.

- **Example:**
- Scenario: You want to load a CSV file named SalesData.csv from Azure Blob Storage.
- Objective: Read the file into the data flow to start the transformation process.

- **Configuration:**
  a. Add a Source transformation to the data flow.
  b. Select the source dataset, e.g., the SalesData CSV file.
  c. Configure Projection to define the schema (e.g., OrderID, ProductID, Quantity, SalesAmount).
  d. Optionally, enable Schema Drift to handle dynamic column changes in the dataset.

- **Outcome:** The data flow reads data from SalesData.csv and makes it available for subsequent transformations.

- **Benefit:** Provides a flexible way to ingest data from various sources and shapes it for processing within the data flow.

2. Derived Column Transformation

The Derived Column transformation allows you to create new columns or modify existing ones using expressions. It's commonly used for data transformation tasks, such as calculations, string formatting, or data type conversions.

- **Use Case:** Add a new column that calculates the total price or formats dates in a specific way.

- **Example:**
- Scenario: You have a dataset with columns Quantity and UnitPrice and want to create a new column TotalPrice.
- Objective: Multiply Quantity by UnitPrice to calculate TotalPrice.

- **Configuration:**
  a. Add a Derived Column transformation to the data flow.
  b. Create a new column named TotalPrice.
  c. Use the expression builder to define the logic: TotalPrice = Quantity * UnitPrice.
  d. You can also modify existing columns. For example, format a column OrderDate to MM-dd-yyyy using toString(OrderDate, 'MM-dd-yyyy').

- **Outcome:** The data flow outputs the dataset with a new column, TotalPrice, containing the computed values.

- **Benefit:** Enhances the dataset by creating new columns or transforming existing ones, enabling more flexible and insightful data analysis.

3. Aggregate

The Aggregate transformation allows you to group data and perform aggregate functions like sum, average, count, min, and max on the grouped data. It’s used to summarize datasets based on specific fields.

- **Use Case:** Calculate the total sales for each region or find the average order value for each customer.

- **Example:**
- Scenario: You have a sales dataset with columns Region, OrderID, and SalesAmount.
- Objective: Calculate the total SalesAmount for each Region.

- **Configuration:**
  a. Add an Aggregate transformation to your data flow.
  b. Set the Group by field to Region.
  c. Use an aggregate expression for a new column, e.g., TotalSales = sum(SalesAmount).

- **Outcome:** The data flow outputs a summarized dataset with two columns: Region and TotalSales.

- **Benefit:** Extracts valuable insights by summarizing large datasets, helping to identify trends like total revenue per region.

4. Join

The Join transformation combines data from two different streams based on a matching key in specified columns. It supports various join types: inner, left, right, and full outer joins.

- **Use Case:** Combine sales data with product information to enrich the dataset for analysis.

- **Example:**
- Scenario: You have two datasets—Sales with columns ProductID, Quantity, and OrderDate, and Products with columns ProductID and ProductName.
- Objective: Merge these datasets to include ProductName in the sales data using ProductID.

- **Configuration:**
- Add a Join transformation to your data flow.
- Select the join type (e.g., Inner Join).
- Set the join condition as Sales@ProductID == Products@ProductID.
- Select the columns to include in the output, such as ProductName, Quantity, and OrderDate.

- **Outcome:** The result is a dataset that includes sales details along with the product names for each order.

- **Benefit:** Creates richer datasets by merging relevant information, making it easier to analyze data in context, such as viewing product names alongside sales figures.

5. Filter

The Filter transformation allows you to remove rows from the data stream based on a specified condition. It’s used to keep only the records that meet certain criteria.

- **Use Case:** Filter out inactive customers or keep only the sales data from the current year.

- **Example:**
- Scenario: You have a dataset with columns OrderDate and Status and want to keep only rows where Status is "Shipped."
- Objective: Filter out rows that do not meet the "Shipped" status.

- **Configuration:**
   a. Add a Filter transformation to your data flow.
   b. Use the Expression Builder to set the condition: Status == 'Shipped'.
   c. Optionally, add more complex conditions using logical operators (e.g., Status == 'Shipped' && year(OrderDate) == 2024).

- **Outcome:** The data flow outputs only the rows where Status is "Shipped."

- **Benefit:** Allows precise data filtering, ensuring that only relevant data is passed to downstream transformations.

6. Conditional Split

The Conditional Split transformation divides the data stream into multiple streams based on conditions. Each condition results in a separate output, similar to a SQL CASE statement.

- **Use Case:** Separate customers into "High Value" and "Low Value" groups based on total spending.

- **Example:**
- Scenario: You have a dataset with a TotalSales column and want to split customers into "VIP" and "Regular" based on their total sales.
- Objective: Create two streams—one for customers with TotalSales > 5000 (VIP) and another for the rest (Regular).

- **Configuration:**
  a. Add a Conditional Split transformation to your data flow.
  b. Define conditions for the split:
     - Condition 1: TotalSales > 5000 for "VIP" customers.
     - Condition 2 (Default): All other records go to "Regular."
  c. Each condition creates a different output stream that you can further transform or write to different sinks.

- **Outcome:** The data flow splits the input dataset into two streams: one for "VIP" customers and one for "Regular" customers.

- **Benefit:** Enables dynamic data routing and processing based on specified conditions, providing flexibility in handling multiple data processing scenarios within a single data flow.

7. Sort

The Sort transformation orders rows in the data stream based on specified columns. It allows you to sort data in ascending or descending order, which can be useful for reports, aggregations, and subsequent transformations.

- **Use Case:** Sort sales records by OrderDate in descending order or arrange customer names alphabetically.

- **Example:**
  - Scenario: You have a dataset with columns OrderDate and SalesAmount and want to sort the data by OrderDate in descending order.
  - Objective: Arrange sales records to show the most recent orders first.
  - Configuration:
    1. Add a Sort transformation to your data flow.
    2. Select the column to sort by (e.g., OrderDate).
    3. Choose the sort direction (ascending or descending). For this example, select Descending.

  - **Outcome:** The data flow outputs the dataset sorted by OrderDate, with the latest orders appearing at the top.

- **Benefit:** Enables organized data presentation and facilitates further operations that may depend on data order, like window functions or ranking.

8. Union

The Union transformation combines data from multiple streams into a single stream. It is similar to the SQL UNION ALL operation, where data from different sources or partitions are appended together.

- **Use Case:** Merge sales data from multiple regions into a consolidated dataset or combine customer information from different sources.

- **Example:**
    - Scenario: You have two datasets—Sales_Region1 and Sales_Region2—both with columns OrderID, ProductID, and SalesAmount, and want to combine them into a single stream.
    - Objective: Create a unified dataset containing sales data from both regions.

    - **Configuration:**
        1. Add a Union transformation to your data flow.
        2. Connect the streams (Sales_Region1 and Sales_Region2) to the Union transformation.
        3. Ensure the columns from each stream align. If there are mismatched columns, use a Select transformation beforehand to rename or map them.

    - **Outcome:** The data flow outputs a single dataset combining records from both Sales_Region1 and Sales_Region2.
    -
- **Benefit:** Provides a simple way to consolidate data from multiple sources, facilitating further processing or analysis in a unified dataset.

9. Pivot

The Pivot transformation rotates rows into columns, allowing you to transform categorical data into a more summarized format. This is useful when you want to create a cross-tabular report or need data in a specific shape for analysis.

Use Case: Convert monthly sales data into a wide format with separate columns for each month.

Example:

Scenario: You have a dataset with columns Year, Month, and SalesAmount and want to pivot the data so that each month becomes its own column.
Objective: Create a table with Year as rows and separate columns for each month (Jan, Feb, Mar, etc.).

Configuration:

Add a Pivot transformation to your data flow.
Select the Group By column, such as Year.
Set the Pivot Key to Month and Aggregate as sum(SalesAmount).
Optionally, specify the column names for the pivoted data (e.g., Jan, Feb, Mar).

Outcome: The data flow outputs a dataset where each year has columns for each month’s total sales (e.g., Jan, Feb, Mar).

Benefit: Converts long-format data into a wide format, making it easier to generate reports and perform comparisons across categories.

10. Unpivot

The Unpivot transformation performs the reverse of a pivot, transforming columns into rows. This is helpful when you need to normalize wide datasets into a more row-based, analyzable format.

- **Use Case:** Convert a wide table with separate columns for each month into a long format with Month and SalesAmount columns.

- **Example:**
  - Scenario: You have a dataset with columns Year, Jan_Sales, Feb_Sales, and Mar_Sales, and you want to unpivot these monthly columns into rows.
  - Objective: Create a table with columns Year, Month, and SalesAmount.

  - **Configuration:**
    1. Add an Unpivot transformation to your data flow.
    2. Select the columns to unpivot (e.g., Jan_Sales, Feb_Sales, Mar_Sales).
    3. Define the Unpivot Key (e.g., Month) to indicate the new column names and the Unpivoted Column (e.g., SalesAmount) to hold the values.

  - **Outcome:** The data flow outputs a dataset with columns Year, Month, and SalesAmount, where each month's sales are now individual rows.

- **Benefit:** Normalizes wide datasets into a long format, making it easier to filter, aggregate, and analyze the data based on different categories.

11. Alter Rows

The Alter Row transformation is used to define row-level policies for data operations like insert, update, delete, or upsert when writing to a database sink. It allows you to control how each row is processed in the destination.

- **Use Case:** Perform conditional updates, inserts, or deletes in a database based on specific criteria, such as updating customer information where Status = 'Active'.

- **Example:**
  - Scenario: You have a dataset of customer records with a Status column. You want to update rows where Status is "Updated" and delete rows where Status is "Inactive".
  - Objective: Define row policies for update and delete operations.

  - **Configuration:**
    1. Add an Alter Row transformation to your data flow.
    2. Use the expression builder to set conditions:
       - isUpdate() for rows where Status == 'Updated'.
       - isDelete() for rows where Status == 'Inactive'.
    3. Select the target policy (Insert, Update, Delete, Upsert) based on the condition.

  - **Outcome:** The data flow marks rows for updates or deletions based on the specified conditions, which will be executed when writing to the sink.

- **Benefit:** Provides fine-grained control over row-level data operations in sinks, facilitating effective data management and synchronization.

12. Lookup

The Lookup transformation searches for matching records in a secondary dataset (reference table) and retrieves additional information. It’s commonly used to enrich data streams by adding details from a reference dataset.

- **Use Case:** Enrich sales data with product details from a product catalog based on ProductID.

- **Example:**
  - Scenario: You have a sales dataset with ProductID and want to add the ProductName and Category from a product reference table.
  - Objective: Look up additional product details to enrich the sales data.

  - **Configuration:**
    1. Add a Lookup transformation to your data flow.
    2. Select the lookup dataset (e.g., Products table) and define the join condition, such as Sales@ProductID == Products@ProductID.
    3. Choose the columns to retrieve from the lookup dataset (e.g., ProductName, Category).

  - **Outcome:** The data flow outputs the original sales dataset, now enriched with ProductName and Category columns from the lookup table.

- **Benefit:** Enhances datasets by pulling in additional details from reference tables, reducing the need for complex joins or nested queries in downstream processes.

13. Flatten

- The Flatten transformation converts complex hierarchical data structures like arrays or nested JSON objects into a flat, tabular format. This is essential for processing data from sources that have nested or repeated fields.

- **Use Case:** Extract individual items from an array in JSON data, such as getting order items from an order record.

- **Example:**
- Scenario: You have a dataset with an Orders column containing nested arrays of OrderItems (e.g., each Order has multiple OrderItems with ItemName and Quantity).
- Objective: Flatten the OrderItems array to create individual rows for each item in every order.

- **Configuration:**
- Add a Flatten transformation to your data flow.
- Select the array column to flatten (e.g., Orders.OrderItems).
- Define the new columns for the flattened data (e.g., ItemName, Quantity).

- **Outcome:** The data flow outputs a flat table where each row represents an individual OrderItem with columns ItemName and Quantity.

- **Benefit:** Makes it easier to analyze and process hierarchical data by transforming it into a simple, flat structure suitable for reporting, aggregation, or loading into relational databases.

14. Cast Transformation

The Cast transformation changes the data type of columns in the data flow. It is commonly used to ensure that columns have the correct data type for downstream processing or to meet the schema requirements of a destination.

- **Use Case:** Convert a string date to a date type or a numeric column from a string to a decimal for calculations.

- **Example:**
  - Scenario: You have a dataset with columns Price as a string and OrderDate as a string. You want to cast Price to a decimal and OrderDate to a date format.
  - Objective: Ensure Price and OrderDate have the correct data types for calculations and filtering.

  - **Configuration:**
    1. Add a Cast transformation to your data flow.
    2. Select the columns to cast:
       - For Price, use the expression toDecimal(Price).
       - For OrderDate, use the expression toDate(OrderDate, 'yyyy-MM-dd').
    3. Apply these expressions to change the data types accordingly.

  - **Outcome:** The data flow outputs a dataset with Price as a decimal and OrderDate as a date, ready for further transformations or loading into a target sink.

- **Benefit:** Ensures data type compatibility, allowing for accurate calculations, filtering, and proper schema alignment in data sinks.

15. Surrogate Key Transformation

The Surrogate Key transformation generates a unique, sequential identifier for each row in the dataset. It is commonly used when there is no natural key in the source data and you need a primary key in the destination.

- **Use Case:** Add a unique identifier to each customer record in a table for loading into a database that requires a primary key.

- **Example:**
  - Scenario: You have a dataset with customer records but no unique identifiers. You want to add a CustomerID to each row.
  - Objective: Generate a new column called CustomerID that provides a unique, sequential number for each row.

  - **Configuration:**
    i. Add a Surrogate Key transformation to your data flow.
    ii. Name the new column (e.g., CustomerID).
    iii. Set the starting value (default is 1) and specify the increment if needed.

  - **Outcome:** The data flow outputs the dataset with an additional CustomerID column, containing unique sequential numbers for each row.

- **Benefit:** Provides a simple way to generate unique identifiers for datasets, useful for loading into databases or for deduplication tasks.

16. Select Transformation

The Select transformation allows you to choose, rename, or drop columns in your dataset. It helps shape the data by controlling which columns are passed to subsequent transformations or sinks.

- **Use Case:** Remove unnecessary columns or rename columns to align with a target schema.

- **Example:**
  - Scenario: You have a dataset with columns FirstName, LastName, Email, and PhoneNumber, but only need FullName and Email for the next step.
  - Objective: Combine FirstName and LastName into FullName, drop PhoneNumber, and pass only the necessary columns.

  - **Configuration:**
    i. Add a Select transformation to your data flow.
    ii. Rename columns as needed (e.g., combine FirstName and LastName using an expression to create FullName).
    iii. Uncheck columns you wish to remove (e.g., PhoneNumber).

  - **Outcome:** The data flow outputs a dataset with only FullName and Email.

- **Benefit:** Simplifies data processing by controlling column selection, renaming columns for clarity, and ensuring the output schema meets the requirements of downstream transformations or sinks.

17. Sink Transformation

The Sink transformation is the endpoint in a data flow where transformed data is written to a target destination. It supports various sinks like Azure Blob Storage, Azure SQL Database, Data Lake, Cosmos DB, etc. This is where data is stored after all processing is completed.

- **Use Case:** Write processed data to a SQL database, store transformed data in a CSV file in Blob Storage, or load data into a Data Lake.

- **Example:**
    - Scenario: You have processed a dataset and want to save the transformed data into an Azure SQL Database table named ProcessedOrders.
    - Objective: Write the data to the target SQL table, with options for insert, update, or upsert operations.

    - **Configuration:**
        1. Add a Sink transformation to your data flow.
        2. Select the target dataset (e.g., an Azure SQL Database table).
        3. Choose the Write Behavior:
            - Insert: Insert all rows into the table.
            - Update: Update existing rows based on a key.
            - Upsert: Insert new rows and update existing ones.
            - Delete: Remove rows based on conditions.
        4. Map columns from the data flow to the sink's schema using Mapping.
        5. (Optional) Enable additional settings like partitioning and batch size for optimized performance.

    - **Outcome:** The data flow writes the transformed dataset into the specified table, following the defined write behavior.

- **Benefit:** Provides flexibility in writing data to various storage destinations, supporting different write operations (insert, update, upsert, delete) to match your specific data integration needs.

Advantages of dataflows

1. No-Code Data Transformation

- Enables data transformations using a visual, drag-and-drop interface, reducing the need for complex coding.
- Simplifies the ETL process, making it accessible to users without deep programming skills.

2. Seamless Integration with Azure Services

- Works natively with Azure services like Blob Storage, Data Lake, and SQL Database.
- Streamlines data ingestion and transformation within the Azure ecosystem.

3. Scalability and Performance

- Leverages Azure Databricks Spark environment for processing, automatically scaling based on data volume.
- Handles large datasets efficiently without manual scaling.

4. Extensive Transformation Capabilities

- Offers a wide range of built-in transformations (joins, aggregates, pivots, etc.).
- Allows complex data manipulations within a single platform.

5. Schema Drift and Flexible Data Handling

- Supports schema drift, dynamically adjusting to changes in incoming data schema.
- Reduces maintenance efforts by handling varying data structures automatically

6. Built-in Data Quality and Error Handling

- Provides functions to clean, validate, and transform data while handling errors.
- Improves data quality by filtering out nulls or incorrect values during processing.

7. Reusable Logic with Parameters

- Allows parameterization for creating reusable data flows across different pipelines.
- Enhances modularity and consistency in data transformation logic.

8. Visual Debugging and Monitoring

- Offers visual debugging tools to inspect data at various transformation stages.
- Simplifies troubleshooting and validation of data flows.

.

9. Compatibility with CI/CD Pipelines

- Integrates with Azure DevOps for continuous integration and deployment.
- Streamlines the development lifecycle with version control and automated testing.

10. Optimized for Cost Management

- Uses Databricks clusters only during runtime and supports auto-shutdown.
- Helps optimize costs by using compute resources only when needed.

11. Security and Compliance

- Integrates with Azure's security features like VNet, Managed Identity, and encryption.
- Ensures data handling meets security and compliance requirements.

Disadvantages of dataflows

1. Learning Curve

- The visual interface, though no-code, can be complex for beginners.
- Requires time to understand various transformations, expressions, and best practices.

2. Limited to Azure Ecosystem

- Data flows are designed primarily for Azure services.
- Integration with non-Azure environments may require additional tools or services.

3. Complex Debugging for Large Flows

- Debugging complex data flows with multiple transformations can become challenging.
- Lack of detailed error messages can make pinpointing issues time-consuming.

4. Performance Overhead

- Data flows rely on Spark clusters, which introduce startup time and may increase latency.
- Not ideal for low-latency, real-time data processing scenarios.

5. Higher Costs for Large Data Volumes

- Uses Azure Databricks clusters, which can become costly when processing large volumes of data frequently.
- Requires careful configuration and monitoring to manage costs effectively

.

6. Limited Control Over Spark Environment

- The Spark environment used in data flows is abstracted, limiting fine-tuning and custom configurations.
- Advanced users who need more control over Spark jobs might find this restrictive.

7. Limited Real-Time Processing

- Data flows are primarily designed for batch processing, not real-time or streaming data.
- May not be suitable for scenarios requiring immediate data transformation and loading.

8. Not Ideal for Highly Custom Transformations

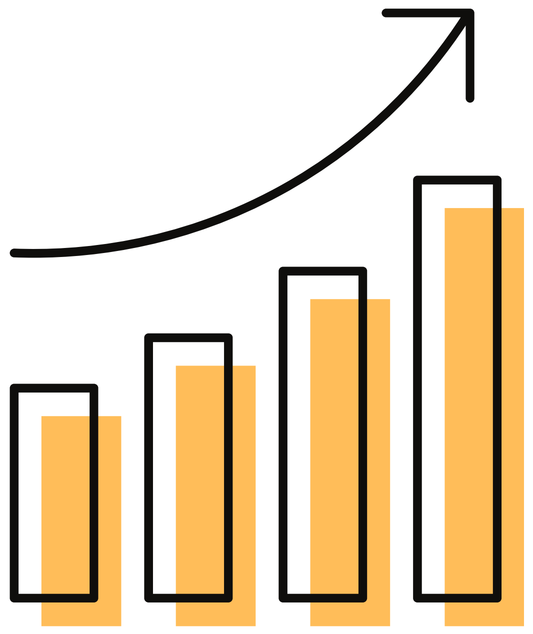
- While data flows provide many built-in transformations, some complex or highly custom transformations may be hard to implement.
- In such cases, using custom code in Databricks or other services might be necessary.

Summary of Data Flows in Azure Data Factory

Data flows in Azure Data Factory provide a visual, no-code environment for designing and executing complex data transformations. They integrate seamlessly with Azure services, offering various built-in transformations like joins, aggregates, and pivots. While data flows are scalable and support schema drift, they may have a learning curve and can become costly for large data volumes. They are best suited for batch processing, data standardization, and ETL scenarios within the Azure ecosystem, providing a flexible and powerful tool for managing data pipelines.

ANIL REDDY CHENCHU

Torture the data, and it will confess to anything



DATA ANALYTICS

Azure Data Factory SQL databricks APACHE Spark Azure Synapse Analytics Power BI

SHARE IF YOU LIKE THE POST

Lets Connect to discuss more on Data

www.linkedin.com/in/chenchuanil