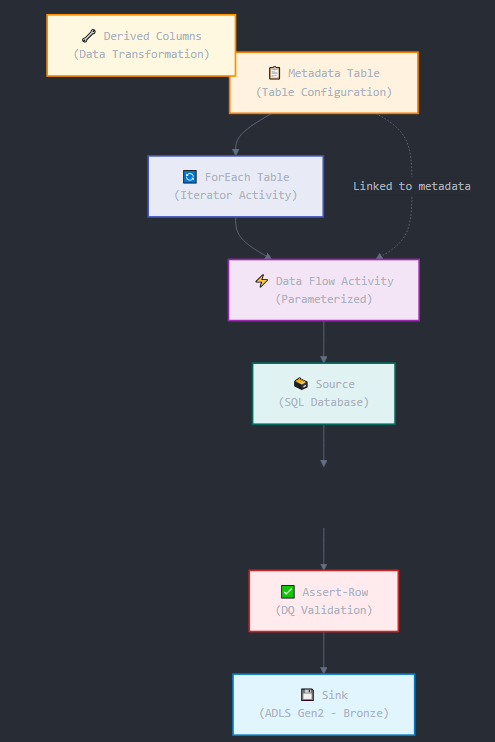
**Case Study: Metadata-Driven Ingestion with Assert-Row DQ in Azure Data Factory**

**Business Scenario**

A fintech company wants to build a **scalable and reusable data ingestion pipeline** to process multiple source tables from Azure SQL Database and load them into a **bronze zone in Azure Data Lake**. Each table has different structures and requires **data quality validation**. The solution should support:

* **Metadata-driven design** for ingesting multiple tables.
* **Custom DQ rules** per table (e.g., non-null, range checks).
* **Flexible mapping** without creating separate data flows for each table.
* Centralized configuration using metadata.

**Target Architecture**



**Step-by-Step Implementation**

**Step 1: Create Metadata Table**

Create a control table in Azure SQL DB (or Excel/Blob CSV) with the following schema:

| **TableName** | **SourceQuery** | **SinkPath** | **DQRules** |
| --- | --- | --- | --- |
| Customer | SELECT \* FROM dbo.Customer | raw/bronze/Customer/ | Name!=null;Email!=null |
| Transactions | SELECT \* FROM dbo.Transactions | raw/bronze/Transactions/ | Amount>0;TransactionDate!=null |
| Products | SELECT \* FROM dbo.Products | raw/bronze/Products/ | Price>0;ProductName!=null |

**Step 2: Create a Parameterized Data Flow**

**Data Flow Name: IngestWithDQ**

**Parameters:**

* pSourceQuery (string)
* pSinkPath (string)
* pDQRules (string)

**Step 3: Add Source Transformation**

* **Source Type**: Azure SQL Database
* **Query**: Use the **parameterized query** pSourceQuery

Turn on **Allow schema drift**  
Set **Import schema** to none or use projection if schema is fixed

**Step 4: Derived Column (Optional)**

* Add calculated columns if needed like IngestionTimestamp = currentUTC().

**Step 5: Assert-Row for Data Quality**

Add an **Assert transformation** to validate rows dynamically based on pDQRules.

**Approach:**

* Use a **custom Flowlet or expression parser** (if rules are simple).
* Or use split(pDQRules, ';') and apply multiple iif() conditions (limited parsing).

**Example Static Approach (basic logic):**

sql

assert(isNull(Name) == false, 'Name cannot be null')

assert(isNull(Email) == false, 'Email cannot be null')

**Advanced Dynamic Option:**

Use **Derived Column** to create a dq\_status column:

sql

dq\_status = iif(isNull(Name) || isNull(Email), 'Fail', 'Pass')

Then apply Conditional Split:

* Pass if dq\_status == 'Pass'
* Fail if dq\_status == 'Fail'

Log or sink failed records separately.

**Step 6: Sink Configuration**

* **Sink Type**: Azure Data Lake Gen2
* **Sink Path**: Use pSinkPath as dynamic path.
* **File Format**: CSV or Parquet
* Enable **Staging** and **Auto Mapping**
* Optional: Add partitioning (e.g., by IngestionDate)

**Step 7: Create a Pipeline to Drive Metadata**

**Pipeline Name: MetadataDrivenIngestion**

1. **Lookup Activity**
   * Query metadata table: SELECT \* FROM MetadataConfig
2. **ForEach Activity**
   * Items: @activity('Lookup1').output.value
   * Inside ForEach:
     + **Execute Data Flow**: IngestWithDQ
       - Map parameters:
         * pSourceQuery: @item().SourceQuery
         * pSinkPath: @item().SinkPath
         * pDQRules: @item().DQRules

**Validation & Monitoring**

**Debugging Data Flow**

* Enable debug mode to test each rule.
* Use **Data Preview** to check if rules are working as intended.

**Logging DQ Failures**

* Optional: Add a Sink for failed rows with path raw/dqfailures/<tablename>/.

**Performance Tuning Recommendations**

| **Optimization** | **Recommendation** |
| --- | --- |
| Source Query Folding | Push filters into SQL query when possible |
| Partitioning | Partition Sink by date or region if applicable |
| Cache Lookups | If joining static DQ tables, cache them |
| Avoid Complex Parsing | Prefer static rules if parsing strings is complex |
| Monitor Skew | Use Monitor → Data Flow run logs |

**Outcome & Benefits**

| **Aspect** | **Benefit** |
| --- | --- |
| Metadata-Driven | Ingest any number of tables without code change |
| Scalable | Just update metadata to ingest a new table |
| Reusable | Central Data Flow used across pipelines |
| Data Quality Rules | Prevents bad data from reaching the Data Lake |
| Modular | Can be extended to include schema validation, alerts |