UDM Framework ETL Patterns

Detailed ETL Methodology and Implementation Patterns

Introduction

The UDM Framework's ETL (Extract, Transform, Load) processes are the heart of the system, orchestrating data movement from diverse sources into a unified warehouse. This document provides detailed patterns, code examples, and best practices for implementing ETL within the framework.

Overview: The Metadata-Driven ETL Philosophy

Traditional ETL vs. UDM ETL

Traditional Approach:

```
New Data Source → Write custom extraction code
                → Write custom transformation logic
                → Write custom load procedures
                → Test and deploy
                = Weeks of development
```

UDM Approach:

```
New Data Source → Configure SourceSystem metadata
                → Map columns in SourceColumnMapping
                → Run standard stored procedures
                → Configure/reuse dataflow templates
                = Days of configuration
```

The Three-Phase ETL Pattern

```
flowchart TB
    subgraph Phase1["PHASE 1: EXTRACTION → Stage Layer"]
        Source1[Source Systems]
        Extract[ETL Tools<br/>SSIS / ADF / Scripts]
        Stage[DS_Stage_*<br/>Raw Tables]
        History[History.*<br/>CDC Tracking]

        Source1 --> Extract
        Extract --> Stage
        Stage --> History
    end

    subgraph Phase2["PHASE 2: PREPARATION → Flow Tables"]
        SP[ReadSourceTable_NewRecords<br/>Stored Procedure]
        Flow[Flow.* Tables<br/>Incremental Batches]
        Mapping[Column Mapping<br/>Applied]

        History --> SP
        SP --> Mapping
        Mapping --> Flow
    end

    subgraph Phase3["PHASE 3: TRANSFORMATION & LOAD → DWH"]
        ADF[Azure Data Factory<br/>Dataflows]
        Lookup[FK Lookups]
        Validate[Business Rules<br/>Validation]
        Split[Conditional Splits]
        DWH[DS_DWH Tables<br/>Success]
        Fallout[Fallout.Fallout<br/>Errors]

        Flow --> ADF
        ADF --> Lookup
        ADF --> Validate
        Lookup --> Split
        Validate --> Split
        Split -->|Valid| DWH
        Split -->|Invalid| Fallout
    end

    Phase1 --> Phase2
    Phase2 --> Phase3
```

```
    style Phase1 fill:#e1f5ff
    style Phase2 fill:#fff4e1
    style Phase3 fill:#e8f5e9
```

Pattern 1: SourceSystem-Driven Loading

Overview

The `Reference.SourceSystem` table is the master control for all ETL processes. Every data integration is defined by a single row in this table.

SourceSystem Table Structure

```
CREATE TABLE [Reference].[SourceSystem](
    [SourceSystemId] [bigint] IDENTITY(1,1) NOT NULL,
    [SourceDescription] [nvarchar](100) NULL,         -- Friendly name
    [SourceDatabase] [nvarchar](100) NULL,            -- DS_Stage_*
    [SourceSchema] [nvarchar](50) NULL,               -- Usually 'dbo'
    [SourceTable] [nvarchar](100) NULL,               -- Source table nar
    [SourceIdentifierColumn] [nvarchar](100) NULL,    -- PK in source
    [TargetSchema] [nvarchar](50) NULL,               -- Sales, Product,
    [TargetTable] [nvarchar](100) NULL,               -- Order, Product,
    [TargetIdentifierColumn] [nvarchar](100) NULL,    -- PK in DWH
    [ValidFromDate] [date] NULL,                      -- When mapping act
    [ValidToDate] [date] NULL,                        -- When mapping exp
    [ParentSourceSystemId] [bigint] NULL,             -- For hierarchies/
    [SourceKeyColumn] [nvarchar](100) NULL,           -- EAV: property na
    [SourceValueColumn] [nvarchar](100) NULL,         -- EAV: property va
    CONSTRAINT [Pk_SourceSystem] PRIMARY KEY ([SourceSystemId])
)
```

Example SourceSystem Entries

Example 1: Standard Dimension Load

```
INSERT INTO Reference.SourceSystem (
    SourceSystemId,
    SourceDescription,
    SourceDatabase,
    SourceSchema,
    SourceTable,
    SourceIdentifierColumn,
    TargetSchema,
    TargetTable,
    TargetIdentifierColumn,
    ValidFromDate,
    ValidToDate
) VALUES (
    42,
    'HealthLabs Products',
    'DS_Stage_HL',
    'dbo',
    'Products',
    'ProductId',           -- PK in source
    'Product',
    'Product',
    'ProductId',           -- PK in DWH
    '2020-01-01',
    '9999-12-31'
)
```

Example 2: EAV Property Load

```
INSERT INTO Reference.SourceSystem (
    SourceSystemId,
    SourceDescription,
    SourceDatabase,
    SourceSchema,
    SourceTable,
    SourceIdentifierColumn,
    TargetSchema,
    TargetTable,
    TargetIdentifierColumn,
    ValidFromDate,
    ValidToDate,
```

```
    ParentSourceSystemId,    -- Links to Product SourceSystem
    SourceKeyColumn,         -- Column containing property names
    SourceValueColumn        -- Column containing property values
) VALUES (
    43,
    'HealthLabs Product Attributes',
    'DS_Stage_HL',
    'dbo',
    'ProductAttributes',
    'AttributeId',
    'Product',
    'ProductProperties',
    'ProductPropertyId',
    '2020-01-01',
    '9999-12-31',
    42,                      -- Parent Product SourceSystemId
    'AttributeName',         -- EAV key column
    'AttributeValue'         -- EAV value column
)
```

SourceColumnMapping Table

Maps individual columns from source to target:

```
CREATE TABLE [Reference].[SourceColumnMapping](
    [SourceColumnMappingId] [bigint] IDENTITY(1,1) NOT NULL,
    [SourceSystemId] [bigint] NOT NULL,              -- FK to SourceSyste
    [SourceColumnName] [nvarchar](100) NULL,         -- Column in source
    [TargetAlias] [nvarchar](100) NOT NULL,          -- Column in Flow/Ta
    CONSTRAINT [PK_SourceColumnMapping] PRIMARY KEY ([SourceColumnMapping
)
```

Example Mappings:

```
-- SourceSystemId = 42 (HealthLabs Products)
INSERT INTO Reference.SourceColumnMapping VALUES
(42, 'ProductId', 'SourceSystemIdentifier'),    -- Source PK → SSI
(42, 'ProductName', 'ProductName'),             -- Direct mapping
(42, 'ProductDescription', 'ProductDescription'),
(42, 'CategoryId', 'ProductGroupId'),           -- Name normalization
```

```
(42, 'Price', 'Price'),
(42, 'Active', 'IsActive')                          -- Name normalization
```

Initialization Workflow

```
flowchart TD
    Start[New Data Source Identified] --> Step1[Step 1: Add SourceSystem

    Step1 --> Insert[INSERT INTO Reference.SourceSystem<br/>Configure sou
    Insert --> Step2[Step 2: Generate Column Mappings]

    Step2 --> InitSP[EXEC InitSourceColumnMapping<br/>Auto-discovers colu
    InitSP --> Step3[Step 3: Manual Mapping Review]

    Step3 --> View[Query VW_ColumnMappingAssistant<br/>View unmapped colu
    View --> Map[Manually map SourceColumnName<br/>to TargetAlias]
    Map --> Step4[Step 4: Populate Flow Table]

    Step4 --> FlowSP[EXEC ReadSourceTable_NewRecords<br/>@SourceSystemId
    FlowSP --> FlowTable[Flow.Product_Product_42<br/>Created & Populated]

    FlowTable --> Ready[Ready for ETL Pipeline]

    style Start fill:#e1f5ff
    style Step1 fill:#fff4e1
    style Step2 fill:#e8f5e9
    style Step3 fill:#f3e5f5
    style Step4 fill:#fce4ec
    style Ready fill:#c8e6c9
```

Step 1: Add SourceSystem Entry

```
-- Manually insert SourceSystem row or use InitSourceSystem SP
INSERT INTO Reference.SourceSystem (...)
VALUES (...)
```

Step 2: Generate Column Mappings

```
-- Stored procedure discovers columns and creates mappings
EXEC [Reference].[InitSourceColumnMapping]
```

Step 3: Manual Mapping Review

```
-- View unmapped columns using helper view
SELECT * FROM [Reference].[VW_ColumnMappingAssistant]
WHERE SourceSystemId = 42

-- Manually update SourceColumnName for each mapping
UPDATE Reference.SourceColumnMapping
SET SourceColumnName = 'ProductName'
WHERE SourceSystemId = 42 AND TargetAlias = 'ProductName'
```

Step 4: Populate Flow Table

```
-- Stored procedure extracts new records to Flow table
EXEC [dbo].[ReadSourceTable_NewRecords] @SourceSystemId = 42
-- Creates/populates: Flow.Product_Product_42
```

Pattern 2: Incremental Loading with Flow Tables

The Flow Pattern

Flow tables are temporary staging tables in the `Flow` schema that hold only new/changed records for a specific SourceSystemId.

Naming Convention: `Flow.[TargetSchema]_[TargetTable]_[SourceSystemId]`

Example: `Flow.Sales_Order_16`

ReadSourceTable_NewRecords Stored Procedure

Purpose: Extract only new records from Stage to Flow

Logic:

1. Query SourceSystem table for source/target configuration
2. Query SourceColumnMapping for column mappings
3. Compare source identifiers to existing DWH records
4. Extract only records not yet in DWH
5. Apply column mappings
6. Insert into Flow table

Pseudo-code:

```
CREATE PROCEDURE [dbo].[ReadSourceTable_NewRecords]
    @SourceSystemId BIGINT
AS
BEGIN
    -- Get source configuration
    DECLARE @SourceDatabase NVARCHAR(100)
    DECLARE @SourceSchema NVARCHAR(50)
    DECLARE @SourceTable NVARCHAR(100)
    DECLARE @SourceIdCol NVARCHAR(100)
    DECLARE @TargetSchema NVARCHAR(50)
    DECLARE @TargetTable NVARCHAR(100)

    SELECT
        @SourceDatabase = SourceDatabase,
        @SourceSchema = SourceSchema,
        @SourceTable = SourceTable,
        @SourceIdCol = SourceIdentifierColumn,
        @TargetSchema = TargetSchema,
        @TargetTable = TargetTable
    FROM Reference.SourceSystem
    WHERE SourceSystemId = @SourceSystemId

    -- Build column mapping list from SourceColumnMapping
    DECLARE @ColumnMappings NVARCHAR(MAX)
    SELECT @ColumnMappings = STRING_AGG(
        CONCAT('[', SourceColumnName, '] AS [', TargetAlias, ']'),
        ', '
    )
    FROM Reference.SourceColumnMapping
    WHERE SourceSystemId = @SourceSystemId

    -- Build Flow table name
```

```
    DECLARE @FlowTable NVARCHAR(200) = CONCAT(
        'Flow.', @TargetSchema, '_', @TargetTable, '_', @SourceSystemId
    )

    -- Create Flow table if doesn't exist
    -- (Simplified - actual implementation more complex)
    IF OBJECT_ID(@FlowTable) IS NULL
    BEGIN
        -- Create table with mapped columns
        EXEC('CREATE TABLE ' + @FlowTable + ' (...)')
    END

    -- Clear existing Flow data
    EXEC('TRUNCATE TABLE ' + @FlowTable)

    -- Insert new records only
    DECLARE @SQL NVARCHAR(MAX) = '
    INSERT INTO ' + @FlowTable + '
    SELECT ' + @ColumnMappings + ',
           ' + @SourceSystemId + ' AS SourceSystemId
    FROM ' + @SourceDatabase + '.' + @SourceSchema + '.' + @SourceTable +
    WHERE NOT EXISTS (
        SELECT 1
        FROM DS_DWH.' + @TargetSchema + '.' + @TargetTable + ' t
        WHERE t.SourceSystemId = ' + @SourceSystemId + '
          AND t.SourceSystemIdentifier = s.' + @SourceIdCol + '
    )'

    EXEC(@SQL)
END
```

Benefits of Flow Tables

1. **Performance**: Only process changed data
2. **Isolation**: Each source system has independent staging
3. **Debugging**: Can inspect Flow tables to see what will be processed
4. **Reprocessing**: Can repopulate Flow tables to retry failed loads
5. **Parallelization**: Multiple sources can populate Flow tables simultaneously

---

Pattern 3: Azure Data Factory Dataflows

Dataflow Architecture

ADF Dataflows transform data from Flow tables into DWH tables, handling:

- Foreign key lookups
- Data validation
- Fallout detection
- Business rule application

Dataflow Structure (LoadSalesOrders Example)

Sources (inputs):

1. `FlowSource`: Read from Flow table
2. `SourceTable`: Original source table (for additional columns)
3. `Leads`: Dimension for FK lookup
4. `Channel`: Dimension for FK lookup
5. `OrderStatus`: Dimension for FK lookup
6. `Fallout`: Existing fallout records (for deduplication)

Sinks (outputs):

1. `SalesOrders`: Target DWH table
2. `FalloutLead`: Records with missing Lead FK (ErrorCode 11)
3. `FalloutChannel`: Records with missing Channel FK (ErrorCode 12)
4. `FalloutOrderStatus`: Records with missing OrderStatus FK (ErrorCode 13)

Transformations:

1. Split by source system
2. Lookup to source table for additional columns
3. Select/filter target columns
4. Lookup dimensions (Leads, Channel, OrderStatus)
5. Conditional splits for fallout routing
6. Derive calculated columns
7. Deduplicate against existing fallout

Dataflow Pattern: FK Lookup with Fallout

Step 1: Lookup Dimension

```
// ADF Dataflow transformation: Lookup
FilterTargetColumns, Leads lookup(
    toString(FilterTargetColumns@LeadId) == Leads@SourceSystemIdentifier,
    multiple: false,
    pickup: 'first',
    asc(Leads@LeadId, true),
    broadcast: 'auto'
) ~> LookupLeads
```

Step 2: Conditional Split on NULL

```
// Split records where LeadId is NULL (FK not found)
AddLeadColumns split(
    isNull(LeadId),
    disjoint: false
) ~> NotFoundLeadIdToFallout@(FalloutMissingLeadId, Rest)
```

Step 3: Route to Fallout

```
// Add fallout metadata
UnionAllErrorcode11 derive(
    ErrorCodeId = 11,
    InsertDate = currentTimestamp()
) ~> SetFalloutParametersErrCode11

// Check if fallout already exists (deduplicate)
SetFalloutParametersErrCode11, Fallout exists(
    SetFalloutParametersErrCode11@ErrorCodeId == Fallout@ErrorCodeId
    && UnionAllErrorcode11@SourceSystemId == Fallout@SourceSystemId
    && UnionAllErrorcode11@SourceSystemIdentifier == Fallout@SourceSyster
    && isNull(SolveDate),
    negate:true,
    broadcast: 'auto'
) ~> FalloutErrorCode11Exists
```

```
// Sink to Fallout table
FalloutErrorCode11Exists sink(
    format: 'table',
    schemaName: 'Fallout',
    tableName: 'Fallout',
    mapColumn(
        ErrorCodeId,
        SourceSystemId,
        SourceSystemIdentifier,
        InsertDate
    )
) ~> FalloutLead
```

Step 4: Continue Pipeline with Valid Records

```
// Only records where LeadId was found continue
NotFoundLeadIdToFallout@Rest, Channel lookup(...) ~> LookupChannel
```

Cascading Lookups Pattern

For records with multiple FK dependencies, lookups cascade:

```
flowchart TD
    Start[Flow Records<br/>Flow.Sales_Order_42] --> Lookup1[Lookup Lead<k

    Lookup1 --> Split1{Lead<br/>Found?}
    Split1 -->|No| Fallout1[Fallout<br/>ErrorCode 11]
    Split1 -->|Yes| Lookup2[Lookup Channel<br/>Marketing.Channel]

    Lookup2 --> Split2{Channel<br/>Found?}
    Split2 -->|No| Fallout2[Fallout<br/>ErrorCode 12]
    Split2 -->|Yes| Lookup3[Lookup OrderStatus<br/>Sales.OrderStatus]

    Lookup3 --> Split3{OrderStatus<br/>Found?}
    Split3 -->|No| Fallout3[Fallout<br/>ErrorCode 13]
    Split3 -->|Yes| Success[Load to DWH<br/>Sales.Order]

    Fallout1 --> FalloutTable[(Fallout.Fallout)]
    Fallout2 --> FalloutTable
    Fallout3 --> FalloutTable
```

```
    style Start fill:#e1f5ff
    style Success fill:#4caf50,color:#fff
    style Fallout1 fill:#ffcdd2
    style Fallout2 fill:#ffcdd2
    style Fallout3 fill:#ffcdd2
    style FalloutTable fill:#f44336,color:#fff
```

Source System Filtering in Lookups

Critical Pattern: When looking up FKs, filter dimensions by same source database:

```
// Filter lookups to same source system database
LookupLeads filter(
    iifNull(Leads@SourceSystemId, 14) == 14
) ~> SS14LeadFilter

LookupChannel filter(
    iifNull(Channel@SourceSystemId, 3) == 3
) ~> SS3ChannelFilter
```

Reason: Prevents cross-contamination when multiple sources provide same dimension.

Pattern 4: EAV / Property Pattern

Overview

The EAV pattern enables flexible attributes without schema changes. It uses the `SourceKeyColumn` and `SourceValueColumn` in SourceSystem to pivot vertical data into property tables.

Source Data Structure (EAV)

Source Table: `ProductAttributes`

```
-- Vertical format (EAV in source)
ProductId  | AttributeName | AttributeValue
```

```
-----------|---------------|---------------
1001       | Color         | Red
1001       | Size          | Large
1001       | Material      | Cotton
1002       | Color         | Blue
1002       | Size          | Medium
```

SourceSystem Configuration

```
INSERT INTO Reference.SourceSystem (
    SourceSystemId,
    SourceDescription,
    SourceDatabase,
    SourceSchema,
    SourceTable,
    SourceIdentifierColumn,
    TargetSchema,
    TargetTable,
    TargetIdentifierColumn,
    ParentSourceSystemId,     -- Links to parent entity
    SourceKeyColumn,          -- 'AttributeName' column
    SourceValueColumn         -- 'AttributeValue' column
) VALUES (
    50,
    'Product Attributes',
    'DS_Stage_Ecom',
    'dbo',
    'ProductAttributes',
    'AttributeId',
    'Product',
    'ProductProperties',
    'ProductPropertyId',
    42,                       -- Parent: Product SourceSystemId
    'AttributeName',
    'AttributeValue'
)
```

PropertyReference Table

Controls which properties are active:

```
CREATE TABLE [Reference].[PropertyReference](
    [PropertyReferenceId] [bigint] IDENTITY(1,1) NOT NULL,
    [PropertyTypeId] [int] NOT NULL,
    [SourceSystemId] [bigint] NOT NULL,
    [SourceColumn] [nvarchar](100) NOT NULL,         -- Property name
    [Ignore] [bit] NOT NULL DEFAULT 1,               -- true = excluded
    CONSTRAINT [PK_PropertyReference] PRIMARY KEY ([PropertyReferenceId])
)
```

Property Loading Workflow

```
flowchart LR
    subgraph Source["SOURCE DATA<br/>(EAV Format)"]
        SourceTable[ProductAttributes<br/>ProductId | AttributeName | Att
    end

    subgraph Config["CONFIGURATION"]
        InitSP[EXEC InitPropertyReference<br/>Discover Properties]
        PropertyRef[PropertyReference Table<br/>Color: Ignore=1<br/>Size:
        Enable[UPDATE Ignore=0<br/>Enable Desired Properties]
    end

    subgraph DWH["DWH RESULT"]
        PropertiesTable[ProductProperties<br/>ProductId | PropertyName |
    end

    SourceTable --> InitSP
    InitSP --> PropertyRef
    PropertyRef --> Enable
    Enable --> PropertiesTable

    style Source fill:#e1f5ff
    style Config fill:#fff4e1
    style DWH fill:#e8f5e9
```

Step 1: Initialize PropertyReference

```
-- Discover available properties
EXEC [Reference].[InitPropertyReference]
```

```
-- Results in PropertyReference entries:
-- PropertyReferenceId | SourceSystemId | SourceColumn | Ignore
-- 1                   | 50             | Color        | 1 (ignored)
-- 2                   | 50             | Size         | 1 (ignored)
-- 3                   | 50             | Material     | 1 (ignored)
```

Step 2: Enable Desired Properties

```
-- Data steward enables properties to load
UPDATE Reference.PropertyReference
SET Ignore = 0
WHERE SourceSystemId = 50
  AND SourceColumn IN ('Color', 'Size', 'Material')
```

Step 3: Load Properties (via pipeline)

- ETL reads PropertyReference for non-ignored properties
- Filters source data to only include enabled properties
- Loads to ProductProperties table

Result in DWH:

```
-- ProductProperties table
ProductPropertyId | ProductId | PropertyName | PropertyValue | ValidFromD
------------------|-----------|--------------|---------------|-----------
1                 | 1001      | Color        | Red           | 2023-01-01
2                 | 1001      | Size         | Large         | 2023-01-01
3                 | 1001      | Material     | Cotton        | 2023-01-01
```

Adding New Properties

Scenario: Source system adds new attribute "Brand"

Workflow:

1. Source data automatically includes new attribute
2. Run `InitPropertyReference` (discovers "Brand")
3. Set `Ignore = 0` for "Brand" in PropertyReference

4. Next pipeline run automatically loads Brand values
5. **No schema changes required**

Pattern 5: Fallout Detection and Resolution

Fallout Table Structure

```
CREATE TABLE [Fallout].[Fallout](
    [FalloutId] [int] IDENTITY(1,1) NOT NULL,
    [ErrorCodeId] [int] NOT NULL,                    -- Type of error
    [SourceSystemId] [bigint] NOT NULL,              -- Which source
    [SourceSystemIdentifier] [nvarchar](100) NOT NULL, -- Which record
    [InsertDate] [datetime] NOT NULL,                -- When error occurr
    [SolveDate] [datetime] NULL,                     -- When resolved
    CONSTRAINT [PK_Fallout2] PRIMARY KEY ([FalloutId])
)
```

Error Code Definitions

ErrorCode Table:

```
CREATE TABLE [Fallout].[ErrorCode](
    [ErrorCodeId] [int] NOT NULL,
    [ErrorDescription] [nvarchar](200) NULL,
    [ErrorCategory] [nvarchar](50) NULL,
    CONSTRAINT [PK_ErrorCode] PRIMARY KEY ([ErrorCodeId])
)
```

Standard Error Codes:

```
INSERT INTO Fallout.ErrorCode VALUES
(11, 'Missing Foreign Key: Lead', 'Referential Integrity'),
(12, 'Missing Foreign Key: Channel', 'Referential Integrity'),
(13, 'Missing Foreign Key: OrderStatus', 'Referential Integrity'),
(14, 'Missing Foreign Key: Product', 'Referential Integrity'),
(15, 'Missing Foreign Key: Customer', 'Referential Integrity'),
(20, 'Invalid Data Type', 'Data Format'),
```

```
(21, 'Invalid Date Format', 'Data Format'),
(22, 'Invalid Numeric Value', 'Data Format'),
(30, 'Business Rule Violation: Negative Amount', 'Business Logic'),
(31, 'Business Rule Violation: Future Date', 'Business Logic'),
(40, 'Duplicate Record Detected', 'Data Quality')
```

Fallout Detection Pattern

In ADF Dataflow:

```
// 1. Lookup dimension
Records, DimensionTable lookup(
    Records@FKColumn == DimensionTable@IdColumn,
    multiple: false,
    pickup: 'first'
) ~> LookupDimension

// 2. Conditional split on NULL
LookupDimension split(
    isNull(FKColumn),
    disjoint: false
) ~> Split@(Fallout, Success)

// 3. Add fallout metadata
Split@Fallout derive(
    ErrorCodeId = 15,  -- Missing FK error code
    InsertDate = currentTimestamp()
) ~> AddFalloutMeta

// 4. Deduplicate (don't re-insert existing fallout)
AddFalloutMeta, ExistingFallout exists(
    AddFalloutMeta@ErrorCodeId == ExistingFallout@ErrorCodeId
    && AddFalloutMeta@SourceSystemId == ExistingFallout@SourceSystemId
    && AddFalloutMeta@SourceSystemIdentifier == ExistingFallout@SourceSys
    && isNull(ExistingFallout@SolveDate),
    negate: true
) ~> NewFalloutOnly

// 5. Sink to Fallout table
NewFalloutOnly sink(...) ~> FalloutSink
```

```
// 6. Continue with successful records
Split@Success ~> NextTransformation
```

Fallout Resolution Workflow

1. Monitor Fallout

```
-- View open fallout issues
SELECT
    ss.SourceDescription,
    ec.ErrorDescription,
    COUNT(*) as FalloutCount,
    MAX(f.InsertDate) as LatestError
FROM Fallout.Fallout f
JOIN Reference.SourceSystem ss ON f.SourceSystemId = ss.SourceSystemId
JOIN Fallout.ErrorCode ec ON f.ErrorCodeId = ec.ErrorCodeId
WHERE f.SolveDate IS NULL
GROUP BY ss.SourceDescription, ec.ErrorDescription
ORDER BY FalloutCount DESC
```

2. Analyze Root Cause

```
-- Get specific fallout records
SELECT
    f.SourceSystemIdentifier,
    f.InsertDate,
    ec.ErrorDescription,
    ss.SourceTable
FROM Fallout.Fallout f
JOIN Reference.SourceSystem ss ON f.SourceSystemId = ss.SourceSystemId
JOIN Fallout.ErrorCode ec ON f.ErrorCodeId = ec.ErrorCodeId
WHERE f.ErrorCodeId = 11  -- Missing Lead FK
  AND f.SolveDate IS NULL
ORDER BY f.InsertDate DESC
```

3. Fix at Source

- Correct data in source system
- Or add missing dimension records

- Or adjust business rules if appropriate

4. Reprocess

- Re-run ETL pipeline for affected SourceSystemId
- Records that now pass validation load to DWH
- Fallout records automatically get `SolveDate` populated

5. Auto-Resolution Logic

```
-- Automatically mark fallout as resolved when record exists in DWH
UPDATE f
SET SolveDate = GETDATE()
FROM Fallout.Fallout f
JOIN Reference.SourceSystem ss ON f.SourceSystemId = ss.SourceSystemId
JOIN DS_DWH.[Schema].[Table] t ON t.SourceSystemId = f.SourceSystemId
                                AND t.SourceSystemIdentifier = f.SourceSy
WHERE f.SolveDate IS NULL
```

Pattern 6: Timeslice Management

Timeslice Fundamentals

Every dimension and some facts include:

- `ValidFromDate`: When record becomes active
- `ValidToDate`: When record becomes inactive (9999-12-31 for current)

Change Detection in Stage Layer

History Tables track source changes:

```
-- Created by BuildHistoryTracking stored procedure
CREATE TABLE [History].[ProductTable](
    [ChangeType] [nvarchar](100) NULL,    -- INSERT, UPDATE, DELETE
    [ChangeTime] [datetime] NULL,         -- Timestamp of change
    -- All columns from original table
    [ProductId] [int] NOT NULL,
```

```
    [ProductName] [nvarchar](200) NULL,
    [Price] [decimal](18,2) NULL
    -- ...
)
```

CDC Logic (triggers or scheduled comparison):

```
-- Detect changes and log to history
INSERT INTO History.ProductTable
SELECT
    'UPDATE' as ChangeType,
    GETDATE() as ChangeTime,
    n.*
FROM dbo.ProductTable n
LEFT JOIN History.ProductTable h ON n.ProductId = h.ProductId
WHERE h.ProductId IS NULL  -- New record
   OR n.ProductName != h.ProductName  -- Changed name
   OR n.Price != h.Price  -- Changed price
```

Timeslice Update Pattern

When source data changes:

```
sequenceDiagram
    participant Source as Source System
    participant ETL as ETL Process
    participant DWH as Product.Product Table

    Note over Source,DWH: Initial State (2023-01-01)
    Source->>ETL: Product PROD-123<br/>Name: "Original"<br/>Price: 19.99
    ETL->>DWH: INSERT ProductId=1001<br/>ValidFromDate: 2023-01-01<br/>Va

    Note over Source,DWH: Change Detected (2023-06-15)
    Source->>ETL: Product PROD-123<br/>Name: "Updated"<br/>Price: 29.99

    Note over ETL: Step 1: Close Old Record
    ETL->>DWH: UPDATE ProductId=1001<br/>SET ValidToDate = 2023-06-15

    Note over ETL: Step 2: Insert New Record
    ETL->>DWH: INSERT ProductId=1525<br/>ProductNK: PROD-123 (same)<br/>V
```

```
    Note over DWH: Result: Two records for PROD-123<br/>Historical: 2023-
```

SQL Implementation:

```
-- Step 1: Close out old record (set ValidToDate)
UPDATE Product.Product
SET ValidToDate = CAST(GETDATE() AS DATE)
WHERE ProductNK = 'PROD-123'
  AND ValidToDate = '9999-12-31'  -- Current record

-- Step 2: Insert new record with changes
INSERT INTO Product.Product (
    ProductNK,
    ProductName,
    Price,
    ValidFromDate,
    ValidToDate,
    SourceSystemId,
    SourceSystemIdentifier
)
VALUES (
    'PROD-123',              -- Same natural key
    'Updated Product Name',  -- New value
    29.99,                   -- New value
    CAST(GETDATE() AS DATE), -- Effective today
    '9999-12-31',            -- Current going forward
    42,
    'PROD-123'
)
```

Result:

```
-- Product table timeline for PROD-123
ProductId | ProductNK  | ProductName           | Price | ValidFromDate |
----------|------------|-----------------------|-------|---------------|-
1001      | PROD-123   | Original Product Name | 19.99 | 2023-01-01    |
1525      | PROD-123   | Updated Product Name  | 29.99 | 2023-06-15    |
```

Visual Timeline:

```
gantt
    title Product PROD-123 Timeslice Timeline
    dateFormat YYYY-MM-DD
    section Historical
    Original Product (Price: 19.99) :done, 1001, 2023-01-01, 2023-06-15
    section Current
    Updated Product (Price: 29.99) :active, 1525, 2023-06-15, 2024-12-31
```

Point-in-Time Queries

Query as of specific date:

```
-- What was the price of PROD-123 on 2023-03-01?
SELECT ProductName, Price
FROM Product.Product
WHERE ProductNK = 'PROD-123'
  AND '2023-03-01' BETWEEN ValidFromDate AND ValidToDate
-- Returns: "Original Product Name", 19.99

-- What is the current price?
SELECT ProductName, Price
FROM Product.Product
WHERE ProductNK = 'PROD-123'
  AND ValidToDate = '9999-12-31'
-- Returns: "Updated Product Name", 29.99
```

Timeslice Joins in Reporting

Join facts to dimensions at point-in-time:

```
-- Orders with product name as it was at order time
SELECT
    o.OrderDate,
    p.ProductName,  -- Name as of OrderDate
    o.Amount
FROM Sales.Order o
JOIN Product.Product p
```

```
    ON o.ProductNK = p.ProductNK
    AND o.OrderDate BETWEEN p.ValidFromDate AND p.ValidToDate
```

Pattern 7: Table Type-Specific Strategies

Strategy for Dimension Tables (Type 2 SCD)

Characteristics:

- Timeslices (ValidFromDate/ValidToDate)
- Natural keys (NK columns)
- Change tracking via history tables

ETL Logic:

1. Compare source to current DWH records (ValidToDate = '9999-12-31')
2. Detect changes in any attribute
3. If changed: Close old record, insert new record
4. If unchanged: Skip
5. If new: Insert with ValidFromDate = today

Strategy for Fact Tables

Characteristics:

- Immutable (rarely updated)
- Large volumes
- Foreign keys to dimensions

ETL Logic:

1. Extract only new records (by identifier comparison)
2. Lookup FKs to dimensions (point-in-time if needed)
3. Validate business rules
4. Insert (rarely update)

Performance Optimization:

- Partition by date

- Batch inserts
- Minimal indexes during load

Strategy for Bridge Tables (Many-to-Many)

Characteristics:

- Link two entities
- Often time-bound

Example: `Bridge.ContactAddress`

```
CREATE TABLE [Bridge].[ContactAddress](
    [ContactAddressId] [bigint] IDENTITY(1,1) NOT NULL,
    [ContactId] [bigint] NOT NULL,
    [AddressId] [bigint] NOT NULL,
    [AddressType] [nvarchar](50) NULL,  -- Home, Work, Billing
    [IsPrimary] [bit] NULL,
    [ValidFromDate] [date] NULL,
    [ValidToDate] [date] NULL,
    CONSTRAINT [PK_ContactAddress] PRIMARY KEY ([ContactAddressId])
)
```

ETL Logic:

1. Load contacts and addresses first (dependencies)
2. Resolve FKs for both sides
3. Detect new/changed relationships
4. Apply timeslices if temporal

Pattern 8: Pipeline Orchestration

Pipeline Structure

Typical Pipeline (for one target table):

```
┌─────────────────────────────────────────────┐
│ PIPELINE: Load_Sales_Orders                 │
```

```
├──────────────────────────────────────────┤
│ 1. Parameters:                           │
│    - SourceSystemId (from trigger)       │
│                                          │
│ 2. Activities:                           │
│    a. Switch on SourceSystemId           │
│    b. For each case:                     │
│       - Execute Dataflow (specific SS)   │
│       - Pass parameters                  │
│                                          │
│ 3. Error Handling:                       │
│    - Log failures                        │
│    - Alert on error                      │
│    - Don't stop other sources            │
└──────────────────────────────────────────┘
```

Master Pipeline Pattern

Load All Sources (for one domain):

```
┌──────────────────────────────────────────┐
│ MASTER PIPELINE: Load_All_Sales          │
├──────────────────────────────────────────┤
│ 1. Load Dimensions (parallel):           │
│    - Load_Sales_Leads                    │
│    - Load_Sales_Publisher                │
│    - Load_Sales_Channel                  │
│    - Load_Sales_OrderStatus              │
│                                          │
│ 2. Load Facts (sequential):              │
│    - Load_Sales_Orders                   │
│    - Load_Sales_Invoices                 │
│                                          │
│ 3. Load Properties (after facts):        │
│    - Load_Sales_OrderProperties          │
└──────────────────────────────────────────┘
```

Dependency Management

Table Insert Order: Use view to determine sequence

```
-- View: Reference.VW_TableInsertOrder
CREATE VIEW [Reference].[VW_TableInsertOrder] AS
SELECT
    TargetSchema,
    TargetTable,
    CASE
        WHEN TargetTable LIKE '%Properties' THEN 3  -- Properties last
        WHEN TargetSchema IN ('Reference') THEN 1   -- Reference first
        WHEN TargetTable IN ('Customer', 'Product', 'Channel') THEN 2  --
        ELSE 2.5  -- Facts after dims
    END as LoadOrder
FROM Reference.SourceSystem
ORDER BY LoadOrder, TargetSchema, TargetTable
```

ETL Best Practices

1. Always Use Incremental Loading

- Never full-reload large tables
- Use `ReadSourceTable_NewRecords` pattern
- Compare identifiers to detect new records

2. Source System Isolation

- Filter lookups by SourceSystemId
- Prevents cross-contamination
- Each source can evolve independently

3. Fail Gracefully with Fallout

- Never let bad data block pipeline
- Route errors to Fallout
- Continue processing valid records
- Monitor and resolve fallout regularly

4. Parameterize Everything

- Use SourceSystemId as parameter
- Build dynamic SQL from metadata
- Reuse dataflow templates

5. Monitor and Log

- Log every pipeline execution
- Track row counts (inserted, updated, fallout)
- Alert on anomalies

6. Test with Fallback

- Test new sources in parallel
- Keep old ETL running during migration
- Compare results before cutover

7. Optimize Lookups

- Use broadcast joins for small dimensions
- Filter dimensions before lookup
- Consider caching frequently-used lookups

8. Handle Timeslices Correctly

- Always check ValidToDate = '9999-12-31' for current
- Use date range joins for point-in-time
- Close old records before inserting new

Common ETL Challenges & Solutions

Challenge 1: Slowly Changing Source Systems

Problem: Source system structure changes (columns added/removed)

Solution:

- Use `allowSchemaDrift: true` in dataflows
- Non-mapped columns ignored automatically

- New columns appear in `VW_ColumnMappingAssistant`
- Add mappings as needed

Challenge 2: Late-Arriving Dimensions

Problem: Fact record arrives before dimension record

Solution:

- Route to fallout (missing FK)
- Load dimension in next cycle
- Reprocess fallout automatically

Challenge 3: Circular Dependencies

Problem: Table A references Table B, which references Table A

Solution:

- Load both without FKs initially
- Update FKs in second pass
- Or use bridge table pattern

Challenge 4: Large Source Tables

Problem: Millions of rows, slow extraction

Solution:

- Partition Flow table extraction by date range
- Process in batches
- Use parallel execution
- Consider CDC at source

Challenge 5: Complex Transformations

Problem: Business logic too complex for dataflow

Solution:

- Use stored procedures for complex logic
- Call from pipeline before dataflow

- Results to temp table, then dataflow reads

Conclusion

The UDM Framework ETL patterns provide:

1. **Metadata-Driven**: Configuration over custom code
2. **Incremental**: Process only changed data
3. **Resilient**: Fallout system handles errors gracefully
4. **Flexible**: EAV pattern for dynamic attributes
5. **Traceable**: Source system tracking on every record
6. **Historical**: Timeslices preserve complete timeline
7. **Scalable**: Parallel processing, partitioning support

By following these patterns, organizations can integrate new data sources in days rather than weeks, maintain data quality through automated monitoring, and build a scalable foundation for analytics.

Next: 05_Implementation_Guide.md - Step-by-step implementation instructions