02 Databricks Lakeflow Concepts

Databricks Lakeflow Concepts

Introduction

This guide explains all fundamental concepts used in the Databricks Lakeflow Declarative Pipeline project. Use these notes to help students understand not just the "how" but the "why" behind each architectural decision.

Part 1: Unity Catalog Fundamentals

1.1 What is Unity Catalog?

Definition: Unity Catalog is Databricks' unified governance solution for data and AI assets. Think of it as a **centralized metadata management system** that provides a single source of truth for all your data.

Key Characteristics:

- **Three-level namespace:** `catalog.schema.table`
- **Cross-cloud compatibility:** Works across AWS, Azure, and GCP
- **Unified governance:** Single place to manage access control, audit logs, and data lineage
- **Fine-grained access control:** Column-level, row-level, and table-level permissions

Real-World Analogy: Think of Unity Catalog like a **library system**:

- The **catalog** is the library building
- **Schemas** are sections (Fiction, Non-Fiction, Reference)
- **Tables** are individual books

Benefits Over Traditional Hive Metastore:

- Centralized management across multiple workspaces
- Built-in data lineage tracking
- ACID transactions at the catalog level
- Easier migration and disaster recovery

1.2 What is a Catalog?

Definition: A catalog is the **top-level container** in Unity Catalog's three-level namespace hierarchy. It's used to organize data assets by business unit, department, project, or environment.

Common Catalog Strategies:

1. **By Environment:**

```
dev_catalog
staging_catalog
production_catalog
```

2. **By Department:**

```
finance_catalog
marketing_catalog
sales_catalog
```

3. **By Project (Our Approach):**

```
lakeflow_dlt
customer_analytics
supply_chain
```

Why We Created `lakeflow_dlt` Catalog:

- Isolates this project's data from other projects
- Makes permissions management easier
- Provides clear project boundaries
- Enables independent lifecycle management

1.3 What is a Schema?

Definition: A schema (also called a database) is the **second-level container** that logically groups related tables, views, and functions. It's where you organize objects by domain or data processing stage.

In Our Project, We Created Four Schemas:

1. `landing_zone`
   - Purpose: Store raw files before processing
   - Contains: Volumes with CSV files
   - Analogy: Airport arrival terminal - data lands here first
2. `bronze`
   - Purpose: Raw, unprocessed data ingestion
   - Contains: Streaming tables with original data types (all strings)
   - Analogy: Raw materials warehouse
3. `silver`
   - Purpose: Cleaned, validated, conformed data
   - Contains: Streaming tables with proper data types + materialized views
   - Analogy: Quality-inspected, organized inventory

4. `gold`
    - Purpose: Business-level aggregates and KPIs
    - Contains: Materialized views optimized for analytics
    - Analogy: Final products ready for consumption

Why This Schema Design Matters:

- **Separation of Concerns:** Each schema has a clear purpose
- **Data Quality Progression:** Data gets cleaner as it moves through layers
- **Access Control:** You can grant different permissions per schema
- **Performance Optimization:** Each layer optimized for its use case

---

Part 2: Volumes - Modern Data Storage

2.1 What is a Volume?

Definition: A volume is a **Unity Catalog object** that provides governance for non-tabular data (files) like CSVs, JSON, Parquet files, images, PDFs, etc.

Think of Volumes as:

- A governed file system within Unity Catalog
- Like tables, but for files instead of structured data
- Have full Unity Catalog features: permissions, lineage, audit logs

Traditional Problem Volumes Solve: Before volumes, you had to:

1. Create external locations (complex)
2. Manage storage credentials separately
3. No centralized governance for files
4. Difficult permission management

With Volumes:

1. Create volume in 3 clicks
2. Upload files through UI
3. Automatic governance
4. Unified permission model

2.2 Managed vs External Volumes

This is a critical concept students often find confusing.

Managed Volumes

Definition: Storage is **fully managed by Databricks**. When you delete the volume, the underlying files are also deleted.

Characteristics:

- Databricks controls the storage location
- Files stored in Unity Catalog's metastore storage
- Full lifecycle management by Databricks
- Simpler to create and manage

Use Cases:

- Temporary staging areas
- Processing zones where data is transient
- When you want Databricks to handle everything
- **Our project uses managed volumes** for the landing zone

Creation Example:

```sql
CREATE VOLUME lakeflow_dlt.landing_zone.fact_and_dimension_files
MANAGED;
```

Path Structure:

```
/Volumes/catalog_name/schema_name/volume_name/
```

External Volumes

Definition: Storage is in an **external cloud storage location** (S3, ADLS, GCS) that you control. Deleting the volume only removes the metadata; files remain.

Characteristics:

- You manage the storage location
- Files persist after volume deletion
- Requires external location and storage credential setup
- More control over storage lifecycle

Use Cases:

- Long-term data archival
- Shared storage across multiple systems
- Regulatory requirements for data location
- When data needs to outlive the Databricks environment

Creation Example:

```sql
CREATE EXTERNAL VOLUME lakeflow_dlt.landing_zone.external_files
LOCATION 's3://my-bucket/data/'
```

Comparison Table

Feature	Managed Volume	External Volume
Storage Control	Databricks	User
Deletion Behavior	Files deleted with volume	Files persist
Setup Complexity	Simple (3 clicks)	Complex (credentials needed)
Use Case	Transient data	Persistent data
Cost Management	Bundled with Databricks	Separate cloud storage bill
Our Project	✅ Used	❌ Not used

Why We Chose Managed Volume:

- Simpler for learning purposes
- Databricks handles all storage complexity
- Perfect for staging data before processing
- No need to manage cloud storage credentials

Part 3: Streaming Live Tables

3.1 What is a Streaming Live Table?

Definition: A Streaming Live Table is a **continuously updating table** that processes data incrementally as it arrives, rather than reprocessing everything each time.

Key Concepts:

1. **Incremental Processing:**
   - Only new or changed data is processed
   - Maintains state to track what's been processed
   - Exactly-once processing guarantees
2. **Continuous Execution:**
   - Always ready to process new data
   - No manual triggers needed
   - Real-time or near-real-time updates
3. **Stateful Operations:**
   - Remembers what data has been processed

- Uses checkpoints to track progress
- Can recover from failures without data loss

Syntax in Our Project:

```sql
CREATE STREAMING LIVE TABLE bronze.fact_sales
COMMENT "Raw fact sales from volume to bronze schema"
AS SELECT * FROM CLOUD_FILES(
  '/Volumes/lakeflow_dlt/landing_zone/fact_and_dimension_files/fact_sales',
  'csv',
  map('header', 'true')
);
```

3.2 Streaming vs Materialized View

This is crucial for students to understand.

Streaming Live Table

Purpose: Continuous, incremental data ingestion and processing

Characteristics:

- Always "listening" for new data
- Maintains checkpoints
- Processes only new records
- Can't be queried directly in production mode
- Lower latency for data updates

When to Use:

- Bronze layer (ingesting raw data)
- Silver layer (transforming streaming data)
- Real-time pipelines
- When data arrives continuously

Example:

```sql
CREATE STREAMING LIVE TABLE silver.fact_sales
AS SELECT ... FROM STREAM(lakeflow_dlt.bronze.fact_sales);
```

Materialized View

Purpose: Pre-computed query results that are periodically refreshed

Characteristics:

- Snapshot of data at a point in time
- Full refresh on each update (by default)
- Can be queried directly
- Better for complex aggregations
- Higher latency acceptable

When to Use:

- Silver layer (complex joins with constraints)
- Gold layer (business aggregations)
- Expensive computations you want to pre-calculate
- When complete refresh is acceptable

Example:

```sql
CREATE OR REFRESH MATERIALIZED VIEW gold.category_sales_summary
AS SELECT category, SUM(revenue)
FROM lakeflow_dlt.silver.cleaned_sales_data
GROUP BY category;
```

Comparison Table

Aspect	Streaming Live Table	Materialized View
Update Pattern	Incremental (append)	Full refresh
Data Latency	Real-time/Near real-time	Batch intervals
Query Access	Limited	Full access
Use Case	Data ingestion/transformation	Aggregations/Analytics
Bronze Layer	✅	❌
Silver Layer	✅ (transformations)	✅ (joins + constraints)
Gold Layer	❌	✅

3.3 The STREAM() Function

What It Does: The `STREAM()` function tells Databricks to read data **incrementally** from the source table.

Without STREAM():

```sql
-- This will fail when creating a streaming table
CREATE STREAMING LIVE TABLE silver.fact_sales
AS SELECT * FROM lakeflow_dlt.bronze.fact_sales;
-- Error: Cannot create streaming table from batch query
```

With STREAM():

```sql
-- This works correctly
CREATE STREAMING LIVE TABLE silver.fact_sales
AS SELECT * FROM STREAM(lakeflow_dlt.bronze.fact_sales);
-- Success: Incremental processing enabled
```

Why It's Critical:

- Enables incremental data processing
- Maintains streaming semantics through the pipeline
- Required when creating streaming tables from streaming sources
- Prevents accidental full table scans

Teaching Analogy: Think of `STREAM()` as a **"tail -f"** command in Unix:

- Without it: Reads entire file every time (batch)
- With it: Only reads new lines added (streaming)

Part 4: AutoLoader and CLOUD_FILES()

4.1 What is AutoLoader?

Definition: AutoLoader is Databricks' **intelligent file ingestion system** that automatically detects and processes new files as they arrive in cloud storage.

Core Capabilities:

1. **Automatic File Discovery:**
   - Continuously monitors directories
   - Detects new files within seconds
   - No manual file tracking needed
2. **Exactly-Once Processing:**
   - Each file processed exactly once
   - No duplicates
   - No missed files
3. **Schema Inference and Evolution:**
   - Automatically detects schema
   - Handles schema changes gracefully
   - Can rescue unparseable data
4. **Scalability:**
   - Handles billions of files
   - Optimized for cloud object stores

- Directory listing optimization

5. **State Management:**
   - Maintains checkpoint of processed files
   - Recovers from failures
   - Tracks file metadata

4.2 CLOUD_FILES() Function

Syntax:

```SQL
CLOUD_FILES(path, format, options)
```

In Our Project:

```SQL
CREATE STREAMING LIVE TABLE bronze.fact_sales
AS SELECT * FROM CLOUD_FILES(
  '/Volumes/lakeflow_dlt/landing_zone/fact_and_dimension_files/fact_sales',
  'csv',
  map('header', 'true')
);
```

Parameters Explained:

1. **Path** (`/Volumes/ ... /fact_sales`):
   - Directory to monitor (not individual file)
   - Can be volume path, S3 bucket, ADLS container
   - Supports wildcards: `/data/year=*/month=*/*.csv`
2. **Format** (`'csv'`):
   - File format: csv, json, parquet, avro, orc, text
   - AutoLoader detects schema based on format
   - Binary formats also supported
3. **Options** (`map('header', 'true')`):
   - Configuration specific to file format
   - For CSV: header, delimiter, quote character
   - For JSON: multiLine, dateFormat
   - For schema evolution: mergeSchema, rescuedDataColumn

Additional Options You Can Use:

```sql
CLOUD_FILES(
  '/path',
  'csv',
  map(
    'header', 'true',
    'delimiter', ',',
    'inferSchema', 'true',
    'cloudFiles.schemaHints', 'price DOUBLE, quantity INT',
    'cloudFiles.schemaEvolutionMode', 'rescue',
    'cloudFiles.inferColumnTypes', 'true'
  )
)
```

4.3 Why Directory Structure Matters

Our Directory Setup:

```
fact_and_dimension_files/
├── fact_sales/
│   ├── fact_sales.csv
│   └── fact_sales_2025.csv
├── dim_customers/
│   └── customer.csv
├── dim_products/
│   └── product.csv
└── dim_regions/
    └── region.csv
```

Why Separate Directories?

1. **Schema Isolation:**
   - Each directory has one schema
   - Prevents schema conflicts
   - Clean separation of concerns
2. **AutoLoader Efficiency:**
   - Monitors specific directory
   - Only processes relevant files
   - Faster file discovery
3. **Incremental Load Testing:**
   - Can add files to specific directory
   - Tests isolated to one table

- No cross-contamination

Common Mistake:

```
❌ BAD:
fact_and_dimension_files/
├── fact_sales.csv
├── customer.csv
├── product.csv
└── region.csv

Problem: All files in one directory, AutoLoader
can't distinguish which schema to apply
```

4.4 How AutoLoader Enables Incremental Load

The Incremental Load Test:

Step 1: Initial Load

- Upload `fact_sales.csv` (50,000 records)
- Pipeline processes all records
- Checkpoint created: `{processed: ['fact_sales.csv']}`

Step 2: Add New File

- Upload `fact_sales_2025.csv` (2,526 records)
- AutoLoader detects new file
- Checkpoint updated: `{processed: ['fact_sales.csv', 'fact_sales_2025.csv']}`

Step 3: Processing

- Only 2,526 new records processed
- No reprocessing of original 50,000
- Total records: 52,526 (assuming no duplicates)

Why This Is Powerful:

- **Efficiency:** Processes only new data
- **Scalability:** Handles millions of files
- **Reliability:** Exactly-once guarantees
- **Simplicity:** No manual file tracking

Behind the Scenes:

```
AutoLoader Checkpoint:
{
  "processedFiles": [
    {
      "path": "/Volumes/ ... /fact_sales/fact_sales.csv",
      "modificationTime": "2024-01-15T10:00:00",
      "size": 1048576,
      "processed": true
    },
    {
      "path": "/Volumes/ ... /fact_sales/fact_sales_2025.csv",
      "modificationTime": "2025-10-29T14:30:00",
      "size": 52428,
      "processed": true
    }
  ]
}
```

Part 5: Medallion Architecture Deep Dive

5.1 What is Medallion Architecture?

Definition: A data architecture pattern that organizes data into three layers (Bronze, Silver, Gold) based on **data quality and refinement level**.

Origin:

- Popularized by Databricks
- Based on decades of data warehousing best practices
- Combines data lake flexibility with data warehouse structure

Core Philosophy:

> "Incremental data refinement: Each layer adds value while preserving raw data"

5.2 Bronze Layer - Raw Data Landing

Purpose: Ingest data exactly as it arrives, with **minimal transformation**

Characteristics:

- **Complete Fidelity:** Exact copy of source data
- **Audit Trail:** Can always trace back to original
- **Schema:** Often all strings or raw bytes
- **Append-Only:** Historical record of all data received

- **Fast Ingestion:** Optimized for speed, not usability

In Our Project:

```sql
CREATE STREAMING LIVE TABLE bronze.fact_sales
AS SELECT * FROM CLOUD_FILES( ... );
```

Data Types: All STRING

```
sale_id: STRING  (will be INT later)
order_date: STRING  (will be DATE later)
quantity: STRING  (will be INT later)
```

Why All Strings?

- Prevents ingestion failures from type mismatches
- Handles dirty data gracefully
- Type conversion happens in Silver layer where you can handle errors

Real-World Analogy: Bronze layer is like a **package receiving dock**:

- Accepts all deliveries as-is
- Doesn't inspect contents
- Keeps receipt of everything received
- Can always verify what was delivered

Best Practices:

- Never filter or drop data in Bronze
- Add metadata columns: ingestion_timestamp, source_file
- Keep forever (or very long retention)
- One Bronze table per source system

5.3 Silver Layer - Cleaned and Conformed

Purpose: Clean, validate, and standardize data for **reliable analytics**

Characteristics:

- **Type Safety:** Proper data types (INT, DATE, DECIMAL)
- **Data Quality:** Validation rules and constraints
- **Standardization:** Consistent formats and naming
- **Enrichment:** Joins, lookups, derived columns
- **Still Detailed:** Maintains grain of Bronze data

In Our Project:

Part 1: Streaming Tables with Type Conversion

```sql
CREATE STREAMING LIVE TABLE silver.fact_sales
AS SELECT
  CAST(sale_id AS INT) AS sale_id,
  TO_DATE(order_date, 'dd/MM/yyyy') AS order_date,
  CAST(customer_id AS INT) AS customer_id,
  CAST(product_id AS INT) AS product_id,
  CAST(quantity AS INT) AS quantity,
  CAST(discount AS INT) AS discount,
  CAST(region_id AS INT) AS region_id,
  CAST(channel AS STRING) AS channel,
  CAST(promo_code AS STRING) AS promo_code
FROM STREAM(lakeflow_dlt.bronze.fact_sales);
```

Part 2: Materialized View with Constraints

```sql
CREATE OR REFRESH MATERIALIZED VIEW silver.cleaned_sales_data
(
  CONSTRAINT sales_quantity_check
    EXPECT (quantity IS NOT NULL)
    ON VIOLATION DROP ROW,

  CONSTRAINT sales_channel_check
    EXPECT (channel IS NOT NULL)
    ON VIOLATION DROP ROW,

  CONSTRAINT sales_promo_code_check
    EXPECT (promo_code IS NOT NULL)
    ON VIOLATION DROP ROW,

  CONSTRAINT customer_email_check
    EXPECT (email IS NOT NULL)
    ON VIOLATION DROP ROW
)
AS SELECT
  FS.sale_id,
  FS.order_date,
  C.first_name,
  C.last_name,
  C.email,
  P.product_name,
  P.category,
  P.price,
  R.region_name,
  R.country,
  FS.quantity * P.price AS revenue
FROM lakeflow_dlt.silver.fact_sales FS
LEFT JOIN lakeflow_dlt.silver.customers C ON FS.customer_id = C.customer_id
LEFT JOIN lakeflow_dlt.silver.products P ON FS.product_id = P.product_id
LEFT JOIN lakeflow_dlt.silver.regions R ON FS.region_id = R.region_id;
```

Why Two Approaches in Silver?

1. **Streaming Tables:** For simple type conversion
2. **Materialized Views:** For complex joins and constraints

Constraint Actions:

```sql
ON VIOLATION DROP ROW      -- Removes bad data
ON VIOLATION FAIL          -- Stops pipeline (strict)
ON VIOLATION QUARANTINE    -- Moves to separate table for review
```

Our Results:

- Input: 50,000 records
- Output: 12,935 records (25.9%)
- Dropped: 37,065 records (74.1%)

Constraint Breakdown:

- `sales_promo_code_check`: 36,533 dropped (73.1%)
- `sales_channel_check`: 999 dropped (2%)
- `customer_email_check`: 971 dropped (1.9%)
- `sales_quantity_check`: 44 dropped (<0.1%)

Production Note: In real projects, 74% drop rate is concerning. Consider:

- Using `QUARANTINE` to investigate
- Making constraints less strict
- Fixing data quality at source

Real-World Analogy: Silver layer is like a **quality inspection facility**:

- Checks each item meets specifications
- Rejects defective items
- Standardizes measurements
- Prepares for final assembly

5.4 Gold Layer - Business Aggregates

Purpose: Business-level aggregations optimized for **analytics and reporting**

Characteristics:

- **Business Metrics:** KPIs, summaries, rankings
- **Aggregated Data:** Rolled up by time, customer, product
- **Denormalized:** Optimized for query performance
- **Business Logic:** Complex calculations in one place
- **End-User Ready:** Directly consumable by BI tools

In Our Project:

Aggregate 1: Category Sales Summary

```sql
CREATE OR REFRESH MATERIALIZED VIEW gold.category_sales_summary
AS SELECT
  category,
  YEAR(order_date) AS year,
  SUM(revenue) AS total_revenue
FROM lakeflow_dlt.silver.cleaned_sales_data
GROUP BY category, YEAR(order_date)
ORDER BY category, year;
```

Output: 56 records (7 categories × 8 years)

Aggregate 2: Customer Revenue Ranking by Region

```sql
CREATE OR REFRESH MATERIALIZED VIEW
  gold.revenue_by_customers_in_each_region_by_ranking
AS SELECT
  region_name,
  customer_id,
  first_name,
  last_name,
  SUM(revenue) AS total_revenue,
  RANK() OVER (PARTITION BY region_name
               ORDER BY SUM(revenue) DESC) AS revenue_rank
FROM lakeflow_dlt.silver.cleaned_sales_data
GROUP BY region_name, customer_id, first_name, last_name
ORDER BY region_name, revenue_rank;
```

Output: ~52 records (top customers per region)

Aggregate 3: Customer Lifetime Value

```sql
CREATE OR REFRESH MATERIALIZED VIEW
  gold.customer_lifetime_value_estimation
AS SELECT
  customer_id,
  first_name,
  last_name,
  email,
  VIP,
  COUNT(DISTINCT sale_id) AS total_orders,
  SUM(quantity) AS total_items_purchased,
  SUM(revenue) AS lifetime_value,
  AVG(revenue) AS average_order_value,
  MIN(order_date) AS first_purchase_date,
  MAX(order_date) AS last_purchase_date,
  DATEDIFF(MAX(order_date), MIN(order_date)) AS customer_tenure_days
FROM lakeflow_dlt.silver.cleaned_sales_data
GROUP BY customer_id, first_name, last_name, email, VIP
ORDER BY lifetime_value DESC;
```

Output: ~4,600 records (one per customer)

Real-World Analogy: Gold layer is like **retail store displays**:

- Products organized for easy browsing
- Promotional displays (top sellers)
- Price comparisons ready
- Everything optimized for customers

Why Gold Layer Matters:

- **Performance:** Pre-computed results = fast queries
- **Consistency:** Same business logic everywhere
- **Simplicity:** Business users don't need SQL expertise
- **Governance:** Single source of truth for metrics

5.5 Medallion Architecture Benefits

1. Data Quality Progression

```
Bronze:  Raw, may contain errors
  ↓
Silver:  Cleaned, validated, usable
  ↓
Gold:    Aggregated, optimized, insights
```

2. Flexibility

- Can rebuild Silver/Gold from Bronze
- Add new transformations without re-ingesting
- Test new logic without affecting production

3. Performance Optimization

- Each layer optimized for its purpose
- Bronze: Fast ingestion
- Silver: Efficient joins
- Gold: Fast analytical queries

4. Clear Ownership

- Data Engineers: Own Bronze & Silver
- Analytics Engineers: Own Gold
- Data Analysts: Consume Gold

5. Compliance and Auditing

- Bronze: Complete audit trail
- Silver: Data quality metrics
- Gold: Business metric validation

Part 6: Declarative Pipelines vs Traditional Approaches

6.1 What is a Declarative Pipeline?

Definition: A pipeline where you **declare what you want** (the desired end state), not how to achieve it (the steps to get there).

Imperative Approach (Traditional):

```python
# You write ALL the logic
df = spark.read.csv("path/to/file.csv")
df_cleaned = df.filter(col("quantity").isNotNull())
df_cleaned = df_cleaned.withColumn("order_date", to_date(col("order_date")))
df_cleaned.write.mode("append").saveAsTable("silver.fact_sales")

# You handle:
# - When to run
# - What changed
# - Error recovery
# - Checkpoint management
# - Schema evolution
```

Declarative Approach (Lakeflow):

```sql
-- You declare what you want
CREATE STREAMING LIVE TABLE silver.fact_sales
(
  CONSTRAINT quantity_check EXPECT (quantity IS NOT NULL)
)
AS SELECT
  CAST(sale_id AS INT),
  TO_DATE(order_date) AS order_date
FROM STREAM(bronze.fact_sales);

-- Databricks handles:
-- ✓ Scheduling
-- ✓ Incremental processing
-- ✓ Error recovery
-- ✓ Checkpoint management
-- ✓ Schema evolution
```

Key Difference:

- **Imperative:** "Here's HOW to process data"
- **Declarative:** "Here's WHAT the data should look like"

6.2 Traditional Approach: External Locations + Spark Notebooks

Typical Traditional Setup:

Step 1: Configure External Storage

```sql
-- Create storage credential
CREATE STORAGE CREDENTIAL my_credential
WITH (AZURE_SERVICE_PRINCIPAL
      'client_id' = '...',
      'client_secret' = '...',
      'tenant_id' = '...');

-- Create external location
CREATE EXTERNAL LOCATION my_external_location
URL 'abfss://container@storage.dfs.core.windows.net/path'
WITH (CREDENTIAL my_credential);

-- Grant permissions
GRANT READ FILES ON EXTERNAL LOCATION my_external_location TO `user@company.com`;
```

Step 2: Write Ingestion Notebook

PYTHON

```python
# Notebook 1: Bronze Ingestion
from pyspark.sql import SparkSession
from delta.tables import DeltaTable

# Configuration
source_path = "abfss://container@storage/data/"
checkpoint_path = "abfss://container@storage/checkpoints/bronze/"
target_table = "catalog.bronze.fact_sales"

# Read with AutoLoader (Structured Streaming)
df = (spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "csv")
  .option("header", "true")
  .load(source_path))

# Write to Delta
(df.writeStream
  .format("delta")
  .option("checkpointLocation", checkpoint_path)
  .option("mergeSchema", "true")
  .trigger(availableNow=True)
  .toTable(target_table))
```

Step 3: Write Transformation Notebook

PYTHON

```python
# Notebook 2: Silver Transformation
from pyspark.sql.functions import *

# Read from Bronze
bronze_df = spark.readStream.table("catalog.bronze.fact_sales")

# Transform
silver_df = (bronze_df
  .withColumn("sale_id", col("sale_id").cast("int"))
  .withColumn("order_date", to_date(col("order_date"), "dd/MM/yyyy"))
  .withColumn("quantity", col("quantity").cast("int"))
  .filter(col("quantity").isNotNull()))

# Write to Silver
(silver_df.writeStream
  .format("delta")
  .option("checkpointLocation", checkpoint_path_silver)
  .trigger(availableNow=True)
  .toTable("catalog.silver.fact_sales"))
```

Step 4: Write Aggregation Notebook

PYTHON

```python
# Notebook 3: Gold Aggregation
gold_df = spark.sql("""
  SELECT
    category,
    YEAR(order_date) as year,
    SUM(revenue) as total_revenue
  FROM catalog.silver.cleaned_sales_data
  GROUP BY category, YEAR(order_date)
""")

gold_df.write.mode("overwrite").saveAsTable("catalog.gold.category_sales")
```

Step 5: Orchestrate with Jobs

PYTHON

```
# Create multi-task job
{
  "name": "Data Pipeline",
  "tasks": [
    {
      "task_key": "bronze_ingestion",
      "notebook_task": {"notebook_path": "/notebooks/bronze"},
      "cluster": {...}
    },
    {
      "task_key": "silver_transformation",
      "depends_on": [{"task_key": "bronze_ingestion"}],
      "notebook_task": {"notebook_path": "/notebooks/silver"},
      "cluster": {...}
    },
    {
      "task_key": "gold_aggregation",
      "depends_on": [{"task_key": "silver_transformation"}],
      "notebook_task": {"notebook_path": "/notebooks/gold"},
      "cluster": {...}
    }
  ]
}
```

6.3 Lakeflow Declarative Approach

Same Pipeline, Declarative Style:

Single Pipeline Definition:

```sql
-- bronze_transformations.sql
CREATE STREAMING LIVE TABLE bronze.fact_sales
AS SELECT * FROM CLOUD_FILES('/Volumes/ ... /fact_sales', 'csv');

-- silver_transformations.sql
CREATE STREAMING LIVE TABLE silver.fact_sales
AS SELECT
  CAST(sale_id AS INT) AS sale_id,
  TO_DATE(order_date, 'dd/MM/yyyy') AS order_date,
  CAST(quantity AS INT) AS quantity
FROM STREAM(lakeflow_dlt.bronze.fact_sales);

-- gold_transformations.sql
CREATE OR REFRESH MATERIALIZED VIEW gold.category_sales_summary
AS SELECT
  category,
  YEAR(order_date) AS year,
  SUM(revenue) AS total_revenue
FROM lakeflow_dlt.silver.cleaned_sales_data
GROUP BY category, YEAR(order_date);
```

That's It! Everything else is automated:

- ✓ No cluster configuration
- ✓ No checkpoint management
- ✓ No orchestration code
- ✓ No error handling code
- ✓ No streaming triggers
- ✓ No dependency management

Thank you!