**Databricks: Delta Lake**

An open-source storage layer that brings ACID transactions, scalable metadata handling, and unifies batch and streaming data processing on your data lake.

**Navigation**: Data > Tables (Delta tables) or Workspace > Notebooks (create/manage Delta tables)

Contents

[Step 1: Create a Delta Table 1](#_Toc202726514)

[Option A: SQL Syntax 1](#_Toc202726515)

[Option B: PySpark DataFrame API 1](#_Toc202726516)

[Step 2: Insert Data 2](#_Toc202726517)

[Step 3: Query Data 2](#_Toc202726518)

[Step 4: Update Records 2](#_Toc202726519)

[Step 5: Delete Records 3](#_Toc202726520)

[Step 6: Merge (Upsert) Records 3](#_Toc202726521)

[Step 7: Describe Table Schema and History 3](#_Toc202726522)

[Step 8: Optimize and Vacuum 4](#_Toc202726523)

[Step 9: Drop the Table 4](#_Toc202726524)

[Step 10: Schedule as Job 4](#_Toc202726525)

[Quick Reference Table 4](#_Toc202726526)

[Databricks Platform Components – Phases, Activities, and Timelines 5](#_Toc202726527)

[1. Delta Lake 5](#_Toc202726528)

[**Delta Lake – RACI Matrix** 5](#_Toc202726529)

# Step 1: Create a Delta Table

## Option A: SQL Syntax

**Managed Delta Table**

sql

CopyEdit

CREATE TABLE my\_catalog.default.customer\_data (

customer\_id STRING,

name STRING,

age INT,

country STRING

)

USING DELTA;

**External Delta Table**

sql

CopyEdit

CREATE TABLE my\_catalog.default.customer\_data\_external

USING DELTA

LOCATION '/mnt/datalake/delta/customer\_data\_external';

## Option B: PySpark DataFrame API

python

CopyEdit

data = [

("C001", "Alice", 30, "USA"),

("C002", "Bob", 45, "UK"),

("C003", "Charlie", 28, "Canada")

]

columns = ["customer\_id", "name", "age", "country"]

df = spark.createDataFrame(data, columns)

df.write.format("delta").saveAsTable("my\_catalog.default.customer\_data\_df")

# Step 2: Insert Data

**SQL**

sql

CopyEdit

INSERT INTO my\_catalog.default.customer\_data

VALUES

('C004', 'Diana', 35, 'Germany'),

('C005', 'Eve', 40, 'France');

**PySpark**

python

CopyEdit

new\_data = [("C006", "Frank", 29, "Italy")]

df\_new = spark.createDataFrame(new\_data, columns)

df\_new.write.format("delta").mode("append").saveAsTable("my\_catalog.default.customer\_data\_df")

# Step 3: Query Data

**SQL**

sql

CopyEdit

SELECT \* FROM my\_catalog.default.customer\_data;

**PySpark**

python

CopyEdit

df = spark.read.table("my\_catalog.default.customer\_data")

df.show()

# Step 4: Update Records

**SQL**

sql

CopyEdit

UPDATE my\_catalog.default.customer\_data

SET age = 50

WHERE customer\_id = 'C002';

# Step 5: Delete Records

**SQL**

sql

CopyEdit

DELETE FROM my\_catalog.default.customer\_data

WHERE age < 30;

# Step 6: Merge (Upsert) Records

**SQL**

sql

CopyEdit

MERGE INTO my\_catalog.default.customer\_data AS target

USING (

SELECT 'C002' AS customer\_id, 'Bob Jr' AS name, 46 AS age, 'UK' AS country

) AS source

ON target.customer\_id = source.customer\_id

WHEN MATCHED THEN UPDATE SET \*

WHEN NOT MATCHED THEN INSERT \*;

# Step 7: Describe Table Schema and History

**View Schema**

sql

CopyEdit

DESCRIBE TABLE my\_catalog.default.customer\_data;

**View Time Travel History**

sql

CopyEdit

DESCRIBE HISTORY my\_catalog.default.customer\_data;

# Step 8: Optimize and Vacuum

**Optimize Table**

sql

OPTIMIZE my\_catalog.default.customer\_data;

**Vacuum Old Files**

sql

CopyEdit

VACUUM my\_catalog.default.customer\_data RETAIN 168 HOURS;

# Step 9: Drop the Table

**SQL**

sql

DROP TABLE my\_catalog.default.customer\_data;

# Step 10: Schedule as Job

In Databricks UI:

1. Go to Workspace > Notebooks and save your notebook.
2. Go to Jobs > Create Job.
3. Select the notebook as the task.
4. Set the schedule (e.g., daily).

# Quick Reference Table

|  |  |  |
| --- | --- | --- |
| **Operation** | **SQL Example** | **PySpark Example** |
| Create Table | CREATE TABLE ... USING DELTA | df.write.format("delta").saveAsTable(...) |
| Insert Data | INSERT INTO ... VALUES ... | df.write.mode("append").saveAsTable(...) |
| Query | SELECT \* FROM ... | spark.read.table(...).show() |
| Update | UPDATE ... SET ... WHERE ... | (Use SQL) |
| Delete | DELETE FROM ... WHERE ... | (Use SQL) |
| Merge (Upsert) | MERGE INTO ... USING ... ON ... | (Use SQL) |
| Optimize | OPTIMIZE ... | (Use SQL) |
| Vacuum | VACUUM ... RETAIN ... | (Use SQL) |
| Describe Schema | DESCRIBE TABLE ... | (Use SQL) |
| View History | DESCRIBE HISTORY ... | (Use SQL) |
| Drop Table | DROP TABLE ... | (Use SQL) |

# 

# ****Databricks Platform Components – Phases, Activities, and Timelines****

## ****1. Delta Lake****

|  |  |  |
| --- | --- | --- |
| **Phase** | **Key Activities** | **Timeline** |
| Phase 1 – Design Standards | Define Delta table naming conventions, partitioning strategies, and schema evolution policies | Month 1 |
| Phase 2 – Data Ingestion Patterns | Establish batch and streaming ingestion workflows | Month 2 |
| Phase 3 – Transaction Validation | Implement ACID transaction validation and time travel testing | Month 2–3 |
| Phase 4 – Performance Optimization | Configure file compaction and optimize storage layout | Month 3 |
| Phase 5 – Enablement & Adoption | Develop guides for Delta operations and provide training | Month 4 |
| Phase 6 – Pilot and Feedback | Run pilot workloads, validate performance, gather feedback | Month 4 |
| Organization-Wide Rollout | Deploy Delta standards across all datasets | Month 5–6 |
| Continuous Improvement | Quarterly reviews and optimization tuning | Month 7 onward |

|  |  |
| --- | --- |
| **Roles:**   * Data Platform Lead (**DPL**) * Data Engineering Team (**DE**) * Security & Compliance Team (**SC**) * Data Science Enablement Team (**DS**) * Business Unit Leads (**BU**) | **Legend:**   * **A** = Accountable (owns the outcome) * **R** = Responsible (executes the work) * **C** = Consulted (provides input) * **I** = Informed (kept updated) |

**Delta Lake – RACI Matrix**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Activity** | **DPL** | **DE** | **SC** | **DS** | **BU** |
| Define Delta table standards and conventions | A | R | C | C | I |
| Configure ingestion and update patterns | A | R | C | C | I |
| Implement ACID transaction validation | C | R | A | I | I |
| Optimize storage and partitioning strategies | C | A/R | I | C | I |
| Validate encryption and compliance | C | C | A/R | I | I |
| Develop guides and enablement materials | C | C | I | A | R |
| Conduct enablement workshops | C | C | I | A | R |
| Pilot adoption with selected datasets | C | R | C | A | R |
| Full rollout across environments | A | R | C | C | R |
| Quarterly reviews and optimization improvements | A | C | R | C | I |
|  |  |  |  |  |  |

**1. Define Delta Table Standards and Conventions**

**Objectives**

* Establish consistent naming.
* Define partitioning and schema conventions.
* Standardize metadata and comments.

**Example Standards**

* Table naming: <domain>\_<subject>\_<granularity>
* Schema: Use lowercase, underscores in column names.
* Partitioning: Partition by date, region, or business unit.

**Example Table Creation Script**

sql

CopyEdit

CREATE TABLE finance\_transactions\_daily (

transaction\_id STRING,

customer\_id STRING,

amount DECIMAL(18,2),

transaction\_date DATE,

region STRING

)

USING DELTA

PARTITIONED BY (transaction\_date)

COMMENT 'Daily financial transactions partitioned by date';

**2. Configure Ingestion and Update Patterns**

**Patterns**

* **Batch Ingestion**: Append data periodically.
* **Streaming Ingestion**: Use Auto Loader or Spark Structured Streaming.
* **Upserts**: Merge CDC or incremental files.

**Example Batch Append (PySpark)**

python

CopyEdit

df = spark.read.format("csv").option("header","true").load("/mnt/raw/transactions")

df.write.format("delta").mode("append").saveAsTable("finance\_transactions\_daily")

**Example Upsert Merge**

sql

CopyEdit

MERGE INTO finance\_transactions\_daily AS target

USING staging\_new\_data AS source

ON target.transaction\_id = source.transaction\_id

WHEN MATCHED THEN UPDATE SET \*

WHEN NOT MATCHED THEN INSERT \*;

**3. Implement ACID Transaction Validation**

**Actions**

* Use Delta Lake transaction log.
* Enable schema enforcement.
* Validate data quality constraints.

**Example Constraint**

sql

CopyEdit

ALTER TABLE finance\_transactions\_daily

ADD CONSTRAINT valid\_amount CHECK (amount >= 0);

**Schema Enforcement Example (PySpark)**

python

CopyEdit

df.write.option("mergeSchema", "false").format("delta").mode("append").saveAsTable("finance\_transactions\_daily")

If schema mismatches, write fails—preserving data integrity.

**4. Optimize Storage and Partitioning Strategies**

**Actions**

* Use compaction (OPTIMIZE).
* Manage small files.
* ZORDER for query acceleration.

**Example OPTIMIZE**

sql

CopyEdit

OPTIMIZE finance\_transactions\_daily

ZORDER BY (transaction\_date, customer\_id);

**Example VACUUM**

sql

CopyEdit

VACUUM finance\_transactions\_daily RETAIN 168 HOURS;

**5. Validate Encryption and Compliance**

**Steps**

* Confirm storage layer encryption (e.g., Azure ADLS SSE, AWS S3 SSE).
* Enable Unity Catalog for fine-grained access control.
* Audit table history.

**Example Table History**

sql

CopyEdit

DESCRIBE HISTORY finance\_transactions\_daily;

**Grant Access**

sql

CopyEdit

GRANT SELECT ON TABLE finance\_transactions\_daily TO `finance\_analysts`;

**6. Develop Guides and Enablement Materials**

**Artifacts to Create**

* Delta table standards document.
* Ingestion playbooks.
* Query performance tuning guides.
* Data governance overview.
* Example notebooks.

**Tip**  
Store guides in a shared location (Wiki, Git repo, Databricks Repos).

**7. Conduct Enablement Workshops**

**Suggested Agenda**

* Introduction to Delta Lake.
* Ingestion patterns (batch, streaming, upsert).
* Querying and optimization.
* ACID guarantees and compliance.
* Hands-on lab: Create and optimize a Delta table.

**Tip**  
Run live in Databricks Notebooks and record sessions for reuse.

**8. Pilot Adoption with Selected Datasets**

**Steps**

* Identify 1–3 pilot datasets.
* Migrate raw and curated tables to Delta.
* Validate ingestion and querying.
* Test governance and lineage.

**Checklist**

* Schema and partitioning validated.
* Access permissions configured.
* Lineage and audit trails tested.

**9. Full Rollout Across Environments**

**Steps**

* Deploy ingestion pipelines to Dev.
* Promote to Test/Staging.
* Final validations.
* Move to Production.
* Update documentation.

**Tip**  
Automate deployments with Databricks Jobs and CI/CD pipelines.

**10. Quarterly Reviews and Optimization Improvements**

**Activities**

* Review table size and query patterns.
* Reevaluate partitioning.
* Audit access logs.
* Validate compliance posture.
* Update enablement materials.

**Metrics to Track**

* Query performance improvements.
* Storage utilization.
* Access audit logs.

**Summary Table (Quick Reference)**

|  |  |
| --- | --- |
| **Step** | **Actions / Examples** |
| Define standards | Naming, partitioning, schema conventions |
| Configure ingestion | Batch append, streaming, upsert merges |
| Implement ACID validation | Constraints, schema enforcement, transaction log |
| Optimize storage | OPTIMIZE, ZORDER, VACUUM |
| Validate encryption & compliance | Storage encryption, Unity Catalog permissions, history audits |
| Develop guides | Documentation, sample notebooks |
| Conduct workshops | Training sessions with hands-on labs |
| Pilot adoption | Test with selected datasets |
| Full rollout | Promote pipelines across environments |
| Quarterly reviews | Reassess performance, compliance, governance |

# Prerequisites for Working with Delta Lake in Databricks

Below is a **complete list grouped by category**:

## 1. ****Databricks Workspace Setup****

* **Databricks Workspace** is created in your cloud (Azure, AWS, or GCP).
* **Cluster** configured:
  + **Databricks Runtime** version **7.0+** (Delta Lake requires this).
  + Recommended: **Databricks Runtime 10.x+** for performance and newer Delta features.
  + For Unity Catalog, use **Databricks Runtime 11.3 LTS+**.
* **Workspace permissions**:
  + You must have **CAN MANAGE** or **CAN RUN** permissions on clusters.
  + You must have **CREATE** permissions for tables and databases.

## 2. ****Storage Configuration****

Delta Lake always writes data to cloud object storage:

* **Azure Databricks**:
  + **ADLS Gen2** configured with proper access (service principal or managed identity).
* **AWS Databricks**:
  + **S3 buckets** configured with correct IAM policies.
* **GCP Databricks**:
  + **GCS buckets** configured with service accounts.
* Storage mount point (e.g., /mnt/raw) created if you’re using DBFS mounts.

## 3. ****Access Control & Governance****

* For **Unity Catalog**:
  + Unity Catalog metastore is set up and assigned to the workspace.
  + You have **Owner** or **Editor** permissions on catalogs and schemas.
* For legacy Hive metastore:
  + You must have **CREATE DATABASE** and **CREATE TABLE** privileges.

## 4. ****Cluster Libraries & Runtime Components****

* **Delta Lake** is built into Databricks Runtime—**no manual library installation needed**.
* If you’re using OSS Spark elsewhere (outside Databricks), you must install:
  + Delta Lake JARs (io.delta:delta-core).
  + Spark configurations for Delta.

## 5. ****Networking & Security****

* Workspace has **network access** to:
  + Cloud storage (S3/ADLS/GCS).
  + Any external data sources you ingest.
* **Encryption** (at rest and in transit) is enabled.
* For compliance, ensure audit logging is configured.

## 6. ****Data Ingestion Readiness****

* Raw data landing zones are prepared.
* Schemas are known or discoverable.
* Sample data is available for testing ingestion.

## 7. ****Tooling****

* **Databricks Notebooks** are created for development.
* Optionally:
  + **Databricks Repos** connected to Git for version control.
  + **Databricks CLI** installed for scripting.
  + **Databricks REST APIs** accessible for automation.

## 8. ****Training / Familiarity****

* Familiarity with:
  + PySpark and/or SQL (most Delta operations use them).
  + Delta Lake concepts:
    - ACID transactions
    - Schema enforcement
    - Time travel
    - Table versioning
    - Optimize/Vacuum

## Quick Prerequisite Checklist

|  |  |
| --- | --- |
| **Area** | **Requirement** |
| **Workspace** | Databricks workspace and cluster created |
| **Runtime** | Databricks Runtime 7.0+ (preferably 10.x or 11.x) |
| **Storage** | Cloud storage configured and accessible |
| **Permissions** | Create and manage tables/databases |
| **Unity Catalog (opt.)** | Metastore assigned and configured |
| **Networking** | Connectivity to storage and data sources |
| **Data readiness** | Landing zones and sample datasets prepared |
| **Tooling** | Notebooks, Repos, CLI, REST APIs configured |
| **Skills** | Familiarity with Spark, Delta Lake, and Databricks workflows |

**1. Delta Lake Prerequisite Detailed Checklist**

|  |  |
| --- | --- |
| **Area** | **Detailed Checklist Items** |
| **Workspace** | - A Databricks workspace has been provisioned in the target cloud (AWS, Azure, or GCP). - Workspace is associated with the correct subscription/account/project. - At least one cluster has been created. - Cluster policies (if applicable) are defined. - Cluster has sufficient worker nodes and permissions to access data. - Workspace admin users and data engineers have access. |
| **Runtime** | - Cluster is running **Databricks Runtime 7.0 or higher**. - For Unity Catalog: **Databricks Runtime 11.3 LTS or higher**. - Delta Lake features are confirmed to be included. - Runtime ML or GPU options are considered if needed. - Spark configuration is adjusted for performance (shuffle partitions, executor memory). |
| **Storage** | - Cloud storage (S3, ADLS Gen2, or GCS) account/container/bucket is created. - Access keys, service principals, or instance profiles are provisioned. - Storage mounts are configured (/mnt/...). - Access policies/IAM roles are tested. - Encryption at rest is enabled. - Storage folder structure for raw/curated data is planned. |
| **Permissions** | - Users/groups are defined in Databricks (or via SCIM/Azure AD sync). - Users have permissions to create databases, tables, and folders. - Workspace admin can assign access controls. - Cluster attach permissions are set. - For Unity Catalog: GRANT/REVOKE usage privileges are planned. |
| **Unity Catalog (optional)** | - A Unity Catalog metastore has been created. - The metastore is assigned to the workspace. - Storage credentials are created and configured. - External locations are registered. - Access to catalogs, schemas, and tables is configured. - Data lineage tracking is enabled. |
| **Networking** | - The workspace has network connectivity to the storage account/bucket. - Private endpoints or VNET/S3 endpoint configurations are validated. - Firewall rules allow necessary traffic. - Outbound internet access is configured if needed. - Any VPN/Peering configurations are completed. |
| **Data readiness** | - Raw datasets are available in landing zones. - Sample files are staged for testing. - Expected schemas are documented. - Data volumes are estimated. - Partitioning strategy is defined. - Data retention and lifecycle policies are planned. |
| **Tooling** | - Notebooks have been created in the workspace. - Databricks Repos connected to Git repositories (optional). - Databricks CLI is installed and configured. - REST API tokens are created if needed. - Jobs UI access is configured for scheduling. |
| **Skills** | - Team is trained on PySpark basics. - Familiarity with Delta Lake operations (CREATE, MERGE, UPDATE, OPTIMIZE, VACUUM). - Knowledge of Unity Catalog governance concepts (if applicable). - Understanding of Spark performance tuning. - Awareness of data compliance requirements (encryption, audit). |

**Example: Workspace Checklist in Detail**

To make it clearer, here’s **one area expanded further**, showing exactly what you check:

**Workspace**

* Databricks workspace created and linked to the right subscription/account.
* Workspace region aligns with data residency requirements.
* At least one cluster created and running.
* Cluster runtime >= 7.0 (recommended 10.x+).
* Workspace admin account validated.
* User groups set up (admins, engineers, analysts).
* Cluster permissions configured (attach/run).

**Copy-Paste Friendly Format**

If you want, here is a plain text version for quick copying into checklists:

diff

CopyEdit

Workspace:

- Databricks workspace created and linked to the right subscription/account.

- Workspace region aligns with data residency requirements.

- Cluster(s) created and running with Databricks Runtime >=7.0.

- Workspace admin and engineering users provisioned.

- Cluster attach/run permissions set.

Runtime:

- Runtime version >=7.0, preferably 10.x or 11.x.

- Delta Lake features included.

- Cluster configurations adjusted for workload (e.g., memory, shuffle partitions).

Storage:

- Cloud storage account/container/bucket created.

- Access keys/IAM roles configured.

- Mount points created (/mnt/...).

- Encryption enabled.

- Folder structure planned.

Permissions:

- Users/groups created.

- Table/database creation privileges tested.

- Cluster attach/run permissions confirmed.

Unity Catalog (if used):

- Metastore created and assigned to workspace.

- External locations and storage credentials set.

- Table permissions configured.

- Lineage tracking enabled.

Networking:

- Storage access connectivity tested.

- Private endpoints/firewalls validated.

- Outbound connectivity as needed.

Data readiness:

- Raw datasets staged in landing zone.

- Sample files available.

- Schema definitions documented.

- Partition strategy decided.

- Data volume estimated.

Tooling:

- Notebooks created.

- Repos connected to Git (optional).

- CLI installed and configured.

- API tokens created.

- Jobs UI access confirmed.

Skills:

- Team trained on Spark and Delta Lake basics.

- Familiarity with ACID operations and optimization.

- Knowledge of compliance and security practices.

|  |
| --- |
|  |
| Option : 1 |
| Walk through a **structured example pipeline** showing **bronze, silver, and gold Delta Lake layers** with:   * **Auto ingestion** * **Checkpoints** * **Watermarking** * **Time-period filtering**   This example is end-to-end, production-style code you can **run in Databricks notebooks**, showing each layer clearly.  Below, I’ll provide **PySpark Structured Streaming code** you can adapt.  **Pipeline Overview**   * **Bronze Layer** Raw ingestion from a cloud storage location. * **Silver Layer** Cleaned, deduplicated, and schema-enforced data. * **Gold Layer** Aggregated business-level metrics.   **1. Bronze Table: Auto Ingestion with Checkpointing**  **Description**   * Use **Auto Loader** (cloudFiles) for continuous ingestion. * Data is stored as-is in Bronze. * Checkpoint directory ensures exactly-once semantics.   **Example PySpark Code**  python  CopyEdit  from pyspark.sql.functions import input\_file\_name, current\_timestamp  bronze\_df = (  spark.readStream.format("cloudFiles")  .option("cloudFiles.format", "json")  .option("cloudFiles.inferColumnTypes", "true")  .load("/mnt/raw/sales/")  .withColumn("source\_file", input\_file\_name())  .withColumn("ingestion\_timestamp", current\_timestamp())  )  bronze\_df.writeStream \  .format("delta") \  .outputMode("append") \  .option("checkpointLocation", "/mnt/checkpoints/bronze\_sales") \  .trigger(availableNow=True) \  .start("/mnt/delta/bronze\_sales")  **Notes**   * availableNow=True = process all files in the directory once. * For continuous ingestion, remove availableNow.   **2. Silver Table: Deduplication + Watermark**  **Description**   * Clean the data. * Enforce schema. * Deduplicate using watermark and event time. * Filter only data within the last 30 days.   **Example PySpark Code**  python  CopyEdit  from pyspark.sql.functions import col, to\_date  # Read Bronze stream  bronze\_stream = (  spark.readStream  .format("delta")  .load("/mnt/delta/bronze\_sales")  )  # Clean and filter  silver\_df = (  bronze\_stream  .withColumn("transaction\_date", to\_date(col("transaction\_date")))  .filter(col("transaction\_date") >= date\_sub(current\_date(), 30))  .dropDuplicates(["transaction\_id"])  .withWatermark("transaction\_date", "7 days")  )  silver\_df.writeStream \  .format("delta") \  .outputMode("append") \  .option("checkpointLocation", "/mnt/checkpoints/silver\_sales") \  .start("/mnt/delta/silver\_sales")  **Notes**   * withWatermark("transaction\_date", "7 days") enables state cleanup for deduplication. * date\_sub ensures only the last 30 days are processed.   **📂 3. Gold Table: Aggregation**  **Description**   * Aggregate cleaned data to compute daily sales totals. * Store results as Gold Delta tables.   **Example PySpark Code**  python  CopyEdit  from pyspark.sql.functions import sum as \_sum  silver\_stream = (  spark.readStream  .format("delta")  .load("/mnt/delta/silver\_sales")  )  gold\_df = (  silver\_stream  .groupBy("transaction\_date")  .agg(\_sum("amount").alias("total\_sales"))  )  gold\_df.writeStream \  .format("delta") \  .outputMode("complete") \  .option("checkpointLocation", "/mnt/checkpoints/gold\_sales") \  .start("/mnt/delta/gold\_sales")  **Notes**   * outputMode("complete") because aggregations need full output. * Gold tables are optimized for BI consumption.   **🟢 Quick Recap: Concepts Used**   | **Concept** | **What It Does** | | --- | --- | | Auto Loader | Automatically detects and ingests new files | | Checkpointing | Tracks progress to avoid duplicates | | Watermarking | Controls late data and state cleanup for deduplication | | Time Filtering | Keeps data within rolling windows (last 30 days) | | Bronze/Silver/Gold | Separates raw, clean, and business-ready layers |   **🟢 Directory Layout**   | **Layer** | **Path** | | --- | --- | | Bronze | /mnt/delta/bronze\_sales | | Silver | /mnt/delta/silver\_sales | | Gold | /mnt/delta/gold\_sales | | Checkpoints | /mnt/checkpoints/... |   **🟢 Optional Optimizations**  After ingestion, you can schedule:  **Optimize Table**  sql  CopyEdit  OPTIMIZE delta.`/mnt/delta/gold\_sales`  ZORDER BY (transaction\_date);  **Vacuum Old Files**  sql  CopyEdit  VACUUM delta.`/mnt/delta/gold\_sales` RETAIN 168 HOURS;  **🟢 How to Run**   1. Create one notebook per layer. 2. Run them in **separate clusters or jobs**, or orchestrate via **Workflows**. 3. Validate data landed in each path. 4. Query each layer with SQL or DataFrames. |
| **Option :2** |
|  |
| **practical and ready for you to import into Databricks**.  I’ll prepare: ✅ Clear **notebook structure** for each layer (Bronze, Silver, Gold) ✅ Ready-to-copy **PySpark code cells** ✅ Example **SQL queries to validate the data**  Below is your **step-by-step deployment guide** with clean scripts.  **📂 Notebook 1 – Bronze Layer Ingestion (Auto Loader)**  **Notebook Name:** 01\_Bronze\_AutoLoader\_Ingestion  **1️⃣ Configure Auto Loader with Checkpointing**  python  CopyEdit  from pyspark.sql.functions import input\_file\_name, current\_timestamp  bronze\_df = (  spark.readStream.format("cloudFiles")  .option("cloudFiles.format", "json") # Change to csv/parquet as needed  .option("cloudFiles.inferColumnTypes", "true")  .load("/mnt/raw/sales/")  .withColumn("source\_file", input\_file\_name())  .withColumn("ingestion\_timestamp", current\_timestamp())  )  bronze\_df.writeStream \  .format("delta") \  .outputMode("append") \  .option("checkpointLocation", "/mnt/checkpoints/bronze\_sales") \  .start("/mnt/delta/bronze\_sales")  **2️⃣ Validation Query (SQL)**  sql  CopyEdit  SELECT \* FROM delta.`/mnt/delta/bronze\_sales` LIMIT 10;  **📂 Notebook 2 – Silver Layer Cleansing + Watermark**  **Notebook Name:** 02\_Silver\_Cleansing\_and\_Watermark  **1️⃣ Read from Bronze**  python  CopyEdit  from pyspark.sql.functions import col, to\_date, current\_date, date\_sub  bronze\_stream = (  spark.readStream  .format("delta")  .load("/mnt/delta/bronze\_sales")  )  **2️⃣ Clean, Deduplicate, Watermark, Filter Time Window**  python  CopyEdit  silver\_df = (  bronze\_stream  .withColumn("transaction\_date", to\_date(col("transaction\_date")))  .filter(col("transaction\_date") >= date\_sub(current\_date(), 30))  .dropDuplicates(["transaction\_id"])  .withWatermark("transaction\_date", "7 days")  )  **3️⃣ Write to Silver**  python  CopyEdit  silver\_df.writeStream \  .format("delta") \  .outputMode("append") \  .option("checkpointLocation", "/mnt/checkpoints/silver\_sales") \  .start("/mnt/delta/silver\_sales")  **4️⃣ Validation Query (SQL)**  sql  CopyEdit  SELECT \* FROM delta.`/mnt/delta/silver\_sales` LIMIT 10;  **📂 Notebook 3 – Gold Layer Aggregation**  **Notebook Name:** 03\_Gold\_Aggregation  **1️⃣ Read from Silver**  python  CopyEdit  from pyspark.sql.functions import sum as \_sum  silver\_stream = (  spark.readStream  .format("delta")  .load("/mnt/delta/silver\_sales")  )  **2️⃣ Aggregate Daily Totals**  python  CopyEdit  gold\_df = (  silver\_stream  .groupBy("transaction\_date")  .agg(\_sum("amount").alias("total\_sales"))  )  **3️⃣ Write to Gold**  python  CopyEdit  gold\_df.writeStream \  .format("delta") \  .outputMode("complete") \  .option("checkpointLocation", "/mnt/checkpoints/gold\_sales") \  .start("/mnt/delta/gold\_sales")  **4️⃣ Validation Query (SQL)**  sql  CopyEdit  SELECT \* FROM delta.`/mnt/delta/gold\_sales` ORDER BY transaction\_date DESC;  **✅ Final Directory Layout**   | **Layer** | **Path** | | --- | --- | | Bronze | /mnt/delta/bronze\_sales | | Silver | /mnt/delta/silver\_sales | | Gold | /mnt/delta/gold\_sales | | Checkpoints | /mnt/checkpoints/... |   **✅ Daily Maintenance**  After ingestion, run:  **Optimize:**  sql  CopyEdit  OPTIMIZE delta.`/mnt/delta/gold\_sales`  ZORDER BY (transaction\_date);  **Vacuum:**  sql  CopyEdit  VACUUM delta.`/mnt/delta/gold\_sales` RETAIN 168 HOURS;  **✅ How to Deploy**   1. Create **three notebooks** named:    * 01\_Bronze\_AutoLoader\_Ingestion    * 02\_Silver\_Cleansing\_and\_Watermark    * 03\_Gold\_Aggregation 2. Copy-paste the code blocks into each notebook. 3. Start each streaming query **in order**. 4. Use the **SQL queries** to confirm data flows properly. 5. (Optional) Create a **Job Workflow** in Databricks to orchestrate all notebooks. |
|  |
| Option 3: Synch with Azure Env |
| ⚙️ Azure Setup Prerequisites (Quick Reminder) Before running the code:  ✅ **Create ADLS Gen2 Storage Account** ✅ **Register an App in Azure AD (Service Principal)** ✅ **Assign the Storage Blob Data Contributor role to the App** ✅ **Get the following credentials:**   * Tenant ID * Client ID * Client Secret   ✅ **Mount ADLS to DBFS** 🟢 Example: Mounting ADLS Gen2 to /mnt/raw and /mnt/checkpoints You only need to do this **once per workspace**:  python  CopyEdit  configs = {  "fs.azure.account.auth.type": "OAuth",  "fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",  "fs.azure.account.oauth2.client.id": "<CLIENT-ID>",  "fs.azure.account.oauth2.client.secret": dbutils.secrets.get(scope="<scope-name>", key="<secret-key-name>"),  "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/<TENANT-ID>/oauth2/token"  }  # Mount raw data  dbutils.fs.mount(  source = "abfss://<container-name>@<storage-account-name>.dfs.core.windows.net/raw",  mount\_point = "/mnt/raw",  extra\_configs = configs)  # Mount checkpoint location  dbutils.fs.mount(  source = "abfss://<container-name>@<storage-account-name>.dfs.core.windows.net/checkpoints",  mount\_point = "/mnt/checkpoints",  extra\_configs = configs)  # Mount Delta tables root  dbutils.fs.mount(  source = "abfss://<container-name>@<storage-account-name>.dfs.core.windows.net/delta",  mount\_point = "/mnt/delta",  extra\_configs = configs)  **Tip:** Use a **Databricks Secret Scope** to store the client secret securely. 🟢 Updated Pipeline Code for Azure Below are the **three notebooks**, Azure-adjusted: 📘 Notebook 1 – Bronze Layer Ingestion **Notebook Name:** 01\_Bronze\_AutoLoader\_Ingestion  python  CopyEdit  from pyspark.sql.functions import input\_file\_name, current\_timestamp  bronze\_df = (  spark.readStream.format("cloudFiles")  .option("cloudFiles.format", "json") # change to "csv" or "parquet" if needed  .option("cloudFiles.inferColumnTypes", "true")  .load("/mnt/raw/sales/")  .withColumn("source\_file", input\_file\_name())  .withColumn("ingestion\_timestamp", current\_timestamp())  )  bronze\_df.writeStream \  .format("delta") \  .outputMode("append") \  .option("checkpointLocation", "/mnt/checkpoints/bronze\_sales") \  .start("/mnt/delta/bronze\_sales")  ✅ This auto-loads any new files placed in /mnt/raw/sales/. 📘 Notebook 2 – Silver Layer Cleansing and Watermark **Notebook Name:** 02\_Silver\_Cleansing\_and\_Watermark  python  CopyEdit  from pyspark.sql.functions import col, to\_date, current\_date, date\_sub  bronze\_stream = (  spark.readStream  .format("delta")  .load("/mnt/delta/bronze\_sales")  )  silver\_df = (  bronze\_stream  .withColumn("transaction\_date", to\_date(col("transaction\_date")))  .filter(col("transaction\_date") >= date\_sub(current\_date(), 30))  .dropDuplicates(["transaction\_id"])  .withWatermark("transaction\_date", "7 days")  )  silver\_df.writeStream \  .format("delta") \  .outputMode("append") \  .option("checkpointLocation", "/mnt/checkpoints/silver\_sales") \  .start("/mnt/delta/silver\_sales") 📘 Notebook 3 – Gold Layer Aggregation **Notebook Name:** 03\_Gold\_Aggregation  python  CopyEdit  from pyspark.sql.functions import sum as \_sum  silver\_stream = (  spark.readStream  .format("delta")  .load("/mnt/delta/silver\_sales")  )  gold\_df = (  silver\_stream  .groupBy("transaction\_date")  .agg(\_sum("amount").alias("total\_sales"))  )  gold\_df.writeStream \  .format("delta") \  .outputMode("complete") \  .option("checkpointLocation", "/mnt/checkpoints/gold\_sales") \  .start("/mnt/delta/gold\_sales") 🟢 Validation Queries After starting each stream, validate results in the **SQL Editor**:  **Bronze:**  sql  CopyEdit  SELECT \* FROM delta.`/mnt/delta/bronze\_sales` LIMIT 10;  **Silver:**  sql  CopyEdit  SELECT \* FROM delta.`/mnt/delta/silver\_sales` LIMIT 10;  **Gold:**  sql  CopyEdit  SELECT \* FROM delta.`/mnt/delta/gold\_sales` ORDER BY transaction\_date DESC; 🟢 Maintenance Commands **Optimize:**  sql  CopyEdit  OPTIMIZE delta.`/mnt/delta/gold\_sales`  ZORDER BY (transaction\_date);  **Vacuum:**  sql  CopyEdit  VACUUM delta.`/mnt/delta/gold\_sales` RETAIN 168 HOURS; 🟢 Orchestration (Optional) To orchestrate all three layers:   * Create a **Job Workflow** in Databricks. * Add each notebook as a task in order:   + Bronze → Silver → Gold. * Use availableNow triggers if you want batch-style runs.   ✅ **You are now Azure-ready!**  If you’d like, I can:   * Help you **generate importable .dbc notebooks**. * Provide **Terraform scripts** to provision storage and mount points. * Review your specific storage and container naming for accuracy. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| how make to use to story telling with vision statement on delta lake as step by step  Fantastic—let’s **elevate this from just technical instructions to a clear, engaging story** you can share in: ✅ Presentations ✅ Vision documents ✅ Stakeholder workshops  Below, I’ll show you **step by step how to build a storytelling narrative**, so your audience sees:   * **Why Delta Lake matters** * **How it solves real problems** * **Where you’re going with it**   I’ll give you: ✅ A **Vision Statement** ✅ A **Story Arc** ✅ Suggested **Slides/Sections** ✅ Example **Narration** you can adapt 🎯 ****1. Start with Your Vision Statement**** **Example Vision Statement:**  “To create a trusted, performant, and scalable foundation for all our data, enabling our teams to innovate confidently and drive business impact through high-quality, reliable insights.”  ✅ **Tip:** Keep it **short**, **inspiring**, and **focused on impact**, not just technology. ✨ ****2. Build Your Story Arc**** Think of it like telling a movie story: 1️⃣ **Problem** (What’s broken today) 2️⃣ **Promise** (How Delta Lake solves this) 3️⃣ **Proof** (How you’ll do it, step by step) 4️⃣ **Future** (What success looks like) 🟢 ****3. The Narrative in Steps**** I’ll break this down **step by step** so you can copy into slides or a document. ✋ Step 1: The Problem **Slide Title:** Why Change is Needed  **Narration:**   * Today, our data lives in many places and formats. * We struggle with inconsistent data quality, missing records, and slow performance. * Our teams spend more time fixing data issues than innovating.   ✅ **Visual:** Picture of spaghetti architecture, failed pipelines, and slow dashboards. 🌟 Step 2: The Promise **Slide Title:** The Promise of Delta Lake  **Narration:**   * Delta Lake provides a reliable, ACID-compliant layer on top of our data lake. * It transforms raw data chaos into an organized, high-quality foundation. * It empowers our teams to trust the data and move faster.   ✅ **Visual:** Before/after diagram: messy lake → clean Delta layers. 🛠️ Step 3: The Path Forward (Step by Step) **Slide Title:** How We Get There  Here is a **step-by-step roadmap** you can present: ****Step 1: Lay the Foundation**** **Objective:** Build infrastructure and governance.  ✅ Provision:   * Cloud storage (ADLS). * Databricks workspace. * Unity Catalog for data governance.   ✅ Define:   * Naming conventions. * Access policies.   **Narration:** We start by laying a strong foundation—secure, governed, and ready to scale. ****Step 2: Ingest and Organize Data (Bronze Layer)**** **Objective:** Capture raw data reliably.  ✅ Use:   * Auto Loader for continuous ingestion. * Checkpoints to ensure nothing is missed.   **Narration:** We bring in all the raw data, capturing every record with full lineage and traceability. ****Step 3: Cleanse and Deduplicate (Silver Layer)**** **Objective:** Turn raw data into trusted, query-ready datasets.  ✅ Apply:   * Schema enforcement. * Deduplication. * Watermarking.   **Narration:** We clean the data, enforce consistency, and remove duplicates—so everyone works from the same truth. ****Step 4: Aggregate and Serve (Gold Layer)**** **Objective:** Create business-ready datasets.  ✅ Build:   * Aggregations. * Optimized tables for analytics.   **Narration:** We transform data into actionable insights, ready for dashboards, AI, and decision-making. ****Step 5: Govern and Optimize**** **Objective:** Keep data fresh, secure, and cost-effective.  ✅ Schedule:   * OPTIMIZE and VACUUM. * Regular access audits.   **Narration:** We put data quality on autopilot and ensure we comply with security and retention policies. 🚀 Step 4: The Future Vision **Slide Title:** What Success Looks Like  **Narration:**   * Every team has instant access to trusted data. * Data engineering effort shifts from fixing to innovating. * We unlock new analytics, machine learning, and faster decision cycles.   ✅ **Visual:** A smooth pipeline flowing to dashboards and ML models. 🟢 ****4. Supporting Proof Points**** In your story, weave in:   * **Examples of slow queries or broken reports** today. * **Metrics you aim to improve**, e.g.,   + Query times reduced by 70%.   + Data freshness improved from weekly to daily. * **User testimonials or quotes** about pain points.  🟢 ****5. How to Deliver This**** ✅ **Workshop/Presentation Tips:**   * Keep slides clean and visual. * Narrate with energy and confidence. * Keep technical details in backup slides. * Show a short **live demo** of a Delta table if possible.   ✅ **Vision Document Structure:**   1. Introduction 2. Why Delta Lake 3. Our Vision Statement 4. The Roadmap (steps 1–5) 5. Benefits and Outcomes 6. Next Steps  ✨ ****Example Vision Statement + Tagline**** “Delta Lake will be our single source of truth—trusted, scalable, and ready for the future.”  “From chaos to confidence.” ✅ ****Summary Cheat Sheet****  | **Step** | **Goal** | | --- | --- | | 1. Lay the Foundation | Setup storage, security, governance. | | 2. Ingest Data | Auto Loader + checkpoints (Bronze). | | 3. Cleanse Data | Deduplication + watermarks (Silver). | | 4. Aggregate Data | Gold tables for BI and ML. | | 5. Govern and Optimize | Automate performance and compliance. | |

|  |
| --- |
| Schema Evolutoins |
| ****End-to-End Example: Schema Evolution with Delta Lake and Auto Loader**** Imagine you have a **sales data feed** landing in cloud storage (e.g., Azure Data Lake Storage or AWS S3). Over time, the schema evolves—**new columns arrive** in the files. 🪜 ****PHASE 1 – Setup Your Environment**** ✅ **1. Create a base Delta table**  Let’s start by creating an initial table:  python  CopyEdit  from pyspark.sql.types import StructType, StructField, StringType, DoubleType, DateType  # Explicit schema  schema = StructType([  StructField("sale\_id", StringType(), True),  StructField("amount", DoubleType(), True),  StructField("sale\_date", DateType(), True)  ])  # Sample initial data  data = [  ("S1", 100.0, "2023-01-01"),  ("S2", 200.5, "2023-01-02")  ]  df = spark.createDataFrame(data, schema=schema)  df.write.format("delta").mode("overwrite").save("/mnt/delta/sales")  ✅ **Check the schema:**  python  CopyEdit  spark.read.format("delta").load("/mnt/delta/sales").printSchema()  objectivec  CopyEdit  root  |-- sale\_id: string (nullable = true)  |-- amount: double (nullable = true)  |-- sale\_date: date (nullable = true) 🪜 ****PHASE 2 – Append New Data Without Schema Evolution**** ✅ **2. Create a new DataFrame with an extra column:**  python  CopyEdit  from pyspark.sql import Row  data\_new = [  Row(sale\_id="S3", amount=300.0, sale\_date="2023-01-03", customer\_id="CUST123")  ]  df\_new = spark.createDataFrame(data\_new)  ✅ **Attempt to append without mergeSchema:**  python  CopyEdit  df\_new.write.format("delta").mode("append").save("/mnt/delta/sales")  ⚠️ **Expected error:**  css  CopyEdit  Detected a schema mismatch between the data and the table... 🪜 ****PHASE 3 – Enable Schema Evolution (mergeSchema)**** ✅ **3. Append with schema evolution:**  python  CopyEdit  df\_new.write \  .option("mergeSchema", "true") \  .format("delta") \  .mode("append") \  .save("/mnt/delta/sales")  ✅ **Check the schema again:**  python  CopyEdit  spark.read.format("delta").load("/mnt/delta/sales").printSchema()  objectivec  CopyEdit  root  |-- sale\_id: string (nullable = true)  |-- amount: double (nullable = true)  |-- sale\_date: date (nullable = true)  |-- customer\_id: string (nullable = true)  ✅ **Query data:**  python  CopyEdit  spark.read.format("delta").load("/mnt/delta/sales").show()  sql  CopyEdit  +-------+------+----------+-----------+  |sale\_id|amount| sale\_date|customer\_id|  +-------+------+----------+-----------+  | S1| 100.0|2023-01-01| null|  | S2| 200.5|2023-01-02| null|  | S3| 300.0|2023-01-03| CUST123|  +-------+------+----------+-----------+ 🪜 ****PHASE 4 – Auto Loader Ingestion with Schema Evolution**** In real-world scenarios, you often use **Auto Loader** to watch cloud directories for new files.  ✅ **4. Auto Loader initial ingestion:**  python  CopyEdit  df\_auto = (  spark.readStream.format("cloudFiles")  .option("cloudFiles.format", "csv")  .option("header", "true")  .schema(schema) # initial schema  .load("/mnt/raw/sales/")  )  (  df\_auto.writeStream.format("delta")  .option("checkpointLocation", "/mnt/checkpoints/sales")  .start("/mnt/delta/sales")  )  ✅ New data arriving in /mnt/raw/sales/ matching the schema will be ingested automatically.  ✅ **5. New files with new columns land in raw folder:**  Example new file:  CopyEdit  sale\_id,amount,sale\_date,customer\_id  S4,250.0,2023-01-04,CUST456  ✅ **To evolve schema automatically:**  You must enable schema evolution during Auto Loader ingestion:  python  CopyEdit  df\_auto\_new = (  spark.readStream.format("cloudFiles")  .option("cloudFiles.format", "csv")  .option("cloudFiles.inferColumnTypes", "true") # Important for CSV  .option("mergeSchema", "true") # Schema evolution  .option("header", "true")  .load("/mnt/raw/sales/")  )  (  df\_auto\_new.writeStream.format("delta")  .option("checkpointLocation", "/mnt/checkpoints/sales")  .option("mergeSchema", "true")  .start("/mnt/delta/sales")  )  ✅ **Auto Loader will:**   * Detect the new column. * Merge it into the Delta table. * Update the schema in metadata. * Fill NULLs in older rows.  🪜 ****PHASE 5 – Track Schema Changes Over Time**** You can see schema history using:  sql  CopyEdit  DESCRIBE HISTORY delta.`/mnt/delta/sales`  This shows:   * Each operation (append, overwrite) * User who did it * Operation parameters (e.g., mergeSchema) * Timestamp  🔄 ****Operations Involving Schema Evolution****  | **Operation** | **Behavior** | | --- | --- | | **Append** | Fails if schema mismatch, unless mergeSchema=true | | **Overwrite** | Replaces data; can optionally replace schema (overwriteSchema) | | **MERGE INTO** | Upserts data; schema evolution possible in Databricks Runtime 9.0+ | | **Auto Loader** | Supports evolving schema during streaming ingestion | | **ALTER TABLE ADD COLUMN** | Manually adds new columns | | **REPLACE TABLE** | Destroys and recreates the table with new schema |  ✨ ****Best Practices Recap****  * **Govern Schema Evolution:** Enable mergeSchema only when you expect schema drift. * **Schema Registry:** Keep track of upstream data contract changes. * **Column Explosion:** Watch out for too many unexpected columns. * **Data Types:** Delta does not auto-convert conflicting types (e.g., string vs int). * **Versioning:** Use DESCRIBE HISTORY and TIMESTAMP AS OF for rollback. |

|  |
| --- |
| Schema Drift – Schema Evolution help it |
| **🎯 Schema Drift – What It Means**  ✅ **Definition (Plain English):**  **Schema drift** is when the structure (schema) of your incoming data *unexpectedly changes over time*.  These changes can happen *without prior notice*, and if your pipelines don’t handle them, you get errors or bad data.  ✅ **Common Symptoms:**   * New columns appear in the data. * Existing columns disappear. * Data types change (e.g., string → integer). * Column names change or get misspelled. * Columns are reordered.   **🪜 Examples to Make It Concrete**  ✅ **Example 1 – New Columns**   * **Yesterday’s data:**   CopyEdit  sale\_id, amount, sale\_date   * **Today’s data:**   CopyEdit  sale\_id, amount, sale\_date, customer\_id  ✅ **Example 2 – Column Type Change**   * **Earlier:**   vbnet  CopyEdit  sale\_id: STRING   * **Now:**   vbnet  CopyEdit  sale\_id: INTEGER  ✅ **Example 3 – Missing Columns**   * **Yesterday:**   CopyEdit  sale\_id, amount, sale\_date   * **Today:**   CopyEdit  sale\_id, sale\_date  ✅ **Example 4 – Renamed Columns**   * **Yesterday:**   nginx  CopyEdit  sale\_id   * **Today:**   nginx  CopyEdit  sales\_id  **⚠️ Why Schema Drift Is a Problem**  If your ingestion or ETL pipeline assumes a *fixed schema*, drift causes:   * **Load failures** (Column not found, Type mismatch) * **Null or bad data** * **Silent data corruption** (wrong values in wrong columns) * **Unexpected table schema changes**   **💡 Schema Drift vs Schema Evolution**  These two terms are related but different:   | **Concept** | **What It Means** | | --- | --- | | **Schema Drift** | The phenomenon of the schema *changing over time*, often unplanned. | | **Schema Evolution** | The capability of your system to *adapt* to those changes (e.g., mergeSchema). |   **Put simply:**   * *Schema drift happens.* * *Schema evolution helps you handle it.*   **🛠️ How Databricks Helps You Manage Schema Drift**  ✅ **Delta Lake**   * mergeSchema allows adding new columns automatically. * overwriteSchema can replace schemas intentionally.   ✅ **Auto Loader**   * Supports schema evolution automatically (.option("mergeSchema", "true")). * Can infer new columns on-the-fly. * Tracks evolving schemas in checkpoints and metadata.   ✅ **Schema Location**   * Auto Loader can store evolving schema in a schema location folder to track it separately.   ✅ **Unity Catalog (if enabled)**   * Provides governance and lineage for schema changes.   **✨ Best Practices to Handle Schema Drift**  ✅ **1. Monitor your schema changes**   * Use DESCRIBE HISTORY and logs.   ✅ **2. Start with explicit schemas**   * Don’t rely solely on schema inference.   ✅ **3. Enable schema evolution only when expected**   * Avoid mergeSchema=true unless you *know* drift will occur.   ✅ **4. Version your data**   * Delta Lake time travel helps recover from unexpected drift.   ✅ **5. Communicate with data providers**   * Schema drift is often a sign of upstream process changes. |
| **3 Options to handle Schema Drift** |
| Fantastic—let’s tackle **all three** of your requests step by step so you get a **complete blueprint**:  1️⃣ **Auto Loader pipelines handling schema drift** 2️⃣ **Bronze–Silver–Gold architecture to isolate and control drift** 3️⃣ **Data quality checks to detect drift early**  I’ll break this down clearly so you can adapt the examples in Databricks. 🟢 1️⃣ Auto Loader Pipelines Handling Schema Drift **Scenario:** You have raw CSV files dropping daily into /mnt/raw/sales/. The schema occasionally evolves—new columns show up.  ✅ **Approach:**   * Use Auto Loader (cloudFiles) with schema evolution. * Store evolving schema in a **schema location**. * Append into a Delta Lake table.   ✅ **PySpark Example:**  python  CopyEdit  # Initial explicit schema  from pyspark.sql.types import StructType, StructField, StringType, DoubleType, DateType  base\_schema = StructType([  StructField("sale\_id", StringType(), True),  StructField("amount", DoubleType(), True),  StructField("sale\_date", DateType(), True)  ])  df\_raw = (  spark.readStream  .format("cloudFiles")  .option("cloudFiles.format", "csv")  .option("header", "true")  # Store evolving schema metadata  .option("cloudFiles.schemaLocation", "/mnt/schema/sales")  .option("mergeSchema", "true")  # Infer types automatically as new columns arrive  .option("cloudFiles.inferColumnTypes", "true")  # Optional: specify base schema (recommended)  .schema(base\_schema)  .load("/mnt/raw/sales/")  )  (  df\_raw.writeStream  .format("delta")  .option("checkpointLocation", "/mnt/checkpoints/bronze\_sales")  .outputMode("append")  .start("/mnt/delta/bronze\_sales")  )  ✅ **Notes:**   * mergeSchema=true: enables automatic schema merging. * cloudFiles.schemaLocation: folder where evolving schemas are versioned. * cloudFiles.inferColumnTypes=true: infers new column types automatically.  🟡 2️⃣ Bronze–Silver–Gold Architecture to Isolate Raw Drift ✅ **Objective:**   * **Bronze:** Store raw ingested data (with schema drift). * **Silver:** Standardize schema and apply data quality checks. * **Gold:** Aggregate and curate trusted data for consumption.   Below is **the typical flow**:  pgsql  CopyEdit  Cloud Storage  |  [Auto Loader Ingest]  |  Bronze Delta Table  |  [Schema Standardization, Data Quality Checks]  |  Silver Delta Table  |  [Transform, Aggregate]  |  Gold Delta Table  ✅ **Step-by-Step Example:** 🚀 Bronze Table (Raw) Your **Bronze** table is the raw landing zone:   * Accept all schema drift. * Append only. * Time-stamped raw ingestion.   Already implemented in **Step 1**. ⚙️ Silver Table (Standardized) In Silver, you:   * Select expected columns. * Cast types. * Apply **data quality filters**. * Fill missing columns with defaults.   ✅ **Example PySpark batch job:**  python  CopyEdit  bronze\_df = spark.read.format("delta").load("/mnt/delta/bronze\_sales")  # Select only known columns, add defaults if missing  from pyspark.sql.functions import col, lit, to\_date  silver\_df = (  bronze\_df  .withColumn("customer\_id", col("customer\_id").cast("string"))  .withColumn("sale\_date", to\_date("sale\_date"))  .select(  "sale\_id",  "amount",  "sale\_date",  "customer\_id"  )  )  # Save standardized table  silver\_df.write.format("delta").mode("overwrite").save("/mnt/delta/silver\_sales") ✨ Gold Table (Curated) Your **Gold** table contains business aggregates, for example:  python  CopyEdit  from pyspark.sql.functions import sum  silver\_df = spark.read.format("delta").load("/mnt/delta/silver\_sales")  gold\_df = (  silver\_df  .groupBy("customer\_id")  .agg(  sum("amount").alias("total\_sales")  )  )  gold\_df.write.format("delta").mode("overwrite").save("/mnt/delta/gold\_customer\_sales")  ✅ **This separates raw, clean, and business-ready data.** 🟣 3️⃣ Data Quality Checks to Detect Schema Drift Early ✅ **Why do this?** Schema drift may still cause:   * Missing columns * Unexpected data types * Null spikes * Duplicate rows   ✅ **Typical checks:**   | **Check** | **What It Detects** | | --- | --- | | **Column presence** | Are all expected columns there? | | **Data types** | Are columns in expected types? | | **Null thresholds** | Too many nulls in critical columns? | | **Row counts** | Is volume consistent? | | **Duplicates** | Duplicate IDs? |   ✅ **Example Data Quality Script in PySpark**  python  CopyEdit  bronze\_df = spark.read.format("delta").load("/mnt/delta/bronze\_sales")  # 1. Column presence  expected\_columns = {"sale\_id", "amount", "sale\_date", "customer\_id"}  actual\_columns = set(bronze\_df.columns)  missing\_columns = expected\_columns - actual\_columns  if missing\_columns:  raise ValueError(f"Missing columns detected: {missing\_columns}")  # 2. Null threshold check  null\_counts = bronze\_df.selectExpr([  "count(\*) as total\_rows",  "count(sale\_id) as non\_null\_sale\_id",  "count(amount) as non\_null\_amount"  ]).collect()[0]  if null\_counts['non\_null\_sale\_id'] < 0.95 \* null\_counts['total\_rows']:  raise ValueError("More than 5% null sale\_id detected.")  # 3. Duplicate check  from pyspark.sql.functions import count  dup\_df = (  bronze\_df.groupBy("sale\_id")  .agg(count("\*").alias("count"))  .filter("count > 1")  )  if dup\_df.count() > 0:  raise ValueError("Duplicate sale\_id detected.")  print("✅ Data quality checks passed.") 🧠 Recap & Blueprint ✅ **Auto Loader Ingestion Pipeline:**   * Use mergeSchema and schemaLocation.   ✅ **Bronze–Silver–Gold Layers:**   * **Bronze:** Raw ingestion (accept drift). * **Silver:** Standardize schema, clean data. * **Gold:** Aggregated, trusted outputs.   ✅ **Data Quality:**   * Validate columns. * Enforce null thresholds. * Deduplicate. * Monitor schema history (DESCRIBE HISTORY). |
|  |
| 🎯 ****Scenario: A Streaming Platform (e.g., Netflix) Handling Evolving User Activity Data**** **Context:**   * They track user events: play, pause, browse, etc. * Event schema evolves over time as app features change. * They must:   + Avoid data loss from schema drift.   + Keep the schema clean and well-governed.   + Evolve their Delta tables to include new fields.   ✅ **This example demonstrates:**   * Schema drift: Unexpected new columns appearing. * Schema evolution: Controlled adoption of new columns into the target tables.  🪜 ****Step-by-Step: Ingestion, Detection, Evolution**** I’ll present each step in **a table** so you can follow easily. 📊 ****Table of Steps****  | **🟢 Step** | **📝 Description** | **🔧 Code Example** | | --- | --- | --- | | **1. Initial Schema Definition** | Define base schema for raw event ingestion. | python from pyspark.sql.types import StructType, StructField, StringType, TimestampType base\_schema = StructType([ StructField("event\_id", StringType()), StructField("user\_id", StringType()), StructField("event\_type", StringType()), StructField("event\_timestamp", TimestampType()) ]) | | **2. Initial Auto Loader Ingestion** | Start streaming ingestion with explicit schema. | python df\_raw = ( spark.readStream .format("cloudFiles") .option("cloudFiles.format", "json") .option("cloudFiles.schemaLocation", "/mnt/schema/user\_events") .schema(base\_schema) .load("/mnt/raw/user\_events/") ) df\_raw.writeStream .format("delta") .option("checkpointLocation", "/mnt/checkpoints/user\_events") .start("/mnt/delta/bronze\_user\_events") | | **3. New Columns Arrive (Schema Drift)** | New fields appear in incoming data, e.g.:  device\_type, app\_version | **Sample new JSON:** json { "event\_id":"e123", "user\_id":"u456", "event\_type":"play", "event\_timestamp":"2023-08-01T12:00:00Z", "device\_type":"mobile", "app\_version":"1.2.3" } | | **4. Schema Drift Detected** | Auto Loader saves inferred schema changes in schemaLocation.  You see it in logs or by inspecting the evolving schema JSON. | **How to inspect schema:** %fs ls /mnt/schema/user\_events | | **5. Enable Schema Evolution** | Update ingestion to merge new columns into the Delta table automatically. | python df\_raw = ( spark.readStream .format("cloudFiles") .option("cloudFiles.format", "json") .option("cloudFiles.schemaLocation", "/mnt/schema/user\_events") .option("mergeSchema", "true") .load("/mnt/raw/user\_events/") ) df\_raw.writeStream .format("delta") .option("checkpointLocation", "/mnt/checkpoints/user\_events") .option("mergeSchema", "true") .start("/mnt/delta/bronze\_user\_events") | | **6. Bronze Table Schema Evolves** | New columns (device\_type, app\_version) are now in the table. | **Check schema:** spark.read.format("delta").load("/mnt/delta/bronze\_user\_events").printSchema() | | **7. Data Validation in Silver Layer** | In Silver, select expected columns, cast types, and set defaults if needed. | python bronze\_df = spark.read.format("delta").load("/mnt/delta/bronze\_user\_events") silver\_df = ( bronze\_df .withColumn("device\_type", bronze\_df["device\_type"].cast("string")) .withColumn("app\_version", bronze\_df["app\_version"].cast("string")) .withColumn("event\_date", to\_date("event\_timestamp")) .select( "event\_id", "user\_id", "event\_type", "event\_timestamp", "device\_type", "app\_version", "event\_date" ) ) silver\_df.write.format("delta").mode("overwrite").save("/mnt/delta/silver\_user\_events") | | **8. Gold Aggregations** | Use new fields to produce insights (e.g., events per device type). | python silver\_df = spark.read.format("delta").load("/mnt/delta/silver\_user\_events") gold\_df = ( silver\_df.groupBy("device\_type").agg(count("\*").alias("event\_count")) ) gold\_df.write.format("delta").mode("overwrite").save("/mnt/delta/gold\_device\_events") | | **9. Data Quality Alerts for Unexpected Schema Drift** | Detect columns that were not expected, raise an alert or log. | python expected = {"event\_id","user\_id","event\_type","event\_timestamp","device\_type","app\_version"} actual = set(bronze\_df.columns) unexpected = actual - expected if unexpected: print("⚠️ WARNING: Unexpected columns detected:", unexpected) | | **10. Schema Governance with Unity Catalog** | Optionally, manage schema versions and permissions centrally. | **Unity Catalog (SQL):** sql CREATE CATALOG analytics; USE CATALOG analytics; CREATE SCHEMA events; CREATE TABLE events.user\_events ( ... ); GRANT SELECT ON TABLE events.user\_events TO data\_analyst; |  🟡 ****What Does This Show?**** This process demonstrates **both schema drift and schema evolution:**  ✅ **Schema Drift:**   * New columns (device\_type, app\_version) unexpectedly arrived in raw data. * Auto Loader detected them automatically.   ✅ **Schema Evolution:**   * By enabling mergeSchema=true, you explicitly accepted the schema changes and evolved the Delta table.  🟢 ****Key Patterns Netflix, Uber, and Similar Companies Use**** ✅ **Separate Raw and Curated Zones**   * **Bronze:** Ingest everything—drift is accepted. * **Silver:** Standardize schema, fill defaults. * **Gold:** Trustworthy analytics.   ✅ **Schema Location**   * Always configure cloudFiles.schemaLocation to track schema evolution.   ✅ **Validation & Alerting**   * Monitor incoming schemas. * Raise alerts for truly unexpected drift.   ✅ **Time Travel**   * Use Delta’s VERSION AS OF or TIMESTAMP AS OF to recover older schema versions if drift causes problems.   ✅ **Unity Catalog (Governance)**   * Central catalog to manage table definitions and permissions.  💡 ****Visual Reference**** **Data Flow Diagram**  pgsql  CopyEdit  [Raw JSON Files with Drift]  |  Auto Loader  (mergeSchema + schemaLocation)  |  [Bronze Delta Table]  |  Data Validation  (Standardize Schema)  |  [Silver Delta Table]  |  Aggregation & Curation  |  [Gold Delta Table]  ✅ **Summary Checklist**   | **✅ What You Need To Do** | **How** | | --- | --- | | **Detect Drift** | Inspect schema location metadata | | **Control Evolution** | Use .option("mergeSchema", "true") only when desired | | **Separate Raw & Clean Data** | Bronze (drift), Silver (clean), Gold (aggregated) | | **Alert on Unexpected Columns** | Compare expected vs. actual columns | | **Govern Schema** | Use Unity Catalog and DESCRIBE HISTORY | |

|  |
| --- |
| Schema drift handling on VARIANT |
| Excellent question—let’s **dig deeper** into how **schema drift** and **schema evolution** handle **VARIANT data types**, what causes drift in them, and how this is managed in Databricks Delta Lake.  This is especially important if you’re working with **semi-structured data** (JSON, Avro, Parquet), which is where schema drift shows up the most.  I’ll cover:  ✅ What VARIANT data is ✅ Why VARIANT increases schema drift ✅ How schema evolution applies to VARIANT ✅ Practical examples step by step ✅ What best practices help you manage it 🎯 1️⃣ What Is VARIANT Data? ✅ **VARIANT** is a flexible column type used in systems like **Snowflake** (in Databricks you typically work with MapType or StructType instead) to store **semi-structured** or **nested data**.  For example:  json  CopyEdit  {  "user\_id": "U123",  "device": {  "type": "mobile",  "os": "iOS"  },  "event\_properties": {  "video\_length": 120,  "playback\_speed": 1.0  }  }  In Delta Lake, you’d store this using:   * StructType * MapType * or a string column with raw JSON   ✅ **How schema drift arises with VARIANT / nested structures:**   * New fields appear under device or event\_properties. * Data types of fields change (e.g., string to integer). * Entire sub-objects are added or removed.  🚩 2️⃣ What Causes Drift With VARIANT? **Examples of schema drift:**   | **Snapshot in time** | **Sample JSON** | | --- | --- | | **Day 1** | json { "user\_id": "U123", "device": { "type": "mobile" } } | | **Day 2** | json { "user\_id": "U123", "device": { "type": "mobile", "os": "iOS" } } | | **Day 3** | json { "user\_id": "U123", "device": { "type": 100 } } | | **Day 4** | json { "user\_id": "U123", "device": "mobile" } |   ✅ These changes are **schema drift** because:   * New nested fields (os) appeared. * Data types changed (type string → integer). * Entire object changed shape (device object → string).  🛠️ 3️⃣ How Schema Evolution Works With Nested Data ✅ In Delta Lake:   * **Schema evolution** (mergeSchema) can **add new fields** to nested structures automatically. * It **does NOT automatically reconcile data type conflicts**.   ✅ What Delta handles:   * Adding new fields to structs * Adding new top-level columns   ✅ What Delta **does NOT** handle automatically:   * Type conflicts within nested structures * Renaming nested fields * Removing fields  🧪 4️⃣ Detailed Example Step by Step Let’s do a **concrete example** in Databricks. 🟢 Step 1 – Define Initial Schema python  CopyEdit  from pyspark.sql.types import StructType, StructField, StringType, StructType, IntegerType  schema = StructType([  StructField("user\_id", StringType()),  StructField("device", StructType([  StructField("type", StringType())  ]))  ])  ✅ **Initial Data:**  json  CopyEdit  {  "user\_id": "U123",  "device": {  "type": "mobile"  }  } 🟡 Step 2 – Load Data Without Drift python  CopyEdit  df = spark.read.schema(schema).json("/mnt/raw/day1/")  df.write.format("delta").mode("overwrite").save("/mnt/delta/user\_events")  ✅ **Schema after Day 1:**  go  CopyEdit  root  |-- user\_id: string  |-- device: struct  |-- type: string 🟠 Step 3 – New Nested Field Arrives (Schema Drift) **Day 2 Data:**  json  CopyEdit  {  "user\_id": "U456",  "device": {  "type": "tablet",  "os": "Android"  }  } 🟣 Step 4 – Append Data With Schema Evolution ✅ This **adds the new nested field** automatically:  python  CopyEdit  df\_new = spark.read.json("/mnt/raw/day2/")  df\_new.write \  .format("delta") \  .option("mergeSchema", "true") \  .mode("append") \  .save("/mnt/delta/user\_events")  ✅ **Schema after Day 2:**  go  CopyEdit  root  |-- user\_id: string  |-- device: struct  |-- type: string  |-- os: string  ✅ **🎉 This is schema evolution at work.** 🔴 Step 5 – Data Type Conflict Appears (Schema Drift) **Day 3 Data:**  json  CopyEdit  {  "user\_id": "U789",  "device": {  "type": 100  }  }  ✅ **Attempting to append this fails:**  python  CopyEdit  df\_bad = spark.read.json("/mnt/raw/day3/")  df\_bad.write \  .format("delta") \  .option("mergeSchema", "true") \  .mode("append") \  .save("/mnt/delta/user\_events")  ❌ **Error:**  go  CopyEdit  Failed to write: Found incompatible data types in column device.type:  - existing type: string  - new type: long  ✅ **Schema evolution does NOT automatically fix data type conflicts.** ⚖️ 5️⃣ How To Handle Type Conflicts ✅ **Option 1 – Cast Before Write**  python  CopyEdit  df\_clean = df\_bad.withColumn("device",  expr("named\_struct('type', cast(device.type as string))")  )  df\_clean.write \  .format("delta") \  .mode("append") \  .save("/mnt/delta/user\_events")  ✅ **Option 2 – Store Entire Row as String JSON**   * Instead of parsing nested columns, keep a raw JSON string column. * Process in Silver layer.   ✅ **Option 3 – Use Auto Loader with Permissive Mode**   * Auto Loader can load unknown columns and nulls, but still won’t auto-fix conflicting types.  💡 6️⃣ Best Practices Table  | **✅ Best Practice** | **💡 Why** | | --- | --- | | Use explicit StructType schemas | Avoids unpredictable type inference | | Enable mergeSchema=true for adding new fields | Lets you evolve schema without recreating tables | | Validate incoming types | Catch conflicts before write | | Separate raw (Bronze) and clean (Silver) data | Keep drift isolated | | Store raw JSON if schema changes frequently | Avoid frequent schema churn in Delta tables | | Use time travel and schema history | Recover previous states if needed |  📝 7️⃣ Recap: How Schema Drift vs Schema Evolution Apply to VARIANT/Nested Data  | **Concept** | **Description** | | --- | --- | | **Schema Drift** | New fields or types appear in nested structures | | **Schema Evolution** | Allows new fields to be merged automatically (mergeSchema=true) | | **NOT handled automatically** | Data type conflicts or field renames |   ✅ **Summary:**   * **Schema drift happens** whenever new keys or types arrive in your nested data. * **Schema evolution helps** automatically add new fields to the Delta table. * **Type conflicts still require you to clean or cast data manually.** |