Highlighting Delta Lake: The Data Lakehouse Storage Layer

Delta Lake is an open-source storage layer that brings ACID (Atomicity, Consistency, Isolation, Durability) transactions, schema enforcement, and time travel capabilities to data lakes. It unifies batch and streaming data processing, transforming your MinIO (S3-compatible) storage into a reliable "data lakehouse."
This guide will demonstrate basic and advanced use cases of Delta Lake, leveraging your **Advanced Track** local environment setup and focusing on its integration with Apache Spark.
Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook**, the **Progressive Path Setup Guide Deep-Dive Addendum**, and the Spark examples in the "Highlighting Apache Spark" document.

Basic Use Case: Reliable Data Appends to the Data Lakehouse

Objective: To demonstrate how Spark Structured Streaming reliably appends data to a Delta Lake table in MinIO, ensuring transactionality and consistency even in streaming scenarios.
Role in Platform: Provide a solid, transactional foundation for raw and curated data storage, enabling consistent reads for downstream consumers.
Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure all Advanced Track services are running:** docker compose up --build -d from your project root.
2. **Verify Spark and MinIO are accessible:** Check their respective container logs and UIs (http://localhost:18080 for Spark History, http://localhost:9001 for MinIO).
3. **Ensure simulate_data.py is running:** Data should be continuously sent to FastAPI and then published to Kafka topics (raw_financial_transactions, raw_insurance_claims).
4. **Spark Streaming Jobs are running:** The pyspark_jobs/streaming_consumer.py jobs (from the "Highlighting Apache Spark" document's Basic Use Case) should be actively consuming from Kafka and writing to s3a://raw-data-bucket/financial_data_delta and s3a://raw-data-bucket/insurance_data_delta.

Steps to Exercise:

1. **Observe Writes:** Let the streaming_consumer.py Spark jobs run for a few minutes.
2. **Inspect MinIO:** Access the MinIO Console (http://localhost:9001).
    - Navigate to raw-data-bucket.
    - Enter financial_data_delta/ (or insurance_data_delta/).
    - You will see a _delta_log directory and numerous .parquet files. The presence of _delta_log confirms it's a Delta Lake table. The .parquet files are the actual data appended in micro-batches by Spark.

3. **Query Data (via Spark-SQL):** You can query the Delta table using Spark's SQL capabilities from within the spark container to verify data content.
   ```
   docker exec -it spark spark-sql \
     --packages io.delta:delta-core_2.12:2.4.0 \
     --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
     --conf
   spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
     --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
     --conf spark.hadoop.fs.s3a.access.key=minioadmin \
     --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
     --conf spark.hadoop.fs.s3a.path.style.access=true \
     -e "SELECT COUNT(*) FROM delta.\`s3a://raw-data-bucket/financial_data_delta\`;"
   ```

   This query will show the growing number of records in the Delta table, demonstrating continuous, transactional appends.

Verification:

- **MinIO Contents:** The _delta_log directory and increasing number of .parquet files in financial_data_delta and insurance_data_delta paths.
- **Spark-SQL Query:** The COUNT(*) query reflects a continuously growing number, indicating that new data is being reliably appended.
- **Spark History Server:** The streaming jobs remain active and show consistent write operations to Delta Lake.

Advanced Use Case 1: Schema Enforcement and Evolution

Objective: To demonstrate how Delta Lake enforces schema by default, preventing bad data from corrupting tables, and how it allows controlled schema evolution to adapt to changing data structures.

Role in Platform: Maintain data quality and flexibility in data lake schemas, preventing silent data corruption.

Setup/Configuration:

1. **Ensure Basic Use Case is running:** financial_data_delta is populated.
2. **pyspark_jobs/streaming_consumer.py:** Ensure this script is using the data_schema as defined, which acts as the target schema for the Delta table.

Steps to Exercise:

1. **Simulate Schema Violation (Expected to Fail without mergeSchema):**
   - Temporarily modify simulate_data.py to send a single batch of financial transactions with a **data type mismatch** for an existing column (e.g., send amount as a string instead of a float).
     - **Original (in simulate_data.py):** "amount": round(random.uniform(1.0, 10000.0), 2),
     - **Temporary change (for a few seconds):** "amount": "FIVE HUNDRED",

- Let simulate_data.py run for 5-10 seconds with this change, then **immediately revert simulate_data.py back to its original (correct float) format.**
- Observe the logs of your financial_transactions Spark streaming job.

2. **Verify Schema Enforcement:**
   - **Expected Behavior (without mergeSchema / overwriteSchema):** The Spark streaming job will likely encounter an error (e.g., AnalysisException: Cannot write unknown type string into float type column ... or similar type mismatch error). The job might stop or continuously restart, demonstrating Delta Lake's strict schema enforcement.
   - **Action:** If the job failed, restart it with the correct data (simulate_data.py reverted).
3. **Demonstrate Schema Evolution (.option("mergeSchema", "true")):**
   - Modify simulate_data.py to add a **new optional column** to the financial transaction data (e.g., is_flagged: bool = False).
     - **Add to transaction_data in simulate_data.py:** "is_flagged": random.choice([True, False]),
   - **Update pyspark_jobs/streaming_consumer.py:** Add this new column to the data_schema definition, and ensure the .writeStream call includes .option("mergeSchema", "true").

     ```
     # In data_schema definition:
     .add("is_flagged", BooleanType(), True) # Add this line

     # In writeStream options:
     .option("mergeSchema", "true") # Ensure this option is present
     ```

   - **Restart both simulate_data.py and the financial_transactions Spark streaming job.**
   - Let them run for a few minutes.
4. **Verify Schema Evolution:**
   - **Query Delta Table:**

     ```
     docker exec -it spark spark-sql \
       --packages io.delta:delta-core_2.12:2.4.0 \
       --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
       --conf
     spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
     \
       --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
       --conf spark.hadoop.fs.s3a.access.key=minioadmin \
       --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
       --conf spark.hadoop.fs.s3a.path.style.access=true \
       -e "DESCRIBE HISTORY delta.\`s3a://raw-data-bucket/financial_data_delta\`;"
     ```

     Look for a new version (version) that includes the schema change.

Then, query the data:

```
docker exec -it spark spark-sql \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
\
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  -e "SELECT transaction_id, amount, is_flagged FROM
delta.\`s3a://raw-data-bucket/financial_data_delta\` LIMIT 20;"
```

Verification:

- **Schema Enforcement (Failure):** When sending invalid data without mergeSchema, the Spark job should log errors and potentially fail, demonstrating that Delta Lake prevents malformed data from being written.
- **Schema Evolution (Success):** After adding the new column and using mergeSchema=true, the DESCRIBE HISTORY command will show a new table version with the updated schema. Queries will show the is_flagged column, with null values for older records and True/False for newly ingested ones. This highlights controlled schema evolution.

Advanced Use Case 2: Time Travel (Data Versioning)

Objective: To demonstrate Delta Lake's "time travel" capability, allowing you to query historical versions of your data, crucial for auditing, reproducing past reports, and recovering from accidental data modifications.
Role in Platform: Provide data versioning for compliance, reproducibility, and disaster recovery.
Setup/Configuration:

1. **Ensure Basic Use Case is running:** financial_data_delta is being continuously written to.
2. **Note a timestamp/version:** Let data stream for some time. Note the timestamp or version from DESCRIBE HISTORY.

Steps to Exercise:

1. **Get Current Table State (Baseline):**
   ```
   docker exec -it spark spark-sql \
     --packages io.delta:delta-core_2.12:2.4.0 \
     --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
     --conf
   spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
   ```

```
--conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
--conf spark.hadoop.fs.s3a.access.key=minioadmin \
--conf spark.hadoop.fs.s3a.secret.key=minioadmin \
--conf spark.hadoop.fs.s3a.path.style.access=true \
-e "SELECT COUNT(*) FROM delta.\`s3a://raw-data-bucket/financial_data_delta\`;"
```

Note the count.

2. **Simulate an "Accidental" Overwrite/Deletion:**
   - Stop the financial_transactions Spark streaming job.
   - Run a temporary Spark batch job to OVERWRITE the entire financial_data_delta table with a very small subset of data, or even empty data, simulating data loss.
     Example pyspark_jobs/temp_overwrite.py (create this file):

```
from pyspark.sql import SparkSession
from delta.tables import DeltaTable
import sys

def create_spark_session(app_name):
    return (SparkSession.builder.appName(app_name)
            .config("spark.jars.packages", "io.delta:delta-core_2.12:2.4.0")
            .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
            .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
            .getOrCreate())

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: temp_overwrite.py <delta_table_path>")
        sys.exit(-1)

    delta_table_path = sys.argv[1]
    spark = create_spark_session("TempOverwrite")

    # Create a very small DataFrame to overwrite the existing table
    small_data = [("OVERWRITE-001", "2024-01-01T10:00:00.000Z",
"ACC-OVERWRITE", 1.0, "USD", "payment", "MER-OVW", "misc")]
    columns = ["transaction_id", "timestamp", "account_id", "amount", "currency",
"transaction_type", "merchant_id", "category"]
    small_df = spark.createDataFrame(small_data, schema=columns) # Use the full
schema for compatibility

    print(f"Overwriting {delta_table_path} with small data...")
    small_df.write.format("delta").mode("overwrite").option("overwriteSchema",
"true").save(delta_table_path)
```

```
print("Overwrite complete.")
spark.stop()
```

- Run this job:

```
docker exec -it spark spark-submit \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
\
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  /opt/bitnami/spark/jobs/temp_overwrite.py \
  s3a://raw-data-bucket/financial_data_delta
```

- Now, query the table normally and note the COUNT(*) again. It should be drastically reduced (e.g., 1 record).

3. **Query History:**

```
docker exec -it spark spark-sql \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  -e "DESCRIBE HISTORY delta.\`s3a://raw-data-bucket/financial_data_delta\`;"
```

   This will show a list of all operations, including your initial appends and the recent OVERWRITE. Note the version (e.g., 0, 1, 2, etc.) and timestamp of a version *before* your overwrite.

4. **Time Travel Query (by Version):**

```
docker exec -it spark spark-sql \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
```

```
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  -e "SELECT COUNT(*) FROM delta.\`s3a://raw-data-bucket/financial_data_delta\`
VERSION AS OF <PREVIOUS_VERSION_NUMBER>;"
```

Replace <PREVIOUS_VERSION_NUMBER> with the version *before* your overwrite. The count should revert to the original large number.

5. **Time Travel Query (by Timestamp):**
```
docker exec -it spark spark-sql \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  -e "SELECT COUNT(*) FROM delta.\`s3a://raw-data-bucket/financial_data_delta\`
TIMESTAMP AS OF 'YYYY-MM-DD HH:MM:SS';"
```

Replace 'YYYY-MM-DD HH:MM:SS' with a timestamp *before* your overwrite.

6. **Rollback (Conceptual):** While not easily demonstrated via simple spark-sql in this context, in a real scenario, you could use RESTORE TABLE command or read a historical version and write it back to the current version.

Verification:

- The current table (SELECT COUNT(*) FROM delta.\...`) shows the limited data after the overwrite.
- Time travel queries (VERSION AS OF or TIMESTAMP AS OF) successfully retrieve the larger dataset from before the overwrite, demonstrating that the historical data is still available.
- DESCRIBE HISTORY clearly shows the sequence of operations and versions.

Advanced Use Case 3: Upserts (MERGE INTO) and Change Data Capture (CDC)

Objective: To demonstrate how Delta Lake supports efficient upsert operations (inserting new records and updating existing ones) and how its transaction log can be used for Change Data Capture (CDC). This is crucial for building slowly changing dimensions and replicating data efficiently.

Role in Platform: Support data warehousing patterns, facilitate efficient data synchronization, and enable real-time data replication.

Setup/Configuration:

1. **Populate a "Target" Delta Table:** Create a new Delta table that will serve as our target for upserts (e.g., s3a://curated-data-bucket/financial_transactions_dim). For simplicity,

start it with some initial data.

2. **Source for Updates:** Use a simple Spark DataFrame or simulate a small Kafka topic with update/insert records.
3. **Spark Merge Script:** Create pyspark_jobs/delta_merge_cdc.py.
   Example pyspark_jobs/delta_merge_cdc.py (conceptual):

```
# pyspark_jobs/delta_merge_cdc.py
import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit, current_timestamp
from delta.tables import DeltaTable # Import DeltaTable class

def create_spark_session(app_name):
    return (SparkSession.builder.appName(app_name)
            .config("spark.jars.packages", "io.delta:delta-core_2.12:2.4.0")
            .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
            .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
            .getOrCreate())

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: delta_merge_cdc.py <target_delta_table_path>")
        sys.exit(-1)

    target_path = sys.argv[1]
    spark = create_spark_session("DeltaMergeAndCDC")
    spark.sparkContext.setLogLevel("WARN")

    # --- Part 1: Initial Population (if table doesn't exist) ---
    initial_data = [
        ("T001", "2024-05-01", "A001", 100.0, "USD", "credit"),
        ("T002", "2024-05-01", "A002", 200.0, "USD", "debit"),
        ("T003", "2024-05-02", "A001", 50.0, "EUR", "payment")
    ]
    initial_schema = ["transaction_id", "date", "account_id", "amount", "currency", "type"]
    initial_df = spark.createDataFrame(initial_data, initial_schema)

    if not DeltaTable.isDeltaTable(spark, target_path):
        print(f"Creating initial Delta table at {target_path}")
        initial_df.write.format("delta").mode("overwrite").save(target_path)
    else:
        print(f"Delta table already exists at {target_path}. Skipping initial population.")
```

```
# Load the target Delta table
deltaTable = DeltaTable.forPath(spark, target_path)
print("Current data in target table:")
deltaTable.toDF().show()

# --- Part 2: Simulate incoming changes (new data + updates) ---
# T002: updated amount, currency
# T004: new transaction
# T005: new transaction
updates_data = [
    ("T002", "2024-05-01", "A002", 250.0, "GBP", "debit"), # Update existing T002
    ("T004", "2024-05-03", "A003", 150.0, "USD", "transfer"), # New T004
    ("T005", "2024-05-03", "A001", 75.0, "EUR", "credit") # New T005
]
updates_df = spark.createDataFrame(updates_data, initial_schema)
print("Incoming updates/inserts:")
updates_df.show()

# --- Part 3: Perform MERGE INTO operation (Upsert) ---
print("Performing MERGE INTO operation...")
deltaTable.alias("target") \
    .merge(
        updates_df.alias("source"),
        "target.transaction_id = source.transaction_id" # Match condition
    ) \
    .whenMatchedUpdate(set = { # If matched, update columns
        "date": "source.date",
        "account_id": "source.account_id",
        "amount": "source.amount",
        "currency": "source.currency",
        "type": "source.type",
        "last_updated": current_timestamp() # Add an audit column
    }) \
    .whenNotMatchedInsert(values = { # If not matched, insert all columns from source
        "transaction_id": "source.transaction_id",
        "date": "source.date",
        "account_id": "source.account_id",
        "amount": "source.amount",
        "currency": "source.currency",
        "type": "source.type",
        "last_updated": current_timestamp()
    }) \
    .execute()
```

```
    print("Data after MERGE INTO:")
    deltaTable.toDF().show()

    # --- Part 4: Demonstrate Change Data Feed (CDC) ---
    # Querying the change data feed requires it to be enabled on the table
    # If not enabled during table creation, you can alter it:
    # spark.sql(f"ALTER TABLE delta.`{target_path}` SET TBLPROPERTIES
(delta.enableChangeDataFeed = true)")
    print("\nDemonstrating Change Data Feed (CDC) - Requires
'delta.enableChangeDataFeed = true' on table.")
    print("Querying changes since last operation:")
    try:
        # Get changes from the last version (current version - 1)
        # You would typically get the version number from `DESCRIBE HISTORY` or track it
        current_version =
deltaTable.history().select("version").orderBy(col("version").desc()).first()["version"]
        # To get changes from the last MERGE, we need to know its version
        # For simplicity here, we'll query changes from version 0

        # This is a bit tricky locally without persistent enablement
        # Best way to ensure CDC is enabled:
        # 1. Create a fresh table with cdc enabled:
        # spark.sql(f"CREATE TABLE IF NOT EXISTS delta.`{target_path}_cdc_enabled`
USING DELTA LOCATION '{target_path}_cdc_enabled' TBLPROPERTIES
(delta.enableChangeDataFeed = true)")
        # 2. Then merge into that table
        # 3. Then query readChangeFeed

        # For conceptual demo, assume CDC is enabled.
        # You can set this explicitly if you create the table via DDL or initial write
        # with .option("delta.enableChangeDataFeed", "true")
        # Or run ALTER TABLE manually in spark-sql before running this script
        # spark.sql(f"ALTER TABLE delta.`{target_path}` SET TBLPROPERTIES
(delta.enableChangeDataFeed = true)")

        # Querying the change data feed from the version before the merge
        # This will show 'insert' for new records and 'update_preimage'/'update_postimage'
for updated ones
        changes_df = spark.read.format("delta") \
            .option("readChangeFeed", "true") \
            .option("startingVersion", str(current_version)) \
            .load(target_path)
```

```
            print(f"Changes from version {current_version}:")
            changes_df.show(truncate=False)

        except Exception as e:
            print(f"Could not demonstrate CDC (Change Data Feed): {e}")
            print("Ensure 'delta.enableChangeDataFeed = true' is set on the Delta table properties.")
            print("You might need to run `ALTER TABLE delta.`<path>` SET TBLPROPERTIES (delta.enableChangeDataFeed = true)` manually in spark-sql, then run this script again after restarting Spark session.")

        spark.stop()
```

Steps to Exercise:

1. **Stop streaming jobs:** Ensure no other jobs are writing to the target path.
2. **Submit Merge/CDC Job:** In a new terminal, submit the delta_merge_cdc.py job.

```
docker exec -it spark spark-submit \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  /opt/bitnami/spark/jobs/delta_merge_cdc.py \
  s3a://curated-data-bucket/financial_transactions_dim
```

3. **Monitor:** Observe the console output.
    - It will show the initial data, then the incoming updates, then the result after the MERGE INTO.
    - It will also attempt to show the CDC.

Verification:

- **Upsert Verification:** After the job runs, query the financial_transactions_dim table (e.g., via spark-sql). You should see:
    - T002 updated with the new amount and currency (250.0, GBP).
    - New records T004 and T005 inserted.
- **CDC Verification (if successful):** The console output for CDC will show a DataFrame containing records with _change_type columns (insert, update_preimage, update_postimage), reflecting exactly what changed in the table during the merge operation. This demonstrates the ability to capture changes for downstream systems.