Delta Lake Introduction

Comprehensive Guide - Day 4

Yash Kavaiya

Databricks 14-Days AI Challenge

January 12, 2026

- **What is Delta Lake?**
  - Definition & Architecture
  - Problems It Solves
- **ACID Transactions**
  - Atomicity, Consistency
  - Isolation, Durability
- **Schema Enforcement**
  - Validation Rules
  - Schema Evolution
- **Delta vs Parquet**
- **Practical Tasks**

Definition

Delta Lake is an **open-source storage layer** that brings reliability, performance, and governance to data lakes.

Key Capabilities:

- Runs on existing data lake infrastructure
- Compatible with Apache Spark APIs
- Adds a "smart layer" on raw files
- Enables database-like behavior

Core Components:

- Data files in **Parquet format**
- Transaction log (`_delta_log`)
- Metadata management
- Version control

Challenge 1: Data Corruption

- Multiple jobs write simultaneously
- Files get corrupted or partially written
- Job failures leave incomplete data

Challenge 2: No Transaction Support

- Can't roll back failed operations
- Data ends up in inconsistent state

Challenge 3: Schema Chaos

- Different teams write different schemas
- "Schema drift" breaks applications

Challenge 4: No Time Travel

- Overwritten data is gone forever
- No easy way to recover mistakes

Directory Structure:

```
my_delta_table/
|-- _delta_log/
|   |-- 00000...000.json
|   |-- 00000...001.json
|   |-- 00000...002.json
|-- part-00000-xxx.parquet
|-- part-00001-xxx.parquet
```

Transaction Log Contains:

- Which data files were added
- Which data files were removed
- Schema changes
- Metadata updates

Each JSON file = One atomic commit

Feature	Description	Benefit
ACID Transactions	All operations atomic & isolated	No partial writes, consistent reads
Schema Enforcement	Validates data against table schema	Prevents bad data from entering
Schema Evolution	Safely add new columns	Adapt to changing requirements
Time Travel	Query previous versions	Audit, rollback, reproduce
Unified Batch/Streaming	Same table for both workloads	Simplified architecture
DML Operations	UPDATE, DELETE, MERGE support	SQL-like data manipulation

What is ACID?

Properties that ensure **reliable transaction processing** in database systems.

A - Atomicity

- "All or Nothing"
- Transaction is indivisible unit
- Either all complete or none

C - Consistency

- "Valid State to Valid State"
- All rules must be satisfied
- Schema & constraints enforced

I - Isolation

- "Transactions Don't Interfere"
- Concurrent execution is safe
- Consistent snapshots

D - Durability

- "Committed Data Persists"
- Survives system failures
- Permanently stored

Without Delta Lake:

- Job writes 1M records
- Crashes at 500K records
- 500K records partially written
- Data corrupted & unusable
- Manual cleanup required

With Delta Lake:

- Job writes 1M records
- Crashes at 500K records
- Transaction not committed
- Partial files ignored
- Table remains valid

Optimistic Concurrency Control: Reads current state, Writes new files, Commits atomically

How It Works:

1. Process A reads table at version 10
2. Process A runs 30-minute query
3. Process B writes new data (version 11)
4. Process A still sees version 10!

Result:
Consistent snapshot throughout query execution

Write Conflict Resolution:

- Process A commits to v11
- Process B attempts v11
- Conflict detected!
- B retries with v11 data
- Process B commits to v12

What is Schema Enforcement?

Schema on Write - Delta Lake validates that incoming data matches the table's schema exactly before writing.

Schema Drift Problem:

- Week 1: `{user_id: 1, name}`
- Week 2: `{userId: 2, userName}`
- Week 3: `{user_id: "3", name}`
- Dashboard breaks!

With Enforcement:

- Schema defined: `user_id INT`
- Week 2: REJECTED
- Week 3: REJECTED
- Data stays consistent!

Check	What It Validates	On Failure
Column Names	All columns exist in table	Exception thrown
Extra Columns	No columns not in table	Exception thrown
Data Types	Type matches table schema	Exception thrown
Nullability	NOT NULL columns have values	Exception thrown
Column Order	N/A	Auto-handled

Note

Schema Evolution can be enabled with `mergeSchema=true` option

Enabling Schema Evolution:

```
# Per-write option
df.write.format("delta")
  .mode("append")
  .option("mergeSchema", "true")
  .save("/delta/products")

# Session-wide setting
spark.conf.set(
  "spark.databricks.delta" +
  ".schema.autoMerge.enabled",
  "true")
```

Supported Operations:

- Add Column - mergeSchema
- Change Type - overwriteSchema
- Rename Column - ALTER TABLE
- Drop Column - Must rewrite table

Existing rows get NULL for new columns

Feature	Parquet	Delta Lake
File Format	Columnar	Columnar (Parquet)
ACID Transactions	No	Yes
Schema Enforcement	No (on read)	Yes (on write)
Time Travel	No	Yes
UPDATE/DELETE	No	Yes
MERGE (Upsert)	No	Yes
Concurrent Writes	Can corrupt	Safe
Streaming Support	Limited	Full
Small File Problem	Common	Auto-compact

Parquet (Complex):

```
# Must read ALL data
df = spark.read.parquet("/path")
# Filter and separate
keep = df.filter(col("id") != 1)
update = df.filter(col("id") == 1)
# Apply changes
updated = update.withColumn(...)
# Overwrite EVERYTHING
final = keep.union(updated)
final.write.mode("overwrite")
     .parquet("/path")
```

Not atomic, slow, unsafe

Delta Lake (Simple):

```
from delta.tables import DeltaTable

table = DeltaTable.forPath(
    spark, "/delta/products")

# Simple atomic UPDATE
table.update(
    condition="product_id = 1",
    set={"price": "899.99"}
)
```

Atomic, fast, safe

Query by Timestamp:

```
# Query data from yesterday
df = spark.read.format("delta")
    .option("timestampAsOf",
            "2024-01-14")
    .load("/delta/data")
```

Query by Version:

```
# Query specific version
df = spark.read.format("delta")
    .option("versionAsOf", 5)
    .load("/delta/data")
```

View History:

```
DESCRIBE HISTORY
  delta.`/delta/data`;
```

Use Cases:

- Audit trail compliance
- Rollback bad changes
- Reproduce ML results
- Debug data issues

Step 1: Read CSV with Schema

```
csv_schema = StructType([
  StructField("employee_id",
    IntegerType(), False),
  StructField("name",
    StringType(), False),
  StructField("department",
    StringType(), True),
  StructField("salary",
    DoubleType(), True)
])

csv_df = spark.read
  .option("header", "true")
  .schema(csv_schema)
  .csv("/data/csv/employees")
```

Step 2: Write as Delta

```
csv_df.write.format("delta")
  .mode("overwrite")
  .save("/delta/employees")

print("Converted to Delta!")
```

Result:

- Parquet files created
- Transaction log initialized
- Schema enforced on reads/writes

Method 1: PySpark API

```
# Create with data
products_df = spark.createDataFrame(
    products_data,
    ["id", "name", "price"]
)

products_df.write.format("delta")
  .mode("overwrite")
  .save("/delta/products")
```

Method 2: SQL

```
CREATE TABLE default.orders (
  order_id INT NOT NULL,
  customer_id INT NOT NULL,
  order_total DOUBLE,
  order_date DATE
)
USING DELTA
LOCATION '/delta/orders';

INSERT INTO default.orders
VALUES (1001, 501, 1299.99,
        '2024-01-10');
```

Write with Partitioning:

```
sales_df.write.format("delta")
  .mode("overwrite")
  .partitionBy("month", "region")
  .save("/delta/sales_partitioned")
```

Create Table As Select:

```
CREATE TABLE high_value_orders
USING DELTA
LOCATION '/delta/high_value'
AS SELECT * FROM orders
   WHERE order_total > 500;
```

Folder Structure:

```
sales_partitioned/
|-- month=2024-01/
|   |-- region=North/
|   |-- region=South/
|   |-- region=East/
|-- month=2024-02/
|   |-- ...
|-- _delta_log/
```

Benefit: Faster queries with partition pruning

Test 1: Wrong Data Type

```
# Table expects: price DOUBLE
bad_data = spark.createDataFrame([
  (106, "Keyboard", "fifty nine")
], ["id", "name", "price"])

try:
    bad_data.write.format("delta")
      .mode("append")
      .save("/delta/products")
except Exception as e:
    print("Blocked! Type mismatch")
```

Schema enforcement works!

Test 2: Extra Column

```
# DataFrame has extra 'brand'
extra = spark.createDataFrame([
  (108, "USB", 24.99, "BrandX")
], ["id", "name", "price", "brand"])

# Enable schema evolution:
extra.write.format("delta")
  .mode("append")
  .option("mergeSchema", "true")
  .save("/delta/products")
```

Now schema evolved!

Strategy 1: Insert Only New

```
delta_table.alias("target").merge(
    incoming_df.alias("source"),
    "target.id = source.id"
).whenNotMatchedInsertAll()
 .execute()
```

Strategy 2: Upsert

```
delta_table.alias("target").merge(
    incoming_df.alias("source"),
    "target.id = source.id"
).whenMatchedUpdateAll()
 .whenNotMatchedInsertAll()
 .execute()
```

MERGE Logic:

- Match on join condition
- Matched: UPDATE or DELETE
- Not Matched: INSERT

Conditional Update:

```
.whenMatchedUpdate(
   set={"email":
        col("source.email")}
)
```

```
MERGE INTO delta.`/delta/customers` AS target
USING incoming_customers AS source
ON target.customer_id = source.customer_id

WHEN MATCHED THEN
    UPDATE SET
        email = source.email,
        name = source.name

WHEN NOT MATCHED THEN
    INSERT (customer_id, name, email, created_date)
    VALUES (source.customer_id, source.name,
            source.email, source.created_date);
```

Result: Atomic operation - updates existing, inserts new, all in one transaction

Reading Data:

```
# Basic read
df = spark.read.format("delta")
  .load("/path/to/delta")

# Read specific version
df = spark.read.format("delta")
  .option("versionAsOf", 5)
  .load("/path")

# Read at timestamp
df = spark.read.format("delta")
  .option("timestampAsOf",
          "2024-01-10")
  .load("/path")
```

Writing Data:

```
# Overwrite
df.write.format("delta")
  .mode("overwrite")
  .save("/path")

# Append
df.write.format("delta")
  .mode("append")
  .save("/path")

# With schema evolution
df.write.format("delta")
  .mode("append")
  .option("mergeSchema", "true")
  .save("/path")
```

Python API:

```python
from delta.tables import DeltaTable

table = DeltaTable.forPath(
    spark, "/path")

# UPDATE
table.update(
    condition="id = 1",
    set={"value": "100"})

# DELETE
table.delete(condition="id = 1")
```

SQL:

```sql
-- Update
UPDATE delta.`/path`
SET name = 'Bob'
WHERE id = 1;

-- Delete
DELETE FROM delta.`/path`
WHERE id = 1;

-- Vacuum (cleanup)
VACUUM delta.`/path`
RETAIN 168 HOURS;
```

History & Details:

```
-- View history
DESCRIBE HISTORY delta.`/path`;

-- View details
DESCRIBE DETAIL delta.`/path`;
```

Optimize:

```
# Compact small files
delta_table.optimize()
  .executeCompaction()
```

Vacuum:

```
# Remove old files (7 days)
delta_table.vacuum(
    retentionHours=168)
```

Common Configs:

- autoOptimize.optimizeWrite
- autoOptimize.autoCompact
- deletedFileRetention

Delta Lake transforms data lakes into reliable lakehouses

- **ACID Transactions** - Data integrity guaranteed
- **Schema Enforcement** - Prevents bad data
- **Time Travel** - Version history access
- **Unified Batch/Streaming** - Same table for all
- **Full DML Support** - UPDATE, DELETE, MERGE
- **Audit History** - Complete changelog

Key Insight

Transaction log + Parquet files = Database reliability + Data lake flexibility

Thank You!

Questions?

Connect with me:

LinkedIn: Yash Kavaiya

Easy AI Labs