Big Data Lab

Lab 2: Lakehouse Architecture & Delta Lake Foundations

Dr. Innocent Nyalala

IIT Madras Zanzibar
School of Engineering and Science

Monday, 16 March 2026 — 2:00–4:40 PM

D0.jpg

Recap: What We Built Last Week

The Modern Big Data Stack (Local):

- We bypassed complex cloud setups by using **Docker Compose**.
- Bootstrapped a fully isolated environment with 9 interlinked containers.
- **Storage:** MinIO (S3-compatible object store)
- **Compute:** Spark Master, Spark Workers, JupyterLab
- **Streaming:** Kafka

Pre-flight Checklist

Before we start today's lab, ensure your environment is running:
`docker ps`
(You should see 9 healthy containers)

JupyterLab: `localhost:8888`
MinIO UI: `localhost:9001`



Agenda

Theory: 2:00–3:10 (70 min)

1. **Storage Evolution:** DW → Lake → Lakehouse
2. **Medallion Architecture:** Bronze / Silver / Gold
3. **Delta Internals:** The Transaction Log
4. **ACID:** Reliability in a distributed world
5. **Core Features:** Time travel, MERGE, Schema evolution, OPTIMIZE

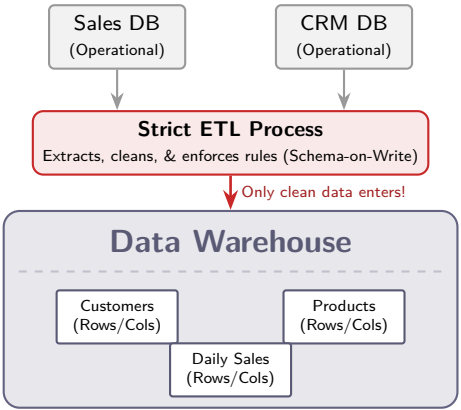
Hands-On: 3:20–4:30 (70 min)

1. **Setup:** Connect PySpark to MinIO (S3A)
2. **Write & Query:** Your first 10K-row Delta table
3. **Time Travel:** Read historical versions
4. **Mutate Data:** MERGE/UPSERT real updates atomically
5. **Maintenance:** OPTIMIZE + Z-ORDER

Objective: By the end of today you will have written, queried, time-travelled, and merged data in Delta Lake on your local MinIO object store. Open your browser: `localhost:9001` now.



1. What is a Data Warehouse? (The Organised Library)

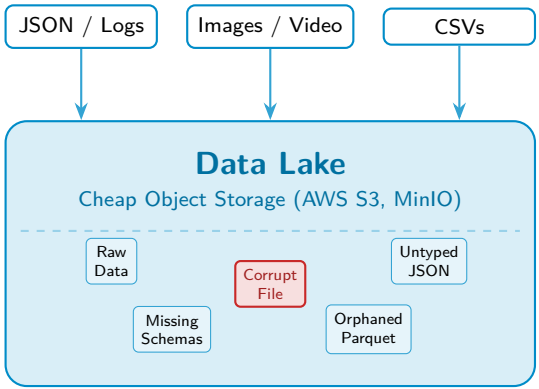


The Analogy: A highly organised university library. Every book has an exact shelf, an index card, and a barcode.

- **What it stores:** Structured, tabular data only (Rows & Columns). No raw text or images.
- **How it works:** Data must be cleaned and formatted *before* it is loaded (Schema-on-Write).
- **Pros:** Blazing fast for BI dashboards, highly reliable (ACID).
- **Cons:** Extremely expensive, rigid, and terrible for modern AI/Machine Learning.

DO.jpg

2. What is a Data Lake? (The Infinite Dumping Ground)

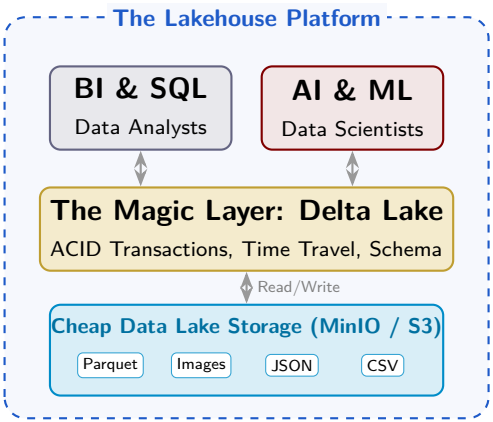


The Analogy: A massive, cheap warehouse where you can just throw boxes of any shape or size.

- **What it stores:** Everything. Structured, semi-structured, and unstructured data.
- **How it works:** Dump data now, figure out what it means later (Schema-on-Read).
- **Pros:** Infinitely scalable, incredibly cheap, great for Data Scientists to explore.
- **Cons:** It easily turns into a "Data Swamp". No transactions, easily corrupted, untrusted by business analysts.



3. What is a Lakehouse? (The Best of Both Worlds)

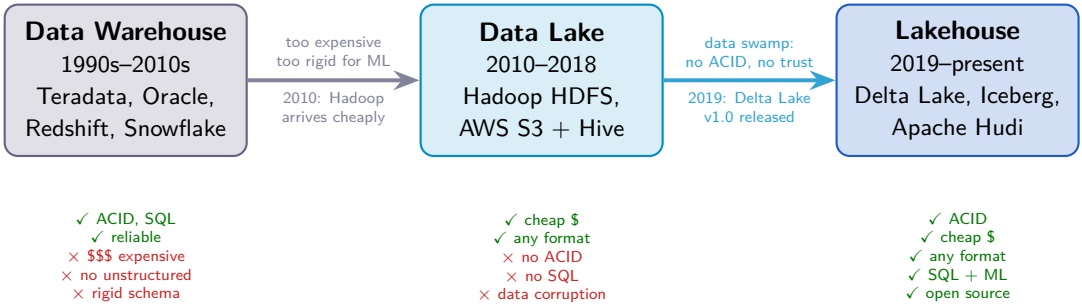


The Analogy: A cheap, massive warehouse, but with a highly intelligent, automated inventory robot managing exactly what comes in and out.

- **The Base:** Keep the cheap, flexible, scalable storage of the Data Lake (MinIO).
- **The Brain:** Add a **transaction layer** (Delta Lake) to enforce schema and ACID rules.
- **The Result:** One unified platform. No more copying data between systems for SQL (Analysts) and Python (Data Scientists).

DO.jpg

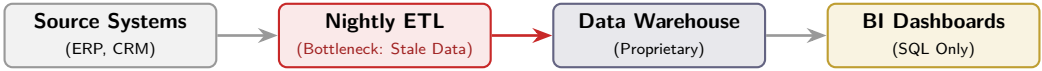
The Data Storage Evolution: A 40-Year Journey



Key insight: The Lakehouse is not a completely new idea. It is a convergence: cheap open file formats (Parquet) on object storage (S3/MinIO), plus a **metadata/transaction layer** on top that adds warehouse-quality reliability. The layer is the innovation; the files are just files.

DO.jpg

Data Warehouses: What They Do Well and Where They Break



Genuine Strengths

- ✓ Full ACID transactions & reliability
- ✓ Optimised columnar SQL (fast)
- ✓ Schema-on-write ensures clean data
- ✓ Mature tooling, enterprise support
- ✓ Perfect for structured/tabular BI

Critical Limitations in 2026

- × **Data types:** No images, video, or JSON
- × **Lock-in:** Proprietary storage formats
- × **Cost:** Exorbitant at petabyte scale
- × **Agility:** Rigid schemas cause downtime
- × **AI/ML:** No raw data for Data Scientists

DO.jpg

Data Lakes: The Cheap Storage that Became a Swamp

The data lake promise (2012–2018):

- Dump *everything* into HDFS/S3 at $23/TB/month
- Schema-on-read: figure out structure at query time
- Train ML models directly on raw Parquet/JSON
- "No ETL needed — just land and query"

What actually happened in practice:

- Nobody deleted old broken files → "data swamp"
- Two Spark jobs wrote to same folder simultaneously → corrupted data
- "We lost 3 hours of data" became a weekly occurrence

The 3 Core Lake Problems

1. No Atomicity (most critical):
Spark job writes 100 files. Fails after 40.
You have a permanently half-written table.
No way to roll back. Data is now corrupt.

2. No Isolation:
Reader runs while writer is mid-write.
Reader sees some old files, some new.
Query returns phantom rows. Metrics are wrong.

3. No Metadata / Versioning:
S3 stores files, not table history.
No way to know: which files are "cur-



Data Lakes: The Cheap Storage that Became a Swamp

The Promise (2012–2018):

- Dump *everything* into S3 at $23/TB/month.
- Schema-on-read: figure it out at query time.
- Train ML models directly on raw files.
- "No ETL needed — just land and query!"

The Reality (The Swamp):

- Unmanaged old files → chaotic "Data Swamp".
- Analysts refuse to trust any numbers.
- "What is the correct version?" — nobody knows.
- Data Scientists spend 60% of their time cleaning.

The 3 Core Technical Failures

1. No Atomicity

The Flaw: A Spark job writes 100 files but crashes after 40.
The Result: You have a permanently half-written table. No way to roll back. Data is corrupt.

2. No Isolation

The Flaw: A reader runs a query while a writer is mid-write.
The Result: The reader sees a mix of old and new files. Dashboards return phantom metrics.

3. No Metadata / History

The Flaw: Object stores (S3/MinIO) just store dumb files.
The Result: No way to know which files belong to the "current" table, or what it looked like yesterday.

.jpg

The Lakehouse: One Architecture, All Workloads

All Workloads on One Platform

BI Dashboards ML Training Real-Time Streaming APIs

← **All in one: no data silos**

Query Engines — Spark / DuckDB / Trino / Presto

Same files, multiple engines. No ETL copy between systems.

← **Same as Data Warehouse but on cheap storage**

Transaction Log Layer — Delta Lake / Apache Iceberg

ACID — Time travel — Schema — Stream + batch

← **The Lakehouse innovation (fills the gap)**

Object Storage — MinIO / AWS S3 / Azure ADLS

Open Parquet — Cheap: $23/TB/mo — Portable: no lock-in

← **What the Data Lake had**

The Lakehouse in one sentence: Parquet files on cheap object storage + a transaction log that adds ACID, versioning, and schema enforcement = **warehouse reliability at lake prices**.
Today you will build this exact architecture locally on your laptop.



The Medallion Architecture: Organizing the Lakehouse

What is the Medallion Architecture?

It is a data design pattern used to logically organize data in a Lakehouse. It uses a series of layers (Bronze $\rightarrow$ Silver $\rightarrow$ Gold) to incrementally and progressively improve the structure and quality of data as it flows through the system.

The core philosophy:

- Start with raw, untrusted data.
- Clean and validate it incrementally.
- Serve highly refined, trusted business metrics at the end.



The Medallion Architecture: Organizing the Lakehouse

What is the Medallion Architecture?

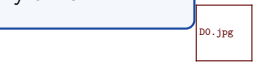
It is a data design pattern used to logically organize data in a Lakehouse. It uses a series of layers (Bronze $\rightarrow$ Silver $\rightarrow$ Gold) to incrementally and progressively improve the structure and quality of data as it flows through the system.

The core philosophy:

- Start with raw, untrusted data.
- Clean and validate it incrementally.
- Serve highly refined, trusted business metrics at the end.

Why is it crucial for Big Data?

1. **Prevents the Data Swamp:** Forces a strict separation between raw data dumps and clean, queryable tables.
2. **Debugging & Lineage:** If a dashboard looks wrong, you can trace the data back through Silver and Bronze to find exactly where the error occurred.
3. **Recomputability:** Because Bronze is immutable and never deleted, you can wipe your Silver/Gold tables and rebuild them from scratch at any time.



Medallion Architecture: Bronze → Silver → Gold



Bronze rules:

- **Never delete** from Bronze
- Append-only; fully immutable
- Full audit trail of raw data
- May have errors — that is OK

Silver rules:

- Exactly-once deduplication
- All types correctly cast
- Joined to dimension tables
- Source of truth for analysts

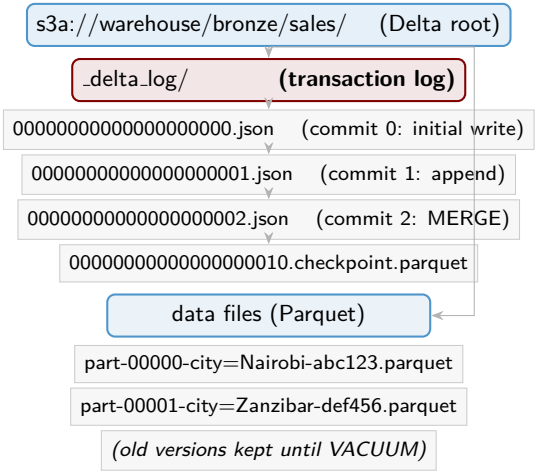
Gold rules:

- Pre-aggregated for performance
- Business-level metrics only
- Feature store inputs for ML
- Directly served in APIs

DO.jpg

Delta Lake Internals: The Transaction Log is Everything

Physical layout on disk (MinIO bucket):



What a commit JSON actually contains

```
// 00000000000000000001.json
{
  "commitInfo": {
    "timestamp": 1710590400000, "operation": "WRITE",
    "operationParameters": {"mode": "Append", "partit
    "numOutputRows": 50000, "numFiles": 5
  },
  "add": {
    "path": "city=Nairobi/part-00001.parquet", "size"
    "stats": {
      "numRecords": 12500, "nullCount": {"amount_usd"
      "minValues": {"amount_usd": 0.5},
      "maxValues": {"amount_usd": 9999.0}
    }
  }
}
```

Key rules:

- Every write = one new `JSON` commit file
- JSON tells Delta which Parquets are "live"
- Old Parquets stay on disk until `VACUUM`
- Checkpoint every 10 commits (faster restart)



ACID: What Each Letter Means in Plain Language

	Property	Plain English	How Delta implements it
A	Atomicity	Write fully succeeds or fully fails. No partial tables, no half-written files.	Write Parquet to temp location, then write JSON commit atomically.
C	Consistency	Every transaction takes the table from one valid state to another. Schema rules respected.	Schema check before every write. Rejected if types are wrong or constraints violated.
I	Isolation	Concurrent readers and writers do not interfere. Readers see a consistent snapshot.	Snapshot isolation: reader picks a log version at start. New commits do not change mid-query.
D	Durability	Once committed, data survives crashes, restarts, and network failures.	Parquet + JSON written to durable object storage (MinIO/S3).

Without ACID (plain Parquet on S3): Writer 1 is halfway through writing 50 files. Writer 2 starts writing. Reader runs now and sees a mix of old and new files. Result: corrupted rows, wrong counts, phantom data. This happened constantly in 2015–2018 data lakes. Delta Lake eliminates this class of bug entirely.



Time Travel: Query Any Past Version of Your Table

Why time travel is critical in production:

- **Audit:** "Show regulator the data as of 31 December 2025"
- **Recovery:** Analyst deleted 10,000 rows by accident — restore in seconds
- **ML reproducibility:** Retrain model on *exact* data version 3 used
- **Debugging:** ETL pipeline produced bad results at 3am. What did the table look like before it ran?
- **A/B testing:** Compare current model against baseline trained on old snapshot

Each write creates version N+1. The `_delta_log` records which Parquet files belong to each version. Reading an old version = Delta loads only the old log entries, which point to the old Parquet files that are still on disk. Old files are preserved until `VACUUM` removes them.



Time Travel: Implementation Examples

Using PySpark DataFrame API:

```
TABLE = "s3a://warehouse/bronze/
    transactions"

# -- Read by version number
    --------------------
df_v0 = spark.read \
    .format("delta") \
    .option("versionAsOf", 0) \
    .load(TABLE)

df_v2 = spark.read \
    .format("delta") \
    .option("versionAsOf", 2) \
    .load(TABLE)

# -- Read by timestamp
    ------------------------
df_jan = spark.read \
    .format("delta") \
    .option("timestampAsOf", "
```

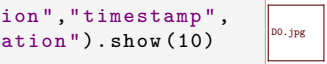
Using Spark SQL API:

```
# -- Using SQL
    --------------------------------

spark.sql(f"""
  SELECT * FROM delta.`{TABLE}`
  VERSION AS OF 2
""").show(5)

spark.sql(f"""
  SELECT * FROM delta.`{TABLE}`
  TIMESTAMP AS OF '2026-01-15'
""").show(5)

# -- See all versions
    -------------------------
spark.sql(f"""
  DESCRIBE HISTORY delta.`{TABLE}`
""").select("version","timestamp",
            "operation").show(10)
```



MERGE / UPSERT: The Most Powerful Delta Operation

Why MERGE exists:

- You have a Delta table of 10 million customers
- Every hour, 50,000 customers update their address
- **Without MERGE:** rewrite entire 10M-row table (≈30 minutes!)
- **With MERGE:** touch only matching rows (≈45 seconds)

MERGE handles three cases in one statement:

WHEN MATCHED → UPDATE
(row exists, has changed)

WHEN MATCHED → DELETE
(row was cancelled/removed)

WHEN NOT MATCHED → INSERT
(brand new row)

Primary use case: CDC (Change Data Capture) — apply a stream of database changes to your Delta Lake table atomically.



MERGE / UPSERT: PySpark Implementation

1. Setup Target & Incoming Data

```
from delta.tables import DeltaTable
from pyspark.sql import functions
    as F

TABLE = "s3a://warehouse/silver/
    customers"

# Existing Delta target table
target = DeltaTable.forPath(spark,
    TABLE)

# Incoming updates (from Kafka /
    CDC feed)
updates = spark.createDataFrame([
    # (id, name, city, status)
    (1001, "Alice Wanjiku", "
        Zanzibar", "active"),
    (1002, "Bob Ochieng", "Nairobi
        ", "active"),
    (1003, "Carol Mwangi", "Mombasa
```

2. Execute Atomic MERGE

```
# -- MERGE: one atomic transaction
    ----------
(target.alias("tgt")
 .merge(
    updates.alias("src"),
    "tgt.id = src.id"           #
        join key
 )
 .whenMatchedDelete(
    condition="src.status = '
        deleted'"
 )
 .whenMatchedUpdate(set={
    "name":         "src.name",
    "city":         "src.city",
    "updated_at": F.
        current_timestamp()
 })
 .whenNotMatchedInsert(values={
    "id":           "src.id",
```



Schema Enforcement vs Schema Evolution

1. Schema Enforcement (ON by default)

- Delta Lake *rejects* any write that does not match the target table's schema.
- Acts as a strict protective firewall to guarantee data quality automatically.

2. Schema Evolution (Opt-in)

- When you deliberately add a new column to a growing schema, use `mergeSchema=true`.
- The schema expands safely; existing rows simply get `NULL` for the new column.

The Golden Rule

Always use enforcement by default. Use evolution **only** when the source genuinely adds a new field (e.g., a new app version or new sensor type). **Never** use it just to bypass a bug in your writer!



Schema Management: PySpark & SQL

Enforcement (Rejects bad writes):

```
# Table schema: id, amount,
    currency
# Trying to write an extra column
bad_df = spark.createDataFrame(
    [(1, 100.0, "USD", "UNKNOWN_COL
        ")],
    ["id", "amount", "currency", "
        extra"])

# This will FAIL and raise an
# AnalysisException: A schema
    mismatch
# was detected...
bad_df.write.format("delta") \
    .mode("append") \
    .save("s3a://warehouse/bronze/
        sales")
```

Evolution (Accepts new columns):

```
# Source sends extra "discount"
    field
new_df = spark.createDataFrame(
    [(2, 200.0, "KES", 0.10)],
    ["id","amount","currency","
        discount"])

# Opt-in to expand the schema
new_df.write.format("delta") \
    .mode("append") \
    .option("mergeSchema", "true")
        \
    .save("s3a://warehouse/bronze/
        sales")

# Or via SQL:
spark.sql("""
  ALTER TABLE delta.`s3a://.../
      sales`
  ADD COLUMNS (discount DOUBLE)
```

D0.jpg

Table Maintenance: OPTIMIZE, Z-ORDER & VACUUM

The Small Files Problem

- Streaming writes 1 file per micro-batch ($\approx$100–500 KB each).
- After 1 week: 10,000 tiny files in S3.
- Each file = 1 separate S3 API call during a query. 10,000 files $\rightarrow$ query takes minutes.
- **OPTIMIZE** compacts them into $\sim$1 GB files (10–50 files instead of 10,000).

Z-ORDER: Co-locate by Column Value

- Reorganizes which rows go into which Parquet file.
- Queries filtering by a Z-ordered column (e.g., `customer_id`) can skip >90% of files.
- Combine with `OPTIMIZE` for maximum read performance.

VACUUM: Cleanup

Deletes old/unreferenced files. **Warning:** After running VACUUM, you CANNOT time-travel older than the retention window. Plan your time travel window accordingly!



Table Maintenance: SQL Implementation

1. OPTIMIZE & Z-ORDER

```
-- Step 1: Compact small files
OPTIMIZE delta.`s3a://warehouse/
    bronze/transactions`;

-- Step 2: Compact AND co-locate by
-- commonly-filtered columns
OPTIMIZE delta.`s3a://warehouse/
    bronze/transactions`
  ZORDER BY (customer_id, txn_date);
```

2. VACUUM (Data Retention)

```
-- Delete unreferenced files
-- Default: keeps 7 days of history
VACUUM delta.`s3a://warehouse/bronze
    /transactions`;

-- Custom retention (min 1 hour for
    testing)
SET spark.databricks.delta.
    retentionDurationCheck.enabled=
    false;
VACUUM delta.`s3a://warehouse/bronze
    /transactions`
  RETAIN 1 HOURS;

-- DRY RUN: see what WOULD be
    deleted
VACUUM delta.`s3a://warehouse/bronze
    /transactions`
  DRY RUN;
```

MinIO: Local S3-Compatible Object Store

Why MinIO?

- Open-source object storage implementing the **AWS S3 API**.
- Your Spark code uses `s3a://` paths — identical to real AWS S3.
- When graduating to production AWS: **zero code change needed**.
- Runs entirely inside your Docker stack — no internet required.

MinIO Concepts

- **Bucket:** Top-level container (like a drive).
- **Object:** Any file stored in a bucket.
- **Prefix:** Folder-like path (no real folders in S3!).
  Example: `bronze/transactions/` is just a name prefix.

Pre-created Buckets in your Stack

- `warehouse` — all Delta Lake tables
- `mlflow-artifacts` — ML model files
- `kafka-connect` — Kafka sink output



MinIO: Connection & CLI Commands

Docker Stack Connection Info

- **API Endpoint:** `http://minio:9000` (Docker) or `localhost:9000` (browser)
- **Console UI:** `http://localhost:9001`
- **Credentials:** `admin` / `bigdata123`
- **S3A Path Format:** `s3a://warehouse/bronze/transactions`
  → bucket: `warehouse`
  → prefix: `bronze/transactions/`

MinIO Client (`mc`) Commands

```
# Inside the minio container, setup alias:
mc alias set local http://minio:9000 \
  admin bigdata123

# List all buckets
mc ls local/

# List objects in warehouse bucket
mc ls local/warehouse/

# Check specific Delta table on disk
mc ls local/warehouse/bronze/transactions/

# View the transaction log files
mc ls local/warehouse/bronze/transactions/_del
```



SparkSession Config for Delta Lake + MinIO

What each config option does:

- `spark.sql.extensions` — loads Delta SQL parser and optimizer.
- `spark.sql.catalog.spark_catalog` — enables `delta.‘path‘` syntax in SQL.
- `fs.s3a.endpoint` — tells S3A where MinIO lives.
- `fs.s3a.path.style.access` — required for MinIO (disables AWS-style subdomains).
- `fs.s3a.impl` — loads the S3A file system connector JAR.
- `adaptive.enabled` — enables Adaptive Query Execution (Lab 5).

Good news:
This full config is already saved in `infrastructure/config/spark-defaults.conf`. JupyterLab auto-loads it at startup. You can also paste the code on the next slide directly into any notebook.



SparkSession: PySpark Implementation

```
from pyspark.sql import SparkSession

spark = (SparkSession.builder
  .appName("Z5008-Lab2-DeltaLake")
  # -- Delta Lake extensions --------------
  .config("spark.sql.extensions",
    "io.delta.sql.DeltaSparkSessionExtension")
  .config("spark.sql.catalog.spark_catalog",
    "org.apache.spark.sql.delta.catalog.DeltaCatalog")

  # -- S3A: connect to MinIO ---------------
  .config("spark.hadoop.fs.s3a.endpoint", "http://minio:9000")
  .config("spark.hadoop.fs.s3a.access.key", "admin")
  .config("spark.hadoop.fs.s3a.secret.key", "bigdata123")
  .config("spark.hadoop.fs.s3a.path.style.access", "true")
  .config("spark.hadoop.fs.s3a.impl", "org.apache.hadoop.fs.s3a.S3AFileSystem")
  .config("spark.hadoop.fs.s3a.connection.ssl.enabled", "false")

  # -- Performance tuning ------------------
  .config("spark.sql.adaptive.enabled", "true")
  .config("spark.sql.shuffle.partitions", "8")
  .getOrCreate()
```



10-Minute Break

Up Next: Hands-On Part 2

- Ensure your Docker stack is running
- Open JupyterLab in your browser
- We will build and query our first Delta tables!



Hands-On: Verify Infrastructure & Access Jupyter

1. Verify Docker Services

```
# In a terminal (or JupyterLab terminal tab)
docker compose -f infrastructure/docker-compose.yml ps
```

Expected Running Services:

- **Spark Cluster:** `spark-master` (8080), `workers` (8081, 8082)
- **Storage:** `minio` (9001)
- **Messaging:** `kafka` (9092)
- **Workspace:** `jupyter` (8888)
- **Monitoring/ML:** `mlflow` (5000), `airflow` (8090), `grafana` (3000)

Troubleshooting: If a container is stopped, bring it back up:

```
docker compose -f infrastructure/docker-compose.yml up -d <service-name>
```

2. Open the Workspace

- Go to `http://localhost:8888` (Token: `bigdata`)
- Open: `work/lab02_delta_lake.ipynb`



Hands-On: Generate Synthetic Retail Data

Generate 10,000 East African retail transactions:

```
import pandas as pd
import random
from pyspark.sql import functions as F

random.seed(42)
n = 10000

# 1. Create a Pandas DataFrame (easier for complex synthetic generation)
pdf = pd.DataFrame({
  "txn_id":      range(1, n+1),
  "customer_id": [f"C{i%500+1:04d}" for i in range(n)],
  "merchant":    ["ShopRite", "Nakumatt", "Jumia", "M-Pesa", "KCB Bank"] * 2000,
  "city":        ["Nairobi", "Dar es Salaam", "Zanzibar", "Kampala", "Mombasa"] * 20
  "category":    ["groceries", "electronics", "finance", "transport", "dining"] * 20
  "amount_usd":  [round(abs(random.gauss(150, 120)), 2) for _ in range(n)],
  "currency":    ["KES", "TZS", "TZS", "KES", "KES"] * 2000,
  "txn_date":    pd.date_range("2026-01-01", periods=n, freq="h"),
  "is_fraud":    [False]*9800 + [True]*200,
})

# 2. Convert to PySpark DataFrame
```



Hands-On 2A: Write to Bronze & Create View

```
# Cell 2: Write to Bronze (partitioned by city for fast queries)
TABLE = "s3a://warehouse/bronze/transactions"

(df.write
   .format("delta")
   .mode("overwrite")
   .partitionBy("city")
   .save(TABLE))
print("Written to Delta Lake.")

# Browser check: Go to http://localhost:9001
# Look for: _delta_log/ and city=Nairobi/, city=Zanzibar/, etc.

# Cell 3: Read back and create SQL view
df_delta = spark.read.format("delta").load(TABLE)
print(f"Rows: {df_delta.count()}")
df_delta.createOrReplaceTempView("transactions")
```



Hands-On 2B: SQL Queries on Delta Table

```
# Cell 4: Total spend and fraud rate by city
spark.sql("""
    SELECT city, COUNT(*) AS num_txns,
           ROUND(SUM(amount_usd), 2) AS total_usd,
           ROUND(AVG(amount_usd), 2) AS avg_usd,
           SUM(CAST(is_fraud AS INT)) AS fraud_count,
           ROUND(100.0 * SUM(CAST(is_fraud AS INT)) / COUNT(*), 2) AS
               fraud_pct
    FROM transactions
    GROUP BY city
    ORDER BY total_usd DESC
""").show()

# Cell 5: Top merchants by fraud rate (where amount > $50)
spark.sql("""
    SELECT merchant, COUNT(*) AS total,
           SUM(CAST(is_fraud AS INT)) AS frauds,
           ROUND(AVG(CAST(is_fraud AS INT))*100, 2) AS fraud_pct,
           ROUND(AVG(amount_usd), 2) AS avg_amount
    FROM transactions
    WHERE amount_usd > 50
    GROUP BY merchant
```

DO.jpg

Hands-On 3A: Append Data & Check History

```
# Cell 6: Append 500 more rows (creates version 1)
pdf_batch2 = pd.DataFrame({
  "txn_id":      range(10001, 10501),
  "customer_id": [f"C{i%500+1:04d}" for i in range(500)],
  "merchant":    ["Equity Bank"] * 500,
  "city":        ["Zanzibar"] * 500,
  "category":    ["finance"] * 500,
  "amount_usd":  [500.0] * 500,
  "currency":    ["TZS"] * 500,
  "txn_date":    pd.date_range("2026-03-16", periods=500, freq="min"),
  "is_fraud":    [False] * 500,
})
(spark.createDataFrame(pdf_batch2)
 .write.format("delta").mode("append").save(TABLE))
print("Appended 500 rows => version 1")

# Cell 7: Check the transaction log history
spark.sql(f"DESCRIBE HISTORY delta.`{TABLE}`") \
     .select("version","timestamp","operation",
             "operationMetrics").show(5, truncate=False)
```



Hands-On 3B: Time Travel & Restore

```python
# Cell 8: Time travel - read old version
v0 = spark.read.format("delta").option("versionAsOf", 0).load(TABLE)
v1 = spark.read.format("delta").option("versionAsOf", 1).load(TABLE)
print(f"Version 0: {v0.count()} rows")    # 10,000
print(f"Version 1: {v1.count()} rows")    # 10,500

# Cell 9: Compare Zanzibar rows between versions
print("Zanzibar in v0:", v0.filter("city='Zanzibar'").count())   # 2000
print("Zanzibar in v1:", v1.filter("city='Zanzibar'").count())   # 2500

# Cell 10: Restore to version 0 (undo the append)
spark.sql(f"RESTORE TABLE delta.`{TABLE}` TO VERSION AS OF 0")
current = spark.read.format("delta").load(TABLE)
print(f"After restore: {current.count()} rows")  # back to 10,000
```



Hands-On 4A: Build a Silver Table

```
# Cell 11: Build a Silver customers table from the Bronze transactions
CUST = "s3a://warehouse/silver/customers"

(spark.sql(f"""
    SELECT customer_id,
           FIRST(city)              AS city,
           COUNT(*)                 AS txn_count,
           ROUND(SUM(amount_usd),2) AS lifetime_value_usd,
           MAX(txn_date)            AS last_txn_date
    FROM delta.`{TABLE}`
    GROUP BY customer_id
""").write.format("delta").mode("overwrite").save(CUST))

print(f"Silver customers: {spark.read.format('delta').load(CUST).count()}")
```



Hands-On 4B: MERGE (CDC Data) & OPTIMIZE

```
# Cell 12: Simulate a CDC batch (updates + deletes + inserts)
from delta.tables import DeltaTable

updates = spark.createDataFrame([
    ("C0001", "Arusha",  9999.99, "active"),     # UPDATE
    ("C0003", None,      0.0,     "cancelled"), # DELETE
    ("C0999", "Zanzibar",0.0,     "new"),        # INSERT
], ["customer_id","city","lifetime_value_usd","status"])

target = DeltaTable.forPath(spark, CUST)
(target.alias("t").merge(updates.alias("u"), "t.customer_id = u.customer_id"
    )
 .whenMatchedDelete(condition="u.status = 'cancelled'")
 .whenMatchedUpdate(set={"city":"u.city", "lifetime_value_usd":"u.
     lifetime_value_usd"})
 .whenNotMatchedInsert(values={"customer_id":"u.customer_id", "city":"u.city
     ",
                               "txn_count":"0",
                               "lifetime_value_usd":"u.lifetime_value_usd",
                               "last_txn_date":"current_timestamp()"})
 .execute())
```



Hands-On Step 5: Structured Streaming to Delta

Delta Lake acts as both a batch table and a streaming sink/source.

```
# Cell 14: Simulate a live data stream (e.g., from Kafka or webhooks)
# We use the built-in 'rate' source to generate 5 rows per second
live_stream = (spark.readStream
    .format("rate")
    .option("rowsPerSecond", 5)
    .load())

# Cell 15: Write the continuous stream directly into Delta Lake
STREAM_PATH = "s3a://warehouse/bronze/live_metrics"
CHECKPOINT  = "s3a://warehouse/checkpoints/live_metrics"

query = (live_stream.writeStream
    .format("delta")
    .outputMode("append")
    .option("checkpointLocation", CHECKPOINT)
    .start(STREAM_PATH))

print(f"Streaming query ID: {query.id}")
# Run this in a new cell to see the count increase live!
# spark.read.format("delta").load(STREAM_PATH).count()
```



Lab 2 Summary: What You Learned and Built Today

Concepts mastered:

- **Architecture:** DW → Lake → Lakehouse, Medallion pipeline design, and MinIO (local S3 storage).
- **Delta Internals:** How the `_delta_log` provides free ACID guarantees, plus schema enforcement & evolution.
- **Data Operations:**
  - Time travel by version/timestamp & `RESTORE`
  - `MERGE`/`UPSERT` to apply CDC changes atomically
  - `OPTIMIZE` + `Z-ORDER` to fix small files

Tables you created today

Bronze:
`s3a://warehouse/bronze/transactions`
- 10,000 rows, partitioned by city
- 3 logged versions (write/append/restore)

Silver:
`s3a://warehouse/silver/customers`
- 500 customers aggregated
- MERGE applied: 2 updated, 1 deleted, 1 inserted

Next Week (Lab 3, 23 March):
Delta Advanced (Change Data Feed, streaming writes) + Apache Iceberg vs Delta.