Delta Lake Advanced

Time Travel, MERGE, OPTIMIZE & VACUUM

Databricks 14-Days AI Challenge

Day 5 - Advanced Delta Lake Features

January 14, 2026

Agenda

- **Delta Lake Architecture**
  - ▷ Transaction Log & Parquet Files
  - ▷ ACID Properties
- **Time Travel**
  - ▷ Version History Queries
  - ▷ Restore Operations
- **MERGE Operations**
  - ▷ Upsert Patterns
  - ▷ SCD Type 2
- **OPTIMIZE & ZORDER**
  - ▷ File Compaction
  - ▷ Data Skipping
- **VACUUM for Cleanup**
  - ▷ Stale File Removal
  - ▷ Retention Configuration
- **Best Practices**
  - ▷ Performance Tips
  - ▷ Common Pitfalls

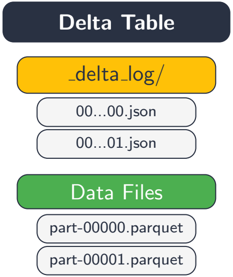
Delta Lake Architecture Overview

What is Delta Lake?

An open-source storage layer that brings **ACID transactions** to Apache Spark and big data workloads.

Core Components:

- ▷ **Transaction Log (_delta_log)**
  - ▷ JSON files recording every change
  - ▷ Enables ACID & time travel
- ▷ **Parquet Data Files**
  - ▷ Efficient columnar storage
  - ▷ Optimized compression



ACID Properties in Delta Lake

Atomicity

Each transaction completely succeeds or completely fails. Partial changes are never visible.

Consistency

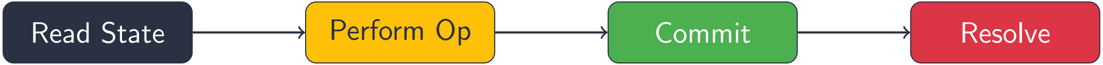
Table always moves from one valid state to another. Schema enforcement ensures data integrity.

Isolation

Concurrent transactions don't interfere. Readers see consistent snapshots while writers make changes.

Durability

Once committed, changes are permanent and survive system failures.



Read State → Perform Op → Commit → Resolve

Time Travel - Query Historical Data

Three Methods to Access Historical Versions:

Method 1: Version Number

```
SELECT * FROM my_table
VERSION AS OF 5;

-- PySpark
df = spark.read.format("delta")
     .option("versionAsOf", 5)
     .load("/path/to/table")
```

Method 2: Timestamp

```
SELECT * FROM my_table
TIMESTAMP AS OF ’2024-01-15
     10:30:00’;
```

Method 3: Version Shorthand

```
-- Version shorthand syntax
SELECT * FROM my_table VERSION AS OF
      5;

-- Timestamp shorthand
SELECT * FROM my_table
TIMESTAMP AS OF ’20240115103000’;
```

View Table History:

```
DESCRIBE HISTORY my_table;
DESCRIBE HISTORY my_table LIMIT 10;
```

Time Travel - Restore Operations

Restoring Previous Versions:

```
-- Restore to a specific version
RESTORE TABLE my_table TO VERSION AS OF 10;

-- Restore to a specific timestamp
RESTORE TABLE my_table TO TIMESTAMP AS OF '2024-01-15 10:30:00';
```

Key Use Cases:

- ▷ Auditing & Compliance
- ▷ Data Recovery
- ▷ Reproducibility (ML)
- ▷ Debugging Issues
- ▷ Rollback Pipelines

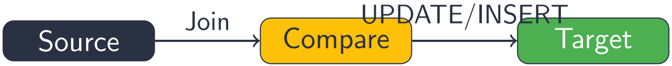
Retention Configuration:

```
-- Log retention (default 30 days)
ALTER TABLE my_table SET
TBLPROPERTIES (
  'delta.logRetentionDuration'
    = '7 days');

-- File retention (default 7 days)
ALTER TABLE my_table SET
TBLPROPERTIES (
  'delta.
      deletedFileRetentionDuration'
```

MERGE Operations (Upserts)

Combine INSERT, UPDATE, DELETE in a single atomic transaction

```sql
MERGE INTO target_table AS target
USING source_table AS source
ON target.id = source.id
WHEN MATCHED THEN
    UPDATE SET
        target.name = source.name,
        target.value = source.value,
        target.updated_at = current_timestamp()
WHEN NOT MATCHED THEN
    INSERT (id, name, value, created_at)
    VALUES (source.id, source.name, source.value, current_timestamp());
```



Source

Join

Compare

UPDATE/INSERT

Target

MERGE Clause Types

WHEN MATCHED:

Update or delete matching rows

```
-- Update matching rows
WHEN MATCHED THEN
    UPDATE SET target.col1 = source.
        col1

-- With condition
WHEN MATCHED AND source.is_deleted =
     true
THEN DELETE
WHEN MATCHED AND source.is_deleted =
     false
THEN UPDATE SET target.col1 = source
    .col1
```

WHEN NOT MATCHED:

Insert new rows

```
-- Insert new rows
WHEN NOT MATCHED THEN
    INSERT (id, name, value)
    VALUES (source.id, source.name,
            source.value)
```

WHEN NOT MATCHED BY SOURCE:

Handle orphaned records

```
-- Delete orphaned records
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
```

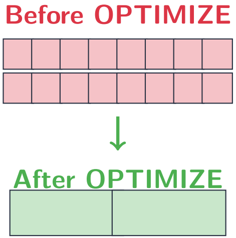
MERGE with PySpark

```
from delta.tables import DeltaTable
from pyspark.sql.functions import current_timestamp

# Load the target Delta table
deltaTable = DeltaTable.forPath(spark, "/path/to/customers")

# Prepare source DataFrame
updates_df = spark.read.parquet("/path/to/updates")

# Perform MERGE
deltaTable.alias("target").merge(
    updates_df.alias("source"),
    "target.customer_id = source.customer_id"
).whenMatchedUpdate(
    condition="source.operation = ’UPDATE’",
    set={"name": "source.name", "email": "source.email",
         "modified_date": current_timestamp()}
).whenMatchedDelete(
    condition="source.operation = ’DELETE’"
).whenNotMatchedInsert(
    condition="source.operation = ’INSERT’",
    values={"customer_id": "source.customer_id", "name": "source.name",
            "created_date": current_timestamp()}
```

The Small File Problem

Why Small Files are Problematic:

- ▷ **Read Overhead**
  - ▷ Each file requires metadata operations
  - ▷ 1000 Ã— 1MB slower than 10 Ã— 100MB
- ▷ **Memory Pressure**
  - ▷ Spark tracks all files in memory
- ▷ **Cloud API Costs**
  - ▷ More LIST and GET operations
  - ▷ Object stores charge per API call



Before OPTIMIZE

↓

After OPTIMIZE

OPTIMIZE Operation

Compacts small files into larger ones (target: 1GB)

SQL Syntax:

```sql
-- Basic OPTIMIZE
OPTIMIZE my_table;

-- Specific partitions
OPTIMIZE my_table
WHERE date >= ’2024-01-01’;
```

PySpark:

```python
from delta.tables import DeltaTable
deltaTable = DeltaTable.forPath(
    spark, "/path/to/table")
deltaTable.optimize()
    .executeCompaction()
```

How OPTIMIZE Works:

1. Identifies small files
2. Groups files (respects partitions)
3. Reads and rewrites data
4. Updates transaction log atomically
5. Original files marked for deletion

Note: Original files removed by VACUUM

ZORDER Optimization

Co-locates related data in the same files for better data skipping

```sql
-- ZORDER on specific columns
OPTIMIZE my_table
ZORDER BY (customer_id);

-- Multiple columns
OPTIMIZE my_table
ZORDER BY (region, customer_id);

-- Combined with partition filter
OPTIMIZE my_table
WHERE date >= ’2024-01-01’
ZORDER BY (customer_id, product_id);
```

Good ZORDER Candidates:

- ✓ High-cardinality filter columns
- ✓ Columns used in JOINs
- ✓ WHERE clause columns

Poor Candidates:

- ✗ Low-cardinality (use partition)
- ✗ Rarely queried columns
- ✗ Already partitioned columns

Partitioning vs ZORDER

Characteristic	Use Partitioning	Use ZORDER
Low cardinality (¡1000)	✓ Yes	✗ No
High cardinality (¿1000)	✗ No	✓ Yes
Used in every query	✓ Yes	Less important
Used in some queries	✗ No	✓ Yes
Equality predicates only	Yes (if low card)	Yes (if high card)
Range predicates	✗ No	✓ Yes

Column Order Matters: First column has strongest locality

```
-- If most queries filter by customer_id first:
```

Auto Optimize Features

Auto Compaction:
Automatically runs OPTIMIZE after writes

```
-- Enable at table level
ALTER TABLE my_table SET
    TBLPROPERTIES (
  ’delta.autoOptimize.autoCompact’
    = ’true’);

-- Enable for all tables in session
SET spark.databricks.delta
  .autoCompact.enabled = true;
```

Optimized Writes:
Coalesces small files during writes

```
ALTER TABLE my_table SET
    TBLPROPERTIES (
  ’delta.autoOptimize.optimizeWrite’
    = ’true’);
```

Feature	When to Use
autoCompact	Streaming
optimizeWrite	Frequent writes
Manual	Batch maintenance

VACUUM for Cleanup

Removes data files no longer referenced by the transaction log

Why Files Become Stale:

1. New file written with updates
2. Old file marked as “removed”
3. Old file still exists (for time travel)
4. **VACUUM cleans them up**

SQL Syntax:

```sql
-- Default 7-day retention
VACUUM my_table;

-- Custom retention
VACUUM my_table RETAIN 168 HOURS;

-- Dry run first!
VACUUM my_table DRY RUN;
```

PySpark:

```python
from delta.tables import DeltaTable

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

# Default retention
deltaTable.vacuum()

# Custom retention (hours)
deltaTable.vacuum(168)
```

Warning: VACUUM affects time travel!

VACUUM Retention & Safety

Recommended Retention Periods:

Scenario	Retention	Reasoning
Production tables	7-30 days	Balance cost & recovery
Development tables	1-7 days	Lower cost, less recovery
Audit-required tables	30-365 days	Compliance requirements
High-frequency updates	7 days min	Protect long queries

Align VACUUM with Time Travel:

```
-- If you need 30 days of time travel
ALTER TABLE my_table SET TBLPROPERTIES (
  'delta.logRetentionDuration' = '30 days',
  'delta.deletedFileRetentionDuration' = '30 days');

-- Then VACUUM with matching retention
VACUUM my_table RETAIN 720 HOURS; -- 30 days
```

OPTIMIZE vs VACUUM Comparison

Aspect	OPTIMIZE	VACUUM
Purpose	Improve query performance	Reduce storage usage
Creates new files	✓ Yes	✗ No
Deletes files	✗ No (marks stale)	✓ Yes
Affects time travel	✗ No	✓ Yes
Storage impact	Temporary increase	Decrease
When to run	After many writes	After OPTIMIZE
Frequency	Daily or weekly	Weekly or monthly

Typical Workflow: Write Data â†’ OPTIMIZE â†’ VACUUM

Best Practices & Performance Tips

Table Design:

- ▷ Partition by low-cardinality columns
- ▷ Aim for 1GB+ partitions
- ▷ Avoid ¿10,000 partitions
- ▷ ZORDER high-cardinality columns

MERGE Optimization:

- ▷ Pre-filter source data
- ▷ Include partition columns in join
- ▷ Batch small merges

OPTIMIZE Schedule:

- ▷ Streaming: Every 1-4 hours
- ▷ Micro-batch: Daily
- ▷ Batch: After each load

VACUUM Guidelines:

- ▷ Never ¡7 days without understanding
- ▷ Always DRY RUN first
- ▷ Run during low-traffic periods
- ▷ Schedule after OPTIMIZE

Common Pitfalls to Avoid

Pitfall	Problem	Solution
Over-partitioning	Too many small files	Fewer partitions, use ZORDER
ZORDER on partition cols	Redundant, no benefit	ZORDER non-partition columns
Aggressive VACUUM	Lose time travel	Match retention to needs
No OPTIMIZE schedule	Small file problem	Automate with Auto Optimize
MERGE without filters	Full table rewrite	Pre-filter, partition pruning
Ignoring file stats	Poor data skipping	ZORDER filtered columns

Quick Reference Card

Time Travel Commands:

```
SELECT * FROM t VERSION AS OF 5;
SELECT * FROM t TIMESTAMP AS OF
  ’2024-01-15’;
DESCRIBE HISTORY t;
RESTORE TABLE t TO VERSION AS OF 5;
```

MERGE Pattern:

```
MERGE INTO target USING source
ON condition
WHEN MATCHED THEN UPDATE SET ...
WHEN NOT MATCHED THEN INSERT ...
```

OPTIMIZE & VACUUM:

```
OPTIMIZE t;
OPTIMIZE t ZORDER BY (col1, col2);
OPTIMIZE t WHERE partition_col = ’v
    ’;
VACUUM t;
VACUUM t RETAIN 168 HOURS;
VACUUM t DRY RUN;
```

Key Properties:

```
’delta.logRetentionDuration’
’delta.deletedFileRetentionDuration’
’delta.autoOptimize.optimizeWrite’
’delta.autoOptimize.autoCompact’
```

Thank You!

Questions?

linkedin.com/in/yashkavaiya

easy-ai-labs.lovable.app

Gen AI Guru