02 Delta Lake

Delta Lake

> Read the original research paper for better understanding: https://www.databricks.com/wp-content/uploads/2020/08/p975-armbrust.pdf

Introduction to Delta Lake

Delta Lake is an **open-source storage layer** that brings reliability to data lakes by enabling **ACID transactions**, **scalable metadata handling**, and unified **batch and streaming** data processing. It transforms traditional data lakes into robust **Lakehouse architectures**, combining the scalability of data lakes with the reliability of data warehouses.

- **Origin:** Developed by Databricks in 2016 and open-sourced in 2019 under the Linux Foundation. It's not controlled by any single company.
- **Core Purpose**: Addresses data lake challenges like lack of ACID compliance, schema enforcement, and time travel, without requiring data movement.
- **Key Benefits**:
  - **ACID Transactions**: Ensures atomicity, consistency, isolation, and durability for reliable updates.
  - **Schema Enforcement & Evolution:** Validates data on write and allows non-breaking schema changes.
  - **Time Travel**: Query historical data versions for audits, rollbacks, or ML reproducibility.
  - **Unified Processing**: Handles batch, streaming, and interactive queries seamlessly.
  - **Scalability**: Manages petabyte-scale tables with billions of files.

Latest Version (as of September 2025): Delta Lake 4.0.0, released on June 6, 2025, built on Apache Spark 4.0. This is the largest release to date, focusing on performance, reliability, and ease of use.

Open Table Formats and Lakehouse Capabilities

Before diving into Delta Lake, understand **open table formats** (OTFs): These add a **logical/metadata layer** on top of data lakes (e.g., stored in Parquet files on S3, ADLS, GCS) to enable Lakehouse features like SQL querying, transactions, and governance.

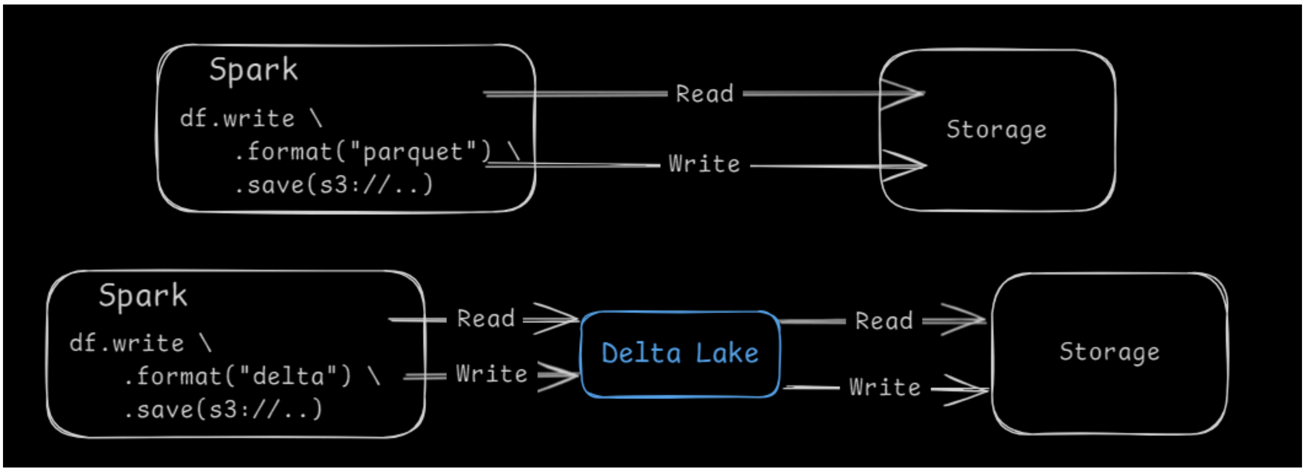
- **Why OTFs?**: Raw data lakes are cheap and scalable but lack performance, reliability, and SQL-like features. OTFs create a "virtual table" (schema + metadata) without moving data, allowing analysts to query as if it were a warehouse.
- **Popular OTFs**:
  - **Delta Lake**: Most mature for Spark-centric environments; backbone of Databricks and Microsoft Fabric.
  - **Apache Iceberg**: Strong in multi-engine support (e.g., Snowflake, AWS Athena); excels in schema evolution and large-scale analytics.
  - **Apache Hudi:** Optimized for real-time streaming, upserts, and CDC (change data capture).

Delta Lake's Position: More mature than alternatives in Spark ecosystems, with tight integration. As of 2025, all three have ~80-90% feature parity, but choice depends on workload (e.g., Delta for unified batch/streaming).

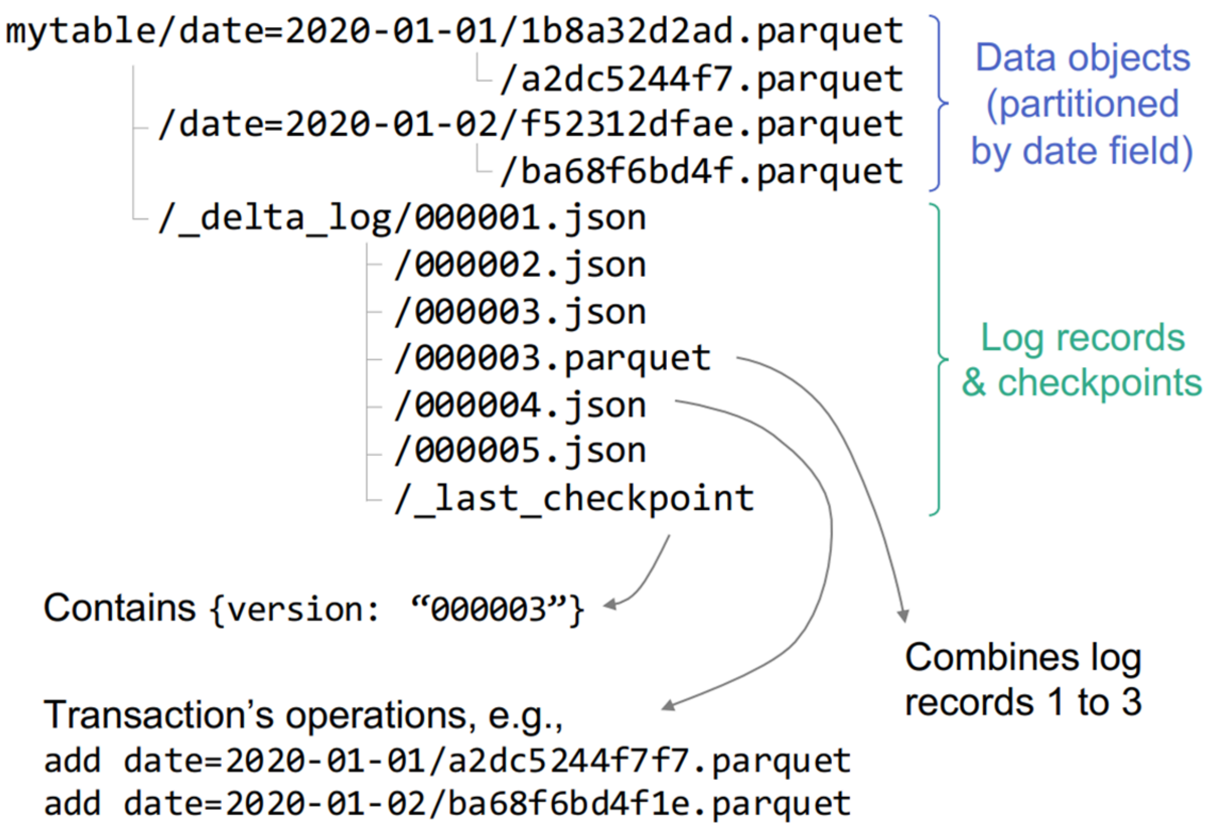
Comparison Table: Delta Lake vs. Iceberg vs. Hudi (2025)

Feature	Delta Lake	Apache Iceberg	Apache Hudi
ACID Transactions	Full support via transaction log	Full support via snapshots	Full via timeline-based commits
Time Travel	Yes (version/timestamp queries)	Yes (snapshot isolation)	Yes (timeline queries)
Schema Evolution	Enforcement + evolution (e.g., add/drop columns)	Advanced (partition evolution, type widening)	Good for upserts, but less flexible
Batch/Streaming	Unified (Structured Streaming)	Batch-focused; streaming via engines	Streaming-optimized (MoR for real-time)
Performance Opts	Z-Ordering, Liquid Clustering, Auto-Compaction	Hidden partitioning, Vectorized I/O	Indexing for upserts, MoR/COW modes
Multi-Engine Support	Spark, Flink, Trino, Hive, Snowflake; UniForm for Iceberg/Hudi	Broad (Spark, Trino, Flink, Snowflake, Athena)	Spark, Flink, Hive; good for CDC
Ecosystem	Databricks, Microsoft Fabric (default)	AWS, Snowflake, Google BigQuery	Uber-inspired; streaming-heavy
Maturity/Adoption	High (60%+ Fortune 500); Spark leader	Rapid growth (Netflix, AWS)	Strong in real-time (fintech, IoT)
Best For	Spark/Databricks users, mixed workloads	Multi-vendor lakehouses, analytics	Real-time ingestion, frequent updates

Interoperability Note: Delta Lake 4.0's **UniForm** allows Delta tables to be read by Iceberg/Hudi clients without data copying, reducing vendor lock-in.



How Delta Lake Works



Delta Lake is **not a file format** (no `.delta` files)—it's a **protocol** built on **Parquet files** (columnar, compressed for analytics) + a **transactional log** (`_delta_log` folder).

- **Structure:**
  - **Parquet Files:** Store actual data (efficient for big data, like CSV/JSON but optimized).
  - **_delta_log Folder:** JSON files for transactions + Parquet checkpoints for fast reads. Uses **MVCC (Multi-Version Concurrency Control)** for optimistic concurrency.
    - Tracks: Metadata, file adds/deletes, schema changes, operations (insert/update/delete).

    - Enables: ACID via atomic commits; prevents corruption in concurrent writes.
- **Conversion Process**:
    - Upgrade existing Parquet to Delta: Add the log folder (e.g., via `CONVERT TO DELTA` in Spark).
    - Example: Day 1 sales Parquet → Enable Delta → Log records metadata. Day 2 adds → Log appends changes.
- **Lakehouse Enablement:** The log turns raw lake data into a "managed table" with SQL features. Feels like a real database—no performance hit for users.

Medallion Architecture: Often used with Delta (Bronze: raw → Silver: cleaned → Gold: curated) for data quality in Databricks/Fabric.

Key Features

- **ACID Transactions:** MVCC ensures serializability; supports **MERGE** for upserts (e.g., CDC).
- **Time Travel**: Query past versions: `SELECT * FROM table VERSION AS OF 5 or TIMESTAMP AS OF '2025-01-01'`. Great for audits/rollbacks.
- **Optimizations** (via `OPTIMIZE` command):
    - **Auto-Compaction/Bin-Packing**: Merges small files (ideal size: 100-300 MB).
    - **Z-Ordering**: Clusters data by columns for faster queries (up to 70% speedup with Photon engine).
    - **Liquid Clustering** (Delta 3.2+): Adaptive, replaces partitioning; auto-tunes for dynamic data (no rewrites for key changes).
    - **Data Skipping/Caching:** Uses metadata stats to skip irrelevant files.
- **Schema Features:** Enforce on write; evolve (add columns, widen types like INT to LONG without rewrite in 4.0).
- **Security/Compliance**: Row-level security (RLS), PII masking; supports GDPR/HIPAA.
- **Streaming**: Exactly-once semantics with Spark Structured Streaming.
- **Delta 4.0 Highlights** (June 2025):
    - **Type Widening**: Evolve types (e.g., INT to LONG) without data rewrite.
    - **Row Tracking Backfill:** Enable on existing tables for row-level lineage (e.g., `_metadata.row_id`).
    - **Variant Data Type**: Flexible ingestion without schema; faster with Spark Variant encoding.
    - **Delta Connect**: Supports Spark Connect for client-server (remote Spark from anywhere).
    - **Zero-Copy Convert:** From Iceberg tables (Spark 3.5+).
    - **AI-Driven Opts**: Auto-tune Z-Orders based on queries (preview).

UniForm (from Delta 3.0, enhanced in 4.0): Generates Iceberg/Hudi metadata asynchronously. Enable: `ALTER TABLE table SET TBLPROPERTIES ('delta.uniform.iceberg.enabled' = 'true')`. Read Delta as Iceberg/Hudi—no copies.

Ecosystem and Adoption

- **Platforms**: Default in **Databricks** (Photon engine for 70% faster queries) and **Microsoft Fabric** (OneLake shortcuts for external querying; Direct Lake Mode in Power BI).
- **Engines**: Spark (native), Flink, Trino, Hive, PrestoDB, Snowflake, BigQuery, Athena, Redshift.
- **APIs**: Scala, Java, Python, Rust (delta-rs 1.0+ for community Rust impl).

- **Adoption:** 10M+ monthly downloads; used by 60%+ Fortune 500. Case: 40% cost reduction, 70% faster queries in Azure Databricks audits.
- **Community:** 190+ developers from 70+ orgs; governed by Linux Foundation. Backward compatible (newer reads older tables).

Use Cases:

- **Analytics/BI:** Medallion pipelines in Fabric/Databricks.
- **Real-Time:** Streaming with exactly-once ingestion.
- **ML:** Feature stores with time travel for reproducible experiments.
- **Compliance:** Time travel for audits (e.g., GDPR).

Practical Example (PySpark in Databricks/Spark)

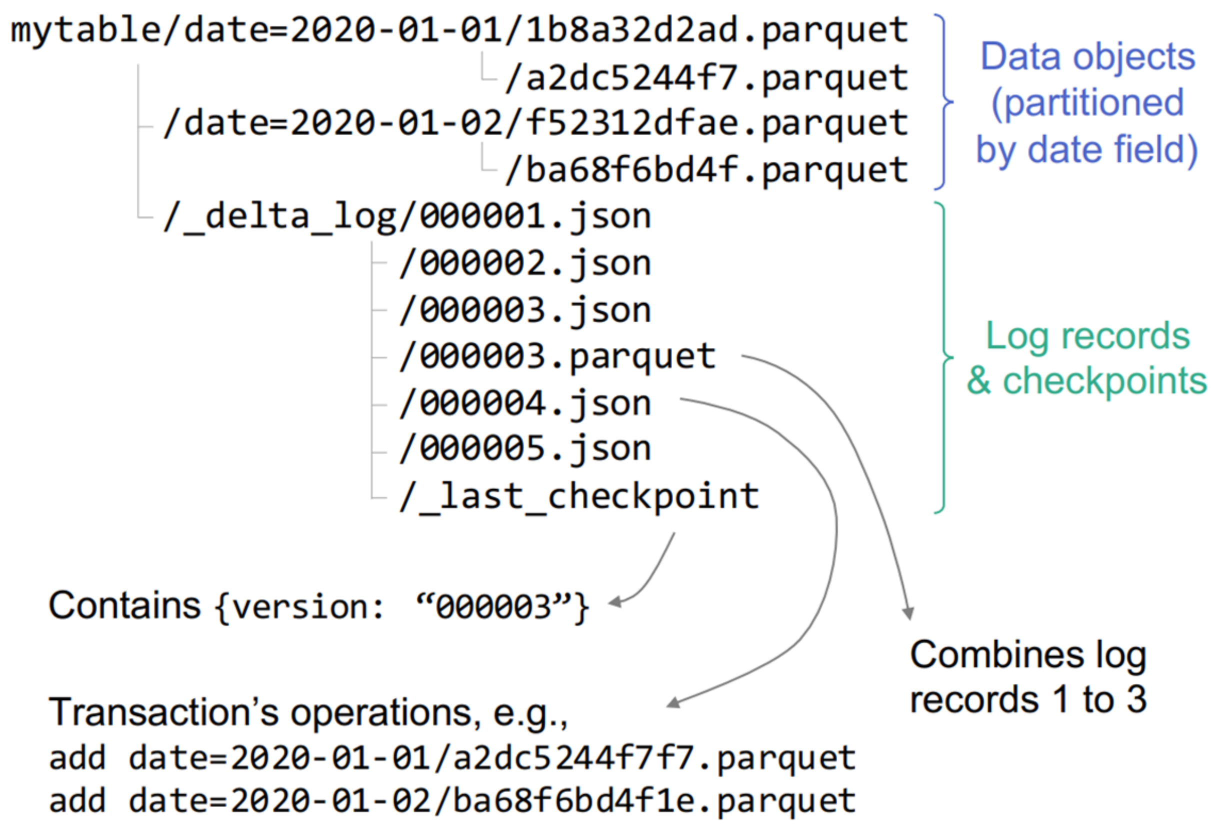
```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("DeltaExample").getOrCreate()

# Create Delta table from Parquet
data = spark.read.parquet("path/to/parquet")
data.write.format("delta").saveAsTable("my_delta_table")

# Time travel query
spark.sql("SELECT * FROM my_delta_table VERSION AS OF 1").show()

# Optimize
spark.sql("OPTIMIZE my_delta_table ZORDER BY (column_name)")

# Enable UniForm (Iceberg compat)
spark.sql("ALTER TABLE my_delta_table SET TBLPROPERTIES
('delta.uniform.iceberg.enabled' = 'true')")
```

Conclusion

Delta Lake is the **backbone of modern Lakehouses**, especially in Spark/Databricks ecosystems. With 4.0's UniForm and performance boosts, it's more interoperable than ever. For alternatives, evaluate based on engines (Iceberg for multi-vendor) or streaming needs (Hudi). Experiment in Databricks Community Edition for hands-on.

Further Reading:

- Official Docs
- GitHub Releases



File Structure

Data Files (Parquet)

- `mytable/date=2020-01-01/` and `date=2020-01-02/` folders contain the actual data
- Files like `1b8a32d2ad.parquet`, `a2dc5244f7.parquet` are the data files
- This shows **partitioning by date field** - data is organized by date for query optimization

Delta Log Folder (`_delta_log/`)

- Contains transaction history as JSON files: `000001.json`, `000002.json`, etc.
- `000003.parquet` - After many transactions, log entries get compacted into parquet format for efficiency
- `_last_checkpoint` - Points to the latest checkpoint for faster table state reconstruction

How Transaction Logging Works

Version Tracking

- Each JSON file represents a transaction/version
- `000003` means this is version 3 of the table
- The `_last_checkpoint` file contains `{"version": "000003"}` indicating the current state

Transaction Operations The diagram shows example operations stored in the log:

```
add date=2020-01-01/a2dc5244f7f7.parquet
add date=2020-01-02/ba68f6bd4f1e.parquet
```

Key Concepts Illustrated

Checkpoint Mechanism

- "Combines log records 1 to 3" means transactions 1, 2, and 3 have been consolidated
- This prevents the log from growing infinitely
- Makes table state reconstruction faster

ACID Properties

- Every data file addition/removal is logged atomically
- You can reconstruct the exact table state at any version
- Enables time travel: `SELECT * FROM mytable VERSION AS OF 2`

Why This Matters

This structure enables:

- **Concurrent reads/writes** without conflicts
- **Time travel queries** to any previous version
- **Fast metadata operations** (schema changes, partition info)
- **Data integrity** through complete audit trail

The genius is that while your data lives in simple parquet files, the `_delta_log` provides database-like ACID guarantees and versioning on top of cheap object storage.