LUMENDATA

Data Sheet

A Quick Technical Guide to Delta Lake

Key Features like ACID Transactions & Time Travel in Databricks Explained

5201 GREAT AMERICAN PARKWAY, SUITE 320
SANTA CLARA, CA 95054
Tel: (855) 695-8636
E-mail: info@lumendata.com
Website: www.lumendata.com

Maintaining data integrity and reliability while enabling complex operations doesn't have to be a daunting task! Delta Lake - an open-source storage layer brings ACID (Atomicity, Consistency, Isolation, Durability) transactions to Apache Spark and big data workloads.

Leveraging Delta Lake's ACID transactions and Time Travel capabilities in Databricks helps solve challenges like corrupted or unreliable data and prevents financial losses and customer dissatisfaction caused by fraudulent transactions. Your organization gains a significant edge in managing data lakes.

With Delta Lake, you get the capabilities that were previously only available in traditional data warehouses, but with the flexibility and scalability of modern cloud-based architectures.

Understanding Delta Lake

Delta Lake is a storage layer that sits atop existing data lakes. It provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing.

Key Features:

1. ACID Transactions
2. Scalable Metadata Handling
3. Time Travel (Data Versioning)
4. Schema Enforcement and Evolution
5. Audit History

We'll dig into all the features one by one.

1. ACID Transactions in Delta Lake

What is it: ACID properties ensure data reliability and consistency: must-haves for production environments!

How Delta Lake implements each ACID property

a) Atomicity: All changes within a transaction are treated as a single operation. Either all changes are committed, or none are.

Example: Updating and deleting data atomically

Explanation:

- We create a Delta table with 'id' and 'square' columns.
- The 'update and delete' function defines two operations: updating even-numbered rows and deleting rows where 'square' > 50.
- If any part of the transaction fails, the entire operation is rolled back, maintaining data consistency.

12:01 PM (4s) 3: Example: Updating and deleting data atomically

```python
from pyspark.sql.functions import *

# Create a Delta table
data = spark.range(0, 10).withColumn("square", col("id") * col("id"))
data.write.format("delta").mode("overwrite").save("/tmp/delta-table_1")

# Create DeltaTable object
deltaTable = DeltaTable.forPath(spark, "/tmp/delta-table_1")

# Verify the results
display(deltaTable.toDF())
```

(9) Spark Jobs

data: pyspark.sql.dataframe.DataFrame = [id: long, square: long]

Table

	id	square
1	2	4
2	3	9
3	4	16
4	7	49
5	8	64
6	9	81
7	0	0
8	1	1
9	5	25
10	6	36

10 rows | 3.86 seconds runtime Refreshed 34 minutes ago

Just now (4s) 4 Python

```python
# Update: Multiply 'square' by 2 for even 'id's
deltaTable.update(
    condition = expr("id % 2 == 0"),
    set = { "square": col("square") * 2 }
)

display(deltaTable.toDF())
```

(17) Spark Jobs

Table

	id	square
1	3	9
2	7	49
3	9	81
4	1	1
5	5	25
6	2	8
7	4	32
8	0	0
9	6	72

11:54 AM (<1s)

```python
# Delete: Remove rows where 'square' > 50
deltaTable.delete("square > 50")
display(deltaTable.toDF())
```

(1) Spark Jobs

Table

	id	square
1	0	0
2	1	1
3	2	8
4	4	32
5	3	9
6	5	25
7	7	49

7 rows | 0.36 seconds runtime

Refreshed 45 minutes ago

b) Consistency: A transaction log is maintained that tracks all changes, ensuring the data remains in a consistent state.

c) Isolation: Optimistic concurrency control is used to handle multiple concurrent reads and writes.

Example of handling concurrent writes ↓

Just now (7s)

Python

```python
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, "/tmp/delta-table_1")

# Writer 1
def update_even_numbers():
    deltaTable.update(
        condition = "id % 2 == 0",
        set = { "id": "id + 1000" }
    )

# Writer 2
def update_odd_numbers():
    deltaTable.update(
        condition = "id % 2 != 0",
        set = { "id": "id + 2000" }
    )

# Both writers can work concurrently without conflicts
update_even_numbers()
update_odd_numbers()
print(f"writers can work concurrently without conflicts...")
```

(31) Spark Jobs

```
writers can work concurrently without conflicts...
```

d) Durability: All committed changes immediately persisted and survive system failures.

2. Scalable Metadata Handling

Traditional data lakes often struggle with managing the metadata (information about your data) associated with petabyte-scale datasets.

How Delta Lake's scalable metadata handling is beneficial:

- Delta Lake distributes and optimizes metadata management to enable fast read and write operations even for petabyte-scale data.
- The transaction log serves as the backbone for Time Travel functionality. Delta Lake can quickly access historical versions of your data by leveraging this readily available metadata.
- By pruning unnecessary data from the transaction log, Delta Lake minimizes storage requirements for metadata management.

Example: Handling many small files

Explanation:

- We create a DataFrame with 10k rows and partition it into 1000 small files.
- Delta Lake efficiently handles the metadata for these numerous partitions.
- When querying, Delta Lake's metadata handling allows for efficient partition pruning, reading only the relevant partitions.

```
12:05 PM (3m)          9: Example: Handling a large number of small files          Python

from pyspark.sql.functions import col

# Generate a large number of small partitions
df = spark.range(0, 10000).withColumn("partition_id", col("id") % 100).repartition(1000)

# Write to Delta format
df.write.format("delta").partitionBy("partition_id").save("/tmp/many-partitions_LD")

# Read and query efficiently
result = spark.read.format("delta").load("/tmp/many-partitions_LD").filter("partition_id == 50")
print(f"Number of matching partitions: {result.rdd.getNumPartitions()}")

(7) Spark Jobs
  Job 131 View (Stages: 1/1)
  Job 132 View (Stages: 1/1, 1 skipped)
      Stage 227 0/4 succeeded View skipped
      Stage 228 1000/1000 succeeded View
  Job 133 View (Stages: 1/1)
  Job 134 View (Stages: 1/1)
  Job 135 View (Stages: 1/1, 1 skipped)
  Job 136 View (Stages: 1/1, 2 skipped)
  Job 137 View (Stages: 1/1)
  df: pyspark.sql.dataframe.DataFrame = [id: long, partition_id: long]
  result: pyspark.sql.dataframe.DataFrame = [id: long, partition_id: long]
Number of matching partitions: 4
```

3. Time Travel (Data Versioning)

Delta Lake's time travel functionality, achieved through versioning, simplifies building data pipelines.

Best advantages:

- Version control allows you to easily track changes made to your data over time.
- In case of bad writes or deletes, you can seamlessly revert to a previous version.
- Version control enables you to reproduce experiments and reports with ease.

- Delta Lake allows you to establish a central repository for your big data within your cloud storage.

Example: Querying different versions of a Delta table

Explanation:

- We create a Delta table and perform an update operation, creating two versions.
- We can query the current version (implicitly) and the previous version using 'versionAsOf'.
- The 'history()' method shows the full history of changes to the table.

Last execution failed — 12: Example: Querying different versions of a Delta table — Python

```python
from delta.tables import *

# Create initial Delta table
data = spark.range(0, 5)
data.write.format("delta").save("/tmp/delta-version-table")

# Version 0: Initial data
print("Version 0:")
spark.read.format("delta").load("/tmp/delta-version-table").show()

# Update the table (creates version 1)
deltaTable = DeltaTable.forPath(spark, "/tmp/delta-version-table")
deltaTable.update(condition = "id % 2 == 0", set = { "id": "id + 100" })

print("Version 1:")
spark.read.format("delta").load("/tmp/delta-version-table").show()

# Query version 0
print("Querying Version 0:")
spark.read.format("delta").option("versionAsOf", 0).load("/tmp/delta-version-table").show()
```

12:10 PM (1s) — 13

```python
# Get table history
display(deltaTable.history())
```

(1) Spark Jobs

Table

	version	timestamp	userId	userName	operation	operationParameters	job
1	2	2024-07-22T06:39:33.000+00:...	35073025997919...	ritesh.chidrewar@lumendata.com	OPTIMIZE	{"predicate":"[]","zOrderBy":"[]","batchId":"0","auto":"true"}	null
2	1	2024-07-22T06:39:28.000+00:...	35073025997919...	ritesh.chidrewar@lumendata.com	UPDATE	{"predicate":"[\"((id#15583L % 2) = 0)\"]"}	null
3	0	2024-07-22T06:39:24.000+00:...	35073025997919...	ritesh.chidrewar@lumendata.com	WRITE	{"mode":"ErrorIfExists","statsOnLoad":"false","partitionBy...	null

3 rows | 0.54 seconds runtime — Refreshed 2 hours ago

4. Schema Enforcement and Evolution

Delta Lake offers two complementary features that guarantee data quality and manageability within your data lake:

- **Schema Enforcement:** Enforces a predefined schema on data written to Delta tables, ensuring data consistency and integrity.

- **Schema Evolution:** Schema evolution provides controlled flexibility for adding new columns to adapt to evolving data needs while maintaining existing data integrity.

Example: Enforcing and evolving schema

Explanation:

- Create a Delta table with an initial schema.
- Attempting to write data with an incompatible schema fails, demonstrating schema enforcement.
- Schema evolution is activated by adding .option('mergeSchema', 'true')
- After schema evolution, we can successfully append data with the new schema.

12:12 PM (4s) 16: Example: Enforcing and evolving schema

```python
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

# Define initial schema
initial_schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True)
])

# Create Delta table with schema
data = spark.createDataFrame([(1, "Alice"), (2, "Bob")], initial_schema)
data.write.format("delta").save("/tmp/schema-table")

# Try to write data with incompatible schema (will fail)
try:
    incompatible_data = spark.createDataFrame([(1, "Charlie", 25)], ["id", "name", "age"])
    incompatible_data.write.format("delta").mode("append").save("/tmp/schema-table")
except Exception as e:
    print(f"Error: {str(e)}")
```

```
(6) Spark Jobs
data: pyspark.sql.dataframe.DataFrame = [id: integer, name: string]
incompatible_data: pyspark.sql.dataframe.DataFrame = [id: long, name: string ... 1 more field]
Error: [DELTA_FAILED_TO_MERGE_FIELDS] Failed to merge fields 'id' and 'id'
```

12:14 PM (3s) 17

```python
from delta.tables import *

# Load the existing Delta table
deltaTable = DeltaTable.forPath(spark, "/tmp/schema-table")

# Define the schema to match the existing table
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])

# New data with the defined schema
new_data = spark.createDataFrame(
    [(3, "Charlie", 25)],
    schema
)

# Write the new data with schema merging enabled
new_data.write.format("delta") \
    .mode("append") \
    .option("mergeSchema", "true") \
    .save("/tmp/schema-table")
```

12:14 PM (1s) 18

```python
# Verify the updated table
updated_table = spark.read.format("delta").load("/tmp/schema-table")
display(updated_table)
```

(3) Spark Jobs

updated_table: pyspark.sql.dataframe.DataFrame = [id: integer, name: string ... 1 more field]

Table

	id	name	age
1	3	Charlie	25
2	1	Alice	null
3	2	Bob	null

5. Audit History/Audit Logs

Delta Lake in Databricks automatically tracks changes made to your data through table versions. Each write operation (inserts, updates, deletes) creates a new version.

- **Audit & Rollback:** Table history enables us to audit data changes and rollback errors by tracking operations and timestamps.
- **Time Travel:** Analyze your data at any point in time by querying specific versions.

Example: Examining table changes

Explanation:

- We created a Delta table and performed various operations (update, delete, insert).
- The 'history()' method provides a detailed audit trail of all operations.
- We can examine specific versions to get detailed metrics about each operation.

12:20 PM (12s) 20: Example: Examining table changes Python

```python
from delta.tables import *
from pyspark.sql.functions import *

# Create a Delta table
data = spark.range(0, 5)
data.write.format("delta").save("/tmp/audit-table_LD")

# Perform some operations
deltaTable = DeltaTable.forPath(spark, "/tmp/audit-table_LD")

# Update
deltaTable.update(condition = "id % 2 == 0", set = { "id": "id + 10" })

# Delete
deltaTable.delete("id > 12")

# Insert
new_data = spark.range(20, 25)
new_data.write.format("delta").mode("append").save("/tmp/audit-table_LD")
```

(43) Spark Jobs

data: pyspark.sql.dataframe.DataFrame = [id: long]

12:21 PM (1s) 22 Python

```python
# Examine the audit history
history = deltaTable.history()
history.select("version", "timestamp", "operation", "operationParameters").show(truncate=False)

# Get details of a specific version
version_details = history.filter("version = 1").select("operationMetrics").collect()[0][0]
print(f"Details of version 1: {version_details}")
#display(version_details)
```

(4) Spark Jobs

history: pyspark.sql.dataframe.DataFrame = [version: long, timestamp: timestamp ... 13 more fields]

```
+-------+-------------------+---------+------------------------------------------------------------------+
|version|timestamp          |operation|operationParameters                                               |
+-------+-------------------+---------+------------------------------------------------------------------+
|4      |2024-07-22 06:50:54|WRITE    |{mode -> Append, statsOnLoad -> false, partitionBy -> []}         |
|3      |2024-07-22 06:50:52|OPTIMIZE |{predicate -> [], zOrderBy -> [], batchId -> 0, auto -> true}     |
|2      |2024-07-22 06:50:51|DELETE   |{predicate -> ["(id#19230L > 12)"]}                               |
|1      |2024-07-22 06:50:49|UPDATE   |{predicate -> ["((id#19230L % 2) = 0)"]}                          |
|0      |2024-07-22 06:50:45|WRITE    |{mode -> ErrorIfExists, statsOnLoad -> false, partitionBy -> []}|
+-------+-------------------+---------+------------------------------------------------------------------+

Details of version 1: {'numDeletionVectorsUpdated': '0', 'numAddedFiles': '3', 'executionTimeMs': '1628', 'numDeletionVectorsRemoved': '0', 'numUpdatedRows': '3', 'numRemovedFiles': '2', 'rewriteTimeMs': '738', 'numRemovedBytes': '988', 'scanTimeMs': '890', 'numCopiedRows': '0', 'numDeletionVectorsAdded': '1', 'numAddedChangeFiles': '0', 'numAddedBytes': '1481'}
```

Delta Lake vs Other Tools

a) Comparison with Hive Tables:

- Hive tables lack ACID properties for file-based tables.
- Delta Lake provides both ACID transactions and time travel.

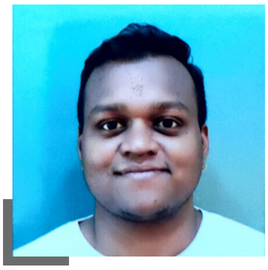
b) Comparison with Parquet:

- Parquet files are immutable. Updates and deletes become challenging.
- Delta Lake allows for easy updates, deletes, and merges while maintaining good read performance.

c) Comparison with traditional data warehouses:

- Delta Lake brings data warehouse-like capabilities to data lakes, allowing for both structured and unstructured data.
- It provides better scalability and cost-effectiveness compared to traditional data warehouses.

Authors



Ritesh Chidrewar
Senior Consultant

About LumenData

LumenData is a leading provider of **Enterprise Data Management, Cloud & Analytics** solutions. We help businesses navigate their data visualization and analytics anxieties and enable them to accelerate their innovation journeys.
Founded in 2008, with locations in multiple countries, LumenData is privileged to serve over 100 leading companies. LumenData is **SOC2 certified** and has instituted extensive controls to protect client data, including adherence to GDPR and CCPA regulations.