DELTA LAKE
WITH SPARK SQL

Delta Lake is an open source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.

delta.io | Documentation | GitHub | Delta Lake on Databricks

CREATE AND QUERY DELTA TABLES

Create and use managed database

```
-- Managed database is saved in the Hive metastore.
Default database is named "default".
DROP DATABASE IF EXISTS dbName;
CREATE DATABASE dbName;
USE dbName  -- This command avoids having to specify
dbName.tableName every time instead of just tableName.
```

Query Delta Lake table by table name (preferred)

```
/* You can refer to Delta Tables by table name, or by
path. Table name is the preferred way, since named tables
are managed in the Hive Metastore (i.e., when you DROP a
named table, the data is dropped also — not the case for
path-based tables.) */
SELECT * FROM [dbName.] tableName
```

Query Delta Lake table by path

```
SELECT * FROM delta.`path/to/delta_table` -- note
backticks
```

Convert Parquet table to Delta Lake format in place

```
-- by table name
CONVERT TO DELTA [dbName.]tableName
[PARTITIONED BY (col_name1 col_type1, col_name2
col_type2)]

-- path-based tables
CONVERT TO DELTA parquet.`/path/to/table` -- note backticks
[PARTITIONED BY (col_name1 col_type1, col_name2 col_type2)]
```

Create Delta Lake table as SELECT * with no upfront schema definition

```
CREATE TABLE [dbName.] tableName
USING DELTA
AS SELECT * FROM tableName | parquet.`path/to/data`
[LOCATION `/path/to/table`]
-- using location = unmanaged table
```

Create table, define schema explicitly with SQL DDL

```
CREATE TABLE [dbName.] tableName (
    id INT [NOT NULL],
    name STRING,
    date DATE,
    int_rate FLOAT)
USING DELTA
[PARTITIONED BY (time, date)] -- optional
```

Copy new data into Delta Lake table (with idempotent retries)

```
COPY INTO [dbName.] targetTable
  FROM (SELECT * FROM "/path/to/table")
  FILEFORMAT = DELTA  -- or CSV, Parquet, ORC, JSON, etc.
```

DELTA LAKE DDL/DML: UPDATE, DELETE, INSERT, ALTER TABLE

Update rows that match a predicate condition

```
UPDATE tableName SET event = 'click' WHERE event = 'clk'
```

Delete rows that match a predicate condition

```
DELETE FROM tableName WHERE "date < '2017-01-01"
```

Insert values directly into table

```
INSERT INTO TABLE tableName VALUES (
    (8003, "Kim Jones", "2020-12-18", 3.875),
    (8004, "Tim Jones", "2020-12-20", 3.750)
);
-- Insert using SELECT statement
INSERT INTO tableName SELECT * FROM sourceTable
-- Atomically replace all data in table with new values
INSERT OVERWRITE loan_by_state_delta VALUES (...)
```

Upsert (update + insert) using MERGE

```
MERGE INTO target
USING updates
ON target.Id = updates.Id
WHEN MATCHED AND target.delete_flag = "true" THEN
  DELETE
WHEN MATCHED THEN
  UPDATE SET *  -- star notation means all columns
WHEN NOT MATCHED THEN
  INSERT (date, Id, data) -- or, use INSERT *
  VALUES (date, Id, data)
```

Insert with Deduplication using MERGE

```
MERGE INTO logs
USING newDedupedLogs
ON logs.uniqueId = newDedupedLogs.uniqueId
WHEN NOT MATCHED
  THEN INSERT *
```

Alter table schema — add columns

```
ALTER TABLE tableName ADD COLUMNS (
    col_name data_type
    [FIRST|AFTER colA_name])
```

Alter table — add constraint

```
-- Add "Not null" constraint:
ALTER TABLE tableName CHANGE COLUMN col_name SET NOT NULL
-- Add "Check" constraint:
ALTER TABLE tableName
ADD CONSTRAINT dateWithinRange CHECK date > "1900-01-01"
-- Drop constraint:
ALTER TABLE tableName DROP CONSTRAINT dateWithinRange
```

TIME TRAVEL

View transaction log (aka Delta Log)

```
DESCRIBE HISTORY tableName
```

Query historical versions of Delta Lake tables

```
SELECT * FROM tableName VERSION AS OF 0
SELECT * FROM tableName@v0 -- equivalent to VERSION AS OF 0
SELECT * FROM tableName TIMESTAMP AS OF "2020-12-18"
```

Find changes between 2 versions of table

```
SELECT * FROM tableName VERSION AS OF 12
EXCEPT ALL SELECT * FROM tableName VERSION AS OF 11
```

TIME TRAVEL (CONTINUED)

Rollback a table to an earlier version

```
-- RESTORE requires Delta Lake version 0.7.0+ & DBR 7.4+.
RESTORE tableName VERSION AS OF 0
RESTORE tableName TIMESTAMP AS OF "2020-12-18"
```

UTILITY METHODS

View table details

```
DESCRIBE DETAIL tableName
DESCRIBE FORMATTED tableName
```

Delete old files with Vacuum

```
VACUUM tableName [RETAIN num HOURS] [DRY RUN]
```

Clone a Delta Lake table

```
-- Deep clones copy data from source, shallow clones don't.
CREATE TABLE [dbName.] targetName
[SHALLOW | DEEP] CLONE sourceName [VERSION AS OF 0]
[LOCATION "path/to/table"]
-- specify location only for path-based tables
```

Interoperability with Python / DataFrames

```
-- Read name-based table from Hive metastore into DataFrame
df = spark.table("tableName")
-- Read path-based table into DataFrame
df = spark.read.format("delta").load("/path/to/delta_table")
```

Run SQL queries from Python

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`/path/to/delta_table`")
```

Modify data retention settings for Delta Lake table

```
-- logRetentionDuration -> how long transaction log history
is kept, deletedFileRetentionDuration -> how long ago a file
must have been deleted before being a candidate for VACCUM.
ALTER TABLE tableName
SET TBLPROPERTIES(
    delta.logRetentionDuration = "interval 30 days",
    delta.deletedFileRetentionDuration = "interval 7 days"
);
SHOW TBLPROPERTIES tableName;
```

PERFORMANCE OPTIMIZATIONS

Compact data files with Optimize and Z-Order

```
*Databricks Delta Lake feature
OPTIMIZE tableName
[ZORDER BY (colNameA, colNameB)]
```

Auto-optimize tables

```
*Databricks Delta Lake feature
ALTER TABLE [table_name | delta.`path/to/delta_table`]
SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true)
```

Cache frequently queried data in Delta Cache

```
*Databricks Delta Lake feature
CACHE SELECT * FROM tableName
-- or:
CACHE SELECT colA, colB FROM tableName WHERE colNameA > 0
```

databricks

DELTA LAKE WITH PYTHON

Delta Lake is an open source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.

delta.io | Documentation | GitHub | API reference | Databricks

READS AND WRITES WITH DELTA LAKE

Read data from pandas DataFrame

```
df = spark.createDataFrame(pdf)
# where pdf is a pandas DF
# then save DataFrame in Delta Lake format as shown below
```

Read data using Apache Spark™

```
# read by path
df = (spark.read.format("parquet"|"csv"|"json"|etc.)
        .load("/path/to/delta_table"))
# read table from Hive metastore
df = spark.table("events")
```

Save DataFrame in Delta Lake format

```
(df.write.format("delta")
  .mode("append"|"overwrite")
  .partitionBy("date")  # optional
  .option("mergeSchema", "true")  # option - evolve schema
  .saveAsTable("events") | .save("/path/to/delta_table")
)
```

Streaming reads (Delta table as streaming source)

```
# by path or by table name
df = (spark.readStream
  .format("delta")
  .schema(schema)
  .table("events") | .load("/delta/events")
)
```

Streaming writes (Delta table as a sink)

```
streamingQuery = (
df.writeStream.format("delta")
  .outputMode("append"|"update"|"complete")
  .option("checkpointLocation", "/path/to/checkpoints")
  .trigger(once=True|processingTime="10 seconds")
  .table("events") | .start("/delta/events")
)
```

CONVERT PARQUET TO DELTA LAKE

Convert Parquet table to Delta Lake format in place

```
deltaTable = DeltaTable.convertToDelta(spark,
"parquet.`/path/to/parquet_table`")

partitionedDeltaTable = DeltaTable.convertToDelta(spark,
"parquet.`/path/to/parquet_table`", "part int")
```

WORKING WITH DELTA TABLES

```
# A DeltaTable is the entry point for interacting with
tables programmatically in Python — for example, to
perform updates or deletes.
from delta.tables import *

deltaTable = DeltaTable.forName(spark, tableName)
deltaTable = DeltaTable.forPath(spark,
delta.`path/to/table`)
```

DELTA LAKE DDL/DML: UPDATES, DELETES, INSERTS, MERGES

Delete rows that match a predicate condition

```
# predicate using SQL formatted string
deltaTable.delete("date < '2017-01-01'")
# predicate using Spark SQL functions
deltaTable.delete(col("date") < "2017-01-01")
```

Update rows that match a predicate condition

```
# predicate using SQL formatted string
deltaTable.update(condition = "eventType = 'clk'",
          set = { "eventType": "'click'" } )
# predicate using Spark SQL functions
deltaTable.update(condition = col("eventType") == "clk",
          set = { "eventType": lit("click") } )
```

Upsert (update + insert) using MERGE

```
# Available options for merges [see documentation for
details]:
.whenMatchedUpdate(...)    | .whenMatchedUpdateAll(...)  |
.whenNotMatchedInsert(...) | .whenMatchedDelete(...)
(deltaTable.alias("target").merge(
        source = updatesDF.alias("updates"),
        condition = "target.eventId = updates.eventId")
  .whenMatchedUpdateAll()
  .whenNotMatchedInsert(
     values = {
        "date": "updates.date",
        "eventId": "updates.eventId",
        "data": "updates.data",
        "count": 1
        }
  ).execute()
)
```

Insert with Deduplication using MERGE

```
(deltaTable.alias("logs").merge(
     newDedupedLogs.alias("newDedupedLogs"),
     "logs.uniqueId = newDedupedLogs.uniqueId")
  .whenNotMatchedInsertAll()
.execute()
)
```

TIME TRAVEL

View transaction log (aka Delta Log)

```
fullHistoryDF = deltaTable.history()
```

Query historical versions of Delta Lake tables

```
# choose only one option: versionAsOf, or timestampAsOf
df = (spark.read.format("delta")
                .option("versionAsOf", 0)
                .option("timestampAsOf", "2020-12-18")
                .load("/path/to/delta_table"))
```

TIME TRAVEL (CONTINUED)

Find changes between 2 versions of a table

```
df1 = spark.read.format("delta").load(pathToTable)
df2 = spark.read.format("delta").option("versionAsOf",
2).load("/path/to/delta_table")
df1.exceptAll(df2).show()
```

Rollback a table by version or timestamp

```
deltaTable.restoreToVersion(0)
deltaTable.restoreToTimestamp('2020-12-01')
```

UTILITY METHODS

Run Spark SQL queries in Python

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`/path/to/delta_table`")
spark.sql("DESCRIBE HISTORY tableName")
```

Compact old files with Vacuum

```
deltaTable.vacuum() # vacuum files older than default
retention period (7 days)
deltaTable.vacuum(100) # vacuum files not required by
versions more than 100 hours old
```

Clone a Delta Lake table

```
deltaTable.clone(target="/path/to/delta_table/",
isShallow=True, replace=True)
```

Get DataFrame representation of a Delta Lake table

```
df = deltaTable.toDF()
```

Run SQL queries on Delta Lake tables

```
spark.sql("SELECT * FROM tableName")
spark.sql("SELECT * FROM delta.`/path/to/delta_table`")
```

PERFORMANCE OPTIMIZATIONS

Compact data files with Optimize and Z-Order

```
*Databricks Delta Lake feature
spark.sql("OPTIMIZE tableName [ZORDER BY (colA, colB)]")
```

Auto-optimize tables

```
*Databricks Delta Lake feature. For existing tables:
spark.sql("ALTER TABLE [table_name |
delta.`path/to/delta_table`]
SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true)
To enable auto-optimize for all new Delta Lake tables:
spark.sql("SET spark.databricks.delta.properties.
defaults.autoOptimize.optimizeWrite = true")
```

Cache frequently queried data in Delta Cache

```
*Databricks Delta Lake feature
spark.sql("CACHE SELECT * FROM tableName")
-- or:
spark.sql("CACHE SELECT colA, colB FROM tableName
           WHERE colNameA > 0")
```

databricks