Get started with Delta using Spark APIs

Note: *In this reading you can see the steps involved in the process of setting up Delta using Spark APIs.*

Delta Lake is included with Azure Databricks. You can start using it today. To quickly get started with Delta Lake, do the following:

Instead of **`parquet`**...

```
CREATE TABLE ...
USING parquet
...

dataframe
    .write
    .format("parquet")
    .save("/data")
```

simply say **`delta`**

```
CREATE TABLE ...
USING delta
...

dataframe
    .write
    .format("delta")
    .save("/data")
```

Using Delta with your existing Parquet tables

Step 1: Convert **`Parquet`** to **`Delta`** tables:

```
CONVERT TO DELTA parquet.`path/to/table` [NO STATISTICS]
[PARTITIONED BY (col_name1 col_type1, col_name2 col_type2, ...)]
```

Step 2: Optimize layout for fast queries:

```
OPTIMIZE events
WHERE date >= current_timestamp() - INTERVAL 1 day
ZORDER BY (eventType)
```

Basic syntax

Two of the core features of Delta Lake are performing upserts (insert/updates) and Time Travel operations. We will explore these concepts more within the notebooks in this module.

To UPSERT means to "UPdate" and "inSERT". In other words, UPSERT is literally TWO operations. It is not supported in traditional data lakes, as running an UPDATE could invalidate data that is accessed by the subsequent INSERT operation.

Using Delta Lake, however, we can do UPSERTS. Delta Lake combines these operations to guarantee atomicity to

- INSERT a row
- if the row already exists, UPDATE the row.

Upsert syntax

Upserting, or merging, in Delta Lake provides fine-grained updates of your data. The following syntax shows how to perform an Upsert:

```
MERGE INTO customers -- Delta table
USING updates
ON customers.customerId = source.customerId
WHEN MATCHED THEN
    UPDATE SET address = updates.address
WHEN NOT MATCHED
    THEN INSERT (customerId, address) VALUES (updates.customerId, updates.addres
s)
```

See update table data syntax documentation.

Time Travel syntax

Because Delta Lake is version controlled, you have the option to query past versions of the data.

Using a single file storage system, you now have access to several versions your historical data, ensuring that your data analysts will be able to replicate their reports (and compare aggregate changes over time) and your data scientists will be able to replicate their experiments.

Other time travel use cases are:

- Re-creating analyses, reports, or outputs (for example, the output of a machine learning model). This could be useful for debugging or auditing, especially in regulated industries.
- Writing complex temporal queries.
- Fixing mistakes in your data.
- Providing snapshot isolation for a set of queries for fast changing tables.

Example of using time travel to reproduce experiments and reports:

```
SELECT count(*) FROM events
TIMESTAMP AS OF timestamp

SELECT count(*) FROM events
VERSION AS OF version
```

```
spark.read.format("delta").option("timestampAsOf", timestamp_string).load("/
events/")
```

If you need to rollback accidental or bad writes:

```
2
3
INSERT INTO my_table
    SELECT * FROM my_table TIMESTAMP AS OF
    date_sub( current_date(), 1)
```

See time travel syntax documentation.