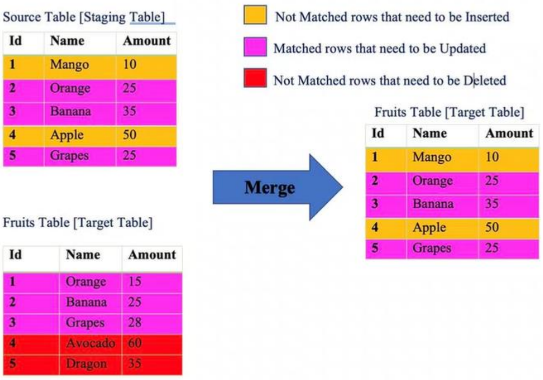
Upserts and Deletes in Delta Lake

- Upsert = Update + Insert
- DELETE operation supported on Delta tables- Ideal for slowly changing dimensions (SCD)



🟧 Not Matched rows that need to be Inserted

🟪 Matched rows that need to be Updated

🟥 Not Matched rows that need to be Deleted

Source Table [Staging Table]

Id	Name	Amount
1	Mango	10
2	Orange	25
3	Banana	35
4	Apple	50
5	Grapes	25

Fruits Table [Target Table]

Id	Name	Amount
1	Orange	15
2	Banana	25
3	Grapes	28
4	Avocado	60
5	Dragon	35

Fruits Table [Target Table]

Id	Name	Amount
1	Mango	10
2	Orange	25
3	Banana	35
4	Apple	50
5	Grapes	25

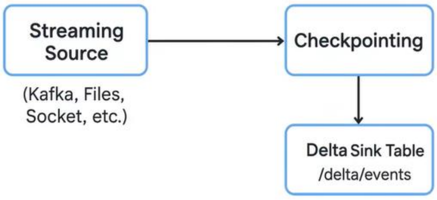
```
MERGE INTO target t
USING source s
ON t.id = s.id
WHEN MATCHED THEN UPDATE SET t.value = s.value
WHEN NOT MATCHED THEN INSERT (id, value) VALUES (s.id, s.value)
```

Streaming with Delta Lake

- Delta Lake supports structured streaming
- Reads and writes are transactional

Auto Loader in Databricks:

- It provides a Structured Streaming source called cloudFiles . Given an input directory path on the cloud file storage, the cloudFiles source automatically processes new files as they arrive, with the option of also processing existing files in that directory



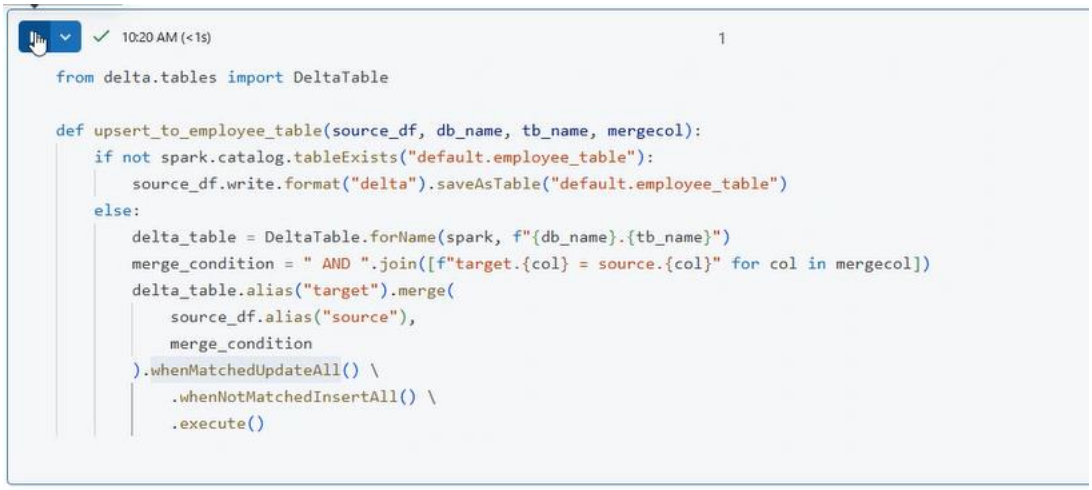
Streaming with Delta Lake

Streaming Read Example

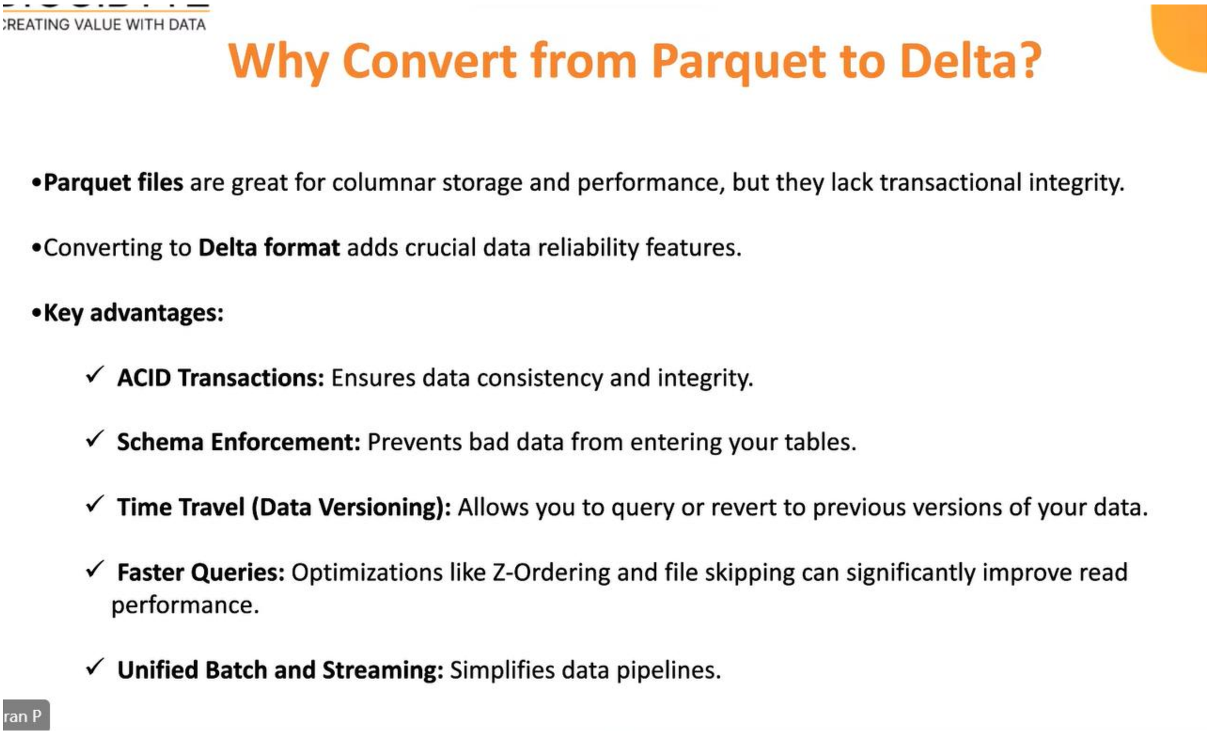
```
stream_df = spark.readStream \
 .format("delta") \
 .load("/delta/events")
```

Streaming Write Example

```
df.writeStream \
 .format("delta") \
 .outputMode("append") \
 .option("checkpointLocation", "/chkpt") \
 .start("/delta/events")
```



```
from delta.tables import DeltaTable

def upsert_to_employee_table(source_df, db_name, tb_name, mergecol):
    if not spark.catalog.tableExists("default.employee_table"):
        source_df.write.format("delta").saveAsTable("default.employee_table")
    else:
        delta_table = DeltaTable.forName(spark, f"{db_name}.{tb_name}")
        merge_condition = " AND ".join([f"target.{col} = source.{col}" for col in mergecol])
        delta_table.alias("target").merge(
            source_df.alias("source"),
            merge_condition
        ).whenMatchedUpdateAll() \
            .whenNotMatchedInsertAll() \
            .execute()
```



:REATING VALUE WITH DATA

Why Convert from Parquet to Delta?

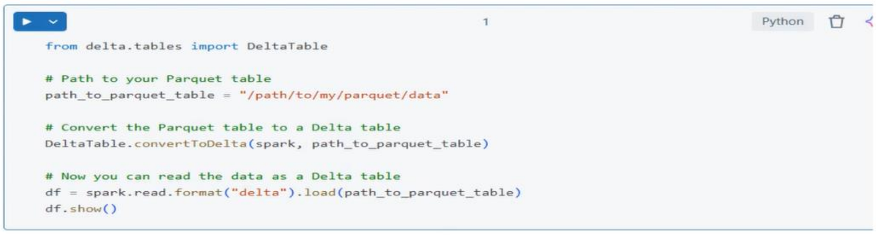
- **Parquet files** are great for columnar storage and performance, but they lack transactional integrity.
- Converting to **Delta format** adds crucial data reliability features.
- **Key advantages:**
  - ✓ **ACID Transactions:** Ensures data consistency and integrity.
  - ✓ **Schema Enforcement:** Prevents bad data from entering your tables.
  - ✓ **Time Travel (Data Versioning):** Allows you to query or revert to previous versions of your data.
  - ✓ **Faster Queries:** Optimizations like Z-Ordering and file skipping can significantly improve read performance.
  - ✓ **Unified Batch and Streaming:** Simplifies data pipelines.

ran P

Converting Parquet to Delta (Method 1: delta.convert)

•The **delta.convert or convertToDelta** command is the simplest and most efficient way to convert an existing Parquet table in place.

•It's a metadata-only operation, meaning it doesn't rewrite any data files. It just creates the **Delta transaction log** on top of the Parquet files.

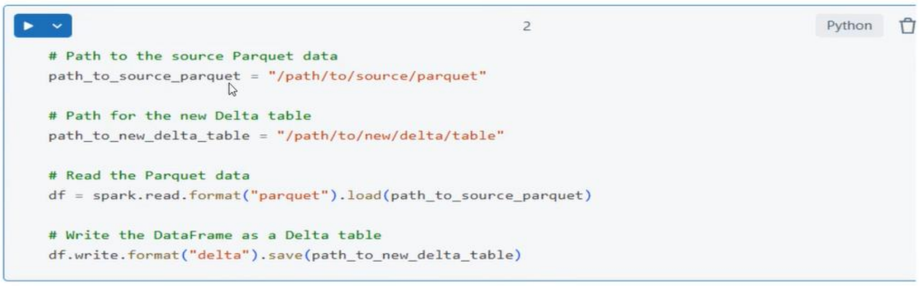


```
from delta.tables import DeltaTable

# Path to your Parquet table
path_to_parquet_table = "/path/to/my/parquet/data"

# Convert the Parquet table to a Delta table
DeltaTable.convertToDelta(spark, path_to_parquet_table)

# Now you can read the data as a Delta table
df = spark.read.format("delta").load(path_to_parquet_table)
df.show()
```

Converting Parquet to Delta (Method 2:spark.write)

• This method is used when you need to **rewrite data** (e.g., changing the partitioning scheme or doing a small transformation) during the conversion.

• You read the Parquet data and then write it out in the Delta format.



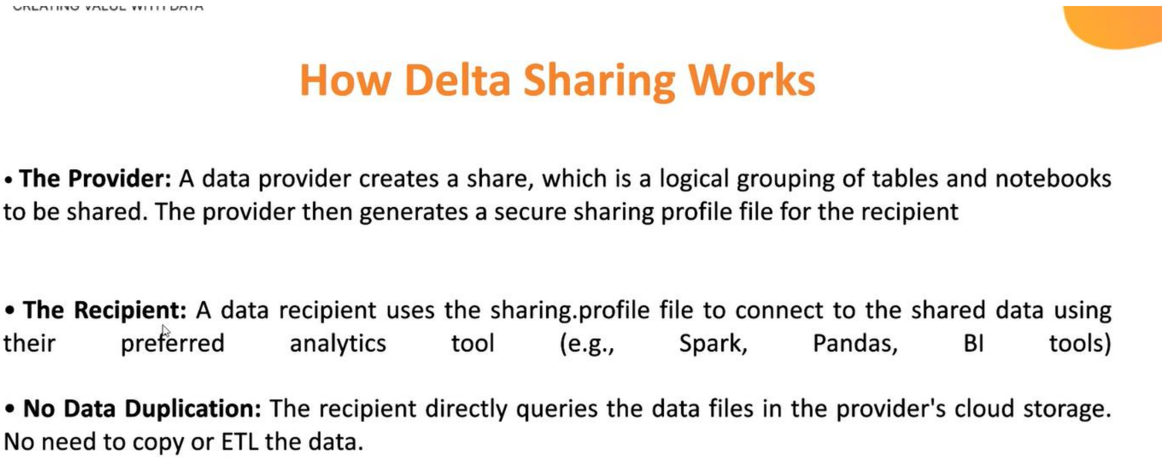
```
# Path to the source Parquet data
path_to_source_parquet = "/path/to/source/parquet"

# Path for the new Delta table
path_to_new_delta_table = "/path/to/new/delta/table"

# Read the Parquet data
df = spark.read.format("parquet").load(path_to_source_parquet)

# Write the DataFrame as a Delta table
df.write.format("delta").save(path_to_new_delta_table)
```

Introduction to Delta Sharing

• **Delta Sharing** is an open protocol developed by Databricks for securely and easily sharing data across different clouds, platforms, and organization.

• It's the first open protocol for data sharing.

• Unlike traditional data sharing methods (e.g., SFTP, API calls) that are complex and slow, Delta Sharing provides **direct access to the data** without needing to copy it.



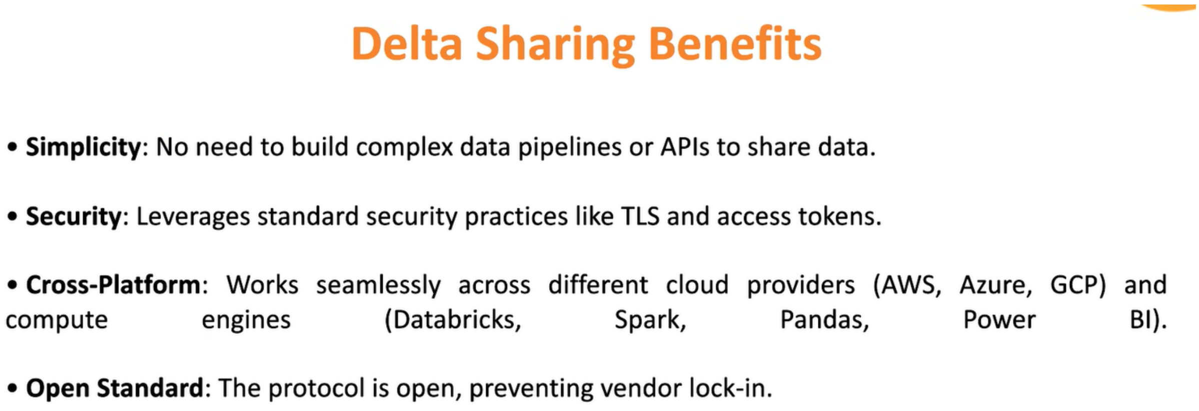
CREATING VALUE WITH DATA

How Delta Sharing Works

- **The Provider:** A data provider creates a share, which is a logical grouping of tables and notebooks to be shared. The provider then generates a secure sharing profile file for the recipient

- **The Recipient:** A data recipient uses the sharing.profile file to connect to the shared data using their preferred analytics tool (e.g., Spark, Pandas, BI tools)

- **No Data Duplication:** The recipient directly queries the data files in the provider's cloud storage. No need to copy or ETL the data.



Delta Sharing Benefits

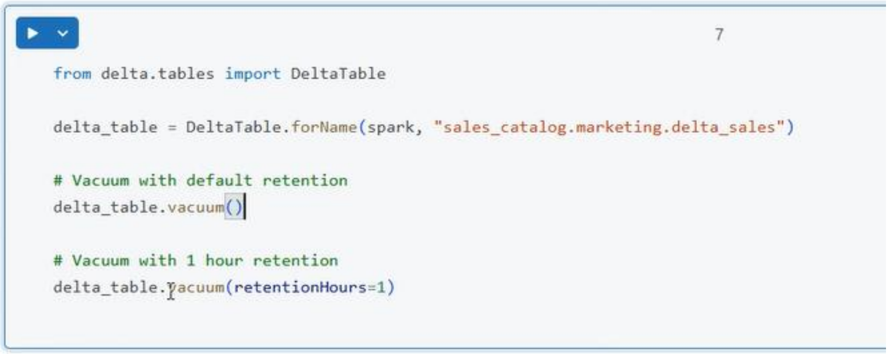
- **Simplicity**: No need to build complex data pipelines or APIs to share data.

- **Security**: Leverages standard security practices like TLS and access tokens.

- **Cross-Platform**: Works seamlessly across different cloud providers (AWS, Azure, GCP) and compute engines (Databricks, Spark, Pandas, Power BI).

- **Open Standard**: The protocol is open, preventing vendor lock-in.

vacuum



```
from delta.tables import DeltaTable

delta_table = DeltaTable.forName(spark, "sales_catalog.marketing.delta_sales")

# Vacuum with default retention
delta_table.vacuum()

# Vacuum with 1 hour retention
delta_table.vacuum(retentionHours=1)
```

Features of Spark