Apache Spark 2 using SQL - DML and Partitioning

DML (Data Manipulation) + Partitioning in Spark SQL (Spark 2, Scala).

1 — Quick conceptual summary

DML in Spark SQL covers commands that manipulate data: INSERT, INSERT OVERWRITE, INSERT INTO, INSERT OVERWRITE DIRECTORY, LOAD DATA, SELECT.

Note: traditional row-level UPDATE/DELETE/MERGE are **not** generically supported for plain Parquet/Hive tables in Spark 2 — use **Delta Lake**, Hive ACID tables, or rewrite patterns (INSERT OVERWRITE / upsert by read+write) for updates. **Partitioning** means physically organizing table data into directory shards (e.g. .../year=2024/month=08/). Partition columns are directory-level values. Partitioning gives **partition pruning** (read only needed folders) and faster queries when filters include partition columns.

2 — Basic DML examples (Spark SQL in Scala)

```
import org.apache.spark.sql.SparkSession
object DMLPartitioningExamples {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .appName("SparkSQL DML & Partitioning")
      .master("local[*]")
      // Use enableHiveSupport() if you want persistent Hive tables / SHOW PARTITIONS
      .enableHiveSupport()
      .getOrCreate()
    import spark.implicits._
    spark.sparkContext.setLogLevel("ERROR")
    // Create a sample DataFrame
    val sales = Seq(
      (1,"2024-08-01","NY","prodA",100.0),
      (2,"2024-08-01","NY","prodB",200.0),
      (3,"2024-08-02","CA","prodA",150.0)
    ).toDF("id","sale_date","state","product","amount")
    // Create external partitioned table (parquet) in current warehouse
    spark.sql("CREATE DATABASE IF NOT EXISTS demo_db")
    spark.sql("USE demo_db")
    // Create table schema with partition columns (year, month) and data columns
    spark.sql(
```

```
      """
        |CREATE TABLE IF NOT EXISTS sales_parq (
        | id INT,
        | sale_date STRING,
        | state STRING,
        | product STRING,
        | amount DOUBLE
        |)
        |USING parquet
        |PARTITIONED BY (year INT, month INT)
      """.stripMargin)
    // Write data into a staging location and then insert with partitioning
    val staged = sales
      .withColumn("year", $"sale_date".substr(1,4).cast("int"))
      .withColumn("month", $"sale_date".substr(6,2).cast("int"))
    // Save staged to a temp view
    staged.createOrReplaceTempView("staged_sales")
    // Simple INSERT INTO (append) using SQL -- dynamic partitions
    // This inserts rows from staged_sales into sales_parq, partitioned by year,month
    spark.sql("INSERT INTO TABLE sales_parq PARTITION (year, month) SELECT id, sale_date, state, product,
amount, year, month FROM staged_sales")
    // Show table content (reads partitioned data)
    spark.sql("SELECT * FROM sales_parq").show(false)
    // INSERT OVERWRITE by partition (overwrite partitions for year=2024)
    spark.sql(
      """
        |INSERT OVERWRITE TABLE sales_parq PARTITION (year=2024, month)
        |SELECT id, sale_date, state, product, amount, month FROM staged_sales WHERE year = 2024
      """.stripMargin)
    // Show partitions (works when Hive support / Hive metastore enabled)
    spark.sql("SHOW PARTITIONS sales_parq").show(truncate=false)
    // Alternative: write DataFrame using DataFrame API with partitionBy
    staged.write.mode("append").partitionBy("year","month").parquet("spark-warehouse/sales_parq")
    spark.stop()
  }
}
```

Notes on above

- CREATE TABLE … PARTITIONED BY (year INT, month INT) defines partition columns. Partition columns are **not** listed among the table's data columns — they become directory keys.
- INSERT INTO TABLE … PARTITION (year, month) SELECT … is the standard way to insert data into partitions (dynamic partitions when you supply the partition columns in SELECT).
- INSERT OVERWRITE TABLE … PARTITION (year=….) can be used for **static** partition overwrite.
- When writing via DataFrame API: df.write.partitionBy("year","month").parquet(path) will create partition directories automatically.

3 — Read, partition pruning & basePath

When you read a partitioned directory, Spark discovers partition columns automatically if the files are laid out as …/year=2024/month=08/part-*.parquet.

```
val df = spark.read.parquet("spark-warehouse/sales_parq")
df.printSchema()
// Partition pruning example:
```

```
val filtered = df.filter($"year" === 2024 && $"month" === 8)
filtered.explain(true)   // shows that only required partitions are read
filtered.show()
```

If reading from multiple partition dirs with different root paths, use option("basePath", "/path/to/root") to ensure correct partition column discovery.

4 — Writing modes & partition behaviour

mode("append") → add new files/partitions.

mode("overwrite") → replaces the entire table or path by default. If you want to overwrite only matching partitions, set:

```
spark.conf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")

// then:

df.write.mode("overwrite").partitionBy("year","month").parquet(path)
```

With "dynamic", Spark overwrites only partitions present in the write dataset.

5 — Partitioning best practices & performance tips

- **Choose partition columns with moderate cardinality.**
  Good: year, month, country (low/moderate cardinality). Bad: user_id / transaction_id (very high cardinality) — creates too many small partitions/files.
- **Avoid excessive small files.** Many small files slow down scheduling and increase metadata overhead. Use coalesce() (or repartition() carefully) before write:

```
df.repartition($"year", $"month").write.parquet(path)
```

- **Partition pruning**: Always filter on partition columns in queries when possible so Spark will skip irrelevant folders.
- **Use Parquet/ORC** for columnar storage — better IO and predicate pushdown.
- **If you need update/delete/upsert** patterns: use Delta Lake, Hudi or Iceberg for ACID upserts / MERGE.

6 — Common practical examples

6.1 Create external partitioned table pointing to path

```
CREATE EXTERNAL TABLE IF NOT EXISTS sales_ext (
  id INT, sale_date STRING, state STRING, product STRING, amount DOUBLE
)
PARTITIONED BY (year INT, month INT)
STORED AS PARQUET
LOCATION 's3a://my-bucket/sales/';
```

You can then MSCK REPAIR TABLE sales_ext (in Hive) or ALTER TABLE ... RECOVER PARTITIONS to discover existing partition directories.

6.2 SHOW PARTITIONS / DESCRIBE

```
SHOW PARTITIONS sales_parq;
DESCRIBE FORMATTED sales_parq;
```

SHOW PARTITIONS lists partition values (works when table is in Hive metastore).

6.3 Load local file into table (LOAD DATA) *(Hive-style)*

LOAD DATA LOCAL INPATH '/tmp/newfile.parquet' INTO TABLE sales_parq PARTITION (year=2024, month=8);
(requires Hive support and table location)

7 — When you need UPDATE / DELETE / MERGE

Spark 2 with plain Parquet: no native row-level UPDATE/DELETE. Options:

- **INSERT OVERWRITE** approach: Read table, perform change in DataFrame, write back with INSERT OVERWRITE or overwrite partitions.
- **Delta Lake** (recommended for production): supports MERGE, UPDATE, DELETE, time travel. Example repo: delta-io/delta.

Short example pattern (overwrite partition to "update" a partition):

```
// read all partitions except partition_to_update
val rest = spark.sql("SELECT * FROM sales_parq WHERE NOT (year=2024 AND month=8)")
val updatedPartition = newDataForPartition  // DataFrame for that partition
// write back
rest.union(updatedPartition)
  .write.mode("overwrite")
  .partitionBy("year","month")
  .parquet("spark-warehouse/sales_parq")
```

8 — Troubleshooting tips

- If partitions don't show up: ensure data folder layout is col=value directories and that table LOCATION points to parent folder. You may need to run MSCK REPAIR TABLE (Hive metastore) or ALTER TABLE ... RECOVER PARTITIONS.

If INSERT INTO errors with dynamic partition: enable Hive dynamic partitions when using Hive-mode:

```
spark.conf.set("hive.exec.dynamic.partition", "true")

spark.conf.set("hive.exec.dynamic.partition.mode", "nonstrict")
```

- If writes produce many small files: increase parallelism or coalesce(1) if you want one file (not recommended for large datasets).

9 — Mini Quiz

1. **What does INSERT OVERWRITE TABLE t PARTITION (year=2024) do?**
2. **True/False:** Partition columns are stored in data files as regular columns.
3. **Which write mode** would you use to overwrite only the partitions present in the write DataFrame (not entire table)?
   a) overwrite b) append c) Set spark.sql.sources.partitionOverwriteMode=dynamic + overwrite d) ignore
4. **Why is partitioning helpful?** (short answer)
5. **Name two techniques** to support row-level upserts/deletes in Spark (production-grade).

(answers at bottom)

10 — Useful GitHub repositories & resources

- **Official Apache Spark examples**: apache/spark → examples/src/main/scala/org/apache/spark/examples/sql — good reference for SQL/DF examples.

- **Spark by Examples**: sparkbyexamples/spark-scala-examples — practical, well organized code snippets.
- **Delta Lake** (ACID + MERGE): delta-io/delta — for upserts, deletes, time travel.
- **Awesome Spark** curated lists: search awesome-spark on GitHub to find many community resources and tutorials.
  (You likely already have these bookmarked; they're great starting points.)

11 — Answers to quiz

1. It overwrites data for the specified partition year=2024 (replaces files under those partitions).
2. False — partition columns are represented by directory names (e.g. year=2024) and not necessarily repeated inside every file (Spark will populate partition column values on read).
3. c) set spark.sql.sources.partitionOverwriteMode=dynamic and use overwrite.
4. Partitioning lets Spark skip reading irrelevant directories (partition pruning), reducing I/O and speeding queries when queries filter on partition columns.
5. Use **Delta Lake** (MERGE/UPDATE/DELETE), **Apache Hudi**, **Apache Iceberg**, or perform read-modify-write with INSERT OVERWRITE per partition.