
Shwetank Singh
GritSetGrow - GSGLearn.com


DATA AND AI

# BATCH PROCESSING WITH SPARK

## DATAFRAME VS SQL, JOB STRUCTURE

www.gsglearn.com

# Batch Processing with Spark --- DataFrame vs SQL, and Job Structure

> ***Goal:*** *Give you a production-ready blueprint for batch jobs in Spark, comparing* ***DataFrame API vs Spark SQL****, and laying out* ***job structure****,* ***idempotency****,* ***testing****, and* ***operational*** *patterns.*
>
> ***Covers:*** *API parity, when to choose which, schema & I/O, joins/aggregations, UDFs, file layout, incremental processing, MERGE/upserts, backfills, orchestration, config knobs, examples in* ***PySpark*** *(with Scala/SQL notes).*

---

## Table of Contents



---

## 1) Mental Model: How Spark Executes

- **Logical Plan → Optimized Plan (Catalyst) → Physical Plan (Tungsten) → Tasks**.
- Both **DataFrame** and **Spark SQL** compile to the same Catalyst plan.
- Shuffles happen on **wide** operations (joins, groupBy, window). Keep an eye on them.

```
-- Inspect a query plan (SQL)
EXPLAIN FORMATTED
SELECT c.customer_id, SUM(o.amount) amt
FROM orders o JOIN customers c USING(customer_id)
GROUP BY c.customer_id;
```

---

## 2) DataFrame vs Spark SQL: Parity, Pros/Cons, and Interop

**Parity**: Almost all relational ops exist in both APIs. Interchange via temp views.

```
# Interop
orders_df.createOrReplaceTempView("orders")
spark.sql("SELECT * FROM orders WHERE dt >= '2025-08-01'")
```

**When to favor DataFrame API**

- Complex programmatic logic, branching, re-usable functions.
- Strong typing (Scala) & IDE refactors.
- Easier unit testing of small transforms.

**When to favor SQL**

- Pure relational transformations, analysts contributing queries.
- Window-heavy logic is often more readable.
- Leverage optimizer hints ( `BROADCAST` , `REPARTITION` , etc.).

**Team pattern**: Keep **business logic** in SQL (views/models), **plumbing** (I/O, params, retries) in DataFrame/PySpark. Mix as needed.

---

## 3) I/O & Schemas: Read/Write Patterns

**Reads**

```
# Parquet/Delta/CSV examples
sales = (spark.read
  .format("parquet")
  .option("mergeSchema", "false")  # prefer explicit schemas
  .load("abfss://lake/silver/sales"))

# Explicit schema (recommended)
from pyspark.sql.types import *
schema = StructType([
    StructField("order_id", LongType(), False),
    StructField("customer_id", LongType(), False),
    StructField("dt", DateType(), False),
    StructField("amount", DecimalType(12,2), True),
])
bronze = spark.read.schema(schema).json("abfss://lake/bronze/orders/*.json")
```

**Writes**

```
# Partitioned write with file sizing
(target_df
  .repartition(200, "dt")                     # writer parallelism & partitioning
  .sortWithinPartitions("dt", "customer_id")  # better encodings
  .write
  .mode("append")
  .option("maxRecordsPerFile", 5_000_000)
  .parquet("abfss://lake/silver/orders"))
```

**SQL equivalents**

```
CREATE TABLE silver.orders USING PARQUET LOCATION 'abfss://lake/silver/orders';
INSERT INTO silver.orders SELECT * FROM bronze.orders_view;
```

**Schema evolution**: Prefer **additive** (new nullable columns). Avoid implicit inference in prod.

---

## 4) Core Transformations & Joins

```
from pyspark.sql import functions as F
# Filters, projections
clean = (bronze
  .filter(F.col("dt") >= F.lit("2025-08-01"))
  .select("order_id","customer_id","dt",
          F.col("amount").cast("decimal(12,2)").alias("amount")))

# Joins
joined = (clean.alias("o")
  .join(customers.alias("c"), "customer_id", "inner")
  .join(products.alias("p"), "sku_id", "left"))

# Aggregations
agg = (joined
  .groupBy("customer_id", F.to_date("dt").alias("d"))
  .agg(F.sum("amount").alias("daily_amount")))
```

**SQL**

```
WITH clean AS (
  SELECT order_id, customer_id, CAST(amount AS DECIMAL(12,2)) AS amount, dt
  FROM bronze.orders WHERE dt >= DATE '2025-08-01'
)
SELECT c.customer_id, DATE(o.dt) AS d, SUM(o.amount) AS daily_amount
FROM clean o JOIN dim_customers c USING (customer_id)
GROUP BY c.customer_id, DATE(o.dt);
```

---

## 5) UDFs vs Built-ins (and Pandas UDFs)

- Prefer **Spark SQL functions** (`functions.*`) → vectorized, optimized, pushdown-friendly.
- **Scala/PySpark UDFs**: easy but slower; break predicate pushdown & codegen.
- **Pandas UDFs** (vectorized) for heavy numeric ops; still avoid if built-ins exist.

```
# Anti-pattern: plain Python UDF for simple math
@F.udf('double')
def bad(x):
    return x * 1.18
```

```
# Better: built-in expr
better = df.select((F.col('amount')*F.lit(1.18)).alias('gross'))
```

---

## 6) Job Structure Templates

**Template A --- Simple Batch (Bronze → Silver)**

```
from argparse import ArgumentParser
from pyspark.sql import functions as F

parser = ArgumentParser()
parser.add_argument('--run-date', required=True)  # e.g., 2025-08-22
args = parser.parse_args()

run_date = args.run_date

# 1) Read partition(s)
bronze = spark.read.schema(schema).json(f"abfss://lake/bronze/orders/dt=
{run_date}/*.json")

# 2) Transform
silver = (bronze
  .filter(F.col('is_valid') == F.lit(True))

.select('order_id','customer_id','sku_id','amount',F.col('dt').cast('date').alias('dt'))


# 3) Write partitioned
(silver
  .repartition(200,'dt')
  .write.mode('append')
  .partitionBy('dt')
  .parquet('abfss://lake/silver/orders'))
```

**Template B --- SQL-First Batch**

```
spark.sql("CREATE TEMP VIEW bronze_orders AS SELECT * FROM
parquet.`abfss://lake/bronze/orders`;")
spark.sql("""
INSERT INTO parquet.`abfss://lake/silver/orders`
SELECT order_id, customer_id, sku_id, CAST(dt AS DATE) dt, amount
FROM bronze_orders WHERE is_valid = true
""")
```

**Template C --- Multi-stage DAG inside one job**

- Stage 1: Load **dimensions** (cached).
- Stage 2: Transform facts with joins.
- Stage 3: Aggregate to daily/monthly tables.
- Stage 4: Optional **optimize/compact** hot partitions.

---

## 7) Incremental Processing & Upserts

**Incremental by watermark (max dt)**

```
from pyspark.sql.functions import col, lit, max as smax
last_dt = (spark.read.parquet('abfss://lake/silver/orders')
             .agg(smax('dt').alias('m')).collect()[0]['m'])
new_data = spark.read.json("abfss://lake/bronze/orders").where(col('dt') >
lit(last_dt))
```

**MERGE/Upsert (Delta/Iceberg/Hudi)**

```
-- Delta Lake example
MERGE INTO silver.orders AS t
USING tmp_upserts AS s
ON t.order_id = s.order_id
WHEN MATCHED THEN UPDATE SET amount = s.amount, dt = s.dt
WHEN NOT MATCHED THEN INSERT *;
```

**Idempotent staging**: write `tmp_upserts` per run_date → MERGE → drop temp.

---

## 8) Backfills & Reprocessing

- Drive by **date ranges**; isolate to **target partitions**.
- Use **overwrite by partition** (dynamic) to avoid full table rewrite.

```
(silver
  .write.mode('overwrite')
  .option('partitionOverwriteMode','dynamic')
  .parquet('abfss://lake/silver/orders'))
```

- Validate with **row counts** and **checksums** against source.

---

## 9) File Layout on Write

- Partition primarily by **date/time**; maybe add one moderate-cardinality key.
- Target **256--1024 MB** files; set `maxRecordsPerFile` and writer parallelism.
- Periodic **compaction** (Delta `OPTIMIZE`, Iceberg `rewrite_data_files`, Hudi clustering/compaction).

---

## 10) Idempotency, Quality Checks & Error Handling

- **Idempotency**: deterministic outputs for the same inputs (MERGE + partition overwrite; avoid non-deterministic UDFs).
- **DQ**: Great Expectations/Deequ or SQL asserts.

```sql
-- Example assert: no null customer_id in silver
SELECT COUNT(*) FROM silver.orders WHERE customer_id IS NULL;
```

- **Quarantine** bad records to an error table with reason.
- **Retry** only safe sections; avoid duplicate appends.

---

## 11) Configs & Performance Knobs (Batch)

```python
# Parallelism
spark.conf.set('spark.sql.shuffle.partitions', '400')  # tune to cluster size/data volume
# Adaptive Query Execution (AQE)
spark.conf.set('spark.sql.adaptive.enabled', 'true')
# Broadcast joins (threshold in bytes)
spark.conf.set('spark.sql.autoBroadcastJoinThreshold', 104857600)  # 100MB
# Skew mitigation (AQE)
spark.conf.set('spark.sql.adaptive.skewJoin.enabled', 'true')
# Caching
facts.cache(); facts.count()  # materialize
```

- Prefer **broadcast joins** for small dimension tables.
- Repartition by **join keys** before heavy joins to reduce skew.

---

## 12) Orchestration: Airflow/Dagster Skeleton

**Airflow DAG (Python)**

```python
from airflow import DAG
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from datetime import datetime, timedelta

with DAG('orders_daily', start_date=datetime(2025,8,1), schedule='@daily', catchup=True) as dag:
    bronze_to_silver = SparkSubmitOperator(
        application='/opt/jobs/bronze_to_silver.py',
        name='bronze_to_silver',
        application_args=['--run-date', '{{ ds }}']
    )

    silver_to_gold = SparkSubmitOperator(
        application='/opt/jobs/silver_to_gold.py',
        name='silver_to_gold',
        application_args=['--run-date', '{{ ds }}']
    )

    bronze_to_silver >> silver_to_gold
```

---

## 13) Observability & Logging

- **Metrics**: input/output rows, bytes written, task time, shuffle read/write.
- **Accumulators** for counters (e.g., bad rows quarantined).
- **Logs**: log params (run date, source paths, target paths, commit IDs).
- **Lineage**: capture input paths & output table/version.

---

## 14) Project Skeleton & Testing

```
repo/
  jobs/
    bronze_to_silver.py
    silver_to_gold.py
  conf/
    base.conf
  tests/
    test_transforms.py
  README.md
```

**Unit test example (PySpark local)**

```
from transforms import compute_daily_amount

def test_daily_amount(spark):
    df = spark.createDataFrame([
        (1, 10, '2025-08-01', 100.0),
        (2, 10, '2025-08-01', 50.0),
    ], 'order_id long, customer_id long, dt string, amount double')
    out = compute_daily_amount(df)
    rows = { (r.customer_id, r.d, r.daily_amount) for r in out.collect() }
    assert rows == { (10, '2025-08-01', 150.0) }
```

---

## 15) End-to-End Examples (DataFrame & SQL)

**DataFrame pipeline**

```
from pyspark.sql import functions as F
bronze = spark.read.json('abfss://lake/bronze/orders/dt=2025-08-22/*.json')
dim = spark.read.parquet('abfss://lake/silver/dim_customers')

silver = (bronze
  .filter('is_valid = true')
  .join(dim, 'customer_id')
  .groupBy('customer_id', F.to_date('dt').alias('d'))
  .agg(F.sum('amount').alias('daily_amount')))

(silver.write
  .mode('append')
```

```
  .partitionBy('d')
  .parquet('abfss://lake/silver/orders_daily'))
```

**SQL pipeline**

```
CREATE TEMP VIEW bronze AS SELECT * FROM json.`abfss://lake/bronze/orders/dt=2025-08-22/*.json`;
CREATE TEMP VIEW dim AS SELECT * FROM parquet.`abfss://lake/silver/dim_customers`;

INSERT INTO parquet.`abfss://lake/silver/orders_daily`
SELECT o.customer_id, DATE(o.dt) AS d, SUM(o.amount) AS daily_amount
FROM bronze o JOIN dim USING (customer_id)
WHERE o.is_valid = true
GROUP BY o.customer_id, DATE(o.dt);
```

---

## 16) Cheat Sheet

- Prefer **explicit schemas**; avoid inference in production.
- Keep logic **pure** (deterministic) for idempotency; use **MERGE** for upserts.
- Partition by **date**; size files to **256--1024 MB**.
- Turn on **AQE**, tune `shuffle.partitions`, and use **broadcast** for small dims.
- Use **SQL** for relational clarity; **DataFrames** for orchestration/parametrization.
- Validate outputs with **row counts/checksums**; quarantine bad records.

# Thank you

Shwetank Singh
**GritSetGrow - GSGLearn.com**