**Spark DataFrame Pipeline Optimization with DAG, Cache, and Partitioning**

**Objective**

To build a real-world DataFrame transformation pipeline, inspect the DAG using Spark UI, and apply cache/partitioning strategies to **double the performance** of a typical aggregation workload.

**Case Study Background: E-Commerce Orders Analysis**

You are a **data engineer** at an e-commerce company. The company stores its **orders, customers, and products** in Parquet files. Your goal is to build an analytics pipeline that computes **monthly sales KPIs** and deliver **fast insights** for the reporting team.

**📂 Data Structure**

**📁 Input Files (Parquet):**

* /data/orders.parquet
* /data/customers.parquet
* /data/products.parquet

**Sample Schema:**

python

orders: order\_id, customer\_id, product\_id, order\_date, quantity, unit\_price

customers: customer\_id, name, country

products: product\_id, product\_name, category

**Step-by-Step Instructions**

**Step 1: Load the data**

python

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("DF Optimization Case Study").getOrCreate()

orders\_df = spark.read.parquet("/data/orders.parquet")

customers\_df = spark.read.parquet("/data/customers.parquet")

products\_df = spark.read.parquet("/data/products.parquet")

**Step 2: Build the base transformation pipeline**

Create a pipeline to compute monthly sales revenue per country and category.

python

from pyspark.sql.functions import month, year, col, sum as \_sum

# Join orders with customers and products

df = orders\_df.join(customers\_df, "customer\_id") \

.join(products\_df, "product\_id")

# Add year, month, revenue

df = df.withColumn("order\_year", year(col("order\_date"))) \

.withColumn("order\_month", month(col("order\_date"))) \

.withColumn("revenue", col("quantity") \* col("unit\_price"))

# Group by month, country, category

sales\_df = df.groupBy("order\_year", "order\_month", "country", "category") \

.agg(\_sum("revenue").alias("monthly\_revenue"))

**Step 3: View the DAG in Spark UI**

* Run: sales\_df.show()
* Open **Spark UI** → **SQL Tab** → View **DAG Visualization** for the job.
* Notice:
  + Shuffle operations due to joins and groupBy
  + No caching, repeated stages if reused

**Step 4: Optimize with cache()**

If you use sales\_df or df multiple times (e.g., writing to multiple sinks), caching helps:

python

df.cache()

sales\_df = df.groupBy("order\_year", "order\_month", "country", "category") \

.agg(\_sum("revenue").alias("monthly\_revenue"))

sales\_df.cache()

sales\_df.count() # Materialize cache

**Step 5: Repartitioning**

If data is skewed by country or category, repartition:

python

# Repartition by country and category to optimize shuffles

df = df.repartition("country", "category")

Use coalesce(n) before writing to reduce small files if needed.

**Step 6: Performance Comparison (×2 speed-up)**

Compare with and without optimization:

python

import time

# Without cache and repartition

start = time.time()

df\_unoptimized = orders\_df.join(customers\_df, "customer\_id") \

.join(products\_df, "product\_id") \

.withColumn("revenue", col("quantity") \* col("unit\_price")) \

.groupBy("country", "category") \

.agg(\_sum("revenue").alias("monthly\_revenue")) \

.collect()

print("Unoptimized Time:", time.time() - start)

# With cache and repartition

start = time.time()

df\_cached = df.groupBy("country", "category") \

.agg(\_sum("revenue").alias("monthly\_revenue")) \

.collect()

print("Optimized Time:", time.time() - start)

Check execution times and Spark UI. With cache and repartition, you should see ~2× speedup.

**Validation Criteria**

| **Metric** | **Goal** |
| --- | --- |
| DAG stages reduced | Yes |
| Execution time improved | ≥ 2× faster |
| Shuffle read/write stats | Improved (see Spark UI) |
| Caching used | Yes (via .cache()) |
| Partitioning improved | Yes (targeted columns) |

**Summary**

* Built a full Spark DF pipeline with joins, aggregations
* Visualized DAG using Spark UI
* Applied caching and partitioning to eliminate recomputation
* Achieved 2×+ speed-up by reducing shuffle and leveraging memory