**Optimizing Spark Jobs for Scalable Sales Data Analytics**

**Project Context**

In a recent project at a retail e-commerce company, I was responsible for optimizing Spark jobs that processed large-scale sales transactions for demand forecasting and inventory management. The pipeline ingested and transformed billions of sales records daily from multiple sales channels before loading them into Google BigQuery for real-time analytics. However, we faced severe performance bottlenecks due to data skew, expensive shuffling, inefficient joins, and excessive partitions.

This document outlines the challenges faced and the Spark optimization techniques implemented to improve performance and reduce processing time.

**Problem 1: Slow Query Performance Due to Data Skew**

Issue:

* The sales\_df dataset contained highly skewed data, where 80% of sales came from just 10 products.
* Some partitions were overloaded, while others remained empty, leading to task stragglers and long execution times.

Solution: Skew Join Optimization (AQE) & Salting

Before Optimization (Without AQE & Salting)

# AQE Disabled

spark.conf.set("spark.sql.adaptive.enabled", "false")

# Join sales transactions with product data

result\_df = sales\_df.join(products\_df, "product\_id")

result\_df.explain(True)

Observed Issues:

* Spark used a Shuffle Hash Join, causing imbalanced partitions.
* Some tasks took 10x longer than others.
* Execution Time: 1 hour 5 minutes.

After Optimization (With AQE & Salting)

# Enable AQE

spark.conf.set("spark.sql.adaptive.enabled", "true")

spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true") # Skew join optimization

# Add a salt column to distribute data evenly

from pyspark.sql.functions import monotonically\_increasing\_id

sales\_df = sales\_df.withColumn("salt", monotonically\_increasing\_id() % 10)

products\_df = products\_df.withColumn("salt", monotonically\_increasing\_id() % 10)

# Join using both "product\_id" and "salt"

result\_df = sales\_df.join(products\_df, ["product\_id", "salt"])

result\_df.explain(True)

Results:

* AQE dynamically split large partitions into smaller, balanced ones.
* Salting ensured even data distribution.
* Execution Time: Reduced to 20 minutes (3x faster).

**Problem 2: Inefficient Joins Leading to High Shuffle Costs**

Issue:

* Joining 500GB+ of sales data with a small (20MB) reference dataset (store\_locations\_df) caused unnecessary shuffle.

Solution: Broadcast Join

Before Optimization (Without Broadcast)

sales\_df = spark.read.parquet("s3://data/sales/")

store\_locations\_df = spark.read.parquet("s3://data/store\_locations/")

result\_df = sales\_df.join(store\_locations\_df, "store\_id") # Causes shuffle

result\_df.explain(True)

Observed Issues:

* Unnecessary shuffle of 500GB due to Shuffle Hash Join.
* Execution Time: 35 minutes.

After Optimization (With Broadcast)

from pyspark.sql.functions import broadcast

# Broadcast small dataset

optimized\_df = sales\_df.join(broadcast(store\_locations\_df), "store\_id")

optimized\_df.explain(True)

Results:

* Eliminated shuffle overhead by broadcasting the small dataset.
* Execution Time: Reduced to 7 minutes (5x faster).

**Problem 3: Excessive Small Partitions Slowing Down Processing**

Issue:

* Data ingestion led to 1500+ small partitions, degrading performance.

Solution: Repartitioning & Coalesce

Before Optimization (Too Many Partitions)

sales\_df = spark.read.parquet("s3://data/sales/")

sales\_df = sales\_df.dropDuplicates(["order\_id"])

sales\_df.write.mode("overwrite").parquet("s3://processed/sales/")

Observed Issues:

* Too many small partitions led to high shuffle overhead.
* Execution Time: 50 minutes.

After Optimization (Balanced Partitions)

sales\_df = spark.read.parquet("s3://data/sales/").repartition(250)

sales\_df = sales\_df.dropDuplicates(["order\_id"])

sales\_df = sales\_df.coalesce(60)

sales\_df.write.mode("overwrite").parquet("s3://processed/sales/")

Results:

* Balanced parallelism before transformations.
* Reduced shuffle overhead before writing.
* Execution Time: Reduced to 22 minutes (2x faster).

**Problem 4: Inefficient Aggregation Using groupByKey()**

Issue:

* Sales forecasting required aggregating revenue per store.
* Using groupByKey() led to large shuffle writes.

**Solution: Use reduceByKey() Instead of groupByKey()**

Before Optimization (With groupByKey - High Memory Usage)

sales\_rdd = sales\_df.rdd.map(lambda row: (row.store\_id, row.sales\_amount))

aggregated\_rdd = sales\_rdd.groupByKey().mapValues(sum)

Observed Issues:

* groupByKey() transferred all values, increasing memory usage.
* Execution Time: 40 minutes.

After Optimization (With reduceByKey - Optimized Aggregation)

sales\_rdd = sales\_df.rdd.map(lambda row: (row.store\_id, row.sales\_amount))

aggregated\_rdd = sales\_rdd.reduceByKey(lambda a, b: a + b)

Results:

* 50% reduction in intermediate shuffle data.
* Execution Time: Reduced to 18 minutes (2x faster).

**Problem 5: Expensive Recomputations Slowing Down Pipeline**

Issue:

* Repeating transformations on the same dataset caused high computation costs.

**Solution: Cache & Persist**

Before Optimization (Without Cache/Persist)

sales\_df = spark.read.parquet("s3://data/sales/")

sales\_df = sales\_df.withColumn("discounted\_price", sales\_df.price \* 0.9)

sales\_df.show()

sales\_df.agg({"discounted\_price": "avg"}).show()

**Observed Issues:**

* Spark recomputed discounted\_price twice, increasing job execution time.
* Execution Time: 25 minutes.

After Optimization (With Cache/Persist)

# Cache the transformed dataset

sales\_df = sales\_df.withColumn("discounted\_price", sales\_df.price \* 0.9)

sales\_df.cache()

# Now the dataset is cached and reused efficiently

sales\_df.show()

sales\_df.agg({"discounted\_price": "avg"}).show()

**Results:**

* Reduced redundant computation.
* **Execution Time: Reduced to 10 minutes (2.5x faster).**