Apache Spark

Repartition vs Coalesce

Complete Guide with Examples & Diagrams

🚀 Introduction

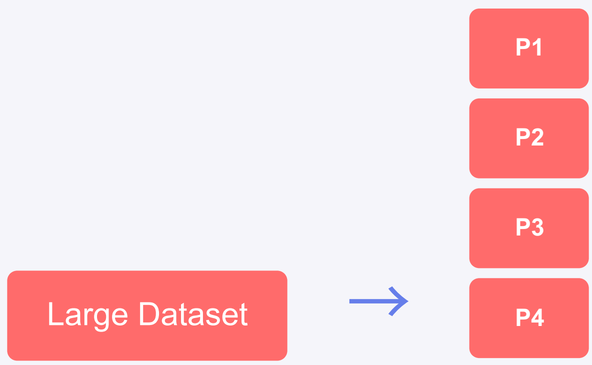
In Apache Spark, managing data partitions is crucial for optimal performance. Two key operations help us control partitions:

What are Partitions?

Partitions are logical divisions of data that can be processed in parallel across different cores or machines in a Spark cluster. Think of them as chunks of your dataset that Spark can work on simultaneously.

Data Partitioning Concept

Original Dataset → Split into Partitions → Process in Parallel



Harshavardhana I
Data Engineer

Repartition

Repartition is a wide transformation that reshuffles data across all partitions in the cluster.

Key Characteristics:

- Performs a **full shuffle** of data across the network
- Can **increase or decrease** the number of partitions
- Ensures **even distribution** of data
- More expensive operation due to network I/O

Visual Example: Repartition from 4 to 3 partitions

Before Repartition (4 partitions):

100 records

150 records

80 records

170 records

↓ repartition(3) ↓

After Repartition (3 partitions):

167 records

167 records

Harshavardhana I
Data Engineer

166
records

Code Examples

```
# Python/PySpark Example from pyspark.sql import
SparkSession spark =
SparkSession.builder.appName("RepartitionExample").getOrCre
# Create a DataFrame df =
spark.range(1000000).toDF("id") # Check current number
of partitions print(f"Original partitions:
{df.rdd.getNumPartitions()}") # Repartition to 8
partitions df_repartitioned = df.repartition(8)
print(f"After repartition:
{df_repartitioned.rdd.getNumPartitions()}") #
Repartition by column (for better data locality)
df_by_column = df.repartition("id")
```

```
// Scala Example import
org.apache.spark.sql.SparkSession val spark =
SparkSession.builder() .appName("RepartitionExample")
.getOrCreate() val df = spark.range(1000000).toDF("id")
// Repartition to specific number val repartitionedDF =
df.repartition(8) // Repartition by column val
repartitionedByCol = df.repartition(col("id"))
```

Coalesce

Coalesce is a narrow transformation that reduces the number of partitions by merging adjacent partitions.

Key Characteristics:

- Performs **minimal data movement**
- Can **only decrease** the number of partitions
- May result in **uneven distribution**
- More efficient than repartition for reducing partitions

Visual Example: Coalesce from 4 to 2 partitions

Before Coalesce (4 partitions):

100 records

150 records

80 records

170 records

↓ coalesce(2) ↓

After Coalesce (2 partitions):

250 records

250 records

Note: Adjacent partitions are merged with minimal shuffling

Code Examples

```
# Python/PySpark Example from pyspark.sql import
SparkSession spark =
SparkSession.builder.appName("CoalesceExample").getOrCreate
# Create a DataFrame with many partitions df =
spark.range(1000000).repartition(16) # Check current number
of partitions print(f"Original partitions:
{df.rdd.getNumPartitions()}") # Coalesce to fewer
partitions df_coalesced = df.coalesce(4) print(f"After
coalesce: {df_coalesced.rdd.getNumPartitions()}") #
Common use case: Single file output df_single =
df.coalesce(1)
df_single.write.mode("overwrite").csv("output_single_file")
```

```
// Scala Example import
org.apache.spark.sql.SparkSession val spark =
SparkSession.builder() .appName("CoalesceExample")
.getOrCreate() val df =
spark.range(1000000).repartition(16) // Coalesce to
fewer partitions val coalescedDF = df.coalesce(4) //
Write to single file df.coalesce(1).write
.mode("overwrite") .option("header", "true")
.csv("output_path")
```

⚖️ Detailed Comparison

Aspect	Repartition	Coalesce
Data Movement	**Full shuffle across network**	**Minimal data movement**
Partition Count	**Can increase/decrease**	**Can only decrease**
Data Distribution	**Even distribution**	**May be uneven**
Performance Cost	**Higher (due to shuffle)**	**Lower (minimal shuffle)**
Network I/O	**High**	**Low**
Use Case	Balancing load, increasing parallelism	Reducing output files, final optimization

When to Use Each

Use Repartition When:

- You need to **increase** partitions
- Data is **highly skewed**
- You want **even distribution**
- Preparing for **intensive operations**
- Partitioning by **specific columns**

Use Coalesce When:

You need to **reduce** partitions
Creating **fewer output files**
Final stage **optimization**
Minimizing **network overhead**
Small datasets after **filtering**

🌍 Real-world Examples

Example 1: Processing Large Log Files

```
# Scenario: Processing 100GB of log files logs_df =
spark.read.text("hdfs://large_logs/") # Initial
partitions might be too few for parallel processing
print(f"Initial partitions:
{logs_df.rdd.getNumPartitions()}") # e.g., 10 #
Repartition for better parallelism logs_df =
logs_df.repartition(50) # More partitions = more
parallelism # Process the data processed_logs =
logs_df.filter(logs_df.value.contains("ERROR")) # After
filtering, data is much smaller - coalesce for fewer
output files
processed_logs.coalesce(5).write.mode("overwrite").parquet(
```

Example 2: ETL Pipeline Optimization

```
# Scenario: Daily sales data processing sales_df =
spark.read.parquet("daily_sales/") # Repartition by date
for better data locality sales_partitioned =
sales_df.repartition("sale_date") # Perform aggregations
daily_summary = sales_partitioned.groupBy("sale_date",
"store_id").sum("amount") # Coalesce before writing
summary (small result set)
daily_summary.coalesce(1).write.mode("overwrite").csv("dail
```

📋 Best Practices

🎯 General Guidelines

- ✓ **Monitor partition sizes:** Aim for 100-200MB per partition
- ✓ **Consider your cluster size:** Number of partitions should be 2-3x the number of cores
- ✓ **Use repartition sparingly:** Only when you need even distribution or more partitions
- ✓ **Coalesce before writes:** Reduce small files problem in output
- ✓ **Test performance:** Always benchmark both approaches for your use case

⚠ Common Pitfalls to Avoid

- ✓ **Over-partitioning:** Too many small partitions increase overhead
- ✓ **Under-partitioning:** Too few large partitions reduce parallelism
- ✓ **Unnecessary repartitioning:** Don't repartition if current distribution is good
- ✓ **Coalescing to 1:** Can create bottlenecks, use only for small datasets
- ✓ **Ignoring data skew:** Some partitions much larger than others

⚡ Performance Tips

Optimization Strategies:

1. Check Partition Distribution

```
# Check partition sizes df.rdd.mapPartitions(lambda
iterator: [sum(1 for _ in iterator)]).collect() #
Check data distribution per partition
df.rdd.glom().map(len).collect()
```

2. Optimal Partition Count Formula

```
# Rule of thumb calculation total_cores =
spark.sparkContext.defaultParallelism
optimal_partitions = total_cores * 2 # to 3 # For
large datasets data_size_gb = 100 partition_size_mb
= 128 optimal_partitions = (data_size_gb * 1024) //
partition_size_mb
```

3. Smart Coalescing

```
# Instead of coalesce(1), use reasonable number #
Bad: Creates bottleneck
df.coalesce(1).write.parquet("output") # Good:
Balanced approach target_partitions = max(1,

df.rdd.getNumPartitions() // 4)
df.coalesce(target_partitions).write.parquet("output")
```

Harshavardhana I
Data Engineer