Repartition vs Coalesce

Introduction:

As Data Engineers, we often focus on how data *moves* — but equally important is **how it's distributed** across the cluster.

That's where `repartition()` and `coalesce()` come in!

Both functions control how Spark divides data into partitions — directly impacting **performance, parallelism, and shuffle operations**.

- Use `repartition(n)` to INCREASE partitions or to reshuffle data using new keys. It triggers a full shuffle.
- Use `coalesce(n)` to DECREASE partitions cheaply by collapsing adjacent partitions. It avoids a full shuffle by default.
- Prefer `repartition` for correctness when keys matter or when increasing parallelism. Prefer `coalesce` at the end of pipelines before writes when only reducing file/partition count.

Why data distribution matters

- Controls parallelism and throughput
- Impacts shuffle size, spill, and cluster cost
- Influences file counts and small-file problems on cloud storage and Delta Lake

---

API quick reference

```
# Increase partitions or reshuffle with/without keys
(df.repartition(200))                # round-robin shuffle

(df.repartition(200, "user_id", "date")) # hash-partition by key

# Decrease partitions without a full shuffle
(df.coalesce(8))                     # collapse adjacent partitions
```

```
# Coalesce but still force a shuffle (rarely needed)
(df.coalesce(8, shuffle=True))
```

How they work under the hood

- `repartition`:
  - Creates a new partitioner. With keys, does a hash-partition by the key tuple. Without keys, round-robin.
  - Always causes a wide dependency and a full shuffle of rows.
- `coalesce`:
  - Narrows the dependency graph by merging existing partitions into fewer tasks.
  - No shuffle by default, which can preserve skew.

Choosing the right tool

- Increase parallelism or rebalance skewed data → `repartition`
- Prepare to write fewer output files → `coalesce` near the sink
- Partition by business keys for downstream joins/bucketing → `repartition(..., keys...)`
- You suspect severe skew and want to smooth it → `repartition` (optionally with salting)

Practical patterns

1) After wide transforms, before heavy joins

```
# Rebalance before a big join
left  = left.repartition(400, "join_key")
right = right.repartition(400, "join_key")
joined = left.join(right, "join_key", "inner")
```

2) Right before writing to avoid small files

```
# Collapse output files for efficient downstream reads
out = df.coalesce(16)
(out
  .write
  .format("delta")
  .mode("overwrite")
  .option("dataChange", "false")  # for optimize-append-like patterns
  .save(path))
```

3) Ingest with drift, then normalize and repartition

```
bronze = spark.readStream.format("cloudFiles").load(raw_path)
silver = (bronze
    .selectExpr("... normalized columns ...")
    .repartition(200)  # smooth bursty ingestion
)
```

Skew and hotspots

- Coalesce can preserve hotspots because it merges partitions as-is.
- Repartition redistributes rows, smoothing hotspots but at shuffle cost.
- For extreme skew, consider key salting:

```
from pyspark.sql import functions as F
salt_buckets = 8
salted = df.withColumn("join_key_salted", F.concat_ws("#", "join_key", (F.rand()*salt_buckets).cast("int")))
# repartition by salted key, then desalt post-join if needed
salted = salted.repartition(400, "join_key_salted")
```

Partition counts: rules of thumb

- Target partitions ≈ 2–4 × total CPU cores available to your job
- For cloud data lakes, aim for 256–1024 MB per output file for balanced read performance
- Start higher, then `coalesce` down right before writes to control file sizes

---

Cost and performance

- `repartition` costs more now (shuffle) but may save later by reducing skew and improving join performance
- `coalesce` is cheap now but can tax downstream stages if skew persists
- Measure with the Spark UI: look at task time variance and shuffle read size

---

Delta Lake and medallion flows

- Bronze: bursts and schema drift often lead to uneven distribution → tolerate, then normalize
- Silver: cleanse and `repartition(keys)` before dimensional joins
- Gold: `coalesce(n)` before publish to control file sizes and scan efficiency

---

Common pitfalls

- Calling `coalesce(1)` early causes single-threaded bottlenecks
- Over-partitioning writes create thousands of tiny files → expensive listing/metadata ops
- Forgetting to align partitioning on both sides of a large join

---

Quick checklist

- Need to increase partitions or rebalance? → `repartition`
- Need fewer output files? → `coalesce` near the sink
- Joining on key(s)? → `repartition(..., keys...)` on both sides
- Seeing skew? → `repartition` and consider salting

- Writing to Delta? → coalesce to target file sizes, then schedule OPTIMIZE if available

---

Interview-ready sound bites

- "Repartition shuffles to a new partitioner. Coalesce narrows by merging without a shuffle."
- "Use repartition for correctness and parallelism. Use coalesce for output file control."
- "Fix skew with repartition and optional salting."

Summary

> Repartition vs Coalesce in PySpark:

> • Repartition = full shuffle, increases partitions, can hash by keys, smooths skew

> • Coalesce = no shuffle, decreases partitions, great before writes

> Rule of thumb: Repartition for balance and joins. Coalesce for output control. Measure in Spark UI, aim 256–1024 MB files.