Understanding Shuffle in Spark

Karthik Kondpak
9989454737

Day 6 — Spark Optimization Topic

6. Understanding Shuffle in Spark

Shuffle isthe **most expensive operation** inSpark.

It is the process where Spark **moves data across executors** to group, sort, or join it.

If your Spark job is slow → **90% chance it is due to shuffle.**

What Is Shuffle?

Shuffle = **Redistributionofdata across partitions** based on a key.

It forces Spark to:

✓ Write data to disk

✓ Transfer data across network

✓ Read it again on target executors

This makes it slow,heavy,andcostly.

When Does Shuffle Happen?

Spark shuffles**wheneveritneedsmatchingkeys to be on the same partition**.

🔸 **Common operations that trigger shuffle:**

Operation	Why Shuffle Happens
`groupBy()`	All rows with same key must meet on one partition
`join()`	Both datasets must align partitions by join key
`distinct()`	Spark groups to remove duplicates
`dropDuplicates()`	Same reason as distinct
`repartition()`	Explicitly reshuffles data
`orderBy()` / `sort()`	Global sort requires moving data
`window functions`	Often require partition alignment

Examples of Shuffle (with code)

GroupBy causes shuffle

```
df.groupBy("customer_id").sum("amount")
```

All rows of each customer_id must come together → shuffle.

Join causes shuffle

```
df1.join(df2, "txn_id")
```

Both datasets move data to align by `txn_id`.

Repartition triggers shuffle

```
df.repartition(50, "state")
```

Spark redistributes entire dataset → heavy shuffle.

Why Shuffle Is Expensive?

Shuffle =

Disk I/O (write + read)

Network transfer

Serialization / deserialization

Large memory usage (spill)

It can slow a job from **seconds → minutes**.

Effects of Shuffle

If shuffle is heavy, you will see:

- Slow stages in Spark UI
- Many spilled tasks
- High GC time
- Straggler tasks

- Memory pressure

How to Reduce Shuffle

Here are the **top strategies professional data engineers use**:

Use Broadcast Join When One Table Is Small

Avoid shuffle-heavy Sort-Merge Join.

```
from pyspark.sql.functions import broadcast
df1.join(broadcast(df2), "id")
```

Use Proper Partitioning Columns

Good:

```
df.repartition("state")
```

Bad:

```
df.repartition(100)
```

(Blind repartition → unnecessary shuffle)

Avoid Unnecessary `distinct()` / `dropDuplicates()`

Prefer:

```
df.select("id").agg(countDistinct("id"))
```

Use Filter Early (Predicate Pushdown + Filter Pushdown)

Reduces data before grouping/joining → less shuffle.

Cache Only When Needed

Avoid recomputing = avoids repeating shuffle.

But:

Use caching **only** when reuse happens.

Increase Parallelism Only When Required

```
spark.conf.set("spark.sql.shuffle.partitions", 200)
```

Too high = tiny partitions → overhead

Too low = huge partitions → slow shuffle

Tune based on cluster + data size.

Use Bucketing for Repeated Joins

If you join same two huge tables daily → bucket them.

```
df.write.bucketBy(50,
"customer_id").sortBy("customer_id")
```

Scenario Example

Dataset: **/delta/retail_transactions**

customer_id	state	amount	date
101	Maharashtra	1200	2025-01-10
102	Karnataka	500	2025-01-11
101	Maharashtra	900	2025-01-11

Goal: Total spend per state.

```
df.groupBy("state").sum("amount")
```

This triggers shuffle because Spark must bring all Maharashtra rows together.

Let's build your Data Engineering journey together!

Call us directly at: 9989454737

https://seekhobigdata.com/