# PySpark Repartition() vs Coalesce()

Spark splits data into partitions and computation is done in parallel for each partition. It is very important to understand how data is partitioned and when you need to manually modify the partitioning to run spark applications efficiently.

Now, diving into our main topic i.e **Repartitioning v/s Coalesce**

## What is Coalesce?

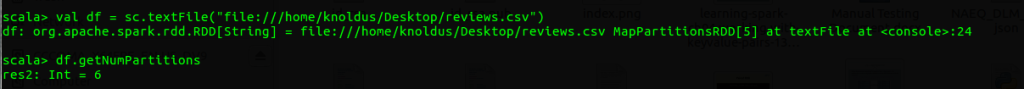
The coalesce method **reduces** the **number of partitions** in a DataFrame. Coalesce **avoids full shuffle**, instead of creating new partitions, it shuffles the data using Hash Partitioner (Default), and **adjusts into existing partitions**, this means it can **only decrease** the number of partitions.

## What is Repartitioning?

The repartition method can be used to either **increase** or **decrease** the **number of partitions** in a DataFrame. Repartition is a **full Shuffle** operation, whole data is taken out from existing partitions and **equally distributed** into **newly formed partitions**.

#### Where to use what?

Let’s look at the below example for the answer.



Now, if I manually pass the number of partitions to 10, see how the data gets distributed:

https://deejeyhome.files.wordpress.com/2019/12/screenshot-from-2019-12-21-14-24-26.png?w=887https://deejeyhome.files.wordpress.com/2019/12/screenshot-from-2019-12-21-14-24-12.png?w=849

Comparatively, **coalesce** took less time as compared with **repartitioning**. And the data gets partitioned as below:

|  |  |
| --- | --- |
| **Repartitioning** | **Coalesce** |
| 19M repartition/part-00000 19M repartition/part-00001  19M repartition/part-00002 19M repartition/part-00003 19M repartition/part-00004 19M repartition/part-00005 19M repartition/part-00006 19M repartition/part-00007 19M repartition/part-00008  19M repartition/part-00009 | 33M coalesce/part-00000  29M coalesce/part-00001  30M coalesce/part-00002  31M coalesce/part-00003  32M coalesce/part-00004  33M coalesce/part-00005 |
|  |  |

If you observe above table when **repartitioned**, data over all the partitions are **equally** **populated**, but when we used **coalesce** the data is **not equally distributed**.

Also, if you observed above coalesce didn’t partition your data to **10 partitions** instead it created **6 partitions**. That means even if you provide a large number of partitions, it partitions your data to the default one in the above case it is **6**.

**Now we understand the behavior and hence back to our initial question, where to use which function?**

**Coalesce use case:** we pass in all 10 above partitions into our RDD and perform some action, the partition which processes the file **part-00000** will finish first followed by others but the executor with **part-00005** will be still running meanwhile 1st executor will be idle. **Hence, the load is not balanced on executors equally.**

**Repartition use case:** All the **executors finish** the job at the same time, and the **resources** are **consumed** **equally** because all input partitions have the same size.

## Summary

* If you have loaded a dataset, includes huge data, and a lot of transformations that need an equal distribution of load on executors, you need to use **Repartition**.
* Once all the transformations are applied and you want to save all the data into fewer files(no. of files = no.of partitions) instead of many files, use **coalesce**.