**Spark Partitioning & Partition Understanding**

Spark/PySpark partitioning is a way to split the data into multiple partitions so that you can execute transformations on multiple partitions in parallel which allows completing the job faster. You can also write partitioned data into a file system (multiple sub-directories) for faster reads by downstream systems.

Spark has several partitioning methods to achieve parallelism, based on your need, you should choose which one to use.

| SPARK DATAFRAME PARTITIONING METHODS (SCALA) | KEY POINTS |
| --- | --- |
| repartition(numPartitions : scala.Int) | Uses RoundRobinPartitioning |
| repartition(partitionExprs : Column\*) | Uses HashPartitioner |
| repartition(numPartitions : scala.Int, partitionExprs : Column\*) | partition = hash(partitionExprs) % numPartitions |
| coalesce(numPartitions : scala.Int) | Use only to reduce the number of partitions. |
| repartitionByRange(partitionExprs : Column\*) | Uses rangepartitioning. Ideal to use on numeric columns. |
| repartitionByRange(numPartitions : scala.Int, partitionExprs : Column\*) |  |
| partitionBy(colNames : \_root\_.scala.Predef.String\*) | Use to write the data into sub-folder |

**Note:** partitionBy() is a method from DataFrameWriter class, all others are from DataFrame.

**1. Understanding Spark Partitioning**

* By default, Spark/PySpark creates partitions that are equal to the number of CPU cores in the machine.
* Data of each partition resides in a single machine.
* Spark/PySpark creates a task for each partition.
* [Spark Shuffle operations](https://sparkbyexamples.com/spark/spark-shuffle-partitions/) move the data from one partition to other partitions.
* Partitioning is an expensive operation as it creates a data shuffle (Data could move between the nodes)
* By default, DataFrame shuffle operations create 200 partitions.

Spark/PySpark supports partitioning in memory (RDD/DataFrame) and partitioning on the disk (File system).

**Partition in memory:** You can partition or repartition the DataFrame by calling [repartition() or coalesce()](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/) transformations.

**Partition on disk:** While writing the PySpark DataFrame back to disk, you can choose how to partition the data based on columns by using partitionBy() of pyspark.sql.DataFrameWriter. This is similar to [Hives partitions](https://sparkbyexamples.com/apache-hive/hive-partitions-explained-with-examples/).

**2. Spark Partitioning Advantages**

As you are aware Spark is designed to process large datasets 100x faster than traditional processing, this wouldn’t have been possible without partitions. Below are some of the advantages of using Spark partitions on memory or on disk.

* Fast accessed to the data.
* Provides the ability to perform an operation on a smaller dataset.

Partitioning at rest (disk) is a feature of many databases and data processing frameworks and it is key to make reads faster.

**3. Default Spark Partitions & Configurations**

Spark by default partitions data based on a number of factors, and the factors differ were you running your job on and what mode.

**3.1 Local mode**

When you running on local in standalone mode, Spark partitions data into the number of CPU cores you have on your system or the value you specify at the time of creating SparkSession object

import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com') \

.master("local[5]").getOrCreate()

Python

Copy

The above example provides local[5] as an argument to master() method meaning to run the job locally with 5 partitions. Though if you have just 2 cores on your system, it still creates 5 partition tasks.

df = spark.range(0,20)

print(df.rdd.getNumPartitions())

Python

Copy

Above example yields output as 5 partitions.

**3.2 HDFS cluster mode**

When you running Spark jobs on the Hadoop cluster the default number of partitions is based on the following.

* On the HDFS cluster, by default, Spark creates one Partition for each block of the file.
* In Version 1 Hadoop the HDFS block size is 64 MB and in Version 2 Hadoop the HDFS block size is 128 MB
* Total number of cores on all executor nodes in a cluster or 2, whichever is larger

For example if you have 640 MB file and running it on Hadoop version 2, creates 5 partitions with each consists on 128 MB blocks (5 blocks \* 128 MB = 640 MB). If you repartition to 10 then it creates 2 partitions for each block.

**3.3 Spark configuration**

* spark.default.parallelism configuration default value set to the number of all cores on all nodes in a cluster, on local it is set to a number of cores on your system.
* spark.sql.shuffle.partitions configuration default value is set to **200** and it is used when you call shuffle operations like <a href="https://sparkbyexamples.com/spark/spark-dataframe-union-and-union-all/">union()</a>, <a href="https://sparkbyexamples.com/spark/using-groupby-on-dataframe/">groupBy()</a>, <a href="https://sparkbyexamples.com/spark/spark-sql-dataframe-join/">join()</a> and many more. This property is available only in DataFrame API but not in RDD.

You can change the values of these properties through programmatically using the below statement.

spark.conf.set("spark.sql.shuffle.partitions", "500")

Python

Copy

You can also set the partition value of these configurations using spark-submit command.

./bin/spark-submit --conf spark.sql.shuffle.partitions=500 --conf spark.default.parallelism=500

Python

Copy

**4. Dynamically Changing Spark Partitions**

When you create an RDD/DataFrame from a file/table, based on certain parameters Spark creates them with a certain number of partitions and it also provides a way to change the partitions runtime in memory and options to partition based on one or multiple columns while writing to disk.

**4.1 repartition() & coalesce()**

While working with partition data we often need to increase or decrease the partitions based on data distribution. Methods repartition() and coalesce() helps us to repartition.

You can find the dataset explained in this article at [GitHub zipcodes.csv file](https://github.com/spark-examples/pyspark-examples/blob/master/resources/simple-zipcodes.csv)

val rdd = spark.sparkContext.parallelize((0,20))

print("From local[5]"+rdd.getNumPartitions())

val rdd1 = spark.sparkContext.parallelize((0,25), 6)

print("parallelize : "+rdd1.getNumPartitions())

#DataFrame repartition()

val df=spark.read.option("header",True) \

.csv("/tmp/resources/simple-zipcodes.csv")

println(df.rdd.getNumPartitions())

#Change DataFrame partitions to 10

newDF=df.repartition(10)

println(newDF.rdd.getNumPartitions())

Scala

Copy

**Note:** When you want to reduce the number of partitions, It is recommended to use [PySpark coalesce() over repartition()](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/) as it uses fewer resources due to less number of shuffles it takes.

**4.2 partitionBy()**

Spark partitionBy() is a function of pyspark.sql.DataFrameWriter class which is used to partition based on one or multiple column values while writing DataFrame to Disk/File system.

When you write Spark DataFrame to disk by calling partitionBy(), PySpark splits the records based on the partition column and stores each partition data into a sub-directory.

#partitionBy()

df.write.option("header",True) \

.partitionBy("state") \

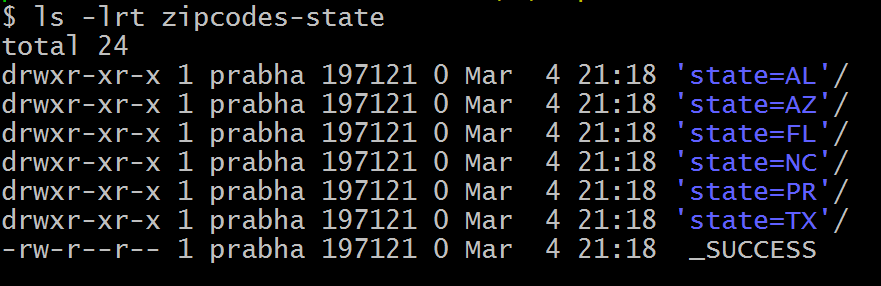
.mode("overwrite") \

.csv("/tmp/zipcodes-state")

Python

Copy

On our DataFrame, we have a total of 6 different states hence, it creates 6 directories as shown below. The name of the sub-directory would be the partition column and its value (partition column=value).



**4.3 partitionBy() Multiple Columns**

You can also create partitions on multiple columns using Spark partitionBy(). Just pass columns you want to partition as arguments to this method.

#partitionBy() multiple columns

df.write.option("header",True) \

.partitionBy("state","city") \

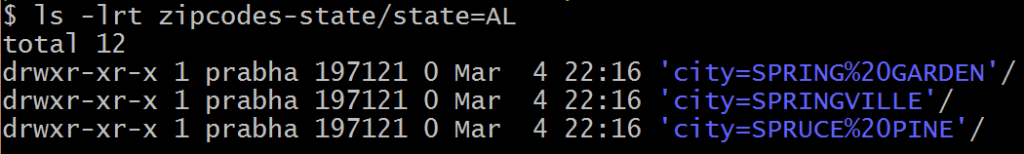
.mode("overwrite") \

.csv("/tmp/zipcodes-state")

Python

Copy

It creates a folder hierarchy for each partition; we have mentioned the first partition as state followed by city hence, it creates a city folder inside the state folder (one folder for each city in a state).



**4.4 repartitionByRange() – Range Partition**

Below is a range partition example using repartitionByRange() transformation.

val data = Seq((1,10),(2,20),(3,10),(4,20),(5,10),

(6,30),(7,50),(8,50),(9,50),(10,30),

(11,10),(12,10),(13,40),(14,40),(15,40),

(16,40),(17,50),(18,10),(19,40),(20,40)

)

import spark.sqlContext.implicits.\_

val dfRange = data.toDF("id","count")

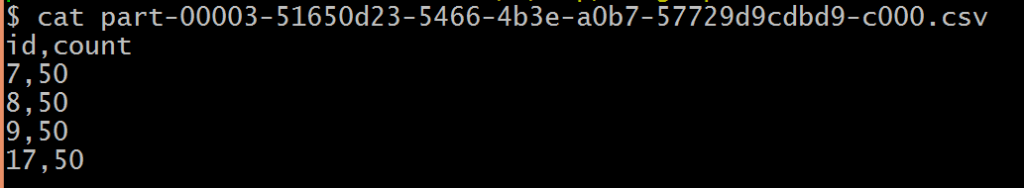
.repartitionByRange(5,col("count"))

dfRange.write.option("header",true).csv("/tmp/range-partition")

Scala

Copy

Let’s check data for one partition.



Above, all data for count=50 are in one partition.

**5. How to Choose Spark Partition Column?**

When using partitionBy(), you have to be very cautious with the number of partitions it creates, as having too many partitions creates too many sub-directories in a directory which brings unnecessarily and overhead to NameNode (if you are using Hadoop) since it must keep all metadata for the file system in memory.

Let’s assume you have a US census table that contains zip code, city, state, and other columns. Creating a partition on the state, splits the table into around 50 partitions, when searching for a zipcode within a state (state=’CA’ and zipCode =’92704′) results in faster as it needs to scan only in a **state=CA** partition directory.

Partition on zipcode may not be a good option as you might end up with too many partitions.

Another good example of partition is on the Date column. Ideally, you should partition on Year/Month but not on a date.

**6. Too Many Partitions Good?**

* If you are a beginner, you would think too many partitions will boost the [Spark Job Performance](https://sparkbyexamples.com/spark/spark-performance-tuning/) actually, it won’t and it’s overkill.
* Spark has to create one task per partition and most of the time goes into creating, scheduling, and managing the tasks then executing.

**7. Too Few Partitions Good?**

* Too few partitions are not good as well, as you may not fully utilize your cluster resources.
* Less parallelism
* Applications may run longer as each partition takes more time to complete.