What is the difference between PySpark’s repartition() and coalesce() functions?

Authored by

stats writer

June 24, 2024

PySpark's repartition() and coalesce() functions are two methods used for partitioning data in a distributed manner. The main difference between these two functions is that repartition() can increase or decrease the number of partitions, while coalesce() can only decrease the number of partitions. Repartition() shuffles and redistributes the data evenly across the new partitions, which can be useful for evenly distributed data or for performing operations that require a specific number of partitions. On the other hand, coalesce() simply merges existing partitions without shuffling the data, making it more efficient for operations that do not require a specific number of partitions. Overall, repartition() is best suited for evenly distributing data, while coalesce() is better for reducing the number of partitions without causing data shuffling.

In PySpark, the choice between `repartition()` and `coalesce()` functions carries importance in optimizing performance and resource utilization. These methods play pivotal roles in reshuffling data across partitions within a DataFrame, yet they differ in their mechanisms and implications.

In simple words, repartition() increases or decreases the partitions, whereas coalesce() only decreases the number of partitions efficiently. In this article, you will learn the difference between PySpark repartition vs coalesce with examples.

One important point to note is PySpark repartition() and `coalesce()` are **very expensive operations** as they **shuffle the data across many partitions**; hence, try to minimize using these as much as possible.

1. PySpark RDD Repartition() vs Coalesce()

Let's create an RDD with partitions and will use this to repartition() and coalesce()

```
# Create spark session with local
rdd = spark.sparkContext.parallelize(range(0,20))
print("From local : "+str(rdd.getNumPartitions()))
```

```
# Use parallelize with 6 partitions
rdd1 = spark.sparkContext.parallelize(range(0,25), 6)
print("parallelize : "+str(rdd1.getNumPartitions()))

rddFromFile = spark.sparkContext.textFile("src/main/resources/test.txt",10)
print("TextFile : "+str(rddFromFile.getNumPartitions()))
```

The above example yields the below output.

```
# Output:
```

```
From local : 5
Parallelize : 6
TextFile : 10
```

The `sparkContext.parallelize()` method in PySpark is used to parallelize a collection into a resilient distributed dataset (RDD). In the given example, `Range(0,20)` creates a range of numbers from 0 to 19 (inclusive). The second argument, `6`, specifies the number of partitions into which the data should be divided.

Let's write this to the file and check the data. Note that your partitions might have different records.

```
rdd1.saveAsTextFile("/tmp/partition")
```

#Writes 6 part files, one for each partition
Partition 1 : 0 1 2
Partition 2 : 3 4 5
Partition 3 : 6 7 8 9
Partition 4 : 10 11 12
Partition 5 : 13 14 15
Partition 6 : 16 17 18 19

1.1 RDD repartition()

`repartition()` is a transformation method available on RDDs (Resilient Distributed Datasets) that redistributes data across a specified number of partitions. When you call `repartition(n)`, where `n` is the desired number of partitions, Spark reshuffles the data in the RDD into exactly `n` partitions.

If you increase/decrease the number of partitions using `repartition()`, Spark will perform a full shuffle of the data across the cluster, which can be an expensive operation, especially for large datasets.

```
# Using repartition
rdd2 = rdd1.repartition(4)
print("Repartition size : "+str(rdd2.getNumPartitions()))
rdd2.saveAsTextFile("/tmp/re-partition")
```

The result shows a "Repartition size" of 4, indicating that the data has been redistributed across partitions. This operation involves a full shuffle, which can be quite costly, especially when handling extremely large datasets in the billions or trillions.

```
# Output:
Partition 1 : 1 6 10 15 19
Partition 2 : 2 3 7 11 16
Partition 3 : 4 8 12 13 17
Partition 4 : 0 5 9 14 18
```

1.2 RDD coalesce()

In PySpark, `coalesce()` is a transformation method available on RDDs (Resilient Distributed Datasets) that reduces the number of partitions without shuffling data across the cluster. When you call `coalesce(n)`, where `n` is the desired number of partitions, Spark merges existing partitions to create `n` partitions.

```
# Using coalesce()
rdd3 = rdd1.coalesce(4)
print("Repartition size : "+str(rdd3.getNumPartitions()))
rdd3.saveAsTextFile("/tmp/coalesce")
```

If we compare the following output with section 1, it becomes apparent that partition 3 has been relocated to partition 2, and partition 6 has been relocated to partition 5. This data movement has occurred between only two partitions.

```
# Output:
Partition 1 : 0 1 2
Partition 2 : 3 4 5 6 7 8 9
Partition 4 : 10 11 12
Partition 5 : 13 14 15 16 17 18 19
```

2. PySpark DataFrame repartition() vs coalesce()

Like RDD, you can't specify the partition/parallelism while creating DataFrame. DataFrame by default internally uses the methods specified in Section 1 to determine the default partition and splits the data for parallelism.

If you are not familiar with DataFrame, I will recommend learning it by following PySpark DataFrame Tutorial before proceeding further with this article.

```
# DataFrame example
```

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com')
.master("local").getOrCreate()
```

df=spark.range(0,20)
print(df.rdd.getNumPartitions())

df.write.mode("overwrite").csv("c:/tmp/partition.csv")

The above example creates 5 partitions as specified in `master("local")` and the data is distributed across all these 5 partitions.

```
# Output:
Partition 1 : 0 1 2 3
Partition 2 : 4 5 6 7
Partition 3 : 8 9 10 11
Partition 4 : 12 13 14 15
Partition 5 : 16 17 18 19
```

2.1 DataFrame repartition()

The DataFrame `repartition()` method in PySpark redistributes (increase or decrease partitions) data evenly across a specified number of partitions, optimizing parallelism and resource usage. It triggers a full shuffle of data and is useful for adjusting the partitioning scheme for downstream operations like joins and aggregations.

This example expands the number of partitions from 5 to 6 by redistributing data across all partitions.

```
# DataFrame repartition
df2 = df.repartition(6)
print(df2.rdd.getNumPartitions())
```

Just increasing 1 partition results in data movements from all partitions.

```
# Output:
Partition 1 : 14 1 5
Partition 2 : 4 16 15
```

```
Partition 3 : 8 3 18
Partition 4 : 12 2 19
Partition 5 : 6 17 7 0
Partition 6 : 9 10 11 13
```

And, even decreasing the partitions also results in moving data from all partitions. hence when you wanted to decrease the partition recommendation is to use coalesce()/

2.2 DataFrame coalesce()

Spark DataFrame `coalesce()` is used only to decrease the number of partitions. This is an optimized or improved version of repartition() where the movement of the data across the partitions is fewer using coalesce.

```
# DataFrame coalesce
df3 = df.coalesce(2)
print(df3.rdd.getNumPartitions())
```

This yields output 2 and the resultant partition looks like

```
# Output:
Partition 1 : 0 1 2 3 8 9 10 11
Partition 2 : 4 5 6 7 12 13 14 15 16 17 18 19
```

Since we are reducing 5 to 2 partitions, the data movement happens only from 3 partitions and it moves to remain 2 partitions.

3. Default Shuffle Partition

In PySpark, the default shuffle partition refers to the number of partitions that Spark uses when performing shuffle operations, such as joins, group-bys, and aggregations. Shuffle operations involve redistributing data across different nodes in the cluster, which can be computationally expensive and affect performance.

By default, Spark sets the number of shuffle partitions to 200. This default value is controlled by the configuration parameter `spark.sql.shuffle.partitions`. You can adjust this setting based on the size of your data and the resources of your cluster to optimize performance.

```
# Default shuffle partition count
df4 = df.groupBy("id").count()
print(df4.rdd.getNumPartitions())
```

Post shuffle operations, you can change the partitions either using coalesce() or repartition().

4. PySpark repartition vs coalesce

Following are differences in a table format.

Feature	Repartition	Coalesce
Description	Adjusts the number of partitions, redistributing data across the specified number of partitions.	Reduces the number of partitions without shuffling data, merging existing partitions.
Full Shuffle	Yes	No
Expensiveness	Can be expensive, especially for large datasets, as it involves a full shuffle of data.	Less expensive than repartitioning, as it minimizes data movement by only combining partitions when possible.
Data Movement	Distributes data across partitions evenly, potentially leading to balanced partition sizes.	May result in imbalanced partition sizes, especially when reducing the number of partitions.
Use Cases	Useful when changing the number of partitions or evenly distributing data across partitions.	Useful when decreasing the number of partitions without incurring the cost of a full shuffle.

Conclusion

In this PySpark repartition() vs coalesce() article, you have learned how to create an RDD with partition, repartition the RDD using coalesce(), repartition DataFrame using repartition() and coalesce() methods, and learned the difference between repartition and coalesce.

Related Articles

Reference

Happy Learning !!