**Introduction**

Spark 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.

In this tutorial we will explore Spark partitioning technics and explain the challenges of partitioning large datasets on disk or in memory.

**What Are Partitions ?**

Partition is a logical division of the data , this idea is derived from Map Reduce (split). Logical data is specifically derived to process the data. Small chunks of data also it can support scalability and speed up the process. Input data, intermediate data, and output data everything is partitioned RDD.

**Advantages of Data Partitioning**

* **Improved performance:** By dividing data into smaller partitions, it can be processed in parallel across multiple machines, leading to faster processing times and improved performance.
* **Scalability:** Partitioning allows for horizontal scalability, meaning that as the amount of data grows, more machines can be added to the cluster to handle the increased load, without having to make changes to the data processing code.
* **Improved fault tolerance:** Partitioning also allows for data to be distributed across multiple machines, which can help to prevent data loss in the event of a single machine failure.
* **Data organization:** Partitioning allows for data to be organized in a more meaningful way, such as by time period or geographic location, which can make it easier to analyze and query the data.

**Factor to consider while partitioning the data**

**Avoid having too big or small files:** Having bigger partitioned data will lead to some of the executor doing the heavy load work, while others are just sitting idle. We need to ensure that no executors in the cluster is sitting idle due to the skewed workload distribution across the executors. This will lead to increased data processing time because of weak utilisation of the cluster.

On the other hand, having too many small files may require lots of shuffling data on disk space, taking a lot of your network compute. The recommendation is to keep your partition file size ranging 256MB to 1GB.

**How to decide the partition key(s)**

* Do not partition by columns having **high cardinality**. For example, don’t use your partition key such as employee\_Id, order\_Id etc. Instead your state code, country code, geo\_code etc.
* Partition data by specific columns that will be mostly used during filter and groupBy operations.

**Dataset Description**

In this tutorial, we will be using the “**bank**” dataset (bank-full.csv). The data is related with direct marketing campaigns (phone calls) of a Portuguese banking institution. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be (‘yes’) or not (‘no’) subscribed.

The file has 17 columns described as follow:

| **Field** | **Description** |
| --- | --- |
| **age** | Age. |
| **job** | Occupation. |
| **marital** | Marital Status. |
| **education** | Education Level. |
| **default** | has credit in default?. |
| **balance** | average yearly balance. |
| **housing** | has housing loan?. |
| **loan** | has personal loan?. |
| **contact** | contact communication type. |
| **day\_of\_week** | last contact day of the week. |
| **month** | last contact month of year. |
| **duration** | last contact duration, in seconds. |
| **campaign** | number of contacts performed during this campaign and for this client. |
| **pdays** | number of days that passed by after the client was last contacted from a previous campaign. |
| **previous** | number of contacts performed before this campaign and for this client. |
| **poutcome** | outcome of the previous marketing campaign. |
| **y** | has the client subscribed a term deposit?. |

To execute the Spark statements of this tutorial we will be using a Zeppelin note with the **spark**interpreter (**%spark**).

%spark

sc.version

**Upload The Input Dataset on HDFS**

To load the the CSV file into a Spark DataFrame, let’s upload it on HDFS. We start by creating a directory on HDFS and then put the file into this directory.

%sh

# upload data to hdfs

hdfs dfs -mkdir -p /tutorials/spark/dataframes/partition

hdfs dfs -put /home/training/Data/bank-full.csv /tutorials/spark/dataframes/partition

1. **Loading The File Into Spark Dataframe**

The easiest way to load the csv file into a Spark dataframe is to use the **read.format** function (Spark SQL API). We will instruct the function to remove the header from the csv file and infer the schema as we didn’t provide it explicitly. The input data is double quoted, so we need to escape these double quotes while loading the file. Also we need to set the delimiter to use the semi-colon (**;**) as column separator.

%spark

// load the file using the SparkSQL API

val bank = spark.read

.format("csv")

.option("header","true")

.option("inferSchema","true")

.option("quote", "\"")

.option("escape", "\"")

.option("delimiter", ";")

.load("/tutorials/spark/dataframes/partition/bank-full.csv")

.cache

1. **Bank Dataframe Exploration**

Let’s explore the newly loaded dataframe. First we will print its schema. Then we will perform a row count on the dataframe*.*

%spark

// Print the dataframe schema

bank.printSchema

1. **Check Data Is Loaded**

Use show to verify that the data have been loaded properly. Include LIMIT to retrieve only the first 10 rows.

%spark

// Show first 10 rows

bank.show(10,false)

Count the number of rows in the dataframe.

%spark

// Show the row count

bank.count

**Spark Partition Functions**

A partition in spark is a logical chunk of data mapped to a single node in a cluster. Partitions are basic units of parallelism. Each partition is processed by a single task slot. In a multicore system, total slots for tasks will be num of executors \* number of cores. Hence the number of partitions decides the task parallelism.

**Memory Partitioning vs. Disk Partitioning**

Spark offers two functions, **coalesce()** and **repartition()** that change the memory partitions for a DataFrame.

**partitionBy()** is a DataFrameWriter method that specifies if the data should be written to disk in folders. By default, Spark does not write data to disk in nested folders.

Memory partitioning is often important independent of disk partitioning. In order to write data on disk properly, you’ll almost always need to repartition the data in memory first.

**Using PartitionBy() Function**

Spark writers allow for data to be partitioned on disk with **partitionBy**. (Some queries can run 50 to 100 times faster on a partitioned data lake, so partitioning is vital for certain queries).

Let’s partition the ‘**bank**‘ dataframe on disk with marital as the partition key. Let’s create one file per partition.

%spark

// partition the dataframe on disk with marital as the partition key.

bank

.repartition(col("marital"))

.write

.mode("overwrite")

.partitionBy("marital")

.parquet("/tutorials/spark/dataframes/partition/output1")

Here’s what the data will look like on disk (HDFS).

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/partitionby-repartition-1.jpg)

When coming to large dataset, it is not a good practice to output one file per disk partition. In such case you might not able to write out all of that data in a single file.

**Spark Repartition() Function**

Spark **repartition()** is used to increase or decrease the RDD, DataFrame, Dataset partitions whereas the **coalesce()** is used to only decrease the number of partitions in an efficient way.

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

As we have four distinct values in **‘marital’** column, let’s run **repartition(4)** to get each row of data in a separate memory partition before running **partitionBy**and see how that impacts how the files get written to disk.

%spark

// partition the dataframe on disk with 4 memory partitions.

bank

.repartition(4)

.write

.mode("overwrite")

.partitionBy("marital")

.parquet("/tutorials/spark/dataframes/partition/output2")

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/partitionby-repartition-2.jpg)

The **partitionBy** writer will write out files on disk for each memory partition. The maximum number of files written out is the number of unique **marital** multiplied by the number of memory partitions.

In this example, we have 4 unique martitals \* 4 memory partitions, so up to **16** files could get written out (if each memory partition had one divorced, one married, one single and one unknown).

**partitionBy with repartition(1)**

If we repartition the data to **one** memory partition before partitioning on disk with **partitionBy**, then we’ll write out a maximum of four files. numMemoryPartitions \* numUniqueCountries = maxNumFiles. => 1 \* 4 = 4.

Let’s take a look at the code.

%spark

// partition the dataframe on disk with 1 memory partition.

bank

.repartition(1)

.write

.mode("overwrite")

.partitionBy("marital")

.parquet("/tutorials/spark/dataframes/partition/output3")

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/partitionby-repartition-3.jpg)

**partitionBy with repartition(x)**

Let’s create **8** memory partitions and scatter the data randomly across the memory partitions (we’ll write out the data to disk in csv format, so we can inspect the contents of a memory partition).

This technique helps us set a maximum number of files per partition when creating a partitioned dataframe. Let’s write out the data to disk and observe the output.

%spark

// partition the dataframe on disk with 8 random memory partitions.

bank

.repartition(8, col("marital"), rand)

.write

.mode("overwrite")

.csv("/tutorials/spark/dataframes/partition/output4")

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/partitionby-repartition-3.jpg)

Let’s look at one of the CSV files that is outputted.

%sh

# List the output directory on HDFS

hdfs dfs -ls /tutorials/spark/dataframes/partition/output4

%sh

# Show the content of one csv file

hdfs dfs -head /tutorials/spark/dataframes/partition/output4/part-00000-e59e7bf8-9569-4cb0-9986-c832bcb8009b-c000.csv

[A white background with black text

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/partitionby-repartition-4.jpg)

**Set Max number of files per file**

Setting the number of memory partitions and fill these partitions randomly is a good, but still not ideal as this approach might generate small files if the memory partition doesn’t have any data for the**‘marital’** column.

We can also set the number of file outputted per partition. Let’s modify the code and set the number of rows per  file.  We’d like our to stored 4000 records per file. So the output will be:

* 2 files for divorced
* 7 files for married
* 3 for single
* 1 file for unknow

This count is based on the number of each **marital** status in the dataset. We can get this count by running groupBy on marital column and get the count for each group.

%spark

// Get the count per marital status

bank.groupBy("marital").count.show

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/partitionby-repartition-6.jpg)

%spark

// Set the maxRecordsPerFile number

bank

.repartition(col("marital"))

.write

.mode("overwrite")

.option("maxRecordsPerFile", 4000)

.partitionBy("marital")

.csv("/tutorials/spark/dataframes/partition/output5")

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/partitionby-repartition-5.jpg)

**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.

The following code generate 2 output file for each partition and the result looks like:

%spark

// Using of coalesce to reduce the number of partitions

bank

.coalesce(2)

.write

.mode("overwrite")

.partitionBy("marital")

.csv("/tutorials/spark/dataframes/partition/output6")

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/partitionby-repartition-7.jpg)

Stop Spark context to free resources.

%spark

sc.stop

**repartition() vs coalesce()**

* Repartition redistributes the data evenly, but at the cost of a shuffle.
* Coalesce works much faster when you reduce the number of partitions because it sticks input partitions together.
* Coalesce doesn’t guarantee uniform data distribution.
* Coalesce is identical to a repartition when you increase the number of partitions.

**Summary**

Repartition is a method which is available for both RDDs and DataFrame. It allows you to change the number of partitions they are split into. This is particularly useful when your partitions are very small and data processing is slow because of it – you can hit repartition with a smaller number of partitions, and your data will be redistributed in between them.

In this tutorial we covered Spark partitionBy and repartition functions and how to use these function to control the number of partition we want to output. coalesce is another method for changing the number of partitions of an RDD or DataFrame. It has a very similar API – just pass a number of desired partitions.

Open Zeppelin Note

[Data Partitioning In Spark](http://localhost:19995/#/notebook/2JKS1KD78)