**PySpark partitionBy() – Write to Disk Example**

PySpark partitionBy() is a function of pyspark.sql.DataFrameWriter class which is used to partition the large dataset (DataFrame) into smaller files based on one or multiple columns while writing to disk, let’s see how to use this with Python examples.

Partitioning the data on the file system is a way to improve the performance of the query when dealing with a large dataset in the Data lake. A Data Lake is a centralized repository of structured, semi-structured, unstructured, and binary data that allows you to store a large amount of data as-is in its original raw format.

By following the concepts in this article, it will help you to create an efficient Data Lake for production size data.

**1. What is PySpark Partition?**

PySpark partition is a way to split a large dataset into smaller datasets based on one or more partition keys. When you create a DataFrame from a file/table, based on certain parameters PySpark creates the DataFrame with a certain number of partitions in memory. This is one of the main advantages of PySpark DataFrame over Pandas DataFrame. Transformations on partitioned data run faster as they execute transformations parallelly for each partition.

PySpark supports partition in two ways; partition in memory (DataFrame) and partition on the disk (File system).

**Partition in memory:** You can partition or repartition the DataFrame by calling repartition() or coalesce() transformations.

**Partition on disk:** While writing the PySpark DataFrame back to disk, you can choose how to partition the data based on columns using partitionBy() of pyspark.sql.DataFrameWriter. This is similar to Hives partitions scheme.

**2. Partition Advantages**

As you are aware PySpark is designed to process large datasets with 100x faster than the tradition processing, this wouldn’t have been possible with out partition. Below are some of the advantages using PySpark partitions on memory or on disk.

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

Partition at rest (disk) is a feature of many databases and data processing frameworks and it is key to make jobs work at scale.

**3. Create DataFrame**

Let’s Create a DataFrame by reading a CSV file. You can find the dataset explained in this article at Github zipcodes.csv file

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

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

df.printSchema()

#Display below schema

root

|-- RecordNumber: string (nullable = true)

|-- Country: string (nullable = true)

|-- City: string (nullable = true)

|-- Zipcode: string (nullable = true)

|-- state: string (nullable = true)

From above DataFrame, I will be using state as a partition key for our examples below.

**4. PySpark partitionBy()**

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

Syntax: partitionBy(self, \*cols)

When you write PySpark 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) \

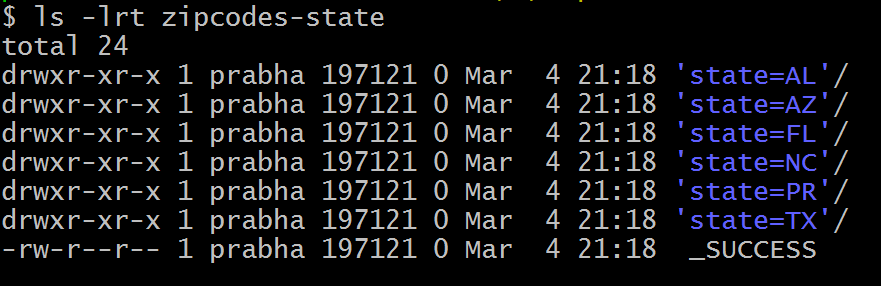
.partitionBy("state") \

.mode("overwrite") \

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

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).

**Note:** While writing the data as partitions, PySpark eliminates the partition column on the data file and adds partition column & value to the folder name, hence it saves some space on storage.To validate this, open any partition file in a text editor and check.

partitionBy(“state”) example output

On each directory, you may see one or more part files (since our dataset is small, all records for each state are kept in a single part file). You can change this behavior by repartition() the data in memory first. Specify the number of partitions (part files) you would want for each state as an argument to the repartition() method.

**5. PySpark partitionBy() Multiple Columns**

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

#partitionBy() multiple columns

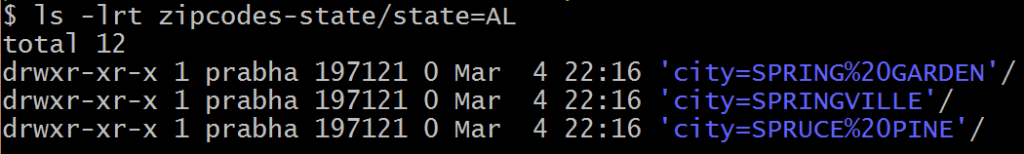
df.write.option("header",True) \

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

.mode("overwrite") \

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

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).

partitonBy(“state”,”city”) multiple columns

**6. Using repartition() and partitionBy() together**

For each partition column, if you wanted to further divide into several partitions, use repartition() and partitionBy() together as explained in the below example.

repartition() creates specified number of partitions in memory. The partitionBy()  will write files to disk for each memory partition and partition column.

#Use repartition() and partitionBy() together

dfRepart.repartition(2)

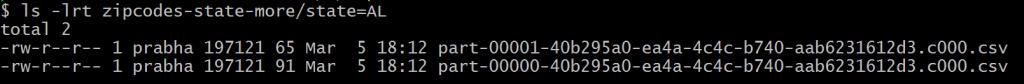
.write.option("header",True) \

.partitionBy("state") \

.mode("overwrite") \

.csv("c:/tmp/zipcodes-state-more")

If you look at the folder, you should see only 2 part files for each state. Dataset has 6 unique states and 2 memory partitions for each state, hence the above code creates a maximum total of 6 x 2 = 12 part files.



**Note:** Since total zipcodes for each US state differ in large, California and Texas have many whereas Delaware has very few, hence it creates a Data Skew (Total rows per each part file differs in large).

**7. Data Skew – Control Number of Records per Partition File**

Use option maxRecordsPerFile if you want to control the number of records for each partition. This is particularly helpful when your data is skewed (Having some partitions with very low records and other partitions with high number of records).

#partitionBy() control number of partitions

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

.option("maxRecordsPerFile", 2) \

.partitionBy("state") \

.mode("overwrite") \

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

The above example creates multiple part files for each state and each part file contains just 2 records.

**8. Read a Specific Partition**

Reads are much faster on partitioned data. This code snippet retrieves the data from a specific partition "state=AL and city=SPRINGVILLE". Here, It just reads the data from that specific folder instead of scanning a whole file (when not partitioned).

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

.csv("c:/tmp/zipcodes-state/state=AL/city=SPRINGVILLE")

dfSinglePart.printSchema()

dfSinglePart.show()

#Displays

root

|-- RecordNumber: string (nullable = true)

|-- Country: string (nullable = true)

|-- Zipcode: string (nullable = true)

+------------+-------+-------+

|RecordNumber|Country|Zipcode|

+------------+-------+-------+

| 54355| US| 35146|

+------------+-------+-------+

While reading specific Partition data into DataFrame, it does not keep the partitions columns on DataFrame hence, you printSchema() and DataFrame is missing state and city columns.

**9. PySpark SQL – Read Partition Data**

This is an example of how to write a Spark DataFrame by preserving the partition columns on DataFrame.

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

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

parqDF.createOrReplaceTempView("ZIPCODE")

spark.sql("select \* from ZIPCODE where state='AL' and city = 'SPRINGVILLE'") \

.show()

#Display

+------------+-------+-------+-----+-----------+

|RecordNumber|Country|Zipcode|state| city|

+------------+-------+-------+-----+-----------+

| 54355| US| 35146| AL|SPRINGVILLE|

+------------+-------+-------+-----+-----------+

The execution of this query is also significantly faster than the query without partition. It filters the data first on state and then applies filters on the city column without scanning the entire dataset.

**10. How to Choose a Partition Column When Writing to File system?**

When creating partitions you have to be very cautious with the number of partitions you would create, as having too many partitions creates too many sub-directories on HDFS 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.

**Conclusion**

While you are create Data Lake out of Azure, HDFS or AWS you need to understand how to partition your data at rest (File system/disk), PySpark partitionBy() and repartition() help you partition the data and eliminating the Data Skew on your large datasets.