How can I use the partitionBy() function in PySpark to write data to disk?

Authored by

stats writer

June 24, 2024

The partitionBy() function in PySpark is a useful tool for writing data to disk in an organized and efficient manner. This function allows for data to be partitioned or separated into smaller groups based on a given criteria, such as a specific column or value. By using partitionBy(), users can improve the performance of writing and reading data by ensuring that related data is stored together, reducing the need for shuffling and improving query speeds. To use this function, simply specify the partitioning criteria and the desired output path, and PySpark will automatically create separate files or directories for each partition. This allows for easier data management and retrieval, making the partitionBy() function a valuable tool for optimizing data storage in PySpark.

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, for example Data lake.

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 100x faster than the traditional processing, this wouldn't have been possible without partition. Below are some of the advantages of using PySpark partitions on memory or on disk.

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. Follow Github zipcodes.csv file to download the CSV file.

```
# Create DataFrame by reading 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 the above DataFrame, I will be using `state` as a partition key for our examples below.

4. PySpark partitionBy()

In PySpark, the `partitionBy()` transformation is used to partition data in an RDD or DataFrame based on the specified partitioner. It is typically applied after certain operations such as `groupBy()` or `join()` to control the distribution of data across partitions.

The partitionBy() is available in `DataFrameWriter` class hence, it is used to write the partition data to the disk.

```
# Syntax partitionBy
partitionBy(self, *cols)
```

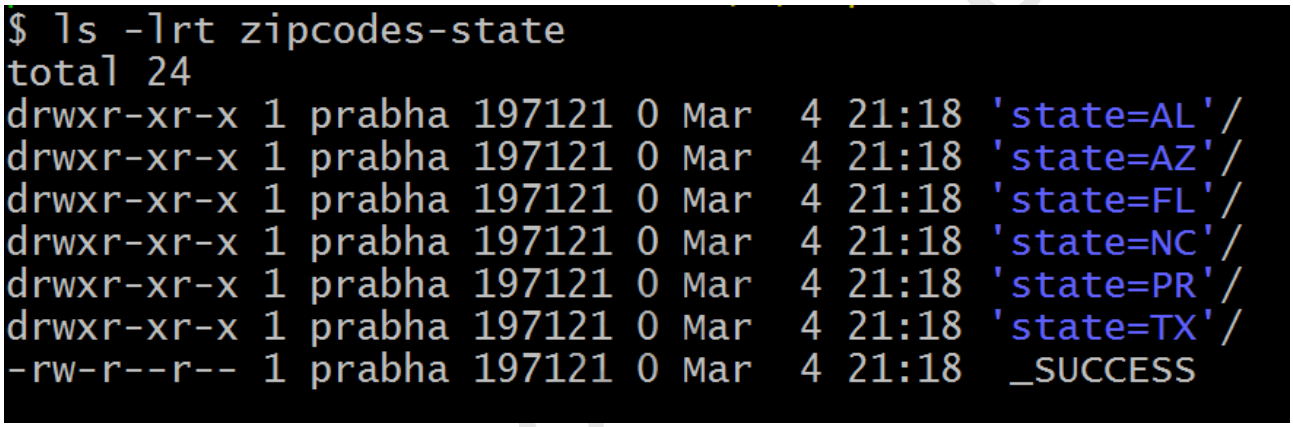
When you write PySpark DataFrame to disk by calling `partitionBy()`, PySpark splits the records based on the partition column and stores each partition data into a sub-directory.

```
# partitionBy() Example
```

```
df.write.option("header",True)
.partitionBy("state")
.mode("overwrite")
.csv("/tmp/zipcodes-state")
```

My data has six states; hence, it created six directories on disk. The name of the sub-directory would be the partition column and its value (partition column=value).

Note: When writing the data as partitions, PySpark eliminates the partition column on the data file and adds the partition column and value to the folder name, saving some storage space.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 the 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`).

```
$ ls -lrt zipcodes-state/state=AL
total 12
drwxr-xr-x 1 prabha 197121 0 Mar  4 22:16 'city=SPRING%20GARDEN'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 22:16 'city=SPRINGVILLE'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 22:16 'city=SPRUCE%20PINE'/
```

partitonBy("state","city") multiple columns

6. Using repartition() and partitionBy() together

The `repartition()` is used to increase or decrease the number of partitions in memory and when you use it with partitionBy(), it further breaks down into multiple partitions based on column data.

```
#Use repartition() and partitionBy() together
dfRepart.repartition(2)
.write.option("header",True)
.partitionBy("state")
.mode("overwrite")
.csv("c:/tmp/zipcodes-state-more")
```

Upon inspection of the folder where the data is written, you'll observe that there are only two partition files for each state. Given that the dataset encompasses six unique states and there are two memory partitions allocated for each state, the code generates a maximum of 12 partition files.

```
$ ls -lrt zipcodes-state-more/state=AL
total 2
-rw-r--r-- 1 prabha 197121 65 Mar  5 18:12 part-00001-40b295a0-ea4a-4c4c-b740-aab6231612d3.c000.csv
-rw-r--r-- 1 prabha 197121 91 Mar  5 18:12 part-00000-40b295a0-ea4a-4c4c-b740-aab6231612d3.c000.csv
```

Note: As the total number of zip codes varies significantly for each US state, with California and Texas containing numerous zip codes while Delaware has relatively few, a data skew arises. This skew results in unequal distribution of rows across partition files, with some containing a large number of rows and others containing only a few.

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 a 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).

```
# Reading from partitioned data
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 partition 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.

```
# Read from partitioned data using sql
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 to 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 unnecessary and overhead to NameNode (if you are using Hadoop) since it must keep all metadata for the file system in memory.

Consider a scenario where you possess a US census table featuring zip codes, cities, states, and additional columns. Employing state as a partition key divides the table into approximately 50 partitions. Consequently, when seeking a zip code within a specific state (e.g., state='CA' and zipCode='92704′), the process is expedited as it solely necessitates scanning within the directory associated with the 'CA' state partition.

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

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

11. Frequently Asked Questions on partitionBy()

How is `partitionBy()` different from `groupBy()` in PySpark?

`partitionBy()` is used for physically organizing data on disk when writing to a file system, while `groupBy()` is used for the logical grouping of data within a DataFrame.

Can I use multiple columns with `partitionBy()`?

Yes, We can specify multiple columns in the `partitionBy()` function to create a hierarchical directory structure. For example:
df.write.partitionBy("column1", "column2").parquet("/path/to/output")

How does partitioning affect query performance?

Partitioning can significantly improve query performance, especially when querying specific subsets of data. It helps skip irrelevant data when reading, reducing the amount of data that needs to be processed.

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.

Hope this give you better idea on partitions in PySpark.

Happy Learning !!

Related Articles

References