R Raymond

Data Partitioning in Spark (PySpark) In-depth Walkthrough

2 years ago 29,610 1

Data partitioning is critical to data processing performance especially for large volume of data processing in Spark. Partitions in Spark won't span across nodes though one node can contains more than one partitions. When processing, Spark assigns one task for each partition and each worker threads can only process one task at a time. Thus, with too few partitions, the application won't utilize all the cores available in the cluster and it can cause

data skewing problem; with too many partitions, it will bring overhead for Spark to manage too many small tasks.

In this post, I'm going to show you how to partition data in Spark appropriately. Python is used as programming language in the examples. You can choose Scala or R if you are more familiar with them.

Starter script

Let's run the following scripts to populate a data frame with 100 records.

```
from pyspark.sql.functions import year, month, dayofmonth
from pyspark.sql import SparkSession
from datetime import date, timedelta
from pyspark.sql.types import IntegerType, DateType, StringType, StructType,
StructField

appName = "PySpark Partition Example"
master = "local[8]"

# Create Spark session with Hive supported.
spark = SparkSession.builder \
    .appName(appName) \
    .master(master) \
    .getOrCreate()

print(spark.version)
# Populate sample data
start_date = date(2019, 1, 1)
data = []
for i in range(0, 50):
```

```
    data.append({"Country": "CN", "Date": start_date +
                 timedelta(days=i), "Amount": 10+i})
    data.append({"Country": "AU", "Date": start_date +
                 timedelta(days=i), "Amount": 10+i})

schema = StructType([StructField('Country', StringType(), nullable=False),
                     StructField('Date', DateType(), nullable=False),
                     StructField('Amount', IntegerType(), nullable=False)])

df = spark.createDataFrame(data, schema=schema)
df.show()
print(df.rdd.getNumPartitions())
```

The above scripts instantiates a SparkSession locally with 8 worker threads. It then populates 100 records (50*2) into a list which is then converted to a data frame.

print(df.rdd.getNumPartitions())

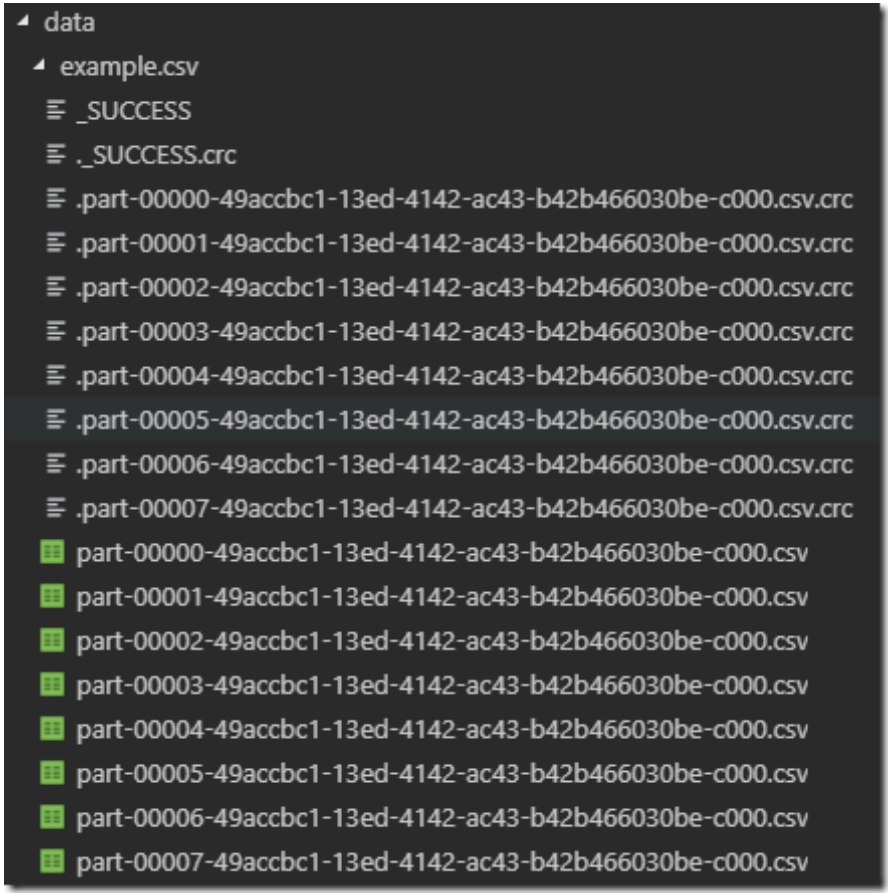
For the above code, it will prints out number 8 as there are 8 worker threads. By default, each thread will read data into one partition.

Write data frame to file system

We can use the following code to write the data into file systems:

```
df.write.mode("overwrite").csv("data/example.csv", header=True)
```

8 sharded files will be generated for each partition:



Each file contains about 12 records while the last one contains 16 records:

```
Country,Date,Amount
CN,2019-02-12,52
AU,2019-02-12,52
CN,2019-02-13,53
AU,2019-02-13,53
CN,2019-02-14,54
AU,2019-02-14,54
CN,2019-02-15,55
AU,2019-02-15,55
CN,2019-02-16,56
AU,2019-02-16,56
CN,2019-02-17,57
AU,2019-02-17,57
CN,2019-02-18,58
AU,2019-02-18,58
CN,2019-02-19,59
AU,2019-02-19,59

```

Repartitioning with coalesce function

There are two functions you can use in Spark to repartition data and **coalesce** is one of them.

This function is defined as the following:

```
def coalesce(numPartitions)
```

Returns a new :class: `DataFrame` that has exactly `numPartitions` partitions.

Similar to coalesce defined on an :class: `RDD`, this operation results in a narrow dependency, e.g. if you go from 1000 partitions to 100 partitions, there will not be a shuffle,

instead each of the 100 new partitions will claim 10 of the current partitions. If a larger number of partitions is requested, it will stay at the current number of partitions.

Now if we run the following code, can you guess how many sharded files will be generated?

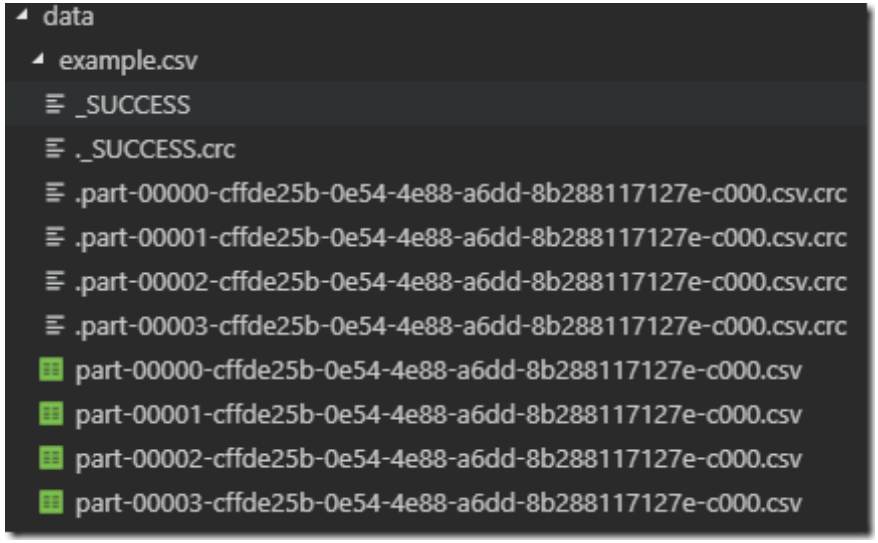
```
df = df.coalesce(16)
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("data/example.csv", header=True)
```

The answer is still 8. This is because coalesce function does' t involve reshuffle of data. In the above code, we want to increate the partitions to 16 but the number of partitions stays at the current (8).

If we decrease the partitions to 4 by running the following code, how many files will be generated?

```
df = df.coalesce(4)
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("data/example.csv", header=True)
```

The answer is 4 as the following screenshot shows:



Repartitioning with repartition function

The other method for repartitioning is **repartition**. It's defined as the follows:

```
def repartition(numPartitions, *cols)
```

Returns a new :class: `DataFrame` partitioned by the given partitioning expressions. The resulting DataFrame is hash partitioned.

`numPartitions` can be an int to specify the target number of partitions or a Column. If it is a Column, it will be used as the first partitioning column. If not specified, the default number of partitions is used.

Added optional arguments to specify the partitioning columns. Also made numPartitions optional if partitioning columns are specified.

Data reshuffle occurs when using this function. Let's try some examples using the above dataset.

Repartition by number

Use the following code to repartition the data to 10 partitions.

```
df = df.repartition(10)
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("data/example.csv", header=True)
```

Spark will try to evenly distribute the data to each partitions. If the total partition number is greater than the actual record count (or RDD size), some partitions will be empty.

After we run the above code, data will be reshuffled to 10 partitions with 10 sharded files generated.

If we repartition the data frame to 1000 partitions, how many sharded files will be generated?

The answer is 100 because the other 900 partitions are empty and each file has one record.

Repartition by column

We can also repartition by columns.

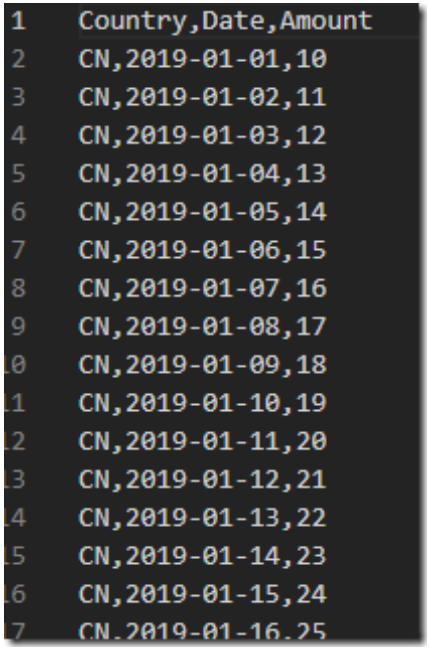
For example, let's run the following code to repartition the data by column **Country**.

```
df = df.repartition("Country")
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("data/example.csv", header=True)
```

The above scripts will create 200 partitions (Spark by default create 200 partitions). However only three sharded files are generated:

- One file stores data for CN country.
- Another file stores data for AU country.
- The other one is empty.

For example, one partition file looks like the following:



```
Country,Date,Amount
CN,2019-01-01,10
CN,2019-01-02,11
CN,2019-01-03,12
CN,2019-01-04,13
CN,2019-01-05,14
CN,2019-01-06,15
CN,2019-01-07,16
CN,2019-01-08,17
CN,2019-01-09,18
CN,2019-01-10,19
CN,2019-01-11,20
CN,2019-01-12,21
CN,2019-01-13,22
CN,2019-01-14,23
CN,2019-01-15,24
CN,2019-01-16,25
```

It includes all the 50 records for 'CN' in **Country** column.

Similarly, if we can also partition the data by **Date** column:

```
df = df.repartition("Date")
print(df.rdd.getNumPartitions())
```

```
df.write.mode("overwrite").csv("data/example.csv", header=True)
```

If you look into the data, you may find the data is probably not partitioned properly as you would expect, for example, one partition file only includes data for both countries and different dates too.

This is because by default Spark use hash partitioning as partition function. You can use range partitioning function or customize the partition functions. I will talk more about this in my other posts.

Partition by multiple columns

In real world, you would probably partition your data by multiple columns. For example, we can implement a partition strategy like the following:

```
data/
        example.csv/
            year=2019/
                month=01/
                    day=01/
                        Country=CN/
                            part....csv
```

With this partition strategy, we can easily retrieve the data by date and country. Of course you can also implement different partition hierarchies based on your requirements. For example, if all your analysis are always performed country by country, you may find the following structure will be easier to access:

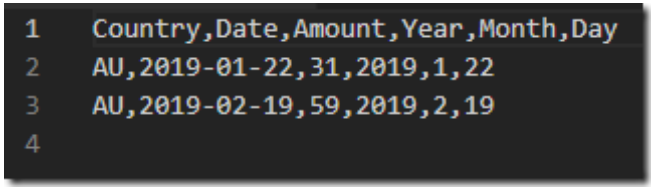
```
data/
    Country=CN/
        example.csv/
            year=2019/
                month=01/
                    day=01/
                        part....csv
```

To implement the above partitioning strategy, we need to derive some new columns (year, month, date).

```
df = df.withColumn("Year", year("Date")).withColumn(
"Month", month("Date")).withColumn("Day", dayofmonth("Date"))
df = df.repartition("Year", "Month", "Day", "Country")
print(df.rdd.getNumPartitions())
df.write.mode("overwrite").csv("data/example.csv", header=True)
```

The above code derives some new columns and then repartition the data frame with those columns.

When you look into the saved files, you may find that all the new columns are also saved and the files still mix different sub partitions. To improve this, we need to match our write partition keys with repartition keys.

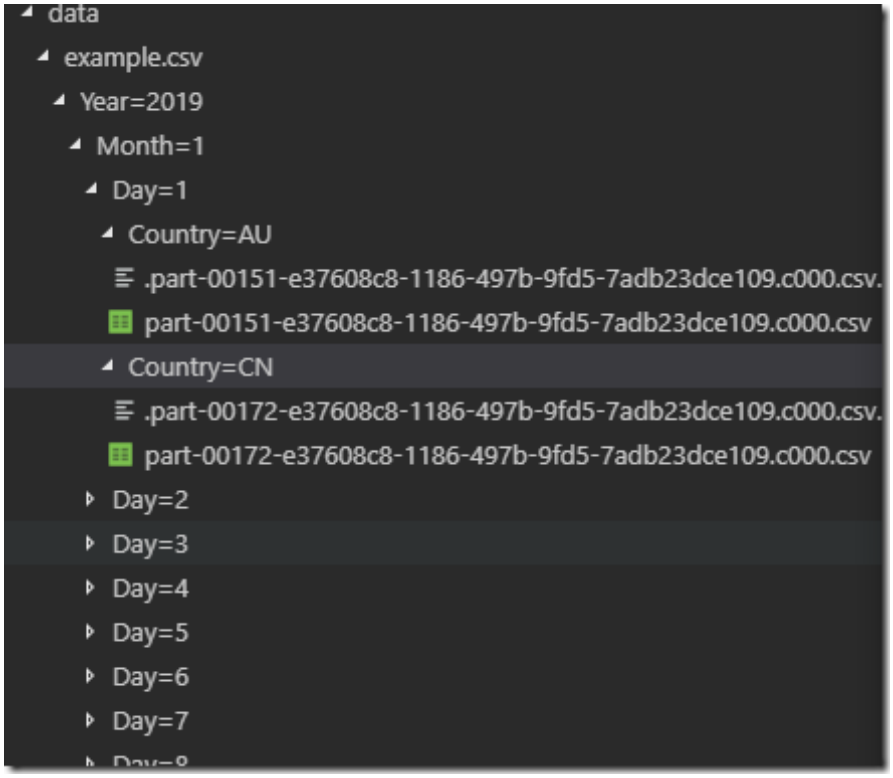


Match repartition keys with write partition keys

To match partition keys, we just need to change the last line to add a **partitionBy** function:

```
df.write.partitionBy("Year", "Month", "Day", "Country").mode(
"overwrite").csv("data/example.csv", header=True)
```

After this change, the partitions are now written into file system as we expect:



By open the files, you will also find that all the partitioning columns/keys are removed from the serialized data files:

```
Date,Amount
2019-01-01,10

```

In this way, the storage cost is also less. With partitioned data, we can also easily append data to new subfolders instead of operating on the complete data set.

Read from partitioned data

Now let's read the data from the partitioned files with the these criteria:

- Year= 2019
- Month=2
- Day=1
- Country=CN

The code can be simple like the following:

```
df = spark.read.csv("data/example.csv/Year=2019/Month=2/Day=1/Country=CN")
print(df.rdd.getNumPartitions())
df.show()
```

The console will print the following output:

```
+----------+------+
|       _c0|   _c1|
+----------+------+
|      Date|Amount|
|2019-02-01|    41|
+----------+------+
```

Can you think about how many partitions there are for this new data frame?

The answer is one for this example (think about why?).

Similarly, we can also query all the data for the second month:

```
df = spark.read.csv("data/example.csv/Year=2019/Month=2")
print(df.rdd.getNumPartitions())
df.show()
```

Now, how should we find all the data for Country CN?

Use wildcards for partition discovery

We can use wildcards. Wildcards are supported for all file formats in partition discovery.

```
df = spark.read.option("basePath", "data/example.csv/").csv(
"data/example.csv/Year=*/Month=*/Day=*/Country=CN")
print(df.rdd.getNumPartitions())
df.show()
```

You can use wildcards in any part of the path for partition discovery. For example, the following code looks data for month 2 of Country AU:

```
df = spark.read.option("basePath", "data/example.csv/").csv(
"data/example.csv/Year=*/Month=2/Day=*/Country=AU")
print(df.rdd.getNumPartitions())
df.show()
```

Summary

Through partitioning, we maximise the parallel usage of Spark cluster, reduce data skewing and storage space to achieve better performance. This is a common design practice in MPP frameworks. When designing serialization partition strategy (write partitions into file systems), you need to take access paths into consideration, for example, are your partition keys commonly used in filters?

However partitioning doesn't mean the more the better as mentioned in the every beginning of this post. Spark recommends 2-3 tasks per CPU core in your cluster. For example, if you have 1000 CPU core in your cluster, the recommended partition number is 2000 to 3000. Sometimes, depends on the distribution and skewness of your source data, you need to tune around to find out the appropriate partitioning strategy.

Simple question

In our example, when we serialize data into file system partitioning by Year, Month, Day and Country, one partition is written into one physical file. However if we use HDFS and also if there is a large amount of data for each partition, will one partition file only exist in one data node?

python spark pyspark spark-advanced

Last modified by Administrator at 24 days ago

Comments (1)

Please log in or register to comment.

log in | register

Log in with external accounts

Log in with Microsoft account

Log in with Google account

Kontext Column

Created for everyone to publish data, programming and cloud related articles.
Follow three steps to create your columns.

Learn more →

More from Kontext

Read and Write XML Files with Python

python python-file-operations

14 0 1 day ago

XML is a commonly used data exchange format in many applications and systems though JSON became more popular nowadays. Compared with JSON, XML supports schema (XSD) validation and can be easily transformed other formats using XSLT. XHTML is also a strict version of HTML that is XML based and used...

Python Programming

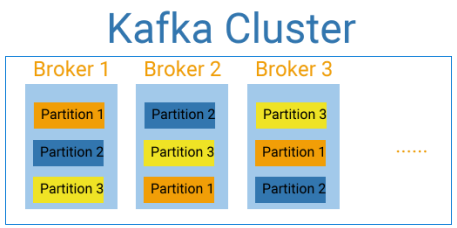
Spark Structured Streaming - Read from and Write into Kafka Topics

spark kafka

34 0 3 days ago

Spark structured streaming provides rich APIs to read from and write to Kafka topics. When reading from Kafka, Kafka sources can be created for both streaming and batch queries. When writing into Kafka, Kafka sinks can be created as destination for both streaming...

Streaming Analytics



Kafka Topic Partitions Walkthrough via Python

kafka python

7 0 4 days ago

Partition is the parallelism unit in a Kafka cluster. Partitions are replicated in Kafka cluster (cluster of brokers) for fault tolerant and throughput. This articles show you how to work with Kafka partitions using Python as programming language. ...

Streaming Analytics

About column

Raymond

Spark

Apache Spark installation guides, performance tuning tips, general tutorials, etc.

*Spark logo is a registered trademark of Apache Spark.

Subscribe RSS

Sign-in with Google and Microsoft accounts

Dark theme

English