How do I read and write Parquet files using PySpark?

Authored by

stats writer

June 24, 2024

PySpark is a powerful Python-based framework for processing big data. One of its key features is the ability to read and write Parquet files, a columnar storage format widely used in big data analytics. To read Parquet files in PySpark, the SparkSession API is used, which provides a simple and efficient way to load the data into a DataFrame. Similarly, the DataFrame API can be used to write data to Parquet files. This allows for efficient data processing and storage, making it a popular choice in big data environments. By utilizing PySpark's capabilities, users can easily read and write Parquet files, providing a seamless and efficient solution for big data processing.

Pyspark SQL provides methods to read Parquet file into DataFrame and write DataFrame to Parquet files, `parquet()` function from `DataFrameReader` and `DataFrameWriter` are used to read from and write/create a Parquet file respectively. Parquet files maintain the schema along with the data hence it is used to process a structured file.

In this article, I will explain how to read from and write a parquet file and also will explain how to partition the data and retrieve the partitioned data with the help of SQL.

Below are the simple statements on how to write and read parquet files in PySpark which I will explain in detail later sections.

```
# Read and Write Parquet file using parquet()
df.write.parquet("/tmp/out/people.parquet")
parDF1=spark.read.parquet("/temp/out/people.parquet")
```

Before, I explain in detail, let's understand What is Parquet file and its advantages over CSV, JSON and other text file formats.

What is Parquet File?

Apache Parquet file is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model, or programming language.

Advantages:

While querying columnar storage, it skips the nonrelevant data very quickly, making faster query execution. As a result aggregation queries consume less time compared to row-oriented databases.

It is able to support advanced nested data structures.

Parquet supports efficient compression options and encoding schemes.

Pyspark SQL provides support for both reading and writing Parquet files that automatically capture the schema of the original data, It also reduces data storage by 75% on average. Pyspark by default supports Parquet in its library hence we don't need to add any dependency libraries.

Apache Parquet Pyspark Example

Since we don't have the parquet file, let's work with writing parquet from a DataFrame. First, create a Pyspark DataFrame from a list of data using `spark.createDataFrame()` method.

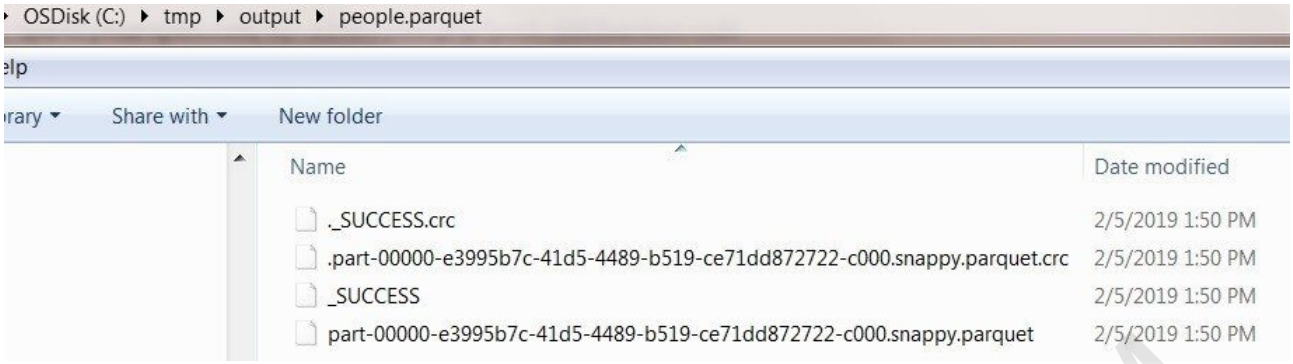
```
# Create Sample data
# Imports
import pyspark
from pyspark.sql import SparkSession
spark=SparkSession.builder.appName("parquetFile").getOrCreate()
data =
columns=
df=spark.createDataFrame(data,columns)
```

above example, it creates a DataFrame with columns firstname, middlename, lastname, dob, gender, salary.

Pyspark Write DataFrame to Parquet file format

Now let's create a parquet file from PySpark DataFrame by calling the `parquet()` function of `DataFrameWriter` class. When you write a DataFrame to parquet file, it automatically preserves column names and their data types. Each part file Pyspark creates has the .parquet file extension. Below is the example,

```
# Write DataFrame to parquet file using write.parquet()
df.write.parquet("/tmp/output/people.parquet")
```



Pyspark Read Parquet file into DataFrame

Pyspark provides a `parquet()` method in `DataFrameReader` class to read the parquet file into dataframe. Below is an example of a reading parquet file to data frame.

```
# Read parquet file using read.parquet()
parDF=spark.read.parquet("/tmp/output/people.parquet")
```

Types of Saving Modes of Parquet File

In PySpark, after writing the DataFrame to the Parquet file, we can specify the mode of saving using `.mode()` method. Below are the types of saving modes available in PySpark from `pyspark.sql.DataFrameWriter.mode` module.

Syntax

`DataFrameWriter.mode`**(***saveMode: Optional***)**

Options :

append: This mode appends the data from the DataFrame to the existing file, if the Destination files already exist. In Case the Destination files do not exists, it will create a new parquet file in the specified location.

Example:

```
df.write.mode("append").parquet("path/to/parquet/file")
```

overwrite: This mode overwrites the destination Parquet file with the data from the DataFrame. If the file does not exist, it creates a new Parquet file.

Example:

```
df.write.mode("overwrite").parquet("path/to/parquet/file")
```

ignore: If the destination Parquet file already exists, this mode does nothing and does not write the DataFrame to the file. If the file does not exist, it creates a new Parquet file.

Example:

```
df.write.mode("ignore").parquet("path/to/parquet/file")
```

error or errorIfExists: This mode raises an error if the destination Parquet file already exists. It does not write the DataFrame to the file. If the file does not exist, it creates a new Parquet file.

Example:

```
df.write.mode("error").parquet("path/to/parquet/file")
```

Append or Overwrite an existing Parquet file

Using append save mode, you can append a dataframe to an existing parquet file. Incase to overwrite use overwrite save mode.

```
# Using append and overwrite to save parquet file
df.write.mode('append').parquet("/tmp/output/people.parquet")
df.write.mode('overwrite').parquet("/tmp/output/people.parquet")
```

Executing SQL queries DataFrame

Pyspark Sql provides to create temporary views on parquet files for executing sql queries. These views are available until your program exists.

```
# Using spark.sql
parqDF.createOrReplaceTempView("ParquetTable")
parkSQL = spark.sql("select * from ParquetTable where salary >= 4000 ")
```

Creating a table on Parquet file

Now let's walk through executing SQL queries on parquet file. In order to execute sql queries, create a temporary view or table directly on the parquet file instead of creating from DataFrame.

```
# Create temp view on Parquet file
spark.sql("CREATE TEMPORARY VIEW PERSON USING parquet OPTIONS (path "/tmp/output/people.parquet")")
spark.sql("SELECT * FROM PERSON").show()
```

Here, we created a temporary view `PERSON` from "`people.parquet`" file. This gives the following results.

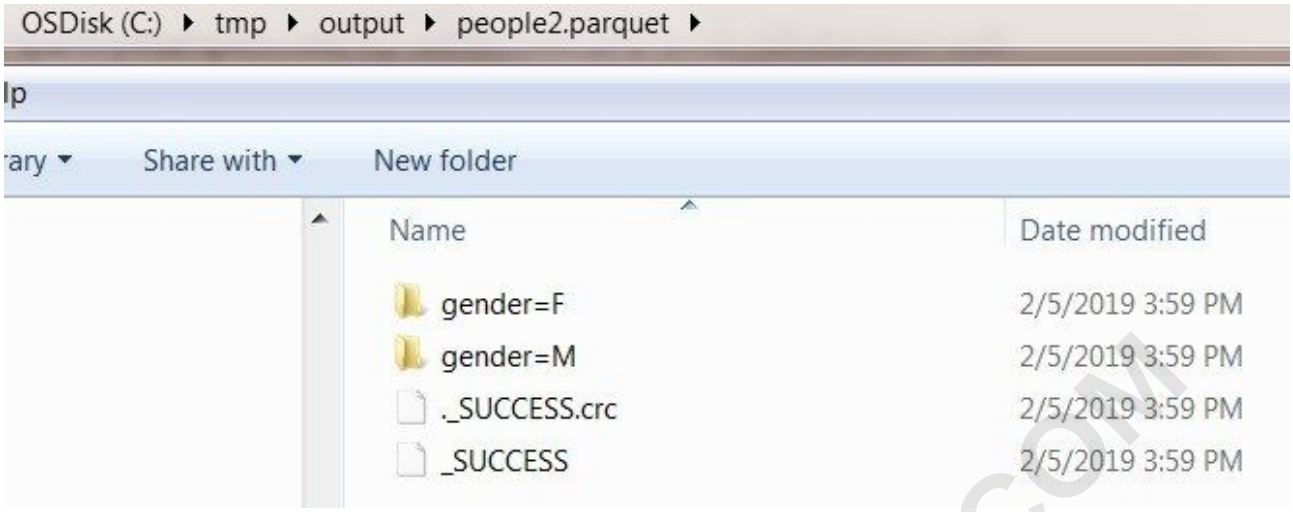
```
# Output
+---------+----------+--------+-----+------+------+
|firstname|middlename|lastname| dob|gender|salary|
+---------+----------+--------+-----+------+------+
| Robert | |Williams|42114| M| 4000|
| Maria | Anne| Jones|39192| F| 4000|
| Michael | Rose| |40288| M| 4000|
| James | | Smith|36636| M| 3000|
| Jen| Mary| Brown| | F| -1|
+---------+----------+--------+-----+------+------+
```

Create Parquet partition file

When we execute a particular query on the PERSON table, it scan's through all the rows and returns the results back. This is similar to the traditional database query execution. In PySpark, we can improve query execution in an optimized way by doing partitions on the data using pyspark `partitionBy()` method. Following is the example of partitionBy().

```
df.write.partitionBy("gender","salary").mode("overwrite").parquet("/tmp/output/people2.parquet")
```

When you check the people2.parquet file, it has two partitions "gender" followed by "salary" inside.



Retrieving from a partitioned Parquet file

The example below explains of reading partitioned parquet file into DataFrame with gender=M.

```
parDF2=spark.read.parquet("/tmp/output/people2.parquet/gender=M")
parDF2.show(truncate=False)
```

Output for the above example is shown below.

```
# Output
+---------+----------+--------+-----+------+
|firstname|middlename|lastname|dob |salary|
+---------+----------+--------+-----+------+
|Robert | |Williams|42114|4000 |
|Michael |Rose | |40288|4000 |
|James | |Smith |36636|3000 |
+---------+----------+--------+-----+------+
```

Creating a table on Partitioned Parquet file

Here, I am creating a table on partitioned parquet file and executing a query that executes faster than the table without partition, hence improving the performance.

```
# Create a temporary view on partitioned Parquet file
```

```
spark.sql("CREATE TEMPORARY VIEW PERSON2 USING parquet OPTIONS (path "/tmp/output/people2.parquet/gender=F")")
spark.sql("SELECT * FROM PERSON2" ).show()
```

Below is the output .

```
# Output
+---------+----------+--------+-----+------+
|firstname|middlename|lastname| dob|salary|
+---------+----------+--------+-----+------+
| Maria | Anne| Jones|39192| 4000|
| Jen| Mary| Brown| | -1|
+---------+----------+--------+-----+------+
```

Complete Example of PySpark read and write Parquet file

```
# Imports
import pyspark
from pyspark.sql import SparkSession
spark=SparkSession.builder.appName("parquetFile").getOrCreate()
data =
columns=
df=spark.createDataFrame(data,columns)
df.write.mode("overwrite").parquet("/tmp/output/people.parquet")
parDF1=spark.read.parquet("/tmp/output/people.parquet")
parDF1.createOrReplaceTempView("parquetTable")
parDF1.printSchema()
parDF1.show(truncate=False)
```

parkSQL = spark.sql("select * from ParquetTable where salary >= 4000 ")
parkSQL.show(truncate=False)

spark.sql("CREATE TEMPORARY VIEW PERSON USING parquet OPTIONS (path "/tmp/output/people.parquet")")
spark.sql("SELECT * FROM PERSON").show()

df.write.partitionBy("gender","salary").mode("overwrite").parquet("/tmp/output/people2.parquet")

parDF2=spark.read.parquet("/tmp/output/people2.parquet/gender=M")
parDF2.show(truncate=False)

```
spark.sql("CREATE TEMPORARY VIEW PERSON2 USING parquet OPTIONS (path "/tmp/output/people2.parquet/gender=F")")
spark.sql("SELECT * FROM PERSON2" ).show()
```

Conclusion:

We have learned how to write a Parquet file from a PySpark DataFrame and reading parquet file to DataFrame and created view/tables to execute SQL queries. Also explained how to do partitions on parquet files to improve performance.

Hope you liked it and, do comment in the comment section.

Related Articles