How can I read a CSV file into a DataFrame using PySpark?

Authored by

stats writer

June 24, 2024

PySpark is a framework used for large-scale data processing that is designed to work with datasets stored in various formats, including CSV. To read a CSV file into a DataFrame using PySpark, the first step is to import the necessary libraries. Then, the CSV file can be loaded into a PySpark DataFrame using the "read" method and specifying the path to the file. This will create a structured table-like data structure that allows for easy manipulation and analysis of the data. The resulting DataFrame can then be used for further data processing and analysis within the PySpark environment.

Reading CSV files into a structured DataFrame becomes easy and efficient with PySpark DataFrame API. By leveraging PySpark's distributed computing model, users can process massive CSV datasets with lightning speed, unlocking valuable insights and accelerating decision-making processes. Whether you're working with gigabytes or petabytes of data, PySpark's CSV file integration offers a flexible and scalable approach to data analysis, empowering organizations to harness the full potential of their data assets

To read a CSV file into PySpark DataFrame use `csv("path")` from `DataFrameReader`. This article explores the process of reading single files, multiple files, or all files from a local directory into a DataFrame using PySpark.

Key Points:

Table of contents:

1. PySpark Read CSV File into DataFrame

By utilizing `DataFrameReader.csv("path")` or `format("csv").load("path")` methods, you can read a CSV file into a PySpark DataFrame. These methods accept a file path as their parameter. When using the format("csv") approach, you should specify data sources like `csv` or `org.apache.spark.sql.csv`.

Refer to dataset zipcodes.csv at GitHub

```
# Import
from pyspark.sql import SparkSession
```

```
# Create SparkSession
spark = SparkSession.builder().master("local")
.appName(arabpsychology.com)
.getOrCreate()

# Read CSV File
```

df = spark.read.csv("/path/zipcodes.csv")
df.printSchema()

Alternatively, you can use the format().load()

```
# Using format().load()
df = spark.read.format("csv")
.load("/path/zipcodes.csv")
# or
df = spark.read.format("org.apache.spark.sql.csv")
.load("/path/zipcodes.csv")
df.printSchema()
```

This example reads the data into DataFrame columns `"_c0"` for the first column and `"_c1"` for the second and so on. and by default data type for all these columns is treated as String.

```
# Output:
root
|-- _c0: string (nullable = true)
|-- _c1: string (nullable = true)
|-- _c2: string (nullable = true)
```

1.1 Using Header Record For Column Names

If you have a header with column names on your input file, you need to explicitly specify `True` for header option using `option("header",True)` not mentioning this, the API treats header as a data record.

```
# Use header record for column names
df2 = spark.read.option("header",True)
.csv("/path/zipcodes.csv")
```

As mentioned earlier, PySpark reads all columns as a string (StringType) by default. I will explain in later sections on how to read the schema (`inferschema`) from the header record and derive the column type based on the data.

1.2 Read Multiple CSV Files

To read multiple CSV files into a PySpark DataFrame, each separated by a comma, you can create a list of file paths and pass it to the `spark.read.csv()` method.

```
# Read multiple CSV files
df = spark.read.csv("path/file1.csv,path/file2.csv,path/file3.csv")
```

1.3 Read all CSV Files from a Directory

To read all CSV files from a directory, specify the directory path as an argument to the csv() method.

```
# Read all files from a directory
df = spark.read.csv("Folder path")
```

2. Reading CSV File Options

PySpark CSV dataset provides multiple options to work with CSV files. Below are some of the most important options explained with examples.

You can either chain option() to use multiple options or use the alternate options() method.

```
# Syntax
option(self, key, value) # Using single options
options(self, **options) # Using multiple options
```

2.1 delimiter

`delimiter` option is used to specify the column delimiter of the CSV file. By default, it is **comma (,)** character, but can be set to any character like **pipe(|)**, **tab (t)**, **space** using this option.

```
# Using delimiter option
df3 = spark.read.options(delimiter=',')
.csv("/path/zipcodes.csv")
```

2.2 inferSchema

The default value set to this option is `False` when setting to `true` it automatically infers column types based on the data. Note that, it requires reading the data one more time to infer the schema.

```
# Using inferschema and delimiter
df4 = spark.read.options(inferSchema='True',delimiter=',')
.csv("/path/zipcodes.csv")
```

Using options with key-value pair.

```
# Define read options
options = {
"inferSchema": "True",
"delimiter": ","
}
```

# Read a CSV file with specified options
df4 = spark.read.options(**options).csv("/path/zipcodes.csv")

Alternatively, you can also write this by chaining `option()` method.

```
# Chaining multiple options
df4 = spark.read.option("inferSchema",True)
.option("delimiter",",")
.csv("/path/zipcodes.csv")
```

2.3 header

This option is used to read the first line of the CSV file as column names. By default the value of this option is `False` , and all column types are assumed to be a string.

```
# Using header option
df3 = spark.read.options(header='True', inferSchema='True', delimiter=',')
.csv("/path/zipcodes.csv")
```

2.4 quotes

When you have a column with a delimiter that used to split the columns, use `quotes` option to specify the quote character, by default it is " and delimiters inside quotes are ignored. but using this option you can set any character.

2.5 nullValues

Using `nullValues` option you can specify the string in a CSV to consider as null. For example, if you want to consider a date column with a value **`"1900-01-01"`** set null on DataFrame.

2.6 dateFormat

`dateFormat` option to used to set the format of the input DateType and TimestampType columns. Supports all `java.text.SimpleDateFormat` formats.

Note: Besides the above options, PySpark CSV API also supports many other options, please refer to this article for details.

3. Specify Custom Schema

Reading CSV files with a user-specified custom schema in PySpark involves defining the schema explicitly before loading the data. You can define the schema for the CSV file by specifying the column names and data types using the StructType and StructField classes. These are from the `pyspark.sql.types module`.

```
# Imports
from pyspark.sql.types import StructType,StructField, StringType, IntegerType
from pyspark.sql.types import ArrayType, DoubleType, BooleanType
```

```
# Using custom schema
schema = StructType()
.add("RecordNumber",IntegerType(),True)
.add("Zipcode",IntegerType(),True)
.add("ZipCodeType",StringType(),True)
.add("City",StringType(),True)
.add("State",StringType(),True)
.add("LocationType",StringType(),True)
.add("Lat",DoubleType(),True)
.add("Long",DoubleType(),True)
```

```
.add("Xaxis",IntegerType(),True)
.add("Yaxis",DoubleType(),True)
.add("Zaxis",DoubleType(),True)
.add("WorldRegion",StringType(),True)
.add("Country",StringType(),True)
.add("LocationText",StringType(),True)
.add("Location",StringType(),True)
.add("Decommisioned",BooleanType(),True)
.add("TaxReturnsFiled",StringType(),True)
.add("EstimatedPopulation",IntegerType(),True)
.add("TotalWages",IntegerType(),True)
.add("Notes",StringType(),True)

df_with_schema = spark.read.format("csv")
.option("header", True)
.schema(schema)
.load("/path/zipcodes.csv")
```

Using a user-specified custom schema provides flexibility in handling CSV files with specific data types or column names, ensuring that the DataFrame accurately represents the data according to the user's requirements.

4. DataFrame Transformations

PySpark DataFrame transformations involve applying various operations to manipulate the data within a DataFrame. These transformations include:

Filtering: Selecting rows from the DataFrame based on specified conditions.**Selecting Columns**: Extracting specific columns from the DataFrame.**Adding Columns**: Creating new columns by performing computations or transformations on existing columns.**Dropping Columns**: Removing unnecessary columns from the DataFrame.**Grouping and Aggregating**: Grouping rows based on certain criteria and computing aggregate statistics, such as sum, average, count, etc., within each group.**Sorting**: Arranging the rows of the DataFrame in a specified order based on column values.**Joining**: Combining two DataFrames based on a common key or condition.**Union**: Concatenating two DataFrames vertically, adding rows from one DataFrame to another.**Pivoting and Melting**: Reshaping the DataFrame from long to wide format (pivoting) or from wide to long format (melting).**Window Functions**: Performing calculations over a sliding window of rows, such as computing moving averages or ranking.

5. Writing PySpark DataFrame to CSV file

To write a PySpark DataFrame to a CSV file, you can use the `write.csv()` method provided by the DataFrame API. This method takes a path as an argument, where the CSV file will be saved. Optionally, you can specify additional parameters such as the delimiter, header inclusion, and whether to overwrite existing files. Here's how you can do it:

```
# Save DataFrame to CSV File
df.write.option("header",True)
.csv("/tmp/spark_output/zipcodes")
```

`option("header", True)`: This specifies an option for the write operation. In this case, it sets the header option to True, indicating that the CSV file should include a header row with column names.

5.1 Options

When writing a DataFrame to a CSV file in PySpark, you can specify various options to customize the output. These options can be set using the `option()` method of the DataFrameWriter class. Here's how to use write options with a CSV file:

```
# Using write options
df2.write.options(header='True', delimiter=',')
.csv("/tmp/spark_output/zipcodes")
```

Here are some commonly used options:

header: Specifies whether to include a header row with column names in the CSV file. Example: `option("header", "true")`.**delimiter**: Specifies the delimiter to use between fields in the CSV file. Example: `option("delimiter", ",")`.**quote**: Specifies the character used for quoting fields in the CSV file. Example: `option("quote", """)`.**escape**: Specifies the escape character used in the CSV file. Example: `option("escape", "\")`.**nullValue**: Specifies the string to represent null values in the CSV file. Example: `option("nullValue", "NA")`.**dateFormat**: Specifies the date format to use for date columns. Example: `option("dateFormat", "yyyy-MM-dd")`.**mode**: Specifies the write mode for the output. Options include "overwrite", "append", "ignore", and "error". Example: `option("mode", "overwrite")`.**compression**: Specifies the compression codec to use for the output file. Example: `option("compression", "gzip")`.

5.2 Saving modes

You can specify different saving modes while writing PySpark DataFrame to disk. These saving modes specify how to write a file to disk.

`overwrite` - Overwrite the existing file if already exists.

`append` - New rows are appended to the existing rows.

`ignore` - When this option is used, it ignores the writing operation when the file already exists.

`error` - This option returns an error when the file already exists. This is a default option.

```
df2.write.mode('overwrite').csv("/tmp/spark_output/zipcodes")
```

# You can also use this
df2.write.format("csv").mode('overwrite').save("/tmp/spark_output/zipcodes")

6. PySpark Read CSV Complete Example

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType, IntegerType
from pyspark.sql.types import ArrayType, DoubleType, BooleanType
from pyspark.sql.functions import col,array_contains
```

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

df = spark.read.csv("/tmp/resources/zipcodes.csv")

df.printSchema()

df2 = spark.read.option("header",True)
.csv("/tmp/resources/zipcodes.csv")
df2.printSchema()

df3 = spark.read.options(header='True', delimiter=',')
.csv("/tmp/resources/zipcodes.csv")
df3.printSchema()

schema = StructType()
.add("RecordNumber",IntegerType(),True)

```
.add("Zipcode",IntegerType(),True)
.add("ZipCodeType",StringType(),True)
.add("City",StringType(),True)
.add("State",StringType(),True)
.add("LocationType",StringType(),True)
.add("Lat",DoubleType(),True)
.add("Long",DoubleType(),True)
.add("Xaxis",IntegerType(),True)
.add("Yaxis",DoubleType(),True)
.add("Zaxis",DoubleType(),True)
.add("WorldRegion",StringType(),True)
.add("Country",StringType(),True)
.add("LocationText",StringType(),True)
.add("Location",StringType(),True)
.add("Decommisioned",BooleanType(),True)
.add("TaxReturnsFiled",StringType(),True)
.add("EstimatedPopulation",IntegerType(),True)
.add("TotalWages",IntegerType(),True)
.add("Notes",StringType(),True)

df_with_schema = spark.read.format("csv")
.option("header", True)
.schema(schema)
.load(/tmp/resources/zipcodes.csv")
df_with_schema.printSchema()

df2.write.option("header",True)
.csv("/tmp/spark_output/zipcodes123")
```

7. Conclusion:

In conclusion, reading CSV files from disk using PySpark offers a versatile and efficient approach to data ingestion and processing. In this article, you have learned the importance of specifying options such as schema, delimiter, and header handling to ensure accurate DataFrame creation. Also, you learned to read a CSV file multiple csv files, all files from a folder e.t.c

Happy Learning !!

Related Articles

References: