PYSPARK DATAFRAME

PySpark – Create DataFrame with Examples:

You can manually c**reate a PySpark DataFrame** using toDF() and createDataFrame() methods, both these function takes different signatures in order to create DataFrame from existing RDD, list, and DataFrame.You can also create PySpark DataFrame from data sources like TXT, CSV, JSON, ORV, Avro, Parquet, XML formats by reading from HDFS, S3, DBFS, Azure Blob file systems e.t.c.

SPARKSESSION	RDD	DATAFRAME
createDataFrame(rdd)	toDF()	toDF(*cols)
createDataFrame(dataList)	toDF(*cols)	
createDataFrame(rowData,columns)		
createDataFrame(dataList,schema)		

In order to create a DataFrame from a list we need the data hence, first, let's create the data and the columns that are needed.

```
columns = ["language","users_count"]
data = [("Java", "20000"), ("Python", "100000"), ("Scala", "3000")]
```

1. Create DataFrame from RDD
One easy way to manually create PySpark DataFrame is from an existing RDD. first, let's create a Spark RDD from a collection List by calling parallelize() function from SparkContext . We would need this **rdd** object for all our examples below.

```
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
rdd = spark.sparkContext.parallelize(data)
```

1.1 Using toDF() function
PySpark RDD's toDF() method is used to create a DataFrame from the existing RDD. Since RDD doesn't have columns, the DataFrame is created with default column names "_1" and "_2" as we have two columns.

```
dfFromRDD1 = rdd.toDF()
dfFromRDD1.printSchema()
```

PySpark printschema() yields the schema of the DataFrame to console.

```
root
 |-- _1: string (nullable = true)
 |-- _2: string (nullable = true)
```

If you wanted to provide column names to the DataFrame use toDF() method with column names as arguments as shown below.

```
columns = ["language","users_count"]
dfFromRDD1 = rdd. toDF(columns)
dfFromRDD1.printSchema()
```

This yields the schema of the DataFrame with column names. use the show() method on PySpark DataFrame to show the DataFrame

```
root
 |-- language: string (nullable = true)
 |-- users: string (nullable = true)
```

By default, the datatype of these columns infers to the type of data. We can change this behavior by supplying schema, where we can specify a column name, data type, and nullable for each field/column.

1.2 Using createDataFrame () from SparkSession

Using createDataFrame () from SparkSession is another way to create manually and it takes rdd object as an argument. and chain with toDF () to specify name to the columns.

```
dfFromRDD2 = spark. createDataFrame(rdd). toDF(*columns)
```

2. Create DataFrame from List Collection

In this section, we will see how to create PySpark DataFrame from a list. These examples would be similar to what we have seen in the above section with RDD, but we use the list data object instead of “rdd” object to create DataFrame.

2.1 Using createDataFrame() from SparkSession

Calling createDataFrame() from SparkSession is another way to create PySpark DataFrame manually, it takes a list object as an argument. and chain with toDF() to specify names to the columns.

```
dfFromData2 = spark.createDataFrame(data).toDF(*columns)
```

2.2 Using createDataFrame() with the Row type

createDataFrame() has another signature in PySpark which takes the collection of Row type and schema for column names as arguments. To use this first we need to convert our “data” object from the list to list of Row.

```
rowData = map(lambda x: Row(*x), data)
dfFromData3 = spark.createDataFrame(rowData,columns)
```

2.3 Create DataFrame with schema

If you wanted to specify the column names along with their data types, you should create the StructType schema first and then assign this while creating a DataFrame.

```
from pyspark.sql.types import StructType,StructField, StringType, IntegerType
data2 = [("James","","Smith","36636","M",3000),
    ("Michael","Rose","","40288","M",4000),
    ("Robert","","Williams","42114","M",4000),
    ("Maria","Anne","Jones","39192","F",4000),
    ("Jen","Mary","Brown","","F",-1)
  ]

schema = StructType([ \
    StructField("firstname",StringType(),True), \
    StructField("middlename",StringType(),True), \
    StructField("lastname",StringType(),True), \
    StructField("id", StringType(), True), \
    StructField("gender", StringType(), True), \
    StructField("salary", IntegerType(), True) \
  ])

df = spark.createDataFrame(data=data2,schema=schema)
df.printSchema()
df.show(truncate=False)
```

This yields below output.

```
root
 |-- firstname: string (nullable = true)
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)

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

3. Create DataFrame from Data sources
In real-time mostly you create DataFrame from data source files like CSV, Text, JSON, XML e.t.c.

PySpark by default supports many data formats out of the box without importing any libraries and to create DataFrame you need to use the appropriate method available in DataFrameReader class.

3.1 Creating DataFrame from CSV
Use csv() method of the DataFrameReader object to create a DataFrame from CSV file. you can also provide options like what delimiter to use, whether you have quoted data, date formats, infer schema, and many more. Please refer PySpark Read CSV into DataFrame

```
df2 = spark.read.csv("/src/resources/file.csv")
```

3.2. Creating from text (TXT) file
Similarly you can also create a DataFrame by reading a from Text file, use text() method of the DataFrameReader to do so.

```
df2 = spark.read.text("/src/resources/file.txt")
```

3.3. Creating from JSON file
PySpark is also used to process semi-structured data files like JSON format. you can use json() method of the DataFrameReader to read JSON file into DataFrame. Below is a simple example.

```
df2 = spark.read.json("/src/resources/file.json")
```

Similarly, we can create DataFrame in PySpark from most of the relational databases which I’ve not covered here and I will leave this to you to explore.

PySpark – Create an Empty DataFrame & RDD

While working with files, sometimes we may not receive a file for processing, however, we still need to create a DataFrame manually with the same schema we expect. If we don’t create with the same schema, our operations/transformations (like union’s) on DataFrame fail as we refer to the columns that may not present.

To handle situations similar to these, we always need to create a DataFrame with the same schema, which means the same column names and datatypes regardless of the file exists or empty file processing.

1. Create Empty RDD in PySpark

Create an empty RDD by using `emptyRDD()` of SparkContext for example `spark.sparkContext.emptyRDD()`.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

#Creates Empty RDD
emptyRDD = spark.sparkContext.emptyRDD()
print(emptyRDD)

#Diplays
#EmptyRDD[188] at emptyRDD
```

Alternatively you can also get empty RDD by using `spark.sparkContext.parallelize([])`.

```
#Creates Empty RDD using parallelize
rdd2= spark.sparkContext.parallelize([])
```

```
print(rdd2)

#EmptyRDD[205] at emptyRDD at NativeMethodAccessorImpl.java:0
#ParallelCollectionRDD[206] at readRDDFromFile at PythonRDD.scala:262
```

Note: If you try to perform operations on empty RDD you going to get `ValueError("RDD is empty")`.

2. Create Empty DataFrame with Schema (StructType)

In order to create an empty PySpark DataFrame manually with schema (column names & data types) first, Create a schema using StructType and StructField .

```
#Create Schema
from pyspark.sql.types import StructType,StructField, StringType
schema = StructType([
  StructField('firstname', StringType(), True),
  StructField('middlename', StringType(), True),
  StructField('lastname', StringType(), True)
  ])
```

Now use the empty RDD created above and pass it to `createDataFrame()` of SparkSession along with the schema for column names & data types.

```
#Create empty DataFrame from empty RDD
df = spark.createDataFrame(emptyRDD,schema)
df.printSchema()
```

This yields below schema of the empty DataFrame.

```
root
 |-- firstname: string (nullable = true)
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
```

3. Convert Empty RDD to DataFrame

You can also create empty DataFrame by converting empty RDD to DataFrame using `toDF()`.

```
#Convert empty RDD to Dataframe
df1 = emptyRDD.toDF(schema)
df1.printSchema()
```

4. Create Empty DataFrame with Schema.

So far I have covered creating an empty DataFrame from RDD, but here will create it manually with schema and without RDD.

```
#Create empty DataFrame directly.
df2 = spark.createDataFrame([], schema)
df2.printSchema()
```

5. Create Empty DataFrame without Schema (no columns)

To create empty DataFrame with out schema (no columns) just create a empty schema and use it while creating PySpark DataFrame.

```
#Create empty DatFrame with no schema (no columns)
df3 = spark.createDataFrame([], StructType([]))
df3.printSchema()

#print below empty schema
#root
```

Convert PySpark RDD to DataFrame

In PySpark, `toDF()` function of the RDD is used to convert RDD to DataFrame. We would need to convert RDD to DataFrame as DataFrame provides more advantages over RDD. For instance, DataFrame is a distributed collection of data organized into named columns similar to Database tables and provides optimization and performance improvements.

- Create PySpark RDD
- Convert PySpark RDD to DataFrame
  - using toDF()
  - using createDataFrame()
  - using RDD row type & schema

1. Create PySpark RDD

First, let's create an RDD by passing Python list object to `sparkContext.parallelize()` function. We would need this `rdd` object for all our examples below.

In PySpark, when you have data in a list meaning you have a collection of data in a PySpark driver memory when you create an RDD, this collection is going to be parallelized.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
dept = [("Finance",10),("Marketing",20),("Sales",30),("IT",40)]
rdd = spark.sparkContext.parallelize(dept)
```

2. Convert PySpark RDD to DataFrame

Converting PySpark RDD to DataFrame can be done using toDF(), createDataFrame(). In this section, I will explain these two methods.

2.1 Using rdd.toDF() function

PySpark provides toDF() function in RDD which can be used to convert RDD into Dataframe

```
df = rdd.toDF()
df.printSchema()
df.show(truncate=False)
```

By default, toDF() function creates column names as "_1" and "_2". This snippet yields below schema.

```
root
 |-- _1: string (nullable = true)
 |-- _2: long (nullable = true)

+---------+---+
|_1       |_2 |
+---------+---+
|Finance  |10 |
|Marketing|20 |
|Sales    |30 |
|IT       |40 |
+---------+---+
```

toDF() has another signature that takes arguments to define column names as shown below.

```
deptColumns = ["dept_name","dept_id"]
df2 = rdd.toDF(deptColumns)
df2.printSchema()
df2.show(truncate=False)
```

Outputs below schema.

```
root
 |-- dept_name: string (nullable = true)
 |-- dept_id: long (nullable = true)

+---------+-------+
|dept_name|dept_id|
+---------+-------+
|Finance  |10     |
|Marketing|20     |
|Sales    |30     |
|IT       |40     |
+---------+-------+
```

2.2 Using PySpark createDataFrame() function

SparkSession class provides createDataFrame() method to create DataFrame and it takes rdd object as an argument.

```
deptDF = spark.createDataFrame(rdd, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)
```

This yields the same output as above.

2.3 Using createDataFrame() with StructType schema

When you infer the schema, by default the datatype of the columns is derived from the data and set's nullable to true for all columns. We can change this behavior by supplying schema using StructType – where we can specify a column name, data type and nullable for each field/column.

If you wanted to know more about StructType, please go through how to use StructType and StructField to define the custom schema.

```
from pyspark.sql.types import StructType,StructField, StringType
deptSchema = StructType([
   StructField('dept_name', StringType(), True),
   StructField('dept_id', StringType(), True)
])

deptDF1 = spark.createDataFrame(rdd, schema = deptSchema)
```

```
deptDF1.printSchema()
deptDF1.show(truncate=False)
```

3. Complete Example

```

import pyspark
from pyspark.sql import SparkSession

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

dept = [("Finance",10),("Marketing",20),("Sales",30),("IT",40)]
rdd = spark.sparkContext.parallelize(dept)

df = rdd.toDF()
df.printSchema()
df.show(truncate=False)

deptColumns = ["dept_name","dept_id"]
df2 = rdd.toDF(deptColumns)
df2.printSchema()
df2.show(truncate=False)

deptDF = spark.createDataFrame(rdd, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)

from pyspark.sql.types import StructType,StructField, StringType
deptSchema = StructType([
    StructField('dept_name', StringType(), True),
    StructField('dept_id', StringType(), True)
])

deptDF1 = spark.createDataFrame(rdd, schema = deptSchema)
deptDF1.printSchema()
deptDF1.show(truncate=False)
```

4. Conclusion:

In this article, you have learned how to convert PySpark RDD to DataFrame, we would need these frequently while working in PySpark as these provides optimization and performance over RDD.

Convert PySpark DataFrame to Pandas

- Post author:NNK
- Post category:Pandas / PySpark / Python

(Spark with Python) PySpark DataFrame can be converted to Python pandas DataFrame using a function toPandas(), In this article, I will explain how to create Pandas DataFrame from PySpark (Spark) DataFrame with examples.
Before we start first understand the main differences between the Pandas & PySpark, operations on Pyspark run faster than Pandas due to its distributed nature and parallel execution on multiple cores and machines.

In other words, pandas run operations on a single node whereas PySpark runs on multiple machines. If you are working on a Machine Learning application where you are dealing with larger datasets, PySpark processes operations many times faster than pandas. Refer to pandas DataFrame Tutorial beginners guide with examples
After processing data in PySpark we would need to convert it back to Pandas DataFrame for a further procession with Machine Learning application or any Python applications.

Prepare PySpark DataFrame

In order to explain with an example first let’s create a PySpark DataFrame.

```
import pyspark
from pyspark.sql import SparkSession

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

data = [("James","","Smith","36636","M",60000),
        ("Michael","Rose","","40288","M",70000),
        ("Robert","","Williams","42114","",400000),
        ("Maria","Anne","Jones","39192","F",500000),
        ("Jen","Mary","Brown","","F",0)]

columns = ["first_name","middle_name","last_name","dob","gender","salary"]
pysparkDF = spark.createDataFrame(data = data, schema = columns)
pysparkDF.printSchema()
pysparkDF.show(truncate=False)
```

This yields below schema and result of the DataFrame.

```
root
 |-- first_name: string (nullable = true)
 |-- middle_name: string (nullable = true)
 |-- last_name: string (nullable = true)
 |-- dob: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: long (nullable = true)

+----------+-----------+---------+-----+------+------+
|first_name|middle_name|last_name|dob  |gender|salary|
+----------+-----------+---------+-----+------+------+
|James     |           |Smith    |36636|M     |60000 |
|Michael   |Rose       |         |40288|M     |70000 |
|Robert    |           |Williams |42114|      |400000|
|Maria     |Anne       |Jones    |39192|F     |500000|
|Jen       |Mary       |Brown    |     |F     |0     |
+----------+-----------+---------+-----+------+------+
```

Convert PySpark Dataframe to Pandas DataFrame

PySpark DataFrame provides a method toPandas() to convert it to Python Pandas DataFrame.
toPandas() results in the collection of all records in the PySpark DataFrame to the driver program and should be done only on a small subset of the data. running on larger dataset’s results in memory error and crashes the application. To deal with a larger dataset, you can also try increasing memory on the driver.

```
pandasDF = pysparkDF.toPandas()
print(pandasDF)
```

This yields the below panda’s DataFrame. Note that pandas add a sequence number to the result as a row Index. You can rename pandas columns by using rename() function.

```
  first_name middle_name last_name    dob gender  salary
0      James                 Smith  36636      M   60000
1    Michael        Rose            40288      M   70000
2     Robert              Williams  42114         400000
3      Maria        Anne     Jones  39192      F  500000
4        Jen        Mary     Brown             F       0
```

I have dedicated Python pandas Tutorial with Examples where I explained pandas concepts in detail.

Convert Spark Nested Struct DataFrame to Pandas

Most of the time data in PySpark DataFrame will be in a structured format meaning one column contains other columns so let’s see how it convert to Pandas. Here is an example with nested struct where we have firstname, middlename and lastname are part of the name column.

```
# Nested structure elements
from pyspark.sql.types import StructType, StructField, StringType,IntegerType
dataStruct = [(("James","","Smith"),"36636","M","3000"), \
     (("Michael","Rose",""),"40288","M","4000"), \
     (("Robert","","Williams"),"42114","M","4000"), \
     (("Maria","Anne","Jones"),"39192","F","4000"), \
     (("Jen","Mary","Brown"),"","F","-1") \
]

schemaStruct = StructType([
       StructField('name', StructType([
           StructField('firstname', StringType(), True),
           StructField('middlename', StringType(), True),
           StructField('lastname', StringType(), True)
           ])),
        StructField('dob', StringType(), True),
       StructField('gender', StringType(), True),
       StructField('salary', StringType(), True)
       ])
df = spark.createDataFrame(data=dataStruct, schema = schemaStruct)
df.printSchema()

pandasDF2 = df.toPandas()
print(pandasDF2)
```

Converting structured DataFrame to Pandas DataFrame results below output.

```
                 name    dob gender salary
0      (James, , Smith)  36636      M   3000
1     (Michael, Rose, )  40288      M   4000
2  (Robert, , Williams)  42114      M   4000
3  (Maria, Anne, Jones)  39192      F   4000
4    (Jen, Mary, Brown)             F     -1
```

Conclusion

In this simple article, you have learned to convert Spark DataFrame to pandas using toPandas() function of the Spark DataFrame. also have seen a similar example with complex nested structure elements. toPandas() results in the collection of all records in the DataFrame to the driver program and should be done on a small subset of the data.

PySpark show() – Display DataFrame Contents in Table

PySpark DataFrame show() is used to display the contents of the DataFrame in a Table Row & Column Format. By default, it shows only 20 Rows, and the column values are truncated at 20 characters.

2. PySpark DataFrame show () Syntax & Example

1.1 Syntax

```
def show (self, n=20, truncate=True, vertical=False):
```

1.2 Example

Use show() method to display the contents of the DataFrame and use pyspark printSchema() method to print the schema.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
columns = ["Seqno","Quote"]
data = [("1", "Be the change that you wish to see in the world"),
    ("2", "Everyone thinks of changing the world, but no one thinks of changing
himself."),
    ("3", "The purpose of our lives is to be happy."),
    ("4", "Be cool.")]
df = spark.createDataFrame(data,columns)
df.show()

# Output
#+-----+--------------------+
#|Seqno|               Quote|
#+-----+--------------------+
#|    1|Be the change tha...|
#|    2|Everyone thinks o...|
#|    3|The purpose of ou...|
#|    4|            Be cool.|
#+-----+--------------------+
```

As you see above, values in the Quote column are truncated at 20 characters, Let's see how to display the full column contents.

```
#Display full column contents
df.show(truncate=False)

#+-----+-----------------------------------------------------------------------------+
#|Seqno|Quote                                                                        |
#+-----+-----------------------------------------------------------------------------+
#|1    |Be the change that you wish to see in the world                              |
#|2    |Everyone thinks of changing the world, but no one thinks of changing himself.|
#|3    |The purpose of our lives is to be happy.                                     |
#|4    |Be cool.                                                                     |
#+-----+-----------------------------------------------------------------------------+
```

By default show() method displays only 20 rows from PySpark DataFrame. The below example limit the rows to 2 and full column contents. Our DataFrame has just 4 rows hence I can’t demonstrate with more than 4 rows. If you have a DataFrame with thousands of rows try changing the value from 2 to 100 to display more than 20 rows.

```
# Display 2 rows and full column contents
df.show(2,truncate=False)

#+-----+-----------------------------------------------------------------------------+
#|Seqno|Quote                                                                        |
#+-----+-----------------------------------------------------------------------------+
#|1    |Be the change that you wish to see in the world                              |
#|2    |Everyone thinks of changing the world, but no one thinks of changing himself.|
#+-----+-----------------------------------------------------------------------------+
```

You can also truncate the column value at the desired length.

```
# Display 2 rows & column values 25 characters
df.show(2,truncate=25)

#+-----+-------------------------+
#|Seqno|                    Quote|
#+-----+-------------------------+
#|    1|Be the change that you...|
#|    2|Everyone thinks of cha...|
#+-----+-------------------------+
#only showing top 2 rows
```

Finally, let’s see how to display the DataFrame vertically record by record.

```
# Display DataFrame rows & columns vertically
```

```
df.show(n=3,truncate=25,vertical=True)

#-RECORD 0-------------------------
# Seqno | 1
# Quote | Be the change that you...
#-RECORD 1-------------------------
# Seqno | 2
# Quote | Everyone thinks of cha...
#-RECORD 2-------------------------
# Seqno | 3
# Quote | The purpose of our liv...
```

PySpark StructType & StructField Explained with ExamplePySpark StructType & StructField classes are used to programmatically specify the schema to the DataFrame and create complex columns like nested struct, array, and map columns. StructType is a collection of StructField's that defines column name, column data type, boolean to specify if the field can be nullable or not and metadata.

In this article, I will explain different ways to define the structure of DataFrame using StructType with PySpark examples. Though PySpark infers a schema from data, sometimes we may need to define our own column names and data types and this article explains how to define simple, nested, and complex schemas.

1. StructType – Defines the structure of the Dataframe

PySpark provides from `pyspark.sql.types import StructType` class to define the structure of the DataFrame.StructType is a collection or list of StructField objects.PySpark printSchema() method on the DataFrame shows StructType columns as `struct`.

2. StructField – Defines the metadata of the DataFrame column

PySpark provides `pyspark.sql.types import StructField` class to define the columns which include column name(String), column type (DataType), nullable column (Boolean) and metadata (MetaData)

3. Using PySpark StructType & StructField with DataFrame

While creating a PySpark DataFrame we can specify the structure using StructType and StructField classes. As specified in the introduction, StructType is a collection of StructField's which is used to define the column name, data type, and a flag for nullable or not. Using StructField we can also add nested struct schema, ArrayType for arrays, and MapType for key-value pairs which we will discuss in detail in later sections.

The below example demonstrates a very simple example of how to create a StructType & StructField on DataFrame and it's usage with sample data to support it.

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType, IntegerType

spark = SparkSession.builder.master("local[1]") \
                .appName('SparkByExamples.com') \
                .getOrCreate()

data = [("James","","Smith","36636","M",3000),
   ("Michael","Rose","","40288","M",4000),
   ("Robert","","Williams","42114","M",4000),
   ("Maria","Anne","Jones","39192","F",4000),
   ("Jen","Mary","Brown","","F",-1)
  ]

schema = StructType([ \
   StructField("firstname",StringType(),True), \
   StructField("middlename",StringType(),True), \
   StructField("lastname",StringType(),True), \
   StructField("id", StringType(), True), \
   StructField("gender", StringType(), True), \
   StructField("salary", IntegerType(), True) \
  ])

df = spark.createDataFrame(data=data,schema=schema)
df.printSchema()
df.show(truncate=False)
```

By running the above snippet, it displays below outputs.

```
root
 |-- firstname: string (nullable = true)
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)

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

4. Defining Nested StructType object struct

While working on DataFrame we often need to work with the nested struct column and this can be defined using StructType.

In the below example column “name” data type is StructType which is nested.

```
structureData = [
   (("James","","Smith"),"36636","M",3100),
   (("Michael","Rose",""),"40288","M",4300),
   (("Robert","","Williams"),"42114","M",1400),
   (("Maria","Anne","Jones"),"39192","F",5500),
   (("Jen","Mary","Brown"),"","F",-1)
  ]
structureSchema = StructType([
        StructField('name', StructType([
             StructField('firstname', StringType(), True),
             StructField('middlename', StringType(), True),
             StructField('lastname', StringType(), True)
             ])),
         StructField('id', StringType(), True),
         StructField('gender', StringType(), True),
         StructField('salary', IntegerType(), True)
         ])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)
df2.printSchema()
df2.show(truncate=False)
```

Outputs below schema and the DataFrame

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)

+--------------------+-----+------+------+
|name                |id   |gender|salary|
+--------------------+-----+------+------+
|[James, , Smith]    |36636|M     |3100  |
|[Michael, Rose, ]   |40288|M     |4300  |
|[Robert, , Williams]|42114|M     |1400  |
|[Maria, Anne, Jones]|39192|F     |5500  |
|[Jen, Mary, Brown]  |     |F     |-1    |
+--------------------+-----+------+------+
```

5. Adding & Changing struct of the DataFrame

Using PySpark SQL function struct(), we can change the struct of the existing DataFrame and add a new StructType to it. The below example demonstrates how to copy the columns from one structure to another and adding a new column. PySpark Column Class also provides some functions to work with the StructType column.

```
from pyspark.sql.functions import col,struct,when
updatedDF = df2.withColumn("OtherInfo",
    struct(col("id").alias("identifier"),
    col("gender").alias("gender"),
    col("salary").alias("salary"),
    when(col("salary").cast(IntegerType()) < 2000,"Low")
      .when(col("salary").cast(IntegerType()) < 4000,"Medium")
      .otherwise("High").alias("Salary_Grade")
  )).drop("id","gender","salary")

updatedDF.printSchema()
updatedDF.show(truncate=False)
```

Here, it copies “gender“, “salary” and “id” to the new struct “otherInfo” and add’s a new column “Salary_Grade“.

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- OtherInfo: struct (nullable = false)
 |    |-- identifier: string (nullable = true)
 |    |-- gender: string (nullable = true)
 |    |-- salary: integer (nullable = true)
 |    |-- Salary_Grade: string (nullable = false)
```

6. Using SQL ArrayType and MapType

SQL StructType also supports ArrayType and MapType to define the DataFrame columns for array and map collections respectively. On the below example, column hobbies defined as ArrayType(StringType) and properties defined as MapType(StringType,StringType) meaning both key and value as String.

```
arrayStructureSchema = StructType([
   StructField('name', StructType([
      StructField('firstname', StringType(), True),
      StructField('middlename', StringType(), True),
      StructField('lastname', StringType(), True)
      ])),
      StructField('hobbies', ArrayType(StringType()), True),
      StructField('properties', MapType(StringType(),StringType()), True)
   ])
```

Outputs the below schema. Note that field Hobbies is array type and properties is map type.

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- hobbies: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

7. Creating StructType object struct from JSON file

If you have too many columns and the structure of the DataFrame changes now and then, it's a good practice to load the SQL StructType schema from JSON file. You can get the schema by using `df2.schema.json()` , store this in a file and will use it to create a the schema from this file.

```
print(df2.schema.json())

{
  "type" : "struct",
  "fields" : [ {
    "name" : "name",
    "type" : {
      "type" : "struct",
      "fields" : [ {
        "name" : "firstname",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      }, {
        "name" : "middlename",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      }, {
        "name" : "lastname",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      } ]
    },
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "dob",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "gender",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "salary",
    "type" : "integer",
    "nullable" : true,
    "metadata" : { }
  } ]
}
```

Alternatively, you could also use df.schema.simpleString(), this will return an relatively simpler schema format.
Now let's load the json file and use it to create a DataFrame.

```
import json
schemaFromJson = StructType.fromJson(json.loads(schema.json))
df3 = spark.createDataFrame(
      spark.sparkContext.parallelize(structureData),schemaFromJson)
df3.printSchema()
```

This prints the same output as the previous section. You can also, have a name, type, and flag for nullable in a comma-separated file and we can use these to create a StructType programmatically, I will leave this to you to explore.

8. Creating StructType object struct from DDL String

Like loading structure from JSON string, we can also create it from DLL (by using fromDDL() static function on SQL StructType class StructType.fromDDL). You can also generate DDL from a schema using toDDL(). printTreeString() on struct object prints the schema similar to printSchemafunction returns.

```
  ddlSchemaStr = "`fullName` STRUCT<`first`: STRING, `last`: STRING,
 `middle`: STRING>,`age` INT,`gender` STRING"
  ddlSchema = StructType.fromDDL(ddlSchemaStr)
  ddlSchema.printTreeString()
```

9. Checking if a Column Exists in a DataFrame

If you want to perform some checks on metadata of the DataFrame, for example, if a column or field exists in a DataFrame or data type of column; we can easily do this using several functions on SQL StructType and StructField.

```
print(df.schema.fieldNames.contains("firstname"))
print(df.schema.contains(StructField("firstname",StringType,true)))
```

This example returns "true" for both scenarios. And for the second one if you have IntegerType instead of StringType it returns false as the datatype for first name column is String, as it checks every property in a field. Similarly, you can also check if two schemas are equal and more.

10. Complete Example of PySpark StructType & StructField

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType,
IntegerType,ArrayType,MapType
from pyspark.sql.functions import col,struct,when

spark = SparkSession.builder.master("local[1]") \
                .appName('SparkByExamples.com') \
                .getOrCreate()

data = [("James","","Smith","36636","M",3000),
   ("Michael","Rose","","40288","M",4000),
   ("Robert","","Williams","42114","M",4000),
   ("Maria","Anne","Jones","39192","F",4000),
   ("Jen","Mary","Brown","","F",-1)
  ]

schema = StructType([
   StructField("firstname",StringType(),True),
   StructField("middlename",StringType(),True),
   StructField("lastname",StringType(),True),
   StructField("id", StringType(), True),
   StructField("gender", StringType(), True),
   StructField("salary", IntegerType(), True)
  ])

df = spark.createDataFrame(data=data,schema=schema)
df.printSchema()
df.show(truncate=False)

structureData = [
   (("James","","Smith"),"36636","M",3100),
   (("Michael","Rose",""),"40288","M",4300),
   (("Robert","","Williams"),"42114","M",1400),
   (("Maria","Anne","Jones"),"39192","F",5500),
   (("Jen","Mary","Brown"),"","F",-1)
  ]
structureSchema = StructType([
      StructField('name', StructType([
          StructField('firstname', StringType(), True),
          StructField('middlename', StringType(), True),
          StructField('lastname', StringType(), True)
          ])),
       StructField('id', StringType(), True),
       StructField('gender', StringType(), True),
         StructField('salary', IntegerType(), True)
         ])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)
df2.printSchema()
df2.show(truncate=False)


updatedDF = df2.withColumn("OtherInfo",
    struct(col("id").alias("identifier"),
    col("gender").alias("gender"),
    col("salary").alias("salary"),
    when(col("salary").cast(IntegerType()) < 2000,"Low")
      .when(col("salary").cast(IntegerType()) < 4000,"Medium")
      .otherwise("High").alias("Salary_Grade")
  )).drop("id","gender","salary")

updatedDF.printSchema()
updatedDF.show(truncate=False)


""" Array & Map"""


arrayStructureSchema = StructType([
    StructField('name', StructType([
       StructField('firstname', StringType(), True),
       StructField('middlename', StringType(), True),
       StructField('lastname', StringType(), True)
       ])),
       StructField('hobbies', ArrayType(StringType()), True),
       StructField('properties', MapType(StringType(),StringType()), True)
    ])
```

PySpark Row using on DataFrame and RDD

In PySpark Row class is available by importing pyspark.sql.Row which is represented as a record/row in DataFrame, one can create a Row object by using named arguments, or create a custom Row like class. In this article I will explain how to use Row class on RDD, DataFrame and its functions.

Before we start using it on RDD & DataFrame, let’s understand some basics of Row class.

Related Article: PySpark Column Class Usage & Functions with Examples

Key Points of Row Class:

- Earlier to Spark 3.0, when used Row class with named arguments, the fields are sorted by name.
- Since 3.0, Rows created from named arguments are not sorted alphabetically instead they will be ordered in the position entered.
- To enable sorting by names, set the environment variable `PYSPARK_ROW_FIELD_SORTING_ENABLED` to `true`.
- Row class provides a way to create a struct-type column as well.
-

1. Create a Row Object

Row class extends the tuple hence it takes variable number of arguments, Row() is used to create the row object. Once the row object created, we can retrieve the data from Row using index similar to tuple.

```
from pyspark.sql import Row
row=Row("James",40)
print(row[0] +","+str(row[1]))
```

This outputs `James,40`. Alternatively you can also write with named arguments. Benefits with the named argument is you can access with field name `row.name`. Below example print “Alice”.

```
row=Row(name="Alice", age=11)
print(row.name)
```

2. Create Custom Class from Row

We can also create a Row like class, for example “Person” and use it similar to Row object. This would be helpful when you wanted to create real time object and refer it’s properties. On below example, we have created a Person class and used similar to Row.

```
Person = Row("name", "age")
p1=Person("James", 40)
p2=Person("Alice", 35)
print(p1.name +","+p2.name)
```

This outputs `James,Alice`

3. Using Row class on PySpark RDD

We can use Row class on PySpark RDD. When you use Row to create an RDD, after collecting the data you will get the result back in Row.

```
from pyspark.sql import SparkSession, Row
```

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

data = [Row(name="James,,Smith",lang=["Java","Scala","C++"],state="CA"),
    Row(name="Michael,Rose,",lang=["Spark","Java","C++"],state="NJ"),
    Row(name="Robert,,Williams",lang=["CSharp","VB"],state="NV")]
rdd=spark.sparkContext.parallelize(data)
print(rdd.collect())
```

This yields below output.

```
[Row(name='James,,Smith', lang=['Java', 'Scala', 'C++'], state='CA'),
Row(name='Michael,Rose,', lang=['Spark', 'Java', 'C++'], state='NJ'),
Row(name='Robert,,Williams', lang=['CSharp', 'VB'], state='NV')]
```

Now, let’s collect the data and access the data using its properties.

```
collData=rdd.collect()
for row in collData:
    print(row.name + "," +str(row.lang))
```

This yields below output.

```
James,,Smith,['Java', 'Scala', 'C++']
Michael,Rose,,['Spark', 'Java', 'C++']
Robert,,Williams,['CSharp', 'VB']
```

Alternatively, you can also do by creating a Row like class “Person”

```
Person=Row("name","lang","state")
data = [Person("James,,Smith",["Java","Scala","C++"],"CA"),
    Person("Michael,Rose,",["Spark","Java","C++"],"NJ"),
    Person("Robert,,Williams",["CSharp","VB"],"NV")]
```

4. Using Row class on PySpark DataFrame

Similarly, Row class also can be used with PySpark DataFrame, By default data in DataFrame represent as Row. To demonstrate, I will use the same data that was created for RDD.

Note that Row on DataFrame is not allowed to omit a named argument to represent that the value is None or missing. This should be explicitly set to None in this case.

```
df=spark.createDataFrame(data)
df.printSchema()
df.show()
```

This yields below output. Note that DataFrame able to take the column names from Row object.

```
root
 |-- name: string (nullable = true)
 |-- lang: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- state: string (nullable = true)

+----------------+------------------+-----+
|            name|              lang|state|
+----------------+------------------+-----+
|    James,,Smith|[Java, Scala, C++]|   CA|
|   Michael,Rose,|[Spark, Java, C++]|   NJ|
|Robert,,Williams|       [CSharp, VB]|   NV|
+----------------+------------------+-----+
```

You can also change the column names by using `toDF()` function

```
columns = ["name","languagesAtSchool","currentState"]
df=spark.createDataFrame(data).toDF(*columns)
df.printSchema()
```

This yields below output, note the column name “languagesAtSchool” from the previous example.

```
root
 |-- name: string (nullable = true)
 |-- languagesAtSchool: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- currentState: string (nullable = true)
```

5. Create Nested Struct Using Row Class

The below example provides a way to create a struct type using the Row class. Alternatively, you can also create struct type using By Providing Schema using PySpark StructType & StructFields

```
#Create DataFrame with struct using Row class
from pyspark.sql import Row
data=[Row(name="James",prop=Row(hair="black",eye="blue")),
      Row(name="Ann",prop=Row(hair="grey",eye="black"))]
df=spark.createDataFrame(data)
df.printSchema()
```

Yields below schema

```
root
 |-- name: string (nullable = true)
 |-- prop: struct (nullable = true)
 |    |-- hair: string (nullable = true)
 |    |-- eye: string (nullable = true)
```

6. Complete Example of PySpark Row usage on RDD & DataFrame

Below is complete example for reference.

```
from pyspark.sql import SparkSession, Row

row=Row("James",40)
print(row[0] +","+str(row[1]))
row2=Row(name="Alice", age=11)
print(row2.name)

Person = Row("name", "age")
p1=Person("James", 40)
p2=Person("Alice", 35)
print(p1.name +","+p2.name)

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

data = [Row(name="James,,Smith",lang=["Java","Scala","C++"],state="CA"),
    Row(name="Michael,Rose,",lang=["Spark","Java","C++"],state="NJ"),
    Row(name="Robert,,Williams",lang=["CSharp","VB"],state="NV")]

#RDD Example 1
rdd=spark.sparkContext.parallelize(data)
collData=rdd.collect()
print(collData)
for row in collData:
    print(row.name + "," +str(row.lang))

# RDD Example 2
Person=Row("name","lang","state")
data = [Person("James,,Smith",["Java","Scala","C++"],"CA"),
    Person("Michael,Rose,",["Spark","Java","C++"],"NJ"),
    Person("Robert,,Williams",["CSharp","VB"],"NV")]
rdd=spark.sparkContext.parallelize(data)
collData=rdd.collect()
print(collData)
```

```
for person in collData:
    print(person.name + "," +str(person.lang))

#DataFrame Example 1
columns = ["name","languagesAtSchool","currentState"]
df=spark.createDataFrame(data)
df.printSchema()
df.show()

collData=df.collect()
print(collData)
for row in collData:
    print(row.name + "," +str(row.lang))

#DataFrame Example 2
columns = ["name","languagesAtSchool","currentState"]
df=spark.createDataFrame(data).toDF(*columns)
df.printSchema()
```

PySpark Column Class | Operators & Functions

pyspark.sql. Column class provides several functions to work with DataFrame to manipulate the Column values, evaluate the boolean expression to filter rows, retrieve a value or part of a value from a DataFrame column, and to work with list, map & struct columns.
In this article, I will cover how to create Column object, access them to perform operations, and finally most used PySpark Column Functions with Examples.

Related Article: PySpark Row Class with Examples
Key Points:

- PySpark Column class represents a single Column in a DataFrame.
- It provides functions that are most used to manipulate DataFrame Columns & Rows.
- Some of these Column functions evaluate a Boolean expression that can be used with filter () transformation to filter the DataFrame Rows.
- Provides functions to get a value from a list column by index, map value by key & index, and finally struct nested column.
- PySpark also provides additional functions pyspark.sql.functions that take Column object and return a Column type.

Note: Most of the pyspark.sql.functions return Column type hence it is very important to know the operation you can perform with Column type.

1. Create Column Class Object

One of the simplest ways to create a Column class object is by using PySpark lit() SQL function, this takes a literal value and returns a Column object.

```
from pyspark.sql.functions import lit
colObj = lit("sparkbyexamples.com")
```

You can also access the Column from DataFrame by multiple ways.

```
data=[("James",23),("Ann",40)]
df=spark.createDataFrame(data).toDF("name.fname","gender")
df.printSchema()
#root
# |-- name.fname: string (nullable = true)
# |-- gender: long (nullable = true)

# Using DataFrame object (df)
df.select(df.gender).show()
df.select(df["gender"]).show()
#Accessing column name with dot (with backticks)
df.select(df["`name.fname`"]).show()

#Using SQL col() function
from pyspark.sql.functions import col
df.select(col("gender")).show()
#Accessing column name with dot (with backticks)
df.select(col("`name.fname`")).show()
```

Below example demonstrates accessing struct type columns. Here I have use PySpark Row class to create a struct type. Alternatively you can also create it by using PySpark StructType & StructField classes

```
#Create DataFrame with struct using Row class
from pyspark.sql import Row
data=[Row(name="James",prop=Row(hair="black",eye="blue")),
      Row(name="Ann",prop=Row(hair="grey",eye="black"))]
df=spark.createDataFrame(data)
df.printSchema()
#root
# |-- name: string (nullable = true)
# |-- prop: struct (nullable = true)
# |    |-- hair: string (nullable = true)
# |    |-- eye: string (nullable = true)

#Access struct column
df.select(df.prop.hair).show()
df.select(df["prop.hair"]).show()
df.select(col("prop.hair")).show()
```

```
#Access all columns from struct
df.select(col("prop.*")).show()
```

2. PySpark Column Operators

PySpark column also provides a way to do arithmetic operations on columns using operators.

```
data=[(100,2,1),(200,3,4),(300,4,4)]
df=spark.createDataFrame(data).toDF("col1","col2","col3")

#Arthmetic operations
df.select(df.col1 + df.col2).show()
df.select(df.col1 - df.col2).show()
df.select(df.col1 * df.col2).show()
df.select(df.col1 / df.col2).show()
df.select(df.col1 % df.col2).show()

df.select(df.col2 > df.col3).show()
df.select(df.col2 < df.col3).show()
df.select(df.col2 == df.col3).show()
```

3. PySpark Column Functions

Let's see some of the most used Column Functions, on below table, I have grouped related functions together to make it easy, click on the link for examples.

COLUMN FUNCTION	FUNCTION DESCRIPTION

alias(*alias, **kwargs)
name(*alias, **kwargs)
	Provides alias to the column or expressions
	name() returns same as alias().

asc()
asc_nulls_first()
asc_nulls_last()
	Returns ascending order of the column.
	asc_nulls_first() Returns null values first then non-null values.
	asc_nulls_last() – Returns null values after non-null values.

astype(dataType)
cast(dataType)
	Used to cast the data type to another type.
	astype() returns same as cast().

between(lowerBound, upperBound)	Checks if the columns values are between lower and upper bound. Returns boolean value.

COLUMN FUNCTION	FUNCTION DESCRIPTION

bitwiseAND(other)
bitwiseOR(other)
bitwiseXOR(other)
	Compute bitwise AND, OR & XOR of this expression with another expression respectively.

contains(other)	Check if String contains in another string.

desc()
desc_nulls_first()
desc_nulls_last()
	Returns descending order of the column.
	desc_nulls_first() -null values appear before non-null values.
	desc_nulls_last() – null values appear after non-null values.

startswith(other)
endswith(other)
	String starts with. Returns boolean expression
	String ends with. Returns boolean expression

eqNullSafe(other)	Equality test that is safe for null values.
getField(name)	Returns a field by name in a StructField and by key in Map.
getItem(key)	Returns a values from Map/Key at the provided position.

isNotNull()
isNull()
	isNotNull() – Returns True if the current expression is NOT null.
	isNull() – Returns True if the current expression is null.

isin(*cols)	A boolean expression that is evaluated to true if the value of this expression is contained by the evaluated values of the arguments.

like(other)
rlike(other)
	Similar to SQL like expression.
	Similar to SQL RLIKE expression (LIKE with Regex).

over(window)	Used with window column
substr(startPos, length)	Return a Column which is a substring of the column.

when(condition, value)
otherwise(value)
	Similar to SQL CASE WHEN, Executes a list of conditions and returns one of multiple possible result expressions.

COLUMN FUNCTION	FUNCTION DESCRIPTION
dropFields(*fieldNames)	Used to drops fields in StructType by name.
withField(fieldName, col)	An expression that adds/replaces a field in StructType by name.

4. PySpark Column Functions Examples

Let’s create a simple DataFrame to work with PySpark SQL Column examples. For most of the examples below, I will be referring DataFrame object name (df.) to get the column.

```
data=[("James","Bond","100",None),
      ("Ann","Varsa","200",'F'),
      ("Tom Cruise","XXX","400",''),
      ("Tom Brand",None,"400",'M')]
columns=["fname","lname","id","gender"]
df=spark.createDataFrame(data,columns)
```

4.1 alias() – Set’s name to Column

On below example df.fname refers to Column object and alias() is a function of the Column to give alternate name. Here, fname column has been changed to first_name & lname to last_name.

On second example I have use PySpark expr() function to concatenate columns and named column as fullName.

```
#alias
from pyspark.sql.functions import expr
df.select(df.fname.alias("first_name"), \
          df.lname.alias("last_name")
   ).show()

#Another example
df.select(expr(" fname ||','|| lname").alias("fullName") \
   ).show()
```

4.2 asc() & desc() – Sort the DataFrame columns by Ascending or Descending order.

```
#asc, desc to sort ascending and descending order repsectively.
df.sort(df.fname.asc()).show()
df.sort(df.fname.desc()).show()
```

4.3 cast() & astype() – Used to convert the data Type.

```
#cast
df.select(df.fname,df.id.cast("int")).printSchema()
```

4.4 between() – Returns a Boolean expression when a column values in between lower and upper bound.

```
#between
df.filter(df.id.between(100,300)).show()
```

4.5 contains() – Checks if a DataFrame column value contains a a value specified in this function.

```
#contains
df.filter(df.fname.contains("Cruise")).show()
```

4.6 startswith() & endswith() – Checks if the value of the DataFrame Column starts and ends with a String respectively.

```
#startswith, endswith()
df.filter(df.fname.startswith("T")).show()
df.filter(df.fname.endswith("Cruise")).show()
```

4.7 eqNullSafe() –

4.8 isNull & isNotNull() – Checks if the DataFrame column has NULL or non NULL values.

```
#isNull & isNotNull
df.filter(df.lname.isNull()).show()
df.filter(df.lname.isNotNull()).show()
```

4.9 like() & rlike() – Similar to SQL LIKE expression

```
#like , rlike
df.select(df.fname,df.lname,df.id) \
  .filter(df.fname.like("%om"))
```

4.10 substr() – Returns a Column after getting sub string from the Column

```
df.select(df.fname.substr(1,2).alias("substr")).show()
```

4.11 when() & otherwise() – It is similar to SQL Case When, executes sequence of expressions until it matches the condition and returns a value when match.

```
#when & otherwise
from pyspark.sql.functions import when
df.select(df.fname,df.lname,when(df.gender=="M","Male") \
            .when(df.gender=="F","Female") \
            .when(df.gender==None ,"") \
            .otherwise(df.gender).alias("new_gender") \
    ).show()
```

4.12 isin() – Check if value presents in a List.

```
#isin
li=["100","200"]
df.select(df.fname,df.lname,df.id) \
  .filter(df.id.isin(li)) \
  .show()
```

4.13 getField() – To get the value by key from MapType column and by stuct child name from StructType column

Rest of the below functions operates on List, Map & Struct data structures hence to demonstrate these I will use another DataFrame with list, map and struct columns. For more explanation how to use Arrays refer to PySpark ArrayType Column on DataFrame Examples & for map refer to PySpark MapType Examples

```
#Create DataFrame with struct, array & map
from pyspark.sql.types import StructType,StructField,StringType,ArrayType,MapType
data=[(("James","Bond"),["Java","C#"],{'hair':'black','eye':'brown'}),
      (("Ann","Varsa"),[".NET","Python"],{'hair':'brown','eye':'black'}),
      (("Tom Cruise",""),["Python","Scala"],{'hair':'red','eye':'grey'}),
      (("Tom Brand",None),["Perl","Ruby"],{'hair':'black','eye':'blue'})]
```

```
schema = StructType([
      StructField('name', StructType([
         StructField('fname', StringType(), True),
         StructField('lname', StringType(), True)])),
      StructField('languages', ArrayType(StringType()),True),
      StructField('properties', MapType(StringType(),StringType()),True)
    ])
df=spark.createDataFrame(data,schema)
df.printSchema()

#Display's to console
root
 |-- name: struct (nullable = true)
 |    |-- fname: string (nullable = true)
 |    |-- lname: string (nullable = true)
 |-- languages: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

getField Example

```
#getField from MapType
df.select(df.properties.getField("hair")).show()

#getField from Struct
df.select(df.name.getField("fname")).show()
```

4.14 getItem() – To get the value by index from MapType or ArrayTupe & ny key for MapType column.

```
#getItem() used with ArrayType
df.select(df.languages.getItem(1)).show()

#getItem() used with MapType
df.select(df.properties.getItem("hair")).show()
```

4.15 dropFields –

```
# TO-DO, getting runtime error
```

4.16 withField() –

```
# TO-DO getting runtime error
```

4.17 over() – Used with Window Functions

```
TO-DO
```

PySpark Select Columns From DataFrame

In PySpark, select() function is used to select single, multiple, column by index, all columns from the list and the nested columns from a DataFrame, PySpark select() is a transformation function hence it returns a new DataFrame with the selected columns.

- Select a Single & Multiple Columns from PySpark
- Select All Columns From List
- Select Columns By Index
- Select a Nested Column
- Other Ways to Select Columns

First, let's create a Dataframe.

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
data = [("James","Smith","USA","CA"),
    ("Michael","Rose","USA","NY"),
    ("Robert","Williams","USA","CA"),
    ("Maria","Jones","USA","FL")
  ]
columns = ["firstname","lastname","country","state"]
df = spark.createDataFrame(data = data, schema = columns)
df.show(truncate=False)
```

1. Select Single & Multiple Columns From PySpark

You can select the single or multiple columns of the DataFrame by passing the column names you wanted to select to the select() function. Since DataFrame is immutable, this creates a new DataFrame with selected columns. show() function is used to show the Dataframe contents.

Below are ways to select single, multiple or all columns.

```
df.select("firstname","lastname").show()
df.select(df.firstname,df.lastname).show()
df.select(df["firstname"],df["lastname"]).show()

#By using col() function
from pyspark.sql.functions import col
df.select(col("firstname"),col("lastname")).show()

#Select columns by regular expression
df.select(df.colRegex("`^.*name*`")).show()
```

2. Select All Columns From List

Sometimes you may need to select all DataFrame columns from a Python list. In the below example, we have all columns in the `columns` list object.

```
# Select All columns from List
df.select(*columns).show()

# Select All columns
df.select([col for col in df.columns]).show()
df.select("*").show()
```

3. Select Columns by Index

Using a python list features, you can select the columns by index.

```
#Selects first 3 columns and top 3 rows
df.select(df.columns[:3]).show(3)

#Selects columns 2 to 4  and top 3 rows
df.select(df.columns[2:4]).show(3)
```

4. Select Nested Struct Columns from PySpark

If you have a nested struct (StructType) column on PySpark DataFrame, you need to use an explicit column qualifier in order to select. If you are new to PySpark and you have not learned StructType yet, I would recommend skipping the rest of the section or first Understand PySpark StructType before you proceed.
First, let’s create a new DataFrame with a struct type.

```
data = [
        (("James",None,"Smith"),"OH","M"),
        (("Anna","Rose",""),"NY","F"),
        (("Julia","","Williams"),"OH","F"),
        (("Maria","Anne","Jones"),"NY","M"),
        (("Jen","Mary","Brown"),"NY","M"),
        (("Mike","Mary","Williams"),"OH","M")
        ]

from pyspark.sql.types import StructType,StructField, StringType
schema = StructType([
    StructField('name', StructType([
         StructField('firstname', StringType(), True),
         StructField('middlename', StringType(), True),
         StructField('lastname', StringType(), True)
         ])),
     StructField('state', StringType(), True),
     StructField('gender', StringType(), True)
     ])
df2 = spark.createDataFrame(data = data, schema = schema)
df2.printSchema()
df2.show(truncate=False) # shows all columns
```

Yields below schema output. If you notice the column name is a struct type which consists of columns firstname, middlename, lastname.

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- state: string (nullable = true)
 |-- gender: string (nullable = true)

+----------------------+-----+------+
|name                  |state|gender|
+----------------------+-----+------+
|[James,, Smith]       |OH   |M     |
|[Anna, Rose, ]        |NY   |F     |
|[Julia, , Williams]   |OH   |F     |
|[Maria, Anne, Jones]  |NY   |M     |
|[Jen, Mary, Brown]    |NY   |M     |
|[Mike, Mary, Williams]|OH   |M     |
+----------------------+-----+------+
```

Now, let’s select struct column.

```
df2.select("name").show(truncate=False)
```

This returns struct column name as is.

```
+----------------------+
|name                  |
+----------------------+
|[James,, Smith]       |
|[Anna, Rose, ]        |
|[Julia, , Williams]   |
|[Maria, Anne, Jones]  |
|[Jen, Mary, Brown]    |
|[Mike, Mary, Williams]|
+----------------------+
```

In order to select the specific column from a nested struct, you need to explicitly qualify the nested struct column name.

```
df2.select("name.firstname","name.lastname").show(truncate=False)
```

This outputs firstname and lastname from the name struct column.

```
+---------+--------+
|firstname|lastname|
+---------+--------+
|James    |Smith   |
|Anna     |        |
|Julia    |Williams|
|Maria    |Jones   |
|Jen      |Brown   |
|Mike     |Williams|
+---------+--------+
```

In order to get all columns from struct column.

```
df2.select("name.*").show(truncate=False)
```

This yields below output.

```
+---------+----------+--------+
|firstname|middlename|lastname|
+---------+----------+--------+
|James    |null      |Smith   |
|Anna     |Rose      |        |
|Julia    |          |Williams|
|Maria    |Anne      |Jones   |
|Jen      |Mary      |Brown   |
|Mike     |Mary      |Williams|
+---------+----------+--------+
```

5. Complete Example

```
import pyspark
from pyspark.sql import SparkSession

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

data = [("James","Smith","USA","CA"),
    ("Michael","Rose","USA","NY"),
    ("Robert","Williams","USA","CA"),
    ("Maria","Jones","USA","FL")
  ]

columns = ["firstname","lastname","country","state"]
df = spark.createDataFrame(data = data, schema = columns)
df.show(truncate=False)

df.select("firstname").show()

df.select("firstname","lastname").show()

#Using Dataframe object name
df.select(df.firstname,df.lastname).show()

# Using col function
from pyspark.sql.functions import col
df.select(col("firstname"),col("lastname")).show()

data = [(("James",None,"Smith"),"OH","M"),
        (("Anna","Rose",""),"NY","F"),
        (("Julia","","Williams"),"OH","F"),
        (("Maria","Anne","Jones"),"NY","M"),
        (("Jen","Mary","Brown"),"NY","M"),
        (("Mike","Mary","Williams"),"OH","M")
        ]

from pyspark.sql.types import StructType,StructField, StringType
schema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
        ])),
    StructField('state', StringType(), True),
    StructField('gender', StringType(), True)
    ])

df2 = spark.createDataFrame(data = data, schema = schema)
df2.printSchema()
df2.show(truncate=False) # shows all columns

df2.select("name").show(truncate=False)
df2.select("name.firstname","name.lastname").show(truncate=False)
df2.select("name.*").show(truncate=False)
```

PySpark Collect() – Retrieve data from DataFrame

PySpark RDD/DataFrame collect() is an action operation that is used to retrieve all the elements of the dataset (from all nodes) to the driver node. We should use the collect() on smaller dataset usually after filter(), group() e.t.c. Retrieving larger datasets results in OutOfMemory error.
In this PySpark article, I will explain the usage of collect() with DataFrame example, when to avoid it, and the difference between collect() and select().

Related Articles:

- How to Iterate PySpark DataFrame through Loop
- How to Convert PySpark DataFrame Column to Python List

In order to explain with example, first, let's create a DataFrame.

```
import pyspark
from pyspark.sql import SparkSession

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

dept = [("Finance",10), \
   ("Marketing",20), \
   ("Sales",30), \
   ("IT",40) \
  ]
deptColumns = ["dept_name","dept_id"]
```

```
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.show(truncate=False)
```

show() function on DataFrame prints the result of DataFrame in a table format. By default, it shows only 20 rows. The above snippet returns the data in a table.

```
+---------+-------+
|dept_name|dept_id|
+---------+-------+
|Finance  |10     |
|Marketing|20     |
|Sales    |30     |
|IT       |40     |
+---------+-------+
```

Now, let’s use the collect() to retrieve the data.

```
dataCollect = deptDF.collect()
print(dataCollect)
```

deptDF.collect() retrieves all elements in a DataFrame as an Array of Row type to the driver node. printing a resultant array yields the below output.

```
[Row(dept_name='Finance', dept_id=10),
Row(dept_name='Marketing', dept_id=20),
Row(dept_name='Sales', dept_id=30),
Row(dept_name='IT', dept_id=40)]
```

Note that collect() is an action hence it does not return a DataFrame instead, it returns data in an Array to the driver. Once the data is in an array, you can use python for loop to process it further.

```
for row in dataCollect:
    print(row['dept_name'] + "," +str(row['dept_id']))
```

If you wanted to get first row and first column from a DataFrame.

```
#Returns value of First Row, First Column which is "Finance"
deptDF.collect()[0][0]
```

Let’s understand what’s happening on above statement.

- deptDF.collect() returns Array of Row type.
- deptDF.collect()[0] returns the first element in an array (1st row).
- deptDF.collect[0][0] returns the value of the first row & first column.

In case you want to just return certain elements of a DataFrame, you should call PySpark select() transformation first.

```
dataCollect = deptDF.select("dept_name").collect()
```

When to avoid Collect()

Usually, collect() is used to retrieve the action output when you have very small result set and calling collect() on an RDD/DataFrame with a bigger result set causes out of memory as it returns the entire dataset (from all workers) to the driver hence we should avoid calling collect() on a larger dataset.

collect () vs select ()

select() is a transformation that returns a new DataFrame and holds the columns that are selected whereas collect() is an action that returns the entire data set in an Array to the driver.

Complete Example of PySpark collect()

Below is complete PySpark example of using collect() on DataFrame, similarly you can also create a program using collect() with RDD.

```
import pyspark
from pyspark.sql import SparkSession

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

dept = [("Finance",10), \
   ("Marketing",20), \
   ("Sales",30), \
   ("IT",40) \
 ]
```

```
deptColumns = ["dept_name","dept_id"]
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)

dataCollect = deptDF.collect()

print(dataCollect)

dataCollect2 = deptDF.select("dept_name").collect()
print(dataCollect2)

for row in dataCollect:
    print(row['dept_name'] + "," +str(row['dept_id']))
```

PySpark withColumn() Usage with Examples

PySpark withColumn() is a transformation function of DataFrame which is used to change the value, convert the datatype of an existing column, create a new column, and many more. In this post, I will walk you through commonly used PySpark DataFrame column operations using withColumn() examples.

- PySpark withColumn – To change column DataType
- Transform/change value of an existing column
- Derive new column from an existing column
- Add a column with the literal value
- Rename column name
- Drop DataFrame column

First, let's create a DataFrame to work with.

```
data = [('James','','Smith','1991-04-01','M',3000),
  ('Michael','Rose','','2000-05-19','M',4000),
  ('Robert','','Williams','1978-09-05','M',4000),
  ('Maria','Anne','Jones','1967-12-01','F',4000),
  ('Jen','Mary','Brown','1980-02-17','F',-1)
]

columns = ["firstname","middlename","lastname","dob","gender","salary"]
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
df = spark.createDataFrame(data=data, schema = columns)
```

1. Change DataType using PySpark withColumn()

By using PySpark withColumn() on a DataFrame, we can cast or change the data type of a column. In order to change data type, you would also need to use cast() function along with withColumn(). The below statement changes the datatype from String to Integer for the salary column.

```
df.withColumn("salary",col("salary").cast("Integer")).show()
```

2. Update The Value of an Existing Column

PySpark withColumn() function of DataFrame can also be used to change the value of an existing column. In order to change the value, pass an existing column name as a first argument and a value to be assigned as a second argument to the withColumn() function. Note that the second argument should be Column type . Also, see Different Ways to Update PySpark DataFrame Column.

```
df.withColumn("salary",col("salary")*100).show()
```

This snippet multiplies the value of "salary" with 100 and updates the value back to "salary" column.

3. Create a Column from an Existing

To add/create a new column, specify the first argument with a name you want your new column to be and use the second argument to assign a value by applying an operation on an existing column. Also, see Different Ways to Add New Column to PySpark DataFrame.

```
df.withColumn("CopiedColumn",col("salary")* -1).show()
```

This snippet creates a new column "CopiedColumn" by multiplying "salary" column with value -1.

4. Add a New Column using withColumn()

In order to create a new column, pass the column name you wanted to the first argument of withColumn() transformation function. Make sure this new column not already present on DataFrame, if it presents it updates the value of that column. On below snippet, PySpark lit() function is used to add a constant value to a DataFrame column. We can also chain in order to add multiple columns.

```
df.withColumn("Country", lit("USA")).show()
df.withColumn("Country", lit("USA")) \
 .withColumn("anotherColumn",lit("anotherValue")) \
 .show()
```

5. Rename Column Name

Though you cannot rename a column using withColumn, still I wanted to cover this as renaming is one of the common operations we perform on DataFrame. To rename an existing column use withColumnRenamed() function on DataFrame.

```
df.withColumnRenamed("gender","sex") \
 .show(truncate=False)
```

6. Drop Column From PySpark DataFrame

Use “drop” function to drop a specific column from the DataFrame.

```
df.drop("salary") \
 .show()
```

Note: Note that all of these functions return the new DataFrame after applying the functions instead of updating DataFrame.

7. PySpark withColumn() Complete Example

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit
from pyspark.sql.types import StructType, StructField, StringType,IntegerType

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

data = [('James','','Smith','1991-04-01','M',3000),
 ('Michael','Rose','','2000-05-19','M',4000),
 ('Robert','','Williams','1978-09-05','M',4000),
 ('Maria','Anne','Jones','1967-12-01','F',4000),
 ('Jen','Mary','Brown','1980-02-17','F',-1)
]

columns = ["firstname","middlename","lastname","dob","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.printSchema()
df.show(truncate=False)

df2 = df.withColumn("salary",col("salary").cast("Integer"))
df2.printSchema()
df2.show(truncate=False)

df3 = df.withColumn("salary",col("salary")*100)
```

```
df3.printSchema()
df3.show(truncate=False)

df4 = df.withColumn("CopiedColumn",col("salary")* -1)
df4.printSchema()

df5 = df.withColumn("Country", lit("USA"))
df5.printSchema()

df6 = df.withColumn("Country", lit("USA")) \
   .withColumn("anotherColumn",lit("anotherValue"))
df6.printSchema()

df.withColumnRenamed("gender","sex") \
  .show(truncate=False)

df4.drop("CopiedColumn") \
.show(truncate=False)
```

PySpark withColumnRenamed to Rename Column on DataFrame

Use PySpark withColumnRenamed() to rename a DataFrame column, we often need to rename one column or multiple (or all) columns on PySpark DataFrame, you can do this in several ways. When columns are nested it becomes complicated.
Since DataFrame's are an immutable collection, you can't rename or update a column instead when using withColumnRenamed() it creates a new DataFrame with updated column names, In this PySpark article, I will cover different ways to rename columns with several use cases like rename nested column, all columns, selected multiple columns with Python/PySpark examples.

Refer to this page, If you are looking for a Spark with Scala example and rename pandas column with examples

1. PySpark withColumnRenamed – To rename DataFrame column name
2. PySpark withColumnRenamed – To rename multiple columns
3. Using StructType – To rename nested column on PySpark DataFrame
4. Using Select – To rename nested columns
5. Using withColumn – To rename nested columns
6. Using col() function – To Dynamically rename all or multiple columns
7. Using toDF() – To rename all or multiple columns

First, let's create our data set to work with.

```
dataDF = [(('James','','Smith'),'1991-04-01','M',3000),
  (('Michael','Rose',''),'2000-05-19','M',4000),
  (('Robert','','Williams'),'1978-09-05','M',4000),
  (('Maria','Anne','Jones'),'1967-12-01','F',4000),
  (('Jen','Mary','Brown'),'1980-02-17','F',-1)
]
```

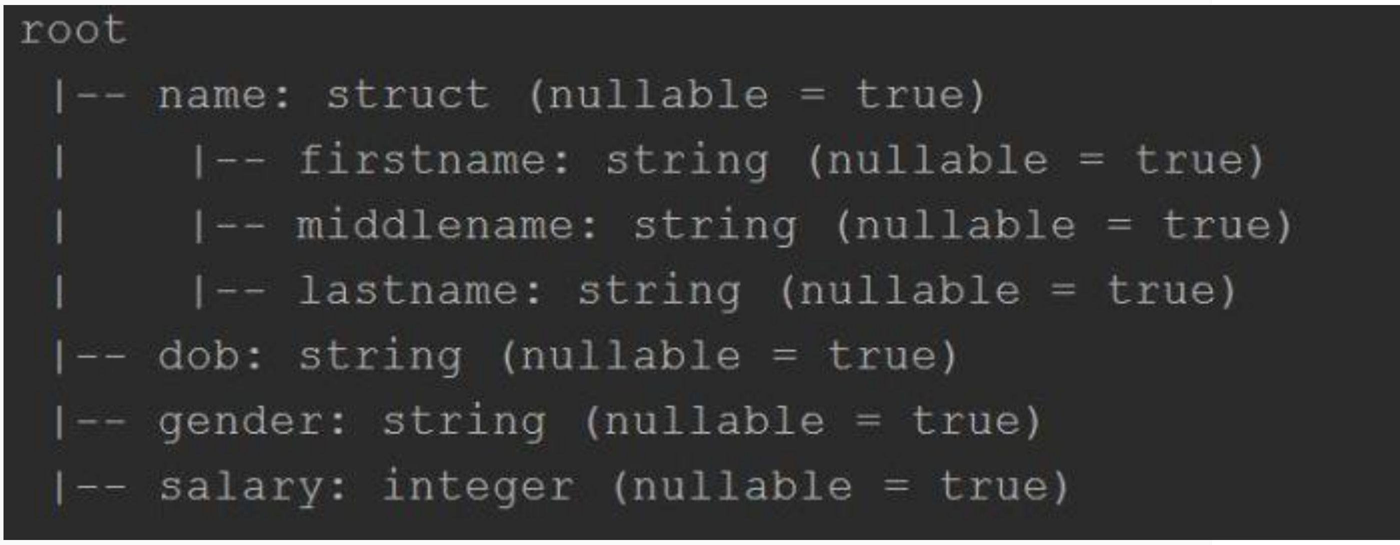
Our base schema with nested structure.

```
from pyspark.sql.types import StructType,StructField, StringType, IntegerType
schema = StructType([
        StructField('name', StructType([
             StructField('firstname', StringType(), True),
             StructField('middlename', StringType(), True),
             StructField('lastname', StringType(), True)
             ])),
         StructField('dob', StringType(), True),
         StructField('gender', StringType(), True),
         StructField('gender', IntegerType(), True)
         ])
```

Let’s create the DataFrame by using parallelize and provide the above schema.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
df = spark.createDataFrame(data = dataDF, schema = schema)
df.printSchema()
```

Below is our schema structure. I am not printing data here as it is not necessary for our examples. This schema has a nested structure.



```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- dob: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)
```

1. PySpark withColumnRenamed – To rename DataFrame column name

PySpark has a withColumnRenamed() function on DataFrame to change a column name. This is the most straight forward approach; this function takes two parameters; the first is your existing column name and the second is the new column name you wish for.

PySpark withColumnRenamed() Syntax:

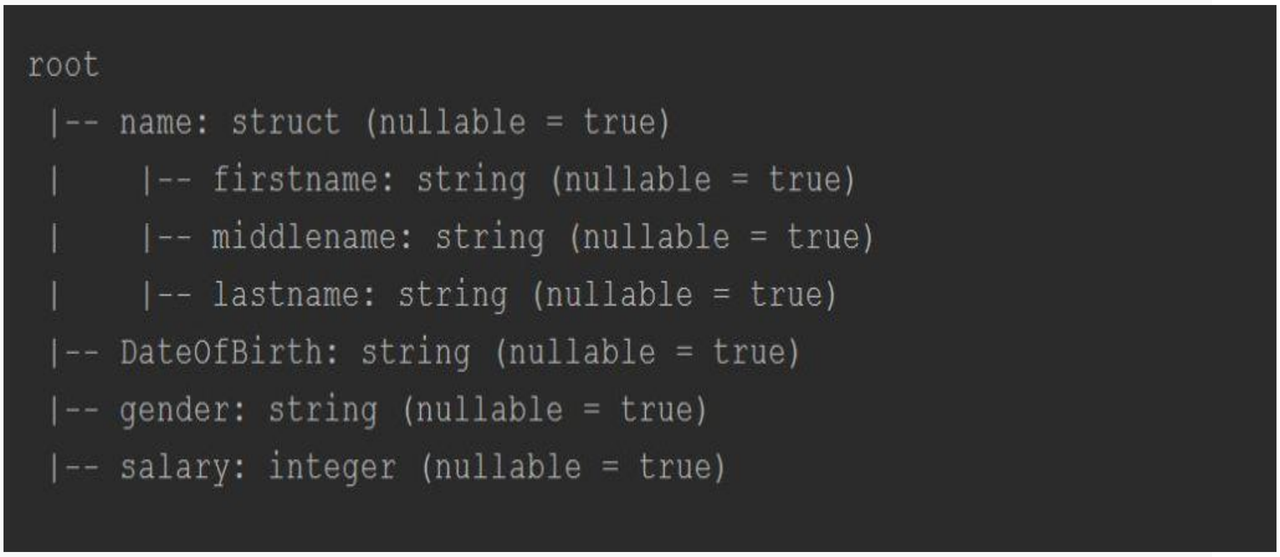
```
withColumnRenamed(existingName, newNam)
```

existingName – The existing column name you want to change
newName – New name of the column
Returns a new DataFrame with a column renamed.

Example

```
df.withColumnRenamed("dob","DateOfBirth").printSchema()
```

The above statement changes column "dob" to "DateOfBirth" on PySpark DataFrame. Note that withColumnRenamed function returns a new DataFrame and doesn't modify the current DataFrame.


```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- DateOfBirth: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)
```

2. PySpark withColumnRenamed – To rename multiple columns

To change multiple column names, we should chain withColumnRenamed functions as shown below. You can also store all columns to rename in a list and loop through to rename all columns, I will leave this to you to explore.

```
df2 = df.withColumnRenamed("dob","DateOfBirth") \
   .withColumnRenamed("salary","salary_amount")
df2.printSchema()
```

This creates a new DataFrame "df2" after renaming dob and salary columns.

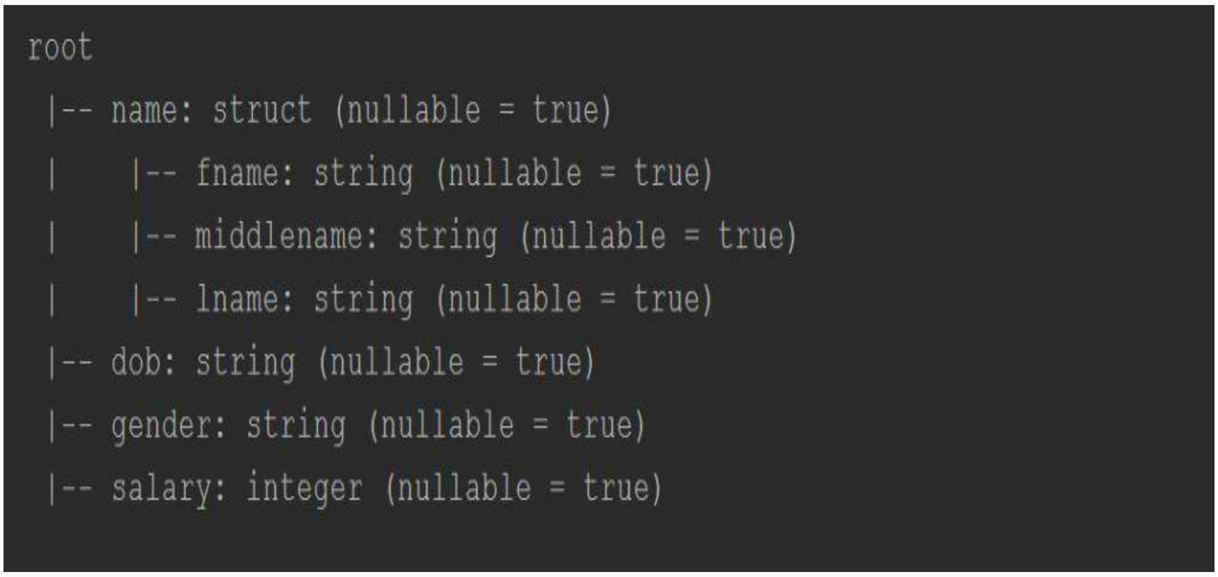
3. Using PySpark StructType – To rename a nested column in Dataframe

Changing a column name on nested data is not straight forward and we can do this by creating a new schema with new DataFrame columns using StructType and use it using cast function as shown below.

```
schema2 = StructType([
    StructField("fname",StringType()),
    StructField("middlename",StringType()),
    StructField("lname",StringType())])

df.select(col("name").cast(schema2), \
     col("dob"), col("gender"),col("salary")) \
   .printSchema()
```

This statement renames firstname to fname and lastname to lname within name structure.



```
root
 |-- name: struct (nullable = true)
 |    |-- fname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lname: string (nullable = true)
 |-- dob: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)
```

4. Using Select – To rename nested elements.

Let's see another way to change nested columns by transposing the structure to flat.

```
from pyspark.sql.functions import *
df.select(col("name.firstname").alias("fname"), \
  col("name.middlename").alias("mname"), \
  col("name.lastname").alias("lname"), \
  col("dob"),col("gender"),col("salary")) \
  .printSchema()
```

5. Using PySpark DataFrame withColumn – To rename nested columns

When you have nested columns on PySpark DatFrame and if you want to rename it, use withColumn on a data frame object to create a new column from an existing and we will need to drop the existing column. Below example creates a “fname” column from “name.firstname” and drops the “name” column

```
from pyspark.sql.functions import *
df4 = df.withColumn("fname",col("name.firstname")) \
      .withColumn("mname",col("name.middlename")) \
      .withColumn("lname",col("name.lastname")) \
      .drop("name")
df4.printSchema()
```

6. Using col() function – To Dynamically rename all or multiple columns

Another way to change all column names on Dataframe is to use col() function.

```
IN progress
```

7. Using toDF() – To change all columns in a PySpark DataFrame

When we have data in a flat structure (without nested) , use toDF() with a new schema to change all column names.

```
newColumns = ["newCol1","newCol2","newCol3","newCol4"]
df.toDF(*newColumns).printSchema()
```

Source code

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType, IntegerType
from pyspark.sql.functions import *

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

dataDF = [(('James','','Smith'),'1991-04-01','M',3000),
  (('Michael','Rose',''),'2000-05-19','M',4000),
  (('Robert','','Williams'),'1978-09-05','M',4000),
  (('Maria','Anne','Jones'),'1967-12-01','F',4000),
  (('Jen','Mary','Brown'),'1980-02-17','F',-1)
]
```

```
schema = StructType([
       StructField('name', StructType([
           StructField('firstname', StringType(), True),
           StructField('middlename', StringType(), True),
           StructField('lastname', StringType(), True)
           ])),
        StructField('dob', StringType(), True),
        StructField('gender', StringType(), True),
        StructField('salary', IntegerType(), True)
        ])

df = spark.createDataFrame(data = dataDF, schema = schema)
df.printSchema()

# Example 1
df.withColumnRenamed("dob","DateOfBirth").printSchema()
# Example 2
df2 = df.withColumnRenamed("dob","DateOfBirth") \
    .withColumnRenamed("salary","salary_amount")
df2.printSchema()

# Example 3
schema2 = StructType([
    StructField("fname",StringType()),
    StructField("middlename",StringType()),
    StructField("lname",StringType())])

df.select(col("name").cast(schema2),
  col("dob"),
  col("gender"),
  col("salary")) \
    .printSchema()

# Example 4
df.select(col("name.firstname").alias("fname"),
  col("name.middlename").alias("mname"),
  col("name.lastname").alias("lname"),
  col("dob"),col("gender"),col("salary")) \
  .printSchema()

# Example 5
df4 = df.withColumn("fname",col("name.firstname")) \
      .withColumn("mname",col("name.middlename")) \
      .withColumn("lname",col("name.lastname")) \
      .drop("name")
```

```
df4.printSchema()

#Example 7
newColumns = ["newCol1","newCol2","newCol3","newCol4"]
df.toDF(*newColumns).printSchema()

# Example 6
'''
not working
old_columns = Seq("dob","gender","salary","fname","mname","lname")
new_columns =
Seq("DateOfBirth","Sex","salary","firstName","middleName","lastName")
columnsList = old_columns.zip(new_columns).map(f=>{col(f._1).as(f._2)})
df5 = df4.select(columnsList:_*)
df5.printSchema()
'''
```

PySpark Where Filter Function | Multiple Conditions

PySpark filter() function is used to filter the rows from RDD/DataFrame based on the given condition or SQL expression, you can also use where() clause instead of the filter() if you are coming from an SQL background, both these functions operate exactly the same.
In this PySpark article, you will learn how to apply a filter on DataFrame columns of string, arrays, struct types by using single and multiple conditions and also applying filter using isin() with PySpark (Python Spark) examples.
Related Article:

- How to Filter Rows with NULL/NONE (IS NULL & IS NOT NULL) in PySpark
- Spark Filter – startsWith(), endsWith() Examples
- Spark Filter – contains(), like(), rlike() Examples

Note: PySpark Column Functions provides several options that can be used with filter().

1. PySpark DataFrame filter() Syntax

Below is syntax of the filter function. condition would be an expression you wanted to filter.

```
filter(condition)
```

Before we start with examples, first let’s create a DataFrame. Here, I am using a DataFrame with StructType and ArrayType columns as I will also be covering examples with struct and array types as-well.

```
from pyspark.sql.types import StructType,StructField
from pyspark.sql.types import StringType, IntegerType, ArrayType
data = [
    (("James","","Smith"),["Java","Scala","C++"],"OH","M"),
    (("Anna","Rose",""),["Spark","Java","C++"],"NY","F"),
    (("Julia","","Williams"),["CSharp","VB"],"OH","F"),
    (("Maria","Anne","Jones"),["CSharp","VB"],"NY","M"),
    (("Jen","Mary","Brown"),["CSharp","VB"],"NY","M"),
    (("Mike","Mary","Williams"),["Python","VB"],"OH","M")
 ]

schema = StructType([
     StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
         StructField('lastname', StringType(), True)
     ])),
     StructField('languages', ArrayType(StringType()), True),
     StructField('state', StringType(), True),
     StructField('gender', StringType(), True)
 ])

df = spark.createDataFrame(data = data, schema = schema)
df.printSchema()
df.show(truncate=False)
```

This yields below schema and DataFrame results.

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- languages: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- state: string (nullable = true)
 |-- gender: string (nullable = true)

+----------------------+------------------+-----+------+
|name                |languages        |state|gender|
+--------------------+------------------+-----+------+
|[James, , Smith]    |[Java, Scala, C++]|OH   |M     |
|[Anna, Rose, ]      |[Spark, Java, C++]|NY   |F     |
|[Julia, , Williams] |[CSharp, VB]      |OH   |F     |
|[Maria, Anne, Jones]|[CSharp, VB]      |NY   |M     |
|[Jen, Mary, Brown]  |[CSharp, VB]      |NY   |M     |
|[Mike, Mary, Williams]|[Python, VB]    |OH   |M     |
+--------------------+------------------+-----+------+
```

2. DataFrame filter() with Column Condition

Use Column with the condition to filter the rows from DataFrame, using this you can express complex condition by referring column names using `dfObject.colname`

```
# Using equals condition
df.filter(df.state == "OH").show(truncate=False)

+--------------------+------------------+-----+------+
|name                |languages        |state|gender|
+--------------------+------------------+-----+------+
|[James, , Smith]    |[Java, Scala, C++]|OH   |M     |
|[Julia, , Williams] |[CSharp, VB]      |OH   |F     |
|[Mike, Mary, Williams]|[Python, VB]    |OH   |M     |
+--------------------+------------------+-----+------+

# not equals condition
df.filter(df.state != "OH") \
   .show(truncate=False)
df.filter(~(df.state == "OH")) \
   .show(truncate=False)
```

Same example can also written as below. In order to use this first you need to import `from pyspark.sql.functions import col`

```
#Using SQL col() function
from pyspark.sql.functions import col
df.filter(col("state") == "OH") \
   .show(truncate=False)
```

3. DataFrame filter() with SQL Expression

If you are coming from SQL background, you can use that knowledge in PySpark to filter DataFrame rows with SQL expressions.

```
#Using SQL Expression
df.filter("gender == 'M'").show()
```

```
#For not equal
df.filter("gender != 'M'").show()
df.filter("gender <> 'M'").show()
```

4. PySpark Filter with Multiple Conditions

In PySpark, to filter() rows on DataFrame based on multiple conditions, you case use either Column with a condition or SQL expression. Below is just a simple example using AND (&) condition, you can extend this with OR(|), and NOT(!) conditional expressions as needed.

```
//Filter multiple condition
df.filter( (df.state  == "OH") & (df.gender  == "M") ) \
    .show(truncate=False)
```

This yields below DataFrame results.

```
+----------------------+------------------+-----+------+
|name                  |languages         |state|gender|
+----------------------+------------------+-----+------+
|[James, , Smith]      |[Java, Scala, C++]|OH   |M     |
|[Mike, Mary, Williams]|[Python, VB]      |OH   |M     |
+----------------------+------------------+-----+------+
```

5. Filter Based on List Values

If you have a list of elements and you wanted to filter that is not in the list or in the list, use isin() function of Column class and it doesn’t have isnotin() function but you do the same using not operator (~)

```
#Filter IS IN List values
li=["OH","CA","DE"]
df.filter(df.state.isin(li)).show()
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
|   [Julia, , Williams]|      [CSharp, VB]|   OH|     F|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+
# Filter NOT IS IN List values
#These show all records with NY (NY is not part of the list)
df.filter(~df.state.isin(li)).show()
df.filter(df.state.isin(li)==False).show()
```

6. Filter Based on Starts With, Ends With, Contains

You can also filter DataFrame rows by using `startswith()`, `endswith()` and `contains()` methods of Column class. For more examples on Column class, refer to PySpark Column Functions.

```
# Using startswith
df.filter(df.state.startswith("N")).show()
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
+--------------------+------------------+-----+------+

#using endswith
df.filter(df.state.endswith("H")).show()

#contains
df.filter(df.state.contains("H")).show()
```

7. PySpark Filter like and rlike

If you have SQL background you must be familiar with `like` and `rlike` (regex like), PySpark also provides similar methods in Column class to filter similar values using wildcard characters. You can use rlike() to filter by checking values case insensitive.

```
data2 = [(2,"Michael Rose"),(3,"Robert Williams"),
    (4,"Rames Rose"),(5,"Rames rose")
  ]
df2 = spark.createDataFrame(data = data2, schema = ["id","name"])

# like - SQL LIKE pattern
df2.filter(df2.name.like("%rose%")).show()
+---+----------+
| id|      name|
+---+----------+
|  5|Rames rose|
+---+----------+
# rlike - SQL RLIKE pattern (LIKE with Regex)
#This check case insensitive
df2.filter(df2.name.rlike("(?i)^*rose$")).show()
+---+------------+
| id|        name|
+---+------------+
|  2|Michael Rose|
|  4|  Rames Rose|
|  5|  Rames rose|
```

8. Filter on an Array column

When you want to filter rows from DataFrame based on value present in an array collection column, you can use the first syntax. The below example uses array_contains() from Pyspark SQL functions which checks if a value contains in an array if present it returns true otherwise false.

```

from pyspark.sql.functions import array_contains
df.filter(array_contains(df.languages,"Java")) \
    .show(truncate=False)
```

This yields below DataFrame results.

```

+----------------+------------------+-----+------+
|name            |languages         |state|gender|
+----------------+------------------+-----+------+
|[James, , Smith]|[Java, Scala, C++]|OH   |M     |
|[Anna, Rose, ]  |[Spark, Java, C++]|NY   |F     |
+----------------+------------------+-----+------+
```

9. Filtering on Nested Struct columns

If your DataFrame consists of nested struct columns, you can use any of the above syntaxes to filter the rows based on the nested column.

```

  //Struct condition
df.filter(df.name.lastname == "Williams") \
    .show(truncate=False)
```

This yields below DataFrame results

```

+----------------------+------------+-----+------+
|name                  |languages   |state|gender|
+----------------------+------------+-----+------+
|[Julia, , Williams]   |[CSharp, VB]|OH   |F     |
|[Mike, Mary, Williams]|[Python, VB]|OH   |M     |
+--------------------+-----------+-----+------+
```

10. Source code of PySpark where filter

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

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

arrayStructureData = [
        (("James","","Smith"),["Java","Scala","C++"],"OH","M"),
        (("Anna","Rose",""),["Spark","Java","C++"],"NY","F"),
        (("Julia","","Williams"),["CSharp","VB"],"OH","F"),
        (("Maria","Anne","Jones"),["CSharp","VB"],"NY","M"),
        (("Jen","Mary","Brown"),["CSharp","VB"],"NY","M"),
        (("Mike","Mary","Williams"),["Python","VB"],"OH","M")
        ]

arrayStructureSchema = StructType([
        StructField('name', StructType([
             StructField('firstname', StringType(), True),
             StructField('middlename', StringType(), True),
             StructField('lastname', StringType(), True)
             ])),
         StructField('languages', ArrayType(StringType()), True),
         StructField('state', StringType(), True),
         StructField('gender', StringType(), True)
         ])


df = spark.createDataFrame(data = arrayStructureData, schema =
arrayStructureSchema)
df.printSchema()
df.show(truncate=False)

df.filter(df.state == "OH") \
    .show(truncate=False)

df.filter(col("state") == "OH") \
    .show(truncate=False)
```

```
df.filter("gender  == 'M'") \
   .show(truncate=False)

df.filter( (df.state  == "OH") & (df.gender  == "M") ) \
   .show(truncate=False)

df.filter(array_contains(df.languages,"Java")) \
   .show(truncate=False)

df.filter(df.name.lastname == "Williams") \
   .show(truncate=False)
```

Examples explained here are also available at PySpark examples GitHub project for reference.

PySpark – Distinct to Drop Duplicate Rows

PySpark distinct() function is used to drop/remove the duplicate rows (all columns) from DataFrame and dropDuplicates() is used to drop rows based on selected (one or multiple) columns. In this article, you will learn how to use distinct() and dropDuplicates() functions with PySpark example.
Before we start, first let's create a DataFrame with some duplicate rows and values on a few columns. We use this DataFrame to demonstrate how to get distinct multiple columns.

```

import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import expr
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [("James", "Sales", 3000), \
   ("Michael", "Sales", 4600), \
   ("Robert", "Sales", 4100), \
   ("Maria", "Finance", 3000), \
   ("James", "Sales", 3000), \
   ("Scott", "Finance", 3300), \
   ("Jen", "Finance", 3900), \
   ("Jeff", "Marketing", 3000), \
   ("Kumar", "Marketing", 2000), \
   ("Saif", "Sales", 4100) \
  ]
columns= ["employee_name", "department", "salary"]
df = spark.createDataFrame(data = data, schema = columns)
df.printSchema()
df.show(truncate=False)
```

Yields below output

```
+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|James        |Sales     |3000  |
|Michael      |Sales     |4600  |
|Robert       |Sales     |4100  |
|Maria        |Finance   |3000  |
|James        |Sales     |3000  |
|Scott        |Finance   |3300  |
|Jen          |Finance   |3900  |
|Jeff         |Marketing |3000  |
|Kumar        |Marketing |2000  |
|Saif         |Sales     |4100  |
+-------------+----------+------+
```

On the above table, record with employer name Robert has duplicate rows, As you notice we have 2 rows that have duplicate values on all columns and we have 4 rows that have duplicate values on department and salary columns.

1. Get Distinct Rows (By Comparing All Columns)

On the above DataFrame, we have a total of 10 rows with 2 rows having all values duplicated, performing distinct on this DataFrame should get us 9 after removing 1 duplicate row.

```
distinctDF = df.distinct()
print("Distinct count: "+str(distinctDF.count()))
distinctDF.show(truncate=False)
```

distinct() function on DataFrame returns a new DataFrame after removing the duplicate records. This example yields the below output.

```
Distinct count: 9
+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|James        |Sales     |3000  |
|Michael      |Sales     |4600  |
|Maria        |Finance   |3000  |
|Robert       |Sales     |4100  |
|Saif         |Sales     |4100  |
|Scott        |Finance   |3300  |
|Jeff         |Marketing |3000  |
|Jen          |Finance   |3900  |
|Kumar        |Marketing |2000  |
+-------------+----------+------+
```

Alternatively, you can also run dropDuplicates() function which returns a new DataFrame after removing duplicate rows.

```
df2 = df.dropDuplicates()
print("Distinct count: "+str(df2.count()))
df2.show(truncate=False)
```

2. PySpark Distinct of Selected Multiple Columns

PySpark doesn't have a distinct method which takes columns that should run distinct on (drop duplicate rows on selected multiple columns) however, it provides another signature of dropDuplicates() function which takes multiple columns to eliminate duplicates.
Note that calling dropDuplicates() on DataFrame returns a new DataFrame with duplicate rows removed.

```
dropDisDF = df.dropDuplicates(["department","salary"])
print("Distinct count of department & salary : "+str(dropDisDF.count()))
dropDisDF.show(truncate=False)
```

Yields below output. If you notice the output, It dropped 2 records that are duplicate.

```
Distinct count of department & salary : 8
+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|Jen          |Finance   |3900  |
|Maria        |Finance   |3000  |
|Scott        |Finance   |3300  |
|Michael      |Sales     |4600  |
|Kumar        |Marketing |2000  |
|Robert       |Sales     |4100  |
|James        |Sales     |3000  |
|Jeff         |Marketing |3000  |
+-------------+----------+------+
```

3. Source Code to Get Distinct Rows

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import expr
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [("James", "Sales", 3000), \
   ("Michael", "Sales", 4600), \
   ("Robert", "Sales", 4100), \
   ("Maria", "Finance", 3000), \
   ("James", "Sales", 3000), \
   ("Scott", "Finance", 3300), \
   ("Jen", "Finance", 3900), \
   ("Jeff", "Marketing", 3000), \
   ("Kumar", "Marketing", 2000), \
   ("Saif", "Sales", 4100) \
  ]
columns= ["employee_name", "department", "salary"]
df = spark.createDataFrame(data = data, schema = columns)
df.printSchema()
df.show(truncate=False)

#Distinct
distinctDF = df.distinct()
print("Distinct count: "+str(distinctDF.count()))
distinctDF.show(truncate=False)

#Drop duplicates
df2 = df.dropDuplicates()
print("Distinct count: "+str(df2.count()))
df2.show(truncate=False)

#Drop duplicates on selected columns
dropDisDF = df.dropDuplicates(["department","salary"])
print("Distinct count of department salary : "+str(dropDisDF.count()))
dropDisDF.show(truncate=False)
}
```

PySpark orderBy() and sort() explained

You can use either sort() or orderBy() function of PySpark DataFrame to sort DataFrame by ascending or descending order based on single or multiple columns, you can also do sorting using PySpark SQL sorting functions, In this article, I will explain all these different ways using PySpark examples.

- Using sort() function
- Using orderBy() function
- Ascending order
- Descending order
- SQL Sort functions

Related: How to sort DataFrame by using Scala
Before we start, first let's create a DataFrame.

```
simpleData = [("James","Sales","NY",90000,34,10000), \
    ("Michael","Sales","NY",86000,56,20000), \
    ("Robert","Sales","CA",81000,30,23000), \
    ("Maria","Finance","CA",90000,24,23000), \
    ("Raman","Finance","CA",99000,40,24000), \
    ("Scott","Finance","NY",83000,36,19000), \
    ("Jen","Finance","NY",79000,53,15000), \
    ("Jeff","Marketing","CA",80000,25,18000), \
    ("Kumar","Marketing","NY",91000,50,21000) \
  ]
columns= ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data = simpleData, schema = columns)
df.printSchema()
df.show(truncate=False)
```

This Yields below output.

```
root
 |-- employee_name: string (nullable = true)
 |-- department: string (nullable = true)
 |-- state: string (nullable = true)
 |-- salary: integer (nullable = false)
 |-- age: integer (nullable = false)
 |-- bonus: integer (nullable = false)

+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|        James|     Sales|   NY| 90000| 34|10000|
|      Michael|     Sales|   NY| 86000| 56|20000|
|       Robert|     Sales|   CA| 81000| 30|23000|
|        Maria|   Finance|   CA| 90000| 24|23000|
|        Raman|   Finance|   CA| 99000| 40|24000|
|        Scott|   Finance|   NY| 83000| 36|19000|
|          Jen|   Finance|   NY| 79000| 53|15000|
|         Jeff| Marketing|   CA| 80000| 25|18000|
|        Kumar| Marketing|   NY| 91000| 50|21000|
+-------------+----------+-----+------+---+-----+
```

DataFrame sorting using the sort() function

PySpark DataFrame class provides sort() function to sort on one or more columns. By default, it sorts by ascending order.

Syntax

```
sort(self, *cols, **kwargs):
```

Example

```
df.sort("department","state").show(truncate=False)
df.sort(col("department"),col("state")).show(truncate=False)
```

The above two examples return the same below output, the first one takes the DataFrame column name as a string and the next takes columns in Column type. This table sorted by the first department column and then the state column.

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|Maria        |Finance   |CA   |90000 |24 |23000|
|Raman        |Finance   |CA   |99000 |40 |24000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Scott        |Finance   |NY   |83000 |36 |19000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
+-------------+----------+-----+------+---+-----+
```

DataFrame sorting using orderBy() function

PySpark DataFrame also provides `orderBy()` function to sort on one or more columns. By default, it orders by ascending.

Example

```
df.orderBy("department","state").show(truncate=False)
df.orderBy(col("department"),col("state")).show(truncate=False)
```

This returns the same output as the previous section.

Sort by Ascending (ASC)

If you wanted to specify the ascending order/sort explicitly on DataFrame, you can use the `asc` method of the `Column` function. for example

```
df.sort(df.department.asc(),df.state.asc()).show(truncate=False)
df.sort(col("department").asc(),col("state").asc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").asc()).show(truncate=False)
```

The above three examples return the same output.

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|Maria        |Finance   |CA   |90000 |24 |23000|
|Raman        |Finance   |CA   |99000 |40 |24000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Scott        |Finance   |NY   |83000 |36 |19000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
+-------------+----------+-----+------+---+-----+
```

Sort by Descending (DESC)

If you wanted to specify the sorting by descending order on DataFrame, you can use the `desc` method of the `Column` function. for example. From our example, let’s use desc on the state column.

```
df.sort(df.department.asc(),df.state.desc()).show(truncate=False)
df.sort(col("department").asc(),col("state").desc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").desc()).show(truncate=False)
```

This yields the below output for all three examples.

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|Scott        |Finance   |NY   |83000 |36 |19000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Raman        |Finance   |CA   |99000 |40 |24000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Robert       |Sales     |CA   |81000 |30 |23000|
+-------------+----------+-----+------+---+-----+
```

Besides `asc()` and `desc()` functions, PySpark also provides `asc_nulls_first()` and `asc_nulls_last()` and equivalent descending functions.

Using Raw SQL

Below is an example of how to sort DataFrame using raw SQL syntax.

```
df.createOrReplaceTempView("EMP")
spark.sql("select employee_name,department,state,salary,age,bonus from EMP ORDER BY department asc").show(truncate=False)
```

The above two examples return the same output as above.

Dataframe Sorting Complete Example

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, asc,desc

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

simpleData = [("James","Sales","NY",90000,34,10000), \
    ("Michael","Sales","NY",86000,56,20000), \
    ("Robert","Sales","CA",81000,30,23000), \
    ("Maria","Finance","CA",90000,24,23000), \
    ("Raman","Finance","CA",99000,40,24000), \
   ("Scott","Finance","NY",83000,36,19000), \
   ("Jen","Finance","NY",79000,53,15000), \
   ("Jeff","Marketing","CA",80000,25,18000), \
   ("Kumar","Marketing","NY",91000,50,21000) \
  ]
columns= ["employee_name","department","state","salary","age","bonus"]

df = spark.createDataFrame(data = simpleData, schema = columns)

df.printSchema()
df.show(truncate=False)

df.sort("department","state").show(truncate=False)
df.sort(col("department"),col("state")).show(truncate=False)

df.orderBy("department","state").show(truncate=False)
df.orderBy(col("department"),col("state")).show(truncate=False)

df.sort(df.department.asc(),df.state.asc()).show(truncate=False)
df.sort(col("department").asc(),col("state").asc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").asc()).show(truncate=False)

df.sort(df.department.asc(),df.state.desc()).show(truncate=False)
df.sort(col("department").asc(),col("state").desc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").desc()).show(truncate=False)

df.createOrReplaceTempView("EMP")
spark.sql("select employee_name,department,state,salary,age,bonus from E
```

PySpark Groupby Explained with Example

Similar to SQL GROUP BY clause, PySpark groupBy() function is used to collect the identical data into groups on DataFrame and perform aggregate functions on the grouped data. In this article, I will explain several groupBy() examples using PySpark (Spark with Python).

Related: How to group and aggregate data using Spark and Scala
Syntax:

```
groupBy(col1 : scala.Predef.String, cols : scala.Predef.String*) :
     org.apache.spark.sql.RelationalGroupedDataset
```

When we perform groupBy() on PySpark Dataframe, it returns GroupedData object which contains below aggregate functions.
count() - Returns the count of rows for each group.

mean() - Returns the mean of values for each group.
max() - Returns the maximum of values for each group.
min() - Returns the minimum of values for each group.
sum() - Returns the total for values for each group.
avg() - Returns the average for values for each group.
agg() - Using agg() function, we can calculate more than one aggregate at a time.
pivot() - This function is used to Pivot the DataFrame which I will not be covered in this article as I already have a dedicated article for Pivot & Unpivot DataFrame.

Preparing Data & creating DataFrame

Before we start, let's create the DataFrame from a sequence of the data to work with. This DataFrame contains columns "employee_name", "department", "state", "salary", "age" and "bonus" columns.
We will use this PySpark DataFrame to run groupBy() on "department" columns and calculate aggregates like minimum, maximum, average, total salary for each group using min(), max() and sum() aggregate functions respectively. and finally, we will also see how to do group and aggregate on multiple columns.

```
simpleData = [("James","Sales","NY",90000,34,10000),
    ("Michael","Sales","NY",86000,56,20000),
    ("Robert","Sales","CA",81000,30,23000),
    ("Maria","Finance","CA",90000,24,23000),
    ("Raman","Finance","CA",99000,40,24000),
    ("Scott","Finance","NY",83000,36,19000),
    ("Jen","Finance","NY",79000,53,15000),
    ("Jeff","Marketing","CA",80000,25,18000),
    ("Kumar","Marketing","NY",91000,50,21000)
  ]

schema = ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data=simpleData, schema = schema)
df.printSchema()
df.show(truncate=False)
```

Yields below output.

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|        James|     Sales|   NY| 90000| 34|10000|
|      Michael|     Sales|   NY| 86000| 56|20000|
|       Robert|     Sales|   CA| 81000| 30|23000|
|        Maria|   Finance|   CA| 90000| 24|23000|
|        Raman|   Finance|   CA| 99000| 40|24000|
|       Scott|   Finance|   NY| 83000| 36|19000|
|         Jen|   Finance|   NY| 79000| 53|15000|
|        Jeff| Marketing|   CA| 80000| 25|18000|
|       Kumar| Marketing|   NY| 91000| 50|21000|
+-------------+----------+-----+------+---+-----+
```

PySpark groupBy and aggregate on DataFrame columns

Let’s do the groupBy() on department column of DataFrame and then find the sum of salary for each department using sum() aggregate function.

```
df.groupBy("department").sum("salary").show(truncate=False)
+----------+-----------+
|department|sum(salary)|
+----------+-----------+
|Sales     |257000     |
|Finance   |351000     |
|Marketing |171000     |
+----------+-----------+
```

Similarly, we can calculate the number of employee in each department using count()

```
df.groupBy("department").count()
```

Calculate the minimum salary of each department using min()

```
df.groupBy("department").min("salary")
```

Calculate the maximin salary of each department using max()

```
```

Calculate the average salary of each department using avg()

```
df.groupBy("department").avg( "salary")
```

Calculate the mean salary of each department using mean()

```
df.groupBy("department").mean( "salary")
```

PySpark groupBy and aggregate on multiple columns

Similarly, we can also run groupBy and aggregate on two or more DataFrame columns, below example does group by on department,state and does sum()
on salary and bonus columns.

```
//GroupBy on multiple columns
df.groupBy("department","state") \
   .sum("salary","bonus") \
   .show(false)
```

This yields the below output.

```
+----------+-----+-----------+----------+
|department|state|sum(salary)|sum(bonus)|
+----------+-----+-----------+----------+
|Finance   |NY   |162000     |34000     |
|Marketing |NY   |91000      |21000     |
|Sales     |CA   |81000      |23000     |
|Marketing |CA   |80000      |18000     |
|Finance   |CA   |189000     |47000     |
|Sales     |NY   |176000     |30000     |
+----------+-----+-----------+----------+
```

similarly, we can run group by and aggregate on tow or more columns for other aggregate functions, please refer below source code for example.

Running more aggregates at a time

Using agg() aggregate function we can calculate many aggregations at a time on a single statement using PySpark SQL aggregate functions sum(), avg(), min(), max() mean() e.t.c. In order to use these, we should import "from pyspark.sql.functions import sum,avg,max,min,mean,count"

```
df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
         avg("salary").alias("avg_salary"), \
         sum("bonus").alias("sum_bonus"), \
         max("bonus").alias("max_bonus") \
     ) \
    .show(truncate=False)
```

This example does group on department column and calculates sum() and avg() of salary for each department and calculates sum() and max() of bonus for each department.

```
+----------+----------+-----------------+---------+---------+
|department|sum_salary|avg_salary       |sum_bonus|max_bonus|
+----------+----------+-----------------+---------+---------+
|Sales     |257000    |85666.66666666667|53000    |23000    |
|Finance   |351000    |87750.0          |81000    |24000    |
|Marketing |171000    |85500.0          |39000    |21000    |
+----------+----------+-----------------+---------+---------+
```

Using filter on aggregate data

Similar to SQL “HAVING” clause, On PySpark DataFrame we can use either where() or filter() function to filter the rows of aggregated data.

```
df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
      avg("salary").alias("avg_salary"), \
     sum("bonus").alias("sum_bonus"), \
     max("bonus").alias("max_bonus")) \
   .where(col("sum_bonus") >= 50000) \
   .show(truncate=False)
```

This removes the sum of a bonus that has less than 50000 and yields below output.

```
+----------+----------+-----------------+---------+---------+
|department|sum_salary|avg_salary       |sum_bonus|max_bonus|
+----------+----------+-----------------+---------+---------+
|Sales     |257000    |85666.66666666667|53000    |23000    |
|Finance   |351000    |87750.0          |81000    |24000    |
+----------+----------+-----------------+---------+---------+
```

PySpark groupBy Example Source code

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,sum,avg,max

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

simpleData = [("James","Sales","NY",90000,34,10000),
   ("Michael","Sales","NY",86000,56,20000),
   ("Robert","Sales","CA",81000,30,23000),
   ("Maria","Finance","CA",90000,24,23000),
   ("Raman","Finance","CA",99000,40,24000),
   ("Scott","Finance","NY",83000,36,19000),
   ("Jen","Finance","NY",79000,53,15000),
   ("Jeff","Marketing","CA",80000,25,18000),
   ("Kumar","Marketing","NY",91000,50,21000)
  ]

schema = ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data=simpleData, schema = schema)
df.printSchema()
df.show(truncate=False)

df.groupBy("department").sum("salary").show(truncate=False)

df.groupBy("department").count().show(truncate=False)


df.groupBy("department","state") \
   .sum("salary","bonus") \
  .show(truncate=False)

df.groupBy("department") \
   .agg(sum("salary").alias("sum_salary"), \
       avg("salary").alias("avg_salary"), \
       sum("bonus").alias("sum_bonus"), \
       max("bonus").alias("max_bonus") \
   ) \
   .show(truncate=False)

df.groupBy("department") \
   .agg(sum("salary").alias("sum_salary"), \
     avg("salary").alias("avg_salary"), \
     sum("bonus").alias("sum_bonus"), \
     max("bonus").alias("max_bonus")) \
   .where(col("sum_bonus") >= 50000) \
   .show(truncate=False)
```

PySpark Join Types | Join Two DataFrames

PySpark Join **is used to combine two DataFrames and by chaining these you can join multiple DataFrames; it supports all basic join type operations available in traditional SQL like INNER, LEFT OUTER, RIGHT OUTER, LEFT ANTI, LEFT SEMI, CROSS, SELF JOIN. PySpark Joins are wider transformations that involve data shuffling across the network.**

PySpark SQL Joins comes with more optimization by default (thanks to DataFrames) however still there would be some performance issues to consider while using.

In this **PySpark SQL Join** tutorial, you will learn different Join syntaxes and using different Join types on two or more DataFrames and Datasets using examples.

1. PySpark Join Syntax

PySpark SQL join has a below syntax and it can be accessed directly from DataFrame.

```
join(self, other, on=None, how=None)
```

join() operation takes parameters as below and returns DataFrame.

- param other: Right side of the join
- param on: a string for the join column name
- param how: default inner. Must be one of inner, cross, outer,full, full_outer, left, left_outer, right, right_outer,left_semi, and left_anti.

You can also write Join expression by adding where() and filter() methods on DataFrame and can have Join on multiple columns.

2. PySpark Join Types

Below are the different Join Types PySpark supports.

Join String	Equivalent SQL Join
inner	INNER JOIN
outer, full, fullouter, full_outer	FULL OUTER JOIN
left, leftouter, left_outer	LEFT JOIN
right, rightouter, right_outer	RIGHT JOIN
cross	
anti, leftanti, left_anti	
semi, leftsemi, left_semi	

PySpark Join Types

Before we jump into PySpark SQL Join examples, first, let’s create an "emp" and "dept" DataFrames. here, column "emp_id" is unique on emp and "dept_id" is unique on the dept dataset’s and emp_dept_id from emp has a reference to dept_id on dept dataset.

```
emp = [(1,"Smith",-1,"2018","10","M",3000), \
   (2,"Rose",1,"2010","20","M",4000), \
   (3,"Williams",1,"2010","10","M",1000), \
   (4,"Jones",2,"2005","10","F",2000), \
   (5,"Brown",2,"2010","40","",-1), \
     (6,"Brown",2,"2010","50","",-1) \
  ]
empColumns = ["emp_id","name","superior_emp_id","year_joined", \
      "emp_dept_id","gender","salary"]

empDF = spark.createDataFrame(data=emp, schema = empColumns)
empDF.printSchema()
empDF.show(truncate=False)

dept = [("Finance",10), \
   ("Marketing",20), \
   ("Sales",30), \
   ("IT",40) \
  ]
deptColumns = ["dept_name","dept_id"]
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)
```

This prints “emp” and “dept” DataFrame to the console. Refer complete example below on how to create spark object.

```

Emp Dataset
+------+--------+---------------+-----------+-----------+------+------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|salary|
+------+--------+---------------+-----------+-----------+------+------+
|1     |Smith   |-1             |2018       |10         |M     |3000  |
|2     |Rose    |1              |2010       |20         |M     |4000  |
|3     |Williams|1              |2010       |10         |M     |1000  |
|4     |Jones   |2              |2005       |10         |F     |2000  |
|5     |Brown   |2              |2010       |40         |      |-1    |
|6     |Brown   |2              |2010       |50         |      |-1    |
+------+--------+---------------+-----------+-----------+------+------+

Dept Dataset
+---------+-------+
|dept_name|dept_id|
+---------+-------+
|Finance  |10     |
|Marketing|20     |
|Sales    |30     |
|IT       |40     |
+---------+-------+
```

3. PySpark Inner Join DataFrame

Inner join is the default join in PySpark and it's mostly used. This joins two datasets on key columns, where keys don't match the rows get dropped from both datasets (emp & dept).

```
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"inner") \
     .show(truncate=False)
```

When we apply Inner join on our datasets, It drops "emp_dept_id" 50 from "emp" and "dept_id" 30 from "dept" datasets. Below is the result of the above Join expression.

```
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
|emp_id|name
|superior_emp_id|year_joined|emp_dept_id|gender|salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
|1     |Smith   |-1             |2018       |10         |M     |3000  |Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000  |Marketing|20     |
|3     |Williams|1              |2010       |10         |M     |1000  |Finance  |10     |
|4     |Jones   |2              |2005       |10         |F     |2000  |Finance  |10     |
|5     |Brown   |2              |2010       |40         |      |-1    |IT       |40     |
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
```

4. PySpark Full Outer Join

Outer a.k.a full, fullouter join returns all rows from both datasets, where join expression doesn't match it returns null on respective record columns.

```
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"outer") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"full") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"fullouter") \
    .show(truncate=False)
```

From our "emp" dataset's "emp_dept_id" with value 50 doesn't have a record on "dept" hence dept columns have null and "dept_id" 30 doesn't have a record in "emp" hence you see null's on emp columns. Below is the result of the above Join expression.

```
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
|emp_id|name
|superior_emp_id|year_joined|emp_dept_id|gender|salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
|2     |Rose    |1              |2010       |20         |M     |4000  |Marketing|20     |
|5     |Brown   |2              |2010       |40         |      |-1    |IT       |40     |
|1     |Smith   |-1             |2018       |10         |M     |3000  |Finance  |10     |
|3    |Williams|1          |2010      |10        |M     |1000  |Finance  |10     |
|4    |Jones   |2          |2005      |10        |F     |2000  |Finance  |10     |
|6    |Brown   |2          |2010      |50        |      |-1    |null     |null   |
|null |null    |null       |null      |null      |null  |null  |Sales    |30     |
+------+--------+--------------+-----------+-----------+------+------+---------+-------+
```

5. PySpark Left Outer Join

Left a.k.a Leftouter join returns all rows from the left dataset regardless of match found on the right dataset when join expression doesn't match, it assigns null for that record and drops records from right where match not found.

```
 empDF.join(deptDF,empDF("emp_dept_id") ==  deptDF("dept_id"),"left")
   .show(false)
 empDF.join(deptDF,empDF("emp_dept_id") ==  deptDF("dept_id"),"leftouter")
   .show(false)
```

From our dataset, "emp_dept_id" 5o doesn't have a record on "dept" dataset hence, this record contains null on "dept" columns (dept_name & dept_id). and "dept_id" 30 from "dept" dataset dropped from the results. Below is the result of the above Join expression.

```
+------+--------+--------------+-----------+-----------+------+------+---------+-------+
|emp_id|name
|superior_emp_id|year_joined|emp_dept_id|gender|salary|dept_name|dept_id|
+------+--------+--------------+-----------+-----------+------+------+---------+-------+
|1    |Smith   |-1         |2018      |10        |M     |3000  |Finance  |10     |
|2    |Rose    |1          |2010      |20        |M     |4000  |Marketing|20     |
|3    |Williams|1          |2010      |10        |M     |1000  |Finance  |10     |
|4    |Jones   |2          |2005      |10        |F     |2000  |Finance  |10     |
|5    |Brown   |2          |2010      |40        |      |-1    |IT       |40     |
|6    |Brown   |2          |2010      |50        |      |-1    |null     |null   |
+------+--------+--------------+-----------+-----------+------+------+---------+-------+
```

6. Right Outer Join

Right a.k.a Rightouter join is opposite of left join, here it returns all rows from the right dataset regardless of math found on the left dataset, when join expression doesn't match, it assigns null for that record and drops records from left where match not found.

```
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"right") \
   .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"rightouter") \
   .show(truncate=False)
```

From our example, the right dataset “dept_id” 30 doesn’t have it on the left dataset “emp” hence, this record contains null on “emp” columns. and “emp_dept_id” 50 dropped as a match not found on left. Below is the result of the above Join expression.

```
+------+--------+--------------+-----------+-----------+------+------+---------+-------+
|emp_id|name
|superior_emp_id|year_joined|emp_dept_id|gender|salary|dept_name|dept_id|
+------+--------+--------------+-----------+-----------+------+------+---------+-------+
|4     |Jones   |2             |2005       |10         |F     |2000  |Finance  |10     |
|3     |Williams|1             |2010       |10         |M     |1000  |Finance  |10     |
|1     |Smith   |-1            |2018       |10         |M     |3000  |Finance  |10     |
|2     |Rose    |1             |2010       |20         |M     |4000  |Marketing|20     |
|null  |null    |null          |null       |null       |null  |null  |Sales    |30     |
|5     |Brown   |2             |2010       |40         |      |-1    |IT       |40     |
+------+--------+--------------+-----------+-----------+------+------+---------+-------+
```

7. Left Semi Join

leftsemi join is similar to inner join difference being leftsemi join returns all columns from the left dataset and ignores all columns from the right dataset. In other words, this join returns columns from the only left dataset for the records match in the right dataset on join expression, records not matched on join expression are ignored from both left and right datasets.
The same result can be achieved using select on the result of the inner join however, using this join would be efficient.

```
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftsemi") \
   .show(truncate=False)
```

Below is the result of the above join expression.

```
leftsemi join
+------+--------+--------------+-----------+-----------+------+------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|salary|
+------+--------+--------------+-----------+-----------+------+------+
|1     |Smith   |-1            |2018       |10         |M     |3000  |
|2     |Rose    |1             |2010       |20         |M     |4000  |
|3     |Williams|1             |2010       |10         |M     |1000  |
|4     |Jones   |2             |2005       |10         |F     |2000  |
|5     |Brown   |2             |2010       |40         |      |-1    |
+------+--------+--------------+-----------+-----------+------+------+
```

8. Left Anti Join

leftanti join does the exact opposite of the leftsemi, leftanti join returns only columns from the left dataset for non-matched records.

```
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftanti") \
   .show(truncate=False)
```

Yields below output

```
+------+-----+---------------+-----------+-----------+------+------+
|emp_id|name |superior_emp_id|year_joined|emp_dept_id|gender|salary|
+------+-----+---------------+-----------+-----------+------+------+
|6     |Brown|2              |2010       |50         |      |-1    |
+------+-----+---------------+-----------+-----------+------+------+
```

9. PySpark Self Join

Joins are not complete without a self join, Though there is no self-join type available, we can use any of the above-explained join types to join DataFrame to itself. below example use inner self join.

```
empDF.alias("emp1").join(empDF.alias("emp2"), \
    col("emp1.superior_emp_id") == col("emp2.emp_id"),"inner") \
    .select(col("emp1.emp_id"),col("emp1.name"), \
      col("emp2.emp_id").alias("superior_emp_id"), \
      col("emp2.name").alias("superior_emp_name")) \
   .show(truncate=False)
```

Here, we are joining emp dataset with itself to find out superior emp_id and name for all employees.

```
+------+--------+---------------+-----------------+
|emp_id|name    |superior_emp_id|superior_emp_name|
+------+--------+---------------+-----------------+
|2     |Rose    |1              |Smith            |
|3     |Williams|1              |Smith            |
|4     |Jones   |2              |Rose             |
|5     |Brown   |2              |Rose             |
|6     |Brown   |2              |Rose             |
+------+--------+---------------+-----------------+
```

4. Using SQL Expression

Since PySpark SQL support native SQL syntax, we can also write join operations after creating temporary tables on DataFrames and use these tables on spark.sql().

```
empDF.createOrReplaceTempView("EMP")
```

```
deptDF.createOrReplaceTempView("DEPT")

joinDF = spark.sql("select * from EMP e, DEPT d where e.emp_dept_id == d.dept_id") \
  .show(truncate=False)

joinDF2 = spark.sql("select * from EMP e INNER JOIN DEPT d ON e.emp_dept_id ==
d.dept_id") \
  .show(truncate=False)
```

5. PySpark SQL Join on multiple DataFrames

When you need to join more than two tables, you either use SQL expression after creating a temporary view on the DataFrame or use the result of join operation to join with another DataFrame like chaining them. for example

```
df1.join(df2,df1.id1 == df2.id2,"inner") \
   .join(df3,df1.id1 == df3.id3,"inner")
```

6. PySpark SQL Join Complete Example

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

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

emp = [(1,"Smith",-1,"2018","10","M",3000), \
    (2,"Rose",1,"2010","20","M",4000), \
    (3,"Williams",1,"2010","10","M",1000), \
    (4,"Jones",2,"2005","10","F",2000), \
    (5,"Brown",2,"2010","40","",-1), \
      (6,"Brown",2,"2010","50","",-1) \
  ]
empColumns = ["emp_id","name","superior_emp_id","year_joined", \
       "emp_dept_id","gender","salary"]

empDF = spark.createDataFrame(data=emp, schema = empColumns)
empDF.printSchema()
empDF.show(truncate=False)


dept = [("Finance",10), \
    ("Marketing",20), \
    ("Sales",30), \
    ("IT",40) \
  ]
deptColumns = ["dept_name","dept_id"]
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"inner") \
     .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"outer") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"full") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"fullouter") \
    .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"left") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftouter") \
   .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"right") \
   .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"rightouter") \
   .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftsemi") \
   .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftanti") \
   .show(truncate=False)

empDF.alias("emp1").join(empDF.alias("emp2"), \
    col("emp1.superior_emp_id") == col("emp2.emp_id"),"inner") \
    .select(col("emp1.emp_id"),col("emp1.name"), \
      col("emp2.emp_id").alias("superior_emp_id"), \
      col("emp2.name").alias("superior_emp_name")) \
   .show(truncate=False)

empDF.createOrReplaceTempView("EMP")
deptDF.createOrReplaceTempView("DEPT")

joinDF = spark.sql("select * from EMP e, DEPT d where e.emp_dept_id == d.dept_id") \
  .show(truncate=False)
```

```
joinDF2 = spark.sql("select * from EMP e INNER JOIN DEPT d ON e.emp_dept_id ==
d.dept_id") \
  .show(truncate=False)
```

PySpark Union and UnionAll Explained

PySpark union() and unionAll() transformations are used to merge two or more DataFrame's of the same schema or structure. In this PySpark article, I will explain both union transformations with PySpark examples.

Dataframe union() – union() method of the DataFrame is used to merge two DataFrame's of the same structure/schema. If schemas are not the same it returns an error.
DataFrame unionAll() – unionAll() is deprecated since Spark "2.0.0" version and replaced with union().
Note: In other SQL languages, Union eliminates the duplicates but UnionAll merges two datasets including duplicate records. But, in PySpark both behave the same and recommend using DataFrame duplicate() function to remove duplicate rows.
First, let's create two DataFrame with the same schema.
First DataFrame

```

import pyspark
from pyspark.sql import SparkSession

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

simpleData = [("James","Sales","NY",90000,34,10000), \
    ("Michael","Sales","NY",86000,56,20000), \
    ("Robert","Sales","CA",81000,30,23000), \
    ("Maria","Finance","CA",90000,24,23000) \
  ]

columns= ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data = simpleData, schema = columns)
df.printSchema()
df.show(truncate=False)
```

This yields the below schema and DataFrame output.

```

root
 |-- employee_name: string (nullable = true)
 |-- department: string (nullable = true)
 |-- state: string (nullable = true)
 |-- salary: long (nullable = true)
 |-- age: long (nullable = true)
 |-- bonus: long (nullable = true)

+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|Maria        |Finance   |CA   |90000 |24 |23000|
+-------------+----------+-----+------+---+-----+
```

Second DataFrame

Now, let's create a second Dataframe with the new records and some records from the above Dataframe but with the same schema.

```
simpleData2 = [("James","Sales","NY",90000,34,10000), \
    ("Maria","Finance","CA",90000,24,23000), \
    ("Jen","Finance","NY",79000,53,15000), \
    ("Jeff","Marketing","CA",80000,25,18000), \
    ("Kumar","Marketing","NY",91000,50,21000) \
  ]
columns2= ["employee_name","department","state","salary","age","bonus"]

df2 = spark.createDataFrame(data = simpleData2, schema = columns2)

df2.printSchema()
df2.show(truncate=False)
```

This yields below output

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
+-------------+----------+-----+------+---+-----+
```

Merge two or more DataFrames using union

DataFrame `union()` method merges two DataFrames and returns the new DataFrame with all rows from two Dataframes regardless of duplicate data.

```
unionDF = df.union(df2)
unionDF.show(truncate=False)
```

As you see below it returns all records.

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|James        |Sales     |NY   |90000 |34 |10000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
+-------------+----------+-----+------+---+-----+
```

Merge DataFrames using unionAll

DataFrame unionAll() method is deprecated since PySpark "2.0.0" version and recommends using the union() method.

```
unionAllDF = df.unionAll(df2)
unionAllDF.show(truncate=False)
```

Returns the same output as above.

Merge without Duplicates

Since the union() method returns all rows without distinct records, we will use the distinct() function to return just one record when duplicate exists.

```
disDF = df.union(df2).distinct()
disDF.show(truncate=False)
```

Yields below output. As you see, this returns only distinct rows.

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Robert       |Sales     |CA   |81000 |30 |23000|
+-------------+----------+-----+------+---+-----+
```

Python

Complete Example of DataFrame Union

```python
import pyspark
from pyspark.sql import SparkSession

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

simpleData = [("James","Sales","NY",90000,34,10000), \
    ("Michael","Sales","NY",86000,56,20000), \
    ("Robert","Sales","CA",81000,30,23000), \
    ("Maria","Finance","CA",90000,24,23000) \
  ]

columns= ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data = simpleData, schema = columns)
df.printSchema()
df.show(truncate=False)

simpleData2 = [("James","Sales","NY",90000,34,10000), \
    ("Maria","Finance","CA",90000,24,23000), \
    ("Jen","Finance","NY",79000,53,15000), \
    ("Jeff","Marketing","CA",80000,25,18000), \
    ("Kumar","Marketing","NY",91000,50,21000) \
  ]
columns2= ["employee_name","department","state","salary","age","bonus"]

df2 = spark.createDataFrame(data = simpleData2, schema = columns2)

df2.printSchema()
df2.show(truncate=False)

unionDF = df.union(df2)
unionDF.show(truncate=False)
disDF = df.union(df2).distinct()
disDF.show(truncate=False)

unionAllDF = df.unionAll(df2)
unionAllDF.show(truncate=False)
```

Spark Merge Two DataFrames with Different Columns or Schema

In Spark or PySpark let's see how to merge/union two DataFrames with a different number of columns (different schema). In Spark 3.1, you can easily achieve this using unionByName() transformation by passing allowMissingColumns with the value true. In order version, this property is not available

```
//Scala
merged_df = df1.unionByName(df2, true)

#PySpark
merged_df = df1.unionByName(df2, allowMissingColumns=True)
```

The difference between unionByName() function and union() is that this function resolves columns by name (not by position). In other words, unionByName() is used to merge two DataFrame's by column names instead of by position.
In case if you are using older than Spark 3.1 version, use below approach to merge DataFrame's with different column names.

- Spark Merge DataFrames with Different Columns (Scala Example)
- PySpark Merge DataFrames with Different Columns (Python Example)

Spark Merge Two DataFrames with Different Columns

In this section I will cover Spark with Scala example of how to merge two different DataFrames, first let's create DataFrames with different number of columns. DataFrame df1 missing column state and salary and df2 missing column age.

```
//Create DataFrame df1 with columns name,dept & age
val data = Seq(("James","Sales",34), ("Michael","Sales",56),
          ("Robert","Sales",30), ("Maria","Finance",24) )
import spark.implicits._
val df1 = data.toDF("name","dept","age")
df1.printSchema()

//root
// |-- name: string (nullable = true)
// |-- dept: string (nullable = true)
// |-- age: long (nullable = true)
```

Second DataFrame

```
//Create DataFrame df1 with columns name,dep,state & salary
val data2=Seq(("James","Sales","NY",9000),("Maria","Finance","CA",9000),
         ("Jen","Finance","NY",7900),("Jeff","Marketing","CA",8000))
```

```
val df2 = data2.toDF("name","dept","state","salary")
df2.printSchema()

//root
// |-- name: string (nullable = true)
// |-- dept: string (nullable = true)
// |-- state: string (nullable = true)
// |-- salary: long (nullable = true)
```

Now create a new DataFrames from existing after adding missing columns. newly added columns contains null values and we add constant column using lit() function.

```

val merged_cols = df1.columns.toSet ++ df2.columns.toSet
import org.apache.spark.sql.functions.{col,lit}
def getNewColumns(column: Set[String], merged_cols: Set[String]) = {
   merged_cols.toList.map(x => x match {
     case x if column.contains(x) => col(x)
     case _ => lit(null).as(x)
   })
}
val new_df1=df1.select(getNewColumns(df1.columns.toSet, merged_cols):_*)
val new_df2=df2.select(getNewColumns(df2.columns.toSet, merged_cols):_*)
```

Finally merge two DataFrame's by using column names

```

//Finally join two dataframe's df1 & df2 by name
val merged_df=new_df1.unionByName(new_df2)
merged_df.show()

//+-------+---------+----+-----+------+
//|   name|     dept| age|state|salary|
//+-------+---------+----+-----+------+
//|  James|    Sales|  34| null|  null|
//|Michael|    Sales|  56| null|  null|
//| Robert|    Sales|  30| null|  null|
//|  Maria|  Finance|  24| null|  null|
//|  James|    Sales|null|   NY|  9000|
//|  Maria|  Finance|null|   CA|  9000|
//|    Jen|  Finance|null|   NY|  7900|
//|   Jeff|Marketing|null|   CA|  8000|
//+-------+---------+----+-----+------+
```

PySpark Merge Two DataFrames with Different Columns

In PySpark to merge two DataFrames with different columns, will use the similar approach explain above and uses unionByName() transformation. First let’s create DataFrame’s with different number of columns.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

#Create DataFrame df1 with columns name,dept & age
data = [("James","Sales",34), ("Michael","Sales",56), \
    ("Robert","Sales",30), ("Maria","Finance",24) ]
columns= ["name","dept","age"]
df1 = spark.createDataFrame(data = data, schema = columns)
df1.printSchema()

#Create DataFrame df1 with columns name,dep,state & salary
data2=[("James","Sales","NY",9000),("Maria","Finance","CA",9000), \
    ("Jen","Finance","NY",7900),("Jeff","Marketing","CA",8000)]
columns2= ["name","dept","state","salary"]
df2 = spark.createDataFrame(data = data2, schema = columns2)
df2.printSchema()
```

Now add missing columns ‘state‘ and ‘salary‘ to df1 and ‘age‘ to df2 with null values.

```
#Add missing columns 'state' & 'salary' to df1
from pyspark.sql.functions import lit
for column in [column for column in df2.columns if column not in df1.columns]:
    df1 = df1.withColumn(column, lit(None))

#Add missing column 'age' to df2
for column in [column for column in df1.columns if column not in df2.columns]:
    df2 = df2.withColumn(column, lit(None))
```

Now merge/union the DataFrames using unionByName(). The difference between unionByName() function and union() is that this function resolves columns by name (not by position). In other words, unionByName() is used to merge two DataFrame’s by column names instead of by position.

```
#Finally join two dataframe's df1 & df2 by name
merged_df=df1.unionByName(df2)
merged_df.show()
```

PySpark UDF Example

PySpark UDF (a.k.a User Defined Function) is the most useful feature of Spark SQL & DataFrame that is used to extend the PySpark build in capabilities. In this article, I will explain what is UDF? why do we need it and how to create and use it on DataFrame select(), withColumn() and SQL using PySpark (Spark with Python) examples.

Note: UDF's are the most expensive operations hence use them only you have no choice and when essential. In the later section of the article, I will explain why using UDF's is an expensive operation in detail.

Table of contents

1. PySpark UDF Introduction

1.1 What is UDF?

UDF's a.k.a User Defined Functions, If you are coming from SQL background, UDF's are nothing new to you as most of the traditional RDBMS databases support User Defined Functions, these functions need to register in the database library and use them on SQL as regular functions.

PySpark UDF's are similar to UDF on traditional databases. In PySpark, you create a function in a Python syntax and wrap it with PySpark SQL udf() or register it as udf and use it on DataFrame and SQL respectively.

1.2 Why do we need a UDF?

UDF's are used to extend the functions of the framework and re-use these functions on multiple DataFrame's. For example, you wanted to convert every first letter of a word in a name string to a capital case; PySpark build-in features don't have this function hence you can create it a UDF and reuse this as needed on many Data

Frames. UDF’s are once created they can be re-used on several DataFrame’s and SQL expressions.

Before you create any UDF, do your research to check if the similar function you wanted is already available in Spark SQL Functions. PySpark SQL provides several predefined common functions and many more new functions are added with every release. hence, It is best to check before you reinventing the wheel.
When you creating UDF’s you need to design them very carefully otherwise you will come across optimization & performance issues.

2. Create PySpark UDF

2.1 Create a DataFrame

Before we jump in creating a UDF, first let’s create a PySpark DataFrame.

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

columns = ["Seqno","Name"]
data = [("1", "john jones"),
    ("2", "tracey smith"),
    ("3", "amy sanders")]

df = spark.createDataFrame(data=data,schema=columns)

df.show(truncate=False)
```

Yields below output.

```
+-----+------------+
|Seqno|Names       |
+-----+------------+
|1    |john jones  |
|2    |tracey smith|
|3    |amy sanders |
+-----+------------+
```

2.2 Create a Python Function

The first step in creating a UDF is creating a Python function. Below snippet creates a function convertCase() which takes a string parameter and converts the first letter of every word to capital letter. UDF’s take parameters of your choice and returns a value.

```
def convertCase(str):
    resStr=""
    arr = str.split(" ")
    for x in arr:
       resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "
    return resStr
```

2.3 Convert a Python function to PySpark UDF

Now convert this function `convertCase()` to UDF by passing the function to PySpark SQL `udf()`, this function is available at `org.apache.spark.sql.functions.udf` package. Make sure you import this package before using it.
PySpark SQL `udf()` function returns `org.apache.spark.sql.expressions.UserDefinedFunction` class object.

```
""" Converting function to UDF """
convertUDF = udf(lambda z: convertCase(z),StringType())
```

Note: The default type of the udf() is StringType hence, you can also write the above statement without return type.

```
""" Converting function to UDF
StringType() is by default hence not required """
convertUDF = udf(lambda z: convertCase(z))
```

3. Using UDF with DataFrame

3.1 Using UDF with PySpark DataFrame select()

Now you can use `convertUDF()` on a DataFrame column as a regular build-in function.

```
df.select(col("Seqno"), \
    convertUDF(col("Name")).alias("Name") ) \
   .show(truncate=False)
```

This results below output.

```
+-----+-------------+
|Seqno|Name         |
+-----+-------------+
|1    |John Jones   |
|2    |Tracey Smith |
|3    |Amy Sanders  |
+-----+-------------+
```

3.2 Using UDF with PySpark DataFrame withColumn()

You could also use udf on DataFrame `withColumn()` function, to explain this I will create another `upperCase()` function which converts the input string to upper case.

```
def upperCase(str):
    return str.upper()
```

Let’s convert `upperCase()` python function to UDF and then use it with DataFrame `withColumn()`. Below example converts the values of “Name” column to upper case and creates a new column “Curated Name”

```
upperCaseUDF = udf(lambda z:upperCase(z),StringType())
```

```
df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \
  .show(truncate=False)
```

This yields below output.

```
+-----+------------+-------------+
|Seqno|Name        |Cureated Name|
+-----+------------+-------------+
|1    |john jones  |JOHN JONES   |
|2    |tracey smith|TRACEY SMITH |
|3    |amy sanders |AMY SANDERS  |
+-----+------------+-------------+
```

3.3 Registering PySpark UDF & use it on SQL

In order to use convertCase() function on PySpark SQL, you need to register the function with PySpark by using spark.udf.register().

```
""" Using UDF on SQL """
spark.udf.register("convertUDF", convertCase,StringType())
df.createOrReplaceTempView("NAME_TABLE")
spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE") \
     .show(truncate=False)
```

This yields the same output as 3.1 example.

4. Creating UDF using annotation

In the previous sections, you have learned creating a UDF is a 2 step process, first, you need to create a Python function, second convert function to UDF using SQL udf() function, however, you can avoid these two steps and create it with just a single step by using annotations.

```
@udf(returnType=StringType())
def upperCase(str):
    return str.upper()

df.withColumn("Cureated Name", upperCase(col("Name"))) \
.show(truncate=False)
```

This results same output as section 3.2

5. Special Handling

5.1 Execution order

One thing to aware is in PySpark/Spark does not guarantee the order of evaluation of subexpressions meaning expressions are not guarantee to evaluated left-to-right or in any other fixed order. PySpark reorders the execution for query optimization and planning hence, AND, OR, WHERE and HAVING expression will have side effects.

So when you are designing and using UDF, you have to be very careful especially with null handling as these results runtime exceptions.

```
"""
No guarantee Name is not null will execute first
If convertUDF(Name) like '%John%' execute first then
you will get runtime error
"""
spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE " + \
         "where Name is not null and convertUDF(Name) like '%John%'") \
     .show(truncate=False)
```

5.2 Handling null check

UDF's are error-prone when not designed carefully. for example, when you have a column that contains the value null on some records

```
""" null check """

columns = ["Seqno","Name"]
data = [("1", "john jones"),
    ("2", "tracey smith"),
    ("3", "amy sanders"),
    ('4',None)]

df2 = spark.createDataFrame(data=data,schema=columns)
df2.show(truncate=False)
df2.createOrReplaceTempView("NAME_TABLE2")

spark.sql("select convertUDF(Name) from NAME_TABLE2") \
     .show(truncate=False)
```

Note that from the above snippet, record with "Seqno 4" has value "None" for "name" column. Since we are not handling null with UDF function, using this on DataFrame returns below error. Note that in Python None is considered null.

```
AttributeError: 'NoneType' object has no attribute 'split'

            at
org.apache.spark.api.python.BasePythonRunner$ReaderIterator.handlePythonExceptio
n(PythonRunner.scala:456)
            at
org.apache.spark.sql.execution.python.PythonUDFRunner$$anon$1.read(PythonUDFR
unner.scala:81)
```

```
        at
org.apache.spark.sql.execution.python.PythonUDFRunner$$anon$1.read(PythonUDFRunner.scala:64)
        at
org.apache.spark.api.python.BasePythonRunner$ReaderIterator.hasNext(PythonRunner.scala:410)
        at
org.apache.spark.InterruptibleIterator.hasNext(InterruptibleIterator.scala:37)
        at scala.collection.Iterator$$anon$12.hasNext(Iterator.scala:440)
```

Below points to remember

- Its always best practice to check for null inside a UDF function rather than checking for null outside.
- In any case, if you can't do a null check in UDF at lease use IF or CASE WHEN to check for null and call UDF conditionally.

```
spark.udf.register("_nullsafeUDF", lambda str: convertCase(str) if not str is None else "" , StringType())

spark.sql("select _nullsafeUDF(Name) from NAME_TABLE2") \
   .show(truncate=False)

spark.sql("select Seqno, _nullsafeUDF(Name) as Name from NAME_TABLE2 " + \
       " where Name is not null and _nullsafeUDF(Name) like '%John%'") \
   .show(truncate=False)
```

This executes successfully without errors as we are checking for null/none while registering UDF.

5.3 Performance concern using UDF

UDF's are a black box to PySpark hence it can't apply optimization and you will lose all the optimization PySpark does on Dataframe/Dataset. When possible you should use Spark SQL built-in functions as these functions provide optimization. Consider creating UDF only when existing built-in SQL function doesn't have it.

6. Complete PySpark UDF Example

Below is complete UDF function example in Scala

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, udf
from pyspark.sql.types import StringType

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

```
columns = ["Seqno","Name"]
data = [("1", "john jones"),
   ("2", "tracey smith"),
   ("3", "amy sanders")]

df = spark.createDataFrame(data=data,schema=columns)

df.show(truncate=False)

def convertCase(str):
   resStr=""
   arr = str.split(" ")
   for x in arr:
      resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "
   return resStr

""" Converting function to UDF """
convertUDF = udf(lambda z: convertCase(z))

df.select(col("Seqno"), \
   convertUDF(col("Name")).alias("Name") ) \
.show(truncate=False)

def upperCase(str):
   return str.upper()

upperCaseUDF = udf(lambda z:upperCase(z),StringType())

df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \
.show(truncate=False)

""" Using UDF on SQL """
spark.udf.register("convertUDF", convertCase,StringType())
df.createOrReplaceTempView("NAME_TABLE")
spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE") \
    .show(truncate=False)

spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE " + \
        "where Name is not null and convertUDF(Name) like '%John%'") \
    .show(truncate=False)

""" null check """

columns = ["Seqno","Name"]
data = [("1", "john jones"),
   ("2", "tracey smith"),
   ("3", "amy sanders"),
   ('4',None)]

df2 = spark.createDataFrame(data=data,schema=columns)
df2.show(truncate=False)
df2.createOrReplaceTempView("NAME_TABLE2")

spark.udf.register("_nullsafeUDF", lambda str: convertCase(str) if not str is None else
"" , StringType())

spark.sql("select _nullsafeUDF(Name) from NAME_TABLE2") \
    .show(truncate=False)

spark.sql("select Seqno, _nullsafeUDF(Name) as Name from NAME_TABLE2 " + \
        " where Name is not null and _nullsafeUDF(Name) like '%John%'") \
    .show(truncate=False)
```

PySpark map() Transformation

PySpark map (map()) is an RDD transformation that is used to apply the transformation function (lambda) on every element of RDD/DataFrame and returns a new RDD. In this article, you will learn the syntax and usage of the RDD map() transformation with an example and how to use it with DataFrame.
RDD map() transformation is used to apply any complex operations like adding a column, updating a column, transforming the data e.t.c, the output of map transformations would always have the same number of records as input.

- **Note1:** DataFrame doesn’t have map() transformation to use with DataFrame hence you need to DataFrame to RDD first.
- **Note2:** If you have a heavy initialization use PySpark mapPartitions() transformation instead of map(), as with mapPartitions() heavy initialization executes only once for each partition instead of every record.

Related: Spark map() vs mapPartitions() Explained with Examples
First, let’s create an RDD from the list.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[1]") \
   .appName("SparkByExamples.com").getOrCreate()

data = ["Project","Gutenberg’s","Alice’s","Adventures",
"in","Wonderland","Project","Gutenberg’s","Adventures",
"in","Wonderland","Project","Gutenberg’s"]
```

```
rdd=spark.sparkContext.parallelize(data)
```

map() Syntax

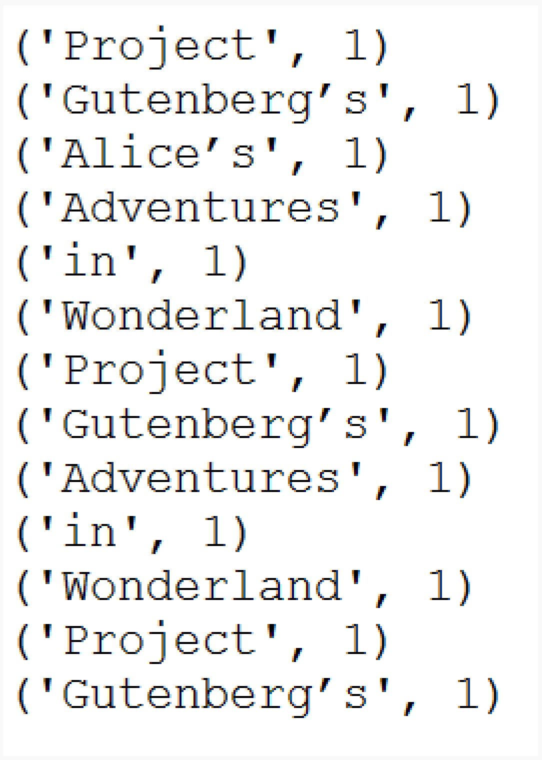
```
map(f, preservesPartitioning=False)
```

PySpark map() Example with RDD

In this PySpark map() example, we are adding a new element with value 1 for each element, the result of the RDD is PairRDDFunctions which contains key-value pairs, word of type String as Key and 1 of type Int as value.

```
rdd2=rdd.map(lambda x: (x,1))
for element in rdd2.collect():
   print(element)
```

This yields below output.



```
('Project', 1)
('Gutenberg's', 1)
('Alice's', 1)
('Adventures', 1)
('in', 1)
('Wonderland', 1)
('Project', 1)
('Gutenberg's', 1)
('Adventures', 1)
('in', 1)
('Wonderland', 1)
('Project', 1)
('Gutenberg's', 1)
```

PySpark map() Example with DataFrame

PySpark DataFrame doesn't have map() transformation to apply the lambda function, when you wanted to apply the custom transformation, you need to convert the DataFrame to RDD and apply the map() transformation. Let's use another dataset to explain this.

```
data = [('James','Smith','M',30),
  ('Anna','Rose','F',41),
  ('Robert','Williams','M',62),
]

columns = ["firstname","lastname","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.show()
+---------+--------+------+------+
|firstname|lastname|gender|salary|
+---------+--------+------+------+
|    James|   Smith|     M|    30|
|     Anna|    Rose|     F|    41|
|   Robert|Williams|     M|    62|
+---------+--------+------+------+

# Refering columns by index.
rdd2=df.rdd.map(lambda x:
    (x[0]+","+x[1],x[2],x[3]*2)
    )
df2=rdd2.toDF(["name","gender","new_salary"]   )
df2.show()
+---------------+------+----------+
|           name|gender|new_salary|
+---------------+------+----------+
|    James,Smith|     M|        60|
|      Anna,Rose|     F|        82|
|Robert,Williams|     M|       124|
+---------------+------+----------+
```

Note that aboveI have used index to get the column values, alternatively, you can also refer to the DataFrame column names while iterating.

```
# Referring Column Names
rdd2=df.rdd.map(lambda x:
    (x["firstname"]+","+x["lastname"],x["gender"],x["salary"]*2)
    )
```

Another alternative

```
# Referring Column Names
rdd2=df.rdd.map(lambda x:
    (x.firstname+","+x.lastname,x.gender,x.salary*2)
    )
```

You can also create a custom function to perform an operation.
Below func1() function executes for every DataFrame row from the lambda function.

```
# By Calling function
```

```
def func1(x):
   firstName=x.firstname
   lastName=x.lastname
   name=firstName+","+lastName
   gender=x.gender.lower()
   salary=x.salary*2
   return (name,gender,salary)
```

Complete PySpark map() example

Below is complete example of PySpark map() transformation.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = ["Project",
"Gutenberg’s",
"Alice’s",
"Adventures",
"in",
"Wonderland",
"Project",
"Gutenberg’s",
"Adventures",
"in",
"Wonderland",
"Project",
"Gutenberg’s"]

rdd=spark.sparkContext.parallelize(data)

rdd2=rdd.map(lambda x: (x,1))
for element in rdd2.collect():
   print(element)

data = [('James','Smith','M',30),
  ('Anna','Rose','F',41),
  ('Robert','Williams','M',62),
]

columns = ["firstname","lastname","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.show()
```

```
rdd2=df.rdd.map(lambda x:
   (x[0]+","+x[1],x[2],x[3]*2)
   )
df2=rdd2.toDF(["name","gender","new_salary"]   )
df2.show()

#Referring Column Names
rdd2=df.rdd.map(lambda x:
   (x["firstname"]+","+x["lastname"],x["gender"],x["salary"]*2)
   )

#Referring Column Names
rdd2=df.rdd.map(lambda x:
   (x.firstname+","+x.lastname,x.gender,x.salary*2)
   )

def func1(x):
   firstName=x.firstname
   lastName=x.lastname
   name=firstName+","+lastName
   gender=x.gender.lower()
   salary=x.salary*2
   return (name,gender,salary)

rdd2=df.rdd.map(lambda x: func1(x))
```

PySpark flatMap() Transformation

PySpark flatMap() is a transformation operation that flattens the RDD/DataFrame (array/map DataFrame columns) after applying the function on every element and returns a new PySpark RDD/DataFrame. In this article, you will learn the syntax and usage of the PySpark flatMap() with an example.
First, let's create an RDD from the list.

```
data = ["Project Gutenberg's",
      "Alice's Adventures in Wonderland",
      "Project Gutenberg's",
      "Adventures in Wonderland",
      "Project Gutenberg's"]
rdd=spark.sparkContext.parallelize(data)
for element in rdd.collect():
   print(element)
```

This yields the below output

```
Project Gutenberg's
Alice's Adventures in Wonderland
Project Gutenberg's
Adventures in Wonderland
Project Gutenberg's
```

flatMap() Syntax

```
flatMap(f, preservesPartitioning=False)
```

flatMap() Example

Now, let's see with an example of how to apply a flatMap() transformation on RDD. In the below example, first, it splits each record by space in an RDD and finally flattens it. Resulting RDD consists of a single word on each record.

```
rdd2=rdd.flatMap(lambda x: x.split(" "))
for element in rdd2.collect():
   print(element)
```

This yields below output.

```
Project
Gutenberg's
Alice's
Adventures
in
Wonderland
Project
Gutenberg's
Adventures
in
Wonderland
Project
Gutenberg's
```

Complete PySpark flatMap() example

Below is the complete example of flatMap() function that works with RDD.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = ["Project Gutenberg’s",
        "Alice’s Adventures in Wonderland",
        "Project Gutenberg’s",
        "Adventures in Wonderland",
        "Project Gutenberg’s"]
rdd=spark.sparkContext.parallelize(data)
for element in rdd.collect():
    print(element)

#Flatmap
rdd2=rdd.flatMap(lambda x: x.split(" "))
for element in rdd2.collect():
    print(element)
```

Using flatMap() transformation on DataFrame

Unfortunately, PySpark DataFame doesn’t have flatMap() transformation however, DataFrame has explode() SQL function that is used to flatten the column. Below is a complete example.

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('pyspark-by-examples').getOrCreate()

arrayData = [
        ('James',['Java','Scala'],{'hair':'black','eye':'brown'}),
        ('Michael',['Spark','Java',None],{'hair':'brown','eye':None}),
        ('Robert',['CSharp',''],{'hair':'red','eye':''}),
        ('Washington',None,None),
        ('Jefferson',['1','2'],{})]
df = spark.createDataFrame(data=arrayData, schema =
['name','knownLanguages','properties'])

from pyspark.sql.functions import explode
df2 = df.select(df.name,explode(df.knownLanguages))
df2.printSchema()
df2.show()
```

This example flattens the array column “knownLanguages” and yields below output

```
root
 |-- name: string (nullable = true)
 |-- col: string (nullable = true)
```

```
+---------+------+
|     name|   col|
+---------+------+
|    James|  Java|
|    James| Scala|
|  Michael| Spark|
|  Michael|  Java|
|  Michael|  null|
|   Robert|CSharp|
|   Robert|      |
|Jefferson|     1|
|Jefferson|     2|
+---------+------+
```

PySpark – Loop/Iterate Through Rows in DataFrame

PySpark provides map(), mapPartitions() to loop/iterate through rows in RDD/DataFrame to perform the complex transformations, and these two returns the same number of records as in the original DataFrame but the number of columns could be different (after add/update).

PySpark also provides foreach() & foreachPartitions() actions to loop/iterate through each Row in a DataFrame but these two returns nothing, In this article, I will explain how to use these methods to get DataFrame column values and process.

- Using map() to loop through DataFrame
- Using foreach() to loop through DataFrame
- Using pandas() to Iterate
- Collect Data As List and Loop Through in Python

PySpark Loop Through Rows in DataFrame Examples

In order to explain with examples, let's create a DataFrame

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [('James','Smith','M',30),('Anna','Rose','F',41),
  ('Robert','Williams','M',62),
]
columns = ["firstname","lastname","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.show()
+---------+--------+------+------+
|firstname|lastname|gender|salary|
+---------+--------+------+------+
|    James|   Smith|     M|    30|
|     Anna|    Rose|     F|    41|
|   Robert|Williams|     M|    62|
+---------+--------+------+------+
```

Mostly for simple computations, instead of iterating through using map() and foreach(), you should use either DataFrame select() or DataFrame withColumn() in conjunction with PySpark SQL functions.

```
from pyspark.sql.functions import concat_ws,col,lit
df.select(concat_ws(",",df.firstname,df.lastname).alias("name"), \
          df.gender,lit(df.salary*2).alias("new_salary")).show()
+---------------+------+----------+
|           name|gender|new_salary|
+---------------+------+----------+
|    James,Smith|     M|        60|
|      Anna,Rose|     F|        82|
|Robert,Williams|     M|       124|
+---------------+------+----------+
```

Below I have map() example to achieve same output as above.

Using map() to Loop Through Rows in DataFrame

PySpark map() Transformation is used to loop/iterate through the PySpark DataFrame/RDD by applying the transformation function (lambda) on every element (Rows and Columns) of RDD/DataFrame. PySpark doesn't have a map() in DataFrame instead it's in RDD hence we need to convert DataFrame to RDD first and then use the map(). It returns an RDD and you should Convert RDD to PySpark DataFrame if needed. If you have a heavy initialization use PySpark mapPartitions() transformation instead of map(), as with mapPartitions() heavy initialization executes only once for each partition instead of every record.

```
# Refering columns by index.
rdd=df.rdd.map(lambda x:
    (x[0]+","+x[1],x[2],x[3]*2)
    )
df2=rdd.toDF(["name","gender","new_salary"])
df2.show()
```

The above example iterates through every row in a DataFrame by applying transformations to the data, since I need a DataFrame back, I have converted the result of RDD to DataFrame with new column names. Note that here I have used index to get the column values, alternatively, you can also refer to the DataFrame column names while iterating.

```
# Referring Column Names
rdd2=df.rdd.map(lambda x:
   (x["firstname"]+","+x["lastname"],x["gender"],x["salary"]*2)
   )
```

Another alternative

```
# Referring Column Names
rdd2=df.rdd.map(lambda x:
   (x.firstname+","+x.lastname,x.gender,x.salary*2)
   )
```

You can also create a custom function to perform an operation.
Below `func1()` function executes for every DataFrame row from the lambda function.

```
# By Calling function
def func1(x):
   firstName=x.firstname
   lastName=x.lastName
   name=firstName+","+lastName
   gender=x.gender.lower()
   salary=x.salary*2
   return (name,gender,salary)

rdd2=df.rdd.map(lambda x: func1(x))
```

Using foreach() to Loop Through Rows in DataFrame

Similar to map(), foreach() also applied to every row of DataFrame, the difference being foreach() is an action and it returns nothing. Below are some examples to iterate through DataFrame using for each.

```
# Foreach example
def f(x): print(x)
df.foreach(f)

# Another example
df.foreach(lambda x:
   print("Data
==>"+x["firstname"]+","+x["lastname"]+","+x["gender"]+","+str(x["salary"]*2))
   )
```

Using pandas() to Iterate

If you have a small dataset, you can also Convert PySpark DataFrame to Pandas and use pandas to iterate through. Use spark.sql.execution.arrow.enabled config to enable Apache Arrow with Spark. Apache Spark uses Apache Arrow which is an in-memory columnar format to transfer the data between Python and JVM.

```
# Using pandas
import pandas as pd
spark.conf.set("spark.sql.execution.arrow.enabled", "true")
pandasDF = df.toPandas()
for index, row in pandasDF.iterrows():
    print(row['firstname'], row['gender'])
```

Collect Data As List and Loop Through

You can also Collect the PySpark DataFrame to Driver and iterate through Python, you can also use toLocalIterator().

```
# Collect the data to Python List
dataCollect = df.collect()
for row in dataCollect:
    print(row['firstname'] + "," +row['lastname'])

#Using toLocalIterator()
dataCollect=df.rdd.toLocalIterator()
for row in dataCollect:
    print(row['firstname'] + "," +row['lastname'])
```

PySpark Random Sample with Example

PySpark provides a pyspark.sql.DataFrame.sample(), pyspark.sql.DataFrame.sampleBy(), RDD.sample(), and RDD.takeSample() methods to get the random sampling subset from the large dataset, In this article I will explain with Python examples.

If you are working as a Data Scientist or Data analyst you are often required to analyze a large dataset/file with billions or trillions of records, processing these large datasets takes some time hence during the analysis phase it is recommended to use a random subset sample from the large files.

1. PySpark SQL sample() Usage & Examples

PySpark sampling (`pyspark.sql.DataFrame.sample()`) is a mechanism to get random sample records from the dataset, this is helpful when you have a larger dataset and wanted to analyze/test a subset of the data for example 10% of the original file. Below is the syntax of the `sample()` function.

```
sample(withReplacement, fraction, seed=None)
```

`fraction` – Fraction of rows to generate, range [0.0, 1.0]. Note that it doesn't guarantee to provide the exact number of the fraction of records.
`seed` – Seed for sampling (default a random seed). Used to reproduce the same random sampling.
`withReplacement` – Sample with replacement or not (default False).
Let's see some examples.

1.1 Using `fraction` to get a random sample in PySpark

By using fraction between 0 to 1, it returns the approximate number of the fraction of the dataset. For example, `0.1` returns 10% of the rows. However, this does not guarantee it returns the exact 10% of the records.
Note: If you run these examples on your system, you may see different results.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
   .master("local[1]") \
   .appName("SparkByExamples.com") \
   .getOrCreate()

df=spark.range(100)
print(df.sample(0.06).collect())
//Output: [Row(id=0), Row(id=2), Row(id=17), Row(id=25), Row(id=26), Row(id=44), Row(id=80)]
```

My DataFrame has 100 records and I wanted to get 6% sample records which are 6 but the sample() function returned 7 records. This proves the sample function doesn't return the exact fraction specified.

1.2 Using `seed` to reproduce the same Samples in PySpark

Every time you run a sample() function it returns a different set of sampling records, however sometimes during the development and testing phase you may need to regenerate the same sample every time as you need to compare the results from your previous run. To get consistent same random sampling uses the same slice value for every run. Change slice value to get different results.

```
print(df.sample(0.1,123).collect())
```

```
//Output: 36,37,41,43,56,66,69,75,83

print(df.sample(0.1,123).collect())
//Output: 36,37,41,43,56,66,69,75,83

print(df.sample(0.1,456).collect())
//Output: 19,21,42,48,49,50,75,80
```

Here, first 2 examples I have used seed value `123` hence the sampling results are the same and for the last example, I have used `456` as a seed value generate different sampling records.

1.3 Sample `withReplacement` (May contain duplicates)

some times you may need to get a random sample with repeated values. By using the value `true`, results in repeated values.

```

print(df.sample(True,0.3,123).collect()) //with Duplicates
//Output: 0,5,9,11,14,14,16,17,21,29,33,41,42,52,52,54,58,65,65,71,76,79,85,96
print(df.sample(0.3,123).collect()) // No duplicates
//Output:
0,4,17,19,24,25,26,36,37,41,43,44,53,56,66,68,69,70,71,75,76,78,83,84,88,94,96,97,9
8
```

On first example, values 14, 52 and 65 are repeated values.

1.4 Stratified sampling in PySpark

You can get Stratified sampling in PySpark without replacement by using `sampleBy()` method. It returns a sampling fraction for each stratum. If a stratum is not specified, it takes zero as the default.

sampleBy() Syntax

```

sampleBy(col, fractions, seed=None)
```

col – column name from DataFrame

fractions – It’s Dictionary type takes key and value.

sampleBy() Example

```

df2=df.select((df.id % 3).alias("key"))
print(df2.sampleBy("key", {0: 0.1, 1: 0.2},0).collect())
//Output: [Row(key=0), Row(key=1), Row(key=1), Row(key=1), Row(key=0),
Row(key=1), Row(key=1), Row(key=0), Row(key=1), Row(key=1), Row(key=1)]
```

2. PySpark RDD Sample

PySpark RDD also provides `sample()` function to get a random sampling, it also has another signature `takeSample()` that returns an Array[T].

RDD `sample()` Syntax & Example

PySpark RDD sample() function returns the random sampling similar to DataFrame and takes a similar types of parameters but in a different order. Since I've already covered the explanation of these parameters on DataFrame, I will not be repeating the explanation on RDD, If not already read I recommend reading the DataFrame section above.

sample() of RDD returns a new RDD by selecting random sampling. Below is a syntax.

```
sample(self, withReplacement, fraction, seed=None)
```

Below is an example of RDD sample() function

```
rdd = spark.sparkContext.range(0,100)
print(rdd.sample(False,0.1,0).collect())
//Output: [24, 29, 41, 64, 86]
print(rdd.sample(True,0.3,123).collect())
//Output: [0, 11, 13, 14, 16, 18, 21, 23, 27, 31, 32, 32, 48, 49, 49, 53, 54, 72, 74, 77,
77, 83, 88, 91, 93, 98, 99]
```

RDD takeSample() Syntax & Example

RDD takeSample() is an action hence you need to careful when you use this function as it returns the selected sample records to driver memory. Returning too much data results in an out-of-memory error similar to collect().

Syntax of RDD takeSample() .

```
takeSample(self, withReplacement, num, seed=None)
```

Example of RDD takeSample()

```
print(rdd.takeSample(False,10,0))
//Output: [58, 1, 96, 74, 29, 24, 32, 37, 94, 91]
print(rdd.takeSample(True,30,123))
//Output: [43, 65, 39, 18, 84, 86, 25, 13, 40, 21, 79, 63, 7, 32, 26,
```

PySpark fillna() & fill() – Replace NULL/None Values

In PySpark, DataFrame.fillna() or DataFrameNaFunctions.fill() is used to replace NULL/None values on all or selected multiple DataFrame columns with either **zero(0), empty string, space, or any constant literal** values.

While working on PySpark DataFrame we often need to replace null values since certain operations on null value return error hence, we need to graciously handle nulls as the first step before processing. Also, while writing to a file, it's always best practice to replace null values, not doing this result nulls on the output file.

As part of the cleanup, sometimes you may need to Drop Rows with NULL/None Values in PySpark DataFrame and Filter Rows by checking IS NULL/NOT NULL conditions.
In this article, I will use both fill() and fillna() to replace null/none values with an empty string, constant value, and zero(0) on Dataframe columns integer, string with Python examples.

- PySpark fillna() and fill() Syntax
- Replace NULL/None Values with Zero (0)
- Replace NULL/None Values with Empty String

Before we start, Let's read a CSV into PySpark DataFrame file, where we have no values on certain rows of String and Integer columns, PySpark assigns null values to these no value columns.
The file we are using here is available at GitHub small_zipcode.csv

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .master("local[1]") \
    .appName("SparkByExamples.com") \
    .getOrCreate()

filePath="resources/small_zipcode.csv"
df = spark.read.options(header='true', inferSchema='true') \
          .csv(filePath)

df.printSchema()
df.show(truncate=False)
```

This yields the below output. As you see columns type, city and population columns have null values.

```
+---+-------+--------+-------------------+-----+----------+
|id |zipcode|type    |city               |state|population|
+---+-------+--------+-------------------+-----+----------+
|1  |704    |STANDARD|null               |PR   |30100     |
|2  |704    |null    |PASEO COSTA DEL SUR|PR   |null      |
|3  |709    |null    |BDA SAN LUIS       |PR   |3700      |
|4  |76166  |UNIQUE  |CINGULAR WIRELESS  |TX   |84000     |
|5  |76177  |STANDARD|null               |TX   |null      |
+---+-------+--------+-------------------+-----+----------+
```

Now, let's see how to replace these null values.

PySpark fillna() & fill() Syntax

PySpark provides DataFrame.fillna() and DataFrameNaFunctions.fill() to replace NULL/None values. These two are aliases of each other and returns the same results.

```
fillna(value, subset=None)
fill(value, subset=None)
```

- **value** – Value should be the data type of int, long, float, string, or dict. Value specified here will be replaced for NULL/None values.
- **subset** – This is optional, when used it should be the subset of the column names where you wanted to replace NULL/None values.

PySpark Replace NULL/None Values with Zero (0)

PySpark fill(value:Long) signatures that are available in DataFrameNaFunctions is used to replace NULL/None values with numeric values either zero(0) or any constant value for all integer and long datatype columns of PySpark DataFrame or Dataset.

```
#Replace 0 for null for all integer columns
df.na.fill(value=0).show()

#Replace 0 for null on only population column
df.na.fill(value=0,subset=["population"]).show()
```

Above both statements yields the same output, since we have just an integer column population with null values Note that it replaces only Integer columns since our value is 0.

```
+---+-------+--------+-------------------+-----+----------+
|id |zipcode|type    |city               |state|population|
+---+-------+--------+-------------------+-----+----------+
|1  |704    |STANDARD|null               |PR   |30100     |
|2  |704    |null    |PASEO COSTA DEL SUR|PR   |0         |
|3  |709    |null    |BDA SAN LUIS       |PR   |3700      |
|4  |76166  |UNIQUE  |CINGULAR WIRELESS  |TX   |84000     |
|5  |76177  |STANDARD|null               |TX   |0         |
+---+-------+--------+-------------------+-----+----------+
```

PySpark Replace Null/None Value with Empty String

Now let’s see how to replace NULL/None values with an empty string or any constant values String on all DataFrame String columns.

```
df.na.fill("").show(false)
```

Yields below output. This replaces all String type columns with empty/blank string for all NULL values.

```
+---+-------+--------+-------------------+-----+----------+
|id |zipcode|type    |city               |state|population|
+---+-------+--------+------------------+-----+----------+
|1  |704    |STANDARD|                  |PR   |30100     |
|2  |704    |        |PASEO COSTA DEL SUR|PR   |null      |
|3  |709    |        |BDA SAN LUIS      |PR   |3700      |
|4  |76166  |UNIQUE  |CINGULAR WIRELESS |TX   |84000     |
|5  |76177  |STANDARD|                  |TX   |null      |
+---+-------+--------+------------------+-----+----------+
```

Now, let's replace NULL's on specific columns, below example replace column type with empty string and column city with value "unknown".

```
df.na.fill("unknown",["city"]) \
   .na.fill("",["type"]).show()
```

Yields below output. This replaces null values with an empty string for type column and replaces with a constant value "unknown" for city column.

```
+---+-------+--------+------------------+-----+----------+
|id |zipcode|type    |city              |state|population|
+---+-------+--------+------------------+-----+----------+
|1  |704    |STANDARD|unknown           |PR   |30100     |
|2  |704    |        |PASEO COSTA DEL SUR|PR   |null      |
|3  |709    |        |BDA SAN LUIS      |PR   |3700      |
|4  |76166  |UNIQUE  |CINGULAR WIRELESS |TX   |84000     |
|5  |76177  |STANDARD|unknown           |TX   |null      |
+---+-------+--------+------------------+-----+----------+
```

Alternatively you can also write the above statement as

```
df.na.fill({"city": "unknown", "type": ""}) \
   .show()
```

Complete Code

Below is complete code with Scala example. You can use it by copying it from here or use the GitHub to download the source code.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
   .master("local[1]") \
   .appName("SparkByExamples.com") \
   .getOrCreate()

filePath="resources/small_zipcode.csv"
df = spark.read.options(header='true', inferSchema='true') \
        .csv(filePath)

df.printSchema()
df.show(truncate=False)


df.fillna(value=0).show()
df.fillna(value=0,subset=["population"]).show()
df.na.fill(value=0).show()
df.na.fill(value=0,subset=["population"]).show()


df.fillna(value="").show()
df.na.fill(value="").show()

df.fillna("unknown",["city"]) \
   .fillna("",["type"]).show()

df.fillna({"city": "unknown", "type": ""}) \
   .show()

df.na.fill("unknown",["city"]) \
   .na.fill("",["type"]).show()

df.na.fill({"city": "unknown", "type": ""}) \
   .show()
```

PySpark Pivot and Unpivot DataFrame

PySpark pivot() function is used to rotate/transpose the data from one column into multiple Dataframe columns and back using unpivot(). Pivot() It is an aggregation where one of the grouping columns values is transposed into individual columns with distinct data.
This tutorial describes and provides a PySpark example on how to create a Pivot table on DataFrame and Unpivot back.

- Pivot PySpark DataFrame
- Pivot Performance improvement in PySpark 2.0
- Unpivot PySpark DataFrame
- Pivot or Transpose without aggregation

Let's create a PySpark DataFrame to work with.

```

import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import expr
#Create spark session
```

```
data = [("Banana",1000,"USA"), ("Carrots",1500,"USA"), ("Beans",1600,"USA"), \
    ("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",400,"China"), \
    ("Carrots",1200,"China"),("Beans",1500,"China"),("Orange",4000,"China"), \
    ("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",2000,"Mexico")]

columns= ["Product","Amount","Country"]
df = spark.createDataFrame(data = data, schema = columns)
df.printSchema()
df.show(truncate=False)
```

DataFrame 'df' consists of 3 columns Product, Amount, and Country as shown below.

```
root
 |-- Product: string (nullable = true)
 |-- Amount: long (nullable = true)
 |-- Country: string (nullable = true)

+-------+------+-------+
|Product|Amount|Country|
+-------+------+-------+
|Banana |1000  |USA    |
|Carrots|1500  |USA    |
|Beans  |1600  |USA    |
|Orange |2000  |USA    |
|Orange |2000  |USA    |
|Banana |400   |China  |
|Carrots|1200  |China  |
|Beans  |1500  |China  |
|Orange |4000  |China  |
|Banana |2000  |Canada |
|Carrots|2000  |Canada |
|Beans  |2000  |Mexico |
+-------+------+-------+
```

Pivot PySpark DataFrame

PySpark SQL provides pivot() function to rotate the data from one column into multiple columns. It is an aggregation where one of the grouping columns values is transposed into individual columns with distinct data. To get the total amount exported to each country of each product, will do group by Product, pivot by Country, and the sum of Amount.

```
pivotDF = df.groupBy("Product").pivot("Country").sum("Amount")
pivotDF.printSchema()
pivotDF.show(truncate=False)
```

This will transpose the countries from DataFrame rows into columns and produces the below output. where ever data is not present, it represents as null by default.

```
root
 |-- Product: string (nullable = true)
 |-- Canada: long (nullable = true)
 |-- China: long (nullable = true)
 |-- Mexico: long (nullable = true)
 |-- USA: long (nullable = true)

+-------+------+-----+------+----+
|Product|Canada|China|Mexico|USA |
+-------+------+-----+------+----+
|Orange |null  |4000 |null  |4000|
|Beans  |null  |1500 |2000  |1600|
|Banana |2000  |400  |null  |1000|
|Carrots|2000  |1200 |null  |1500|
+-------+------+-----+------+----+
```

Pivot Performance improvement in PySpark 2.0

version 2.0 on-wards performance has been improved on Pivot, however, if you are using the lower version; note that pivot is a very expensive operation hence, it is recommended to provide column data (if known) as an argument to function as shown below.

```
countries = ["USA","China","Canada","Mexico"]
pivotDF = df.groupBy("Product").pivot("Country", countries).sum("Amount")
pivotDF.show(truncate=False)
```

Another approach is to do two-phase aggregation. PySpark 2.0 uses this implementation in order to improve the performance Spark-13749

```
pivotDF = df.groupBy("Product","Country") \
      .sum("Amount") \
      .groupBy("Product") \
      .pivot("Country") \
      .sum("sum(Amount)") \
pivotDF.show(truncate=False)
```

The above two examples return the same output but with better performance.

Unpivot PySpark DataFrame

Unpivot is a reverse operation, we can achieve by rotating column values into rows values. PySpark SQL doesn't have unpivot function hence will use the stack() function. Below code converts column countries to row.

```
from pyspark.sql.functions import expr
```

```
unpivotExpr = "stack(3, 'Canada', Canada, 'China', China, 'Mexico', Mexico) as
(Country,Total)"
unPivotDF = pivotDF.select("Product", expr(unpivotExpr)) \
   .where("Total is not null")
unPivotDF.show(truncate=False)
unPivotDF.show()
```

It converts pivoted column “country” to rows.

```
+-------+-------+-----+
|Product|Country|Total|
+-------+-------+-----+
| Orange|  China| 4000|
|  Beans|  China| 1500|
|  Beans| Mexico| 2000|
| Banana| Canada| 2000|
| Banana|  China|  400|
|Carrots| Canada| 2000|
|Carrots|  China| 1200|
+-------+-------+-----+
```

Transpose or Pivot without aggregation

Can we do PySpark DataFrame transpose or pivot without aggregation?

Off course you can, but unfortunately, you can’t achieve using the Pivot function. However, pivoting or transposing the DataFrame structure without aggregation from rows to columns and columns to rows can be easily done using PySpark and Scala hack. please refer to this example.

Complete Example

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import expr
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [("Banana",1000,"USA"), ("Carrots",1500,"USA"), ("Beans",1600,"USA"), \
     ("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",400,"China"), \
     ("Carrots",1200,"China"),("Beans",1500,"China"),("Orange",4000,"China"), \
     ("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",2000,"Mexico")]

columns= ["Product","Amount","Country"]
```

```
df = spark.createDataFrame(data = data, schema = columns)
df.printSchema()
df.show(truncate=False)

pivotDF = df.groupBy("Product").pivot("Country").sum("Amount")
pivotDF.printSchema()
pivotDF.show(truncate=False)

pivotDF = df.groupBy("Product","Country") \
      .sum("Amount") \
      .groupBy("Product") \
      .pivot("Country") \
      .sum("sum(Amount)")
pivotDF.printSchema()
pivotDF.show(truncate=False)

""" unpivot """
unpivotExpr = "stack(3, 'Canada', Canada, 'China', China, 'Mexico', Mexico) as
(Country,Total)"
unPivotDF = pivotDF.select("Product", expr(unpivotExpr)) \
    .where("Total is not null")
unPivotDF.show(truncate=False)
```

PySpark partitionBy() – Write to Disk Example

- Post author:NNK
- Post category: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 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.

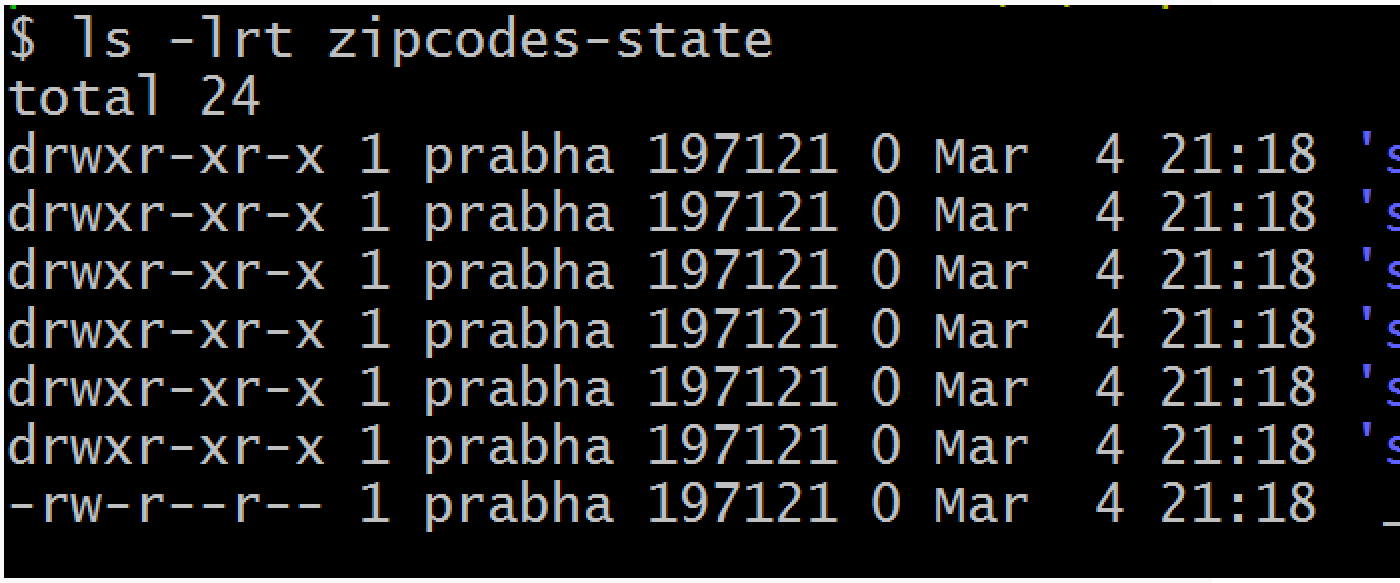
```
Syntax: partitionBy(self, *cols)
```

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.



```
$ ls -lrt zipcodes-state
total 24
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
-rw-r--r-- 1 prabha 197121 0 Mar  4 21:18 _
```

partitionBy(“state”) example output

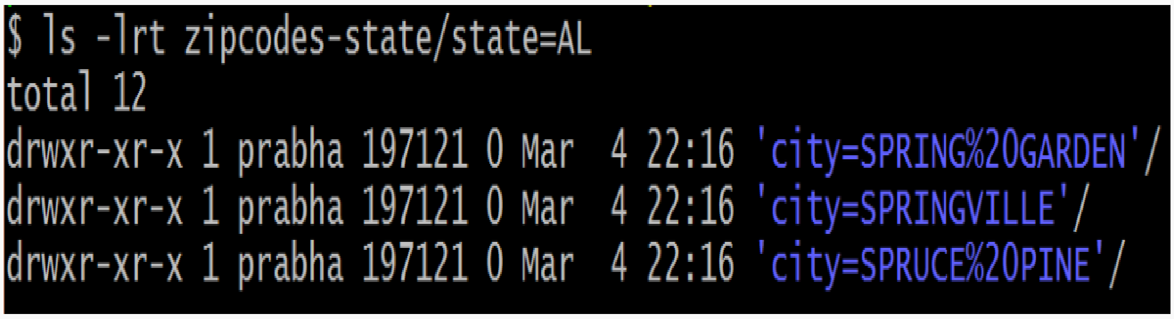
On each directory, you may see one or more part files (since our dataset is small, all records for each state are kept in a single part file). You can change this behavior by repartition() the data in memory first. Specify the number of partitions (part files) you would want for each state as an argument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



```
$ 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'/
```

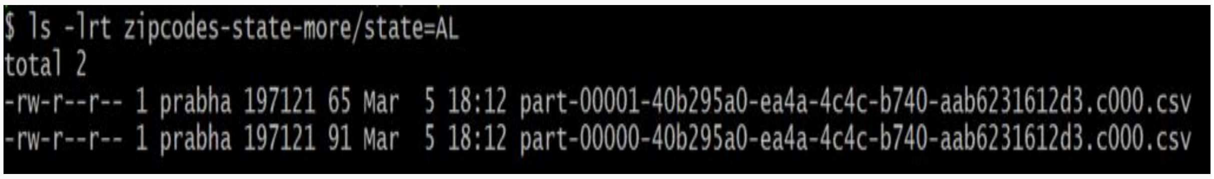
partitonBy("state","city") multiple columns

6. Using repartition() and partitionBy() together

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.



```
$ 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: 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.

PySpark ArrayType Column With

Examples:PySpark pyspark.sql.types.ArrayType (ArrayType extends DataType class) is used to define an array data type column on DataFrame that holds the same type of elements, In this article, I will explain how to create a DataFrame ArrayType column using org.apache.spark.sql.types.ArrayType class and applying some SQL functions on the array columns with examples.

While working with structured files (Avro, Parquet e.t.c) or semi-structured (JSON) files, we often get data with complex structures like MapType, ArrayType, StructType e.t.c. I will try my best to cover some mostly used functions on ArrayType columns.

What is PySpark ArrayType

PySpark ArrayType is a collection data type that extends the DataType class which is a superclass of all types in PySpark. All elements of ArrayType should have the same type of elements.

Create PySpark ArrayType

You can create an instance of an ArrayType using ArraType() class, This takes arguments valueType and one optional argument valueContainsNull to specify if a value can accept null, by default it takes True. valueType should be a PySpark type that extends DataType class.

```
from pyspark.sql.types import StringType, ArrayType
arrayCol = ArrayType(StringType(),False)
```

Create PySpark ArrayType Column Using StructType

Let’s create a DataFrame with few array columns by using PySpark StructType & StructField classes.

```
data = [
 ("James,,Smith",["Java","Scala","C++"],["Spark","Java"],"OH","CA"),
 ("Michael,Rose,",["Spark","Java","C++"],["Spark","Java"],"NY","NJ"),
 ("Robert,,Williams",["CSharp","VB"],["Spark","Python"],"UT","NV")
]

from pyspark.sql.types import StringType, ArrayType,StructType,StructField
schema = StructType([
    StructField("name",StringType(),True),
    StructField("languagesAtSchool",ArrayType(StringType()),True),
    StructField("languagesAtWork",ArrayType(StringType()),True),
    StructField("currentState", StringType(), True),
    StructField("previousState", StringType(), True)
  ])

df = spark.createDataFrame(data=data,schema=schema)
df.printSchema()
df.show()
```

This snippet creates two Array columns languagesAtSchool and languagesAtWork which defines languages learned at School and languages using at work. For the rest of the article, I will use these array columns of DataFrame and provide examples of PySpark SQL array functions. printSchema() and show() from above snippet display below output.

```
root
 |-- name: string (nullable = true)
 |-- languagesAtSchool: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- languagesAtWork: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- currentState: string (nullable = true)
 |-- previousState: string (nullable = true)
+----------------+------------------+---------------+------------+-------------+
|            name| languagesAtSchool|languagesAtWork|currentState|previousState|
+----------------+------------------+---------------+------------+-------------+
|    James,,Smith|[Java, Scala, C++]|  [Spark, Java]|          OH|           CA|
|   Michael,Rose,|[Spark, Java, C++]|  [Spark, Java]|          NY|           NJ|
|Robert,,Williams|      [CSharp, VB]|[Spark, Python]|          UT|           NV|
+----------------+------------------+---------------+------------+-------------+
```

PySpark ArrayType (Array) Functions

PySpark SQL provides several Array functions to work with the ArrayType column, In this section, we will see some of the most commonly used SQL functions.

explode()

Use explode() function to create a new row for each element in the given array column. There are various PySpark SQL explode functions available to work with Array columns.

```
from pyspark.sql.functions import explode
df.select(df.name,explode(df.languagesAtSchool)).show()

+----------------+------+
|            name|   col|
+----------------+------+
|    James,,Smith|  Java|
|   James,,Smith| Scala|
|   James,,Smith|   C++|
|  Michael,Rose,| Spark|
|  Michael,Rose,|  Java|
|  Michael,Rose,|   C++|
|Robert,,Williams|CSharp|
|Robert,,Williams|    VB|
+----------------+------+
```

Split()

split() sql function returns an array type after splitting the string column by delimiter. Below example split the name column by comma delimiter.

```
from pyspark.sql.functions import split
df.select(split(df.name,",").alias("nameAsArray")).show()

+--------------------+
|         nameAsArray|
+--------------------+
|    [James, , Smith]|
|   [Michael, Rose, ]|
|[Robert, , Williams]|
+--------------------+
```

array()

Use array() function to create a new array column by merging the data from multiple columns. All input columns must have the same data type. The below example combines the data from currentState and previousState and creates a new column states.

```
from pyspark.sql.functions import array
df.select(df.name,array(df.currentState,df.previousState).alias("States")).show()
+----------------+--------+
|            name|  States|
+----------------+--------+
|    James,,Smith|[OH, CA]|
|   Michael,Rose,|[NY, NJ]|
|Robert,,Williams|[UT, NV]|
+----------------+--------+
```

array_contains()

array_contains() sql function is used to check if array column contains a value. Returns null if the array is null, true if the array contains the value, and false otherwise.

```
from pyspark.sql.functions import array_contains
df.select(df.name,array_contains(df.languagesAtSchool,"Java")
   .alias("array_contains")).show()

+----------------+--------------+
|            name|array_contains|
+----------------+--------------+
|    James,,Smith|          true|
|   Michael,Rose,|          true|
|Robert,,Williams|         false|
+----------------+--------------+
```

PySpark MapType (Dict) Usage with Examples

PySpark MapType (also called map type) is a data type to represent Python Dictionary (dict) to store key-value pair, a MapType object comprises three fields, keyType (a DataType), valueType (a DataType) and valueContainsNull (a BooleanType).

What is PySpark MapType

PySpark MapType is used to represent map key-value pair similar to python Dictionary (Dict), it extends DataType class which is a superclass of all types in PySpark and takes two mandatory arguments keyType and valueType of type DataType and one optional boolean argument valueContainsNull. keyType and valueType can be any type that extends the DataType class. for e.g StringType, IntegerType, ArrayType, MapType, StructType (struct) e.t.c.

1. Create PySpark MapType

In order to use MapType data type first, you need to import it from pyspark.sql.types.MapType and use MapType() constructor to create a map object.

```
from pyspark.sql.types import StringType, MapType
mapCol = MapType(StringType(),StringType(),False)
```

MapType Key Points:

- The First param keyType is used to specify the type of the key in the map.
- The Second param valueType is used to specify the type of the value in the map.
- Third parm valueContainsNull is an optional boolean type that is used to specify if the value of the second param can accept Null/None values.
- The key of the map won’t accept None/Null values.
- PySpark provides several SQL functions to work with MapType.

2. Create MapType From StructType

Let's see how to create a MapType by using PySpark StructType & StructField, StructType() constructor takes list of StructField, StructField takes a fieldname and type of the value.

```
from pyspark.sql.types import StructField, StructType, StringType, MapType
schema = StructType([
   StructField('name', StringType(), True),
   StructField('properties', MapType(StringType(),StringType()),True)
])
```

Now let's create a DataFrame by using above StructType schema.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
dataDictionary = [
      ('James',{'hair':'black','eye':'brown'}),
      ('Michael',{'hair':'brown','eye':None}),
      ('Robert',{'hair':'red','eye':'black'}),
      ('Washington',{'hair':'grey','eye':'grey'}),
      ('Jefferson',{'hair':'brown','eye':''})
      ]
df = spark.createDataFrame(data=dataDictionary, schema = schema)
df.printSchema()
df.show(truncate=False)
```

df.printSchema() yields the Schema and df.show() yields the DataFrame output.

```
root
 |-- Name: string (nullable = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)

+----------+-----------------------------+
|Name      |properties                   |
+----------+-----------------------------+
|James     |[eye -> brown, hair -> black]|
|Michael   |[eye ->, hair -> brown]      |
|Robert    |[eye -> black, hair -> red]  |
|Washington|[eye -> grey, hair -> grey]  |
|Jefferson |[eye -> , hair -> brown]     |
+----------+-----------------------------+
```

3. Access PySpark MapType Elements

Let’s see how to extract the key and values from the PySpark DataFrame Dictionary column. Here I have used PySpark map transformation to read the values of properties (MapType column)

```
df3=df.rdd.map(lambda x: \
   (x.name,x.properties["hair"],x.properties["eye"])) \
   .toDF(["name","hair","eye"])
df3.printSchema()
df3.show()

root
 |-- name: string (nullable = true)
 |-- hair: string (nullable = true)
 |-- eye: string (nullable = true)

+----------+-----+-----+
|      name| hair|  eye|
+----------+-----+-----+
|     James|black|brown|
|   Michael|brown| null|
|    Robert|  red|black|
|Washington| grey| grey|
| Jefferson|brown|     |
+----------+-----+-----+
```

Let’s use another way to get the value of a key from Map using getItem() of Column type, this method takes a key as an argument and returns a value.

```
df.withColumn("hair",df.properties.getItem("hair")) \
  .withColumn("eye",df.properties.getItem("eye")) \
  .drop("properties") \
  .show()

df.withColumn("hair",df.properties["hair"]) \
  .withColumn("eye",df.properties["eye"]) \
  .drop("properties") \
  .show()
```

4. Functions

Below are some of the MapType Functions with examples.

4.1 – explode

```
from pyspark.sql.functions import explode
df.select(df.name,explode(df.properties)).show()

+----------+----+-----+
|      name| key|value|
+----------+----+-----+
|     James| eye|brown|
|     James|hair|black|
|   Michael| eye| null|
|   Michael|hair|brown|
|    Robert| eye|black|
|    Robert|hair|  red|
|Washington| eye| grey|
|Washington|hair| grey|
| Jefferson| eye|     |
| Jefferson|hair|brown|
+----------+----+-----+
```

4.2 map_keys() – Get All Map Keys

```
from pyspark.sql.functions import map_keys
df.select(df.name,map_keys(df.properties)).show()

+----------+--------------------+
|      name|map_keys(properties)|
+----------+--------------------+
|     James|         [eye, hair]|
|   Michael|         [eye, hair]|
|    Robert|         [eye, hair]|
|Washington|         [eye, hair]|
| Jefferson|         [eye, hair]|
+----------+--------------------+
```

In case if you wanted to get all map keys as Python List. **WARNING**: **This runs very slow**.

```
from pyspark.sql.functions import explode,map_keys
keysDF = df.select(explode(map_keys(df.properties))).distinct()
keysList = keysDF.rdd.map(lambda x:x[0]).collect()
print(keysList)
#['eye', 'hair']
```

4.3 map_values() – Get All map Values

```
from pyspark.sql.functions import map_values
df.select(df.name,map_values(df.properties)).show()

+----------+---------------------+
|      name|map_values(properties)|
+----------+---------------------+
|     James|       [brown, black]|
|   Michael|            [, brown]|
|    Robert|         [black, red]|
|Washington|         [grey, grey]|
| Jefferson|            [, brown]|
+----------+---------------------+
```

Conclusion

MapType is a map data structure that is used to store key key-value pairs similar to Python Dictionary (Dic), keys and values type of map should be of a type that extends DataType. Key won’t accept null/None values whereas map of the key can have None/Null value.