Pyspark

Pyspark DataFrame & SQL

Ways to Rename Column on DataFrame

```
from pyspark.sql.types import StrucType, StrucField, StringType, IntegerType

data = [()]
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)
         ])

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

StrucType --> columns 类型 StrucField --> column 类型 others --> type of column

1.Using withColumnRenamed()

```
# month --> withColumnRenamed(existingName, newName)
# return a new df and doesn't modify the current df
df.withColumnRenamed('A', 'B')
```

2.Using StrucType to Rename a Nested Column in DataFrame

```
from pyspark.sql.functions import *
schema2 = StructType([
    StructField("fname",StringType()),
```

```
    StructField("middlename",StringType()),
    StructField("lname",StringType())])

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

3.Using Select

```
from pyspark.sql.functions import *
df.select(col('existing').alias('new'),col('...').alias('...'),)
```

4.Using withColumn

```
from pyspark.sql.functions import *
df.withColumn('newColumn', col('existingColumn')).drop('existingColumn')
```

5.Using toDF()

```
newColumns = ['A', 'B', 'C']
df.toDF(*newColumns)
```

Usage of withColumn

1.Change DateType of a Column

```
#
df.withColumn('A', col('A').cast('Integer'))
```

2.Update The Value of a Column

```
df.withColumn('A', col('A')*100)
```

3.Create a New Column from an Existing

```
df.withColumn('newColumn', col('existingColumn')-100)
```

4.Create a New Column

```
from pyspark.sql.functions import lit
df.withColumn('newColumn', lit('AAA'))
# lit() function is used to add a constant value to the column
```

Filter data from DataFrame

1. Filter with Column Condition

```
df.filter(df.aColumn == 'aaa')
# same as filter of pandas.DataFrame

from pyspark.sql.functions import col
df.filter(col('aColumn') == 'aaa')

# filter multiple condition
df.filter((df.A == 'aaa')&(df.B == 'bbb'))
```

2.Filter with SQL Expression

```
df.filter("aColumn <> 'aaa'")
```

3.Filter Based on a List Values

```
# Using isin()
aList = ['aaa', 'bbb', 'ccc']
df.filter(df.A.isin(aList))
```

4.Filter a StringType Column

```
# Function: startswith/endswith/contains/like/rlike
# rlike --> like with regex pattern
```

5.Filter on an Array Column

```
# using array_contains()
from pyspark.sql.functions import array_contains
df.filter(array_contains(df.anArrayColumn, 'aaa'))
```

5.Filter on a Nested Struct Column

```
df.filter(df.aNestedStructColumn.Columname == "aaa")
```

Sort Values

1.Usage of sort()

```
df.sort('ColumnA', 'ColumnB').show(truncate=False)
df.sort(col('ColumnA'), col('ColumnB')).show(truncate=True)
# Ascending order is used by default
# Show 20 characters at most if truncate is True by default, or show all content

# Using asc/desc method to specify sort on DataFrame
from pyspark.sql.functions import asc, desc
df.sort(df.ColumnA.desc(), df.ColumnB.asc())
```

2.Sorting in SQL

```
# create a view on this DataFrame as first
df.createOrReplaceTempView('temp')
# read data from view using SQL
spark.sql('select ColumnA, ColumnB from temp order by ColumnA desc')
```

Explode array and map columns to rows

1.Using explode

```
# explode array columns
from pyspark.sql.function import explode
df2 = df.select(df.name, explode(df.ArrayColumn))

'''
eg: df --> +------+-----------------+
           |  name|      ArrayColumn|
           +------+-----------------+
           |     A|    [aaa,bbb,ccc]|

    df2--> +------+--------+
           |  name|     col|
           +------+--------+
           |     A|     aaa|
           |     A|     bbb|
           |     A|     ccc|
'''

# explode map columns
```

```
from pyspark.sql.function import explode
df2 = df.select(df.name, explode(df.MapColumn))

'''
eg: df --> +------+--------------------------------+
           |  name|                        MapColumn|
           +------+--------------------------------+
           |     A|{'1':'aaa', '2':'bbb', '3':None}|

    df2--> +------+-- --------+
           |  name| key| value|
           +------+-----------+
           |     A|  1 |   aaa|
           |     A|  2 |   bbb|
'''
# if the value is None, explode will ignore elements
```

2.Using explode_outer

```
# this function is same as explode, while explode returns null when elements is
null or empty.
from pyspark.sql.functions import explode_outer

df.selet(df.name, explode_outer(df.ArrayColumn)).show()
```

3.posexplode and posexplode_outer

```
# posexplode methond records the position when creating a row for each element.

from pyspark.sql.functions import posexplode_outer
df.select($"name",posexplode_outer($"ArrayColumn")).show()

'''
eg: df --> +------+-----------------+
           |  name|      ArrayColumn|
           +------+-----------------+
           |     A|   [aaa,bbb,None]|

    df2--> +------+-- --------+
           |  name| pos|   col|
           +------+-----------+
           |     A|  0 |   aaa|
           |     A|  1 |   bbb|
           |     A|  2 |  Null|
'''
```

Explode Nested Array into Rows

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, flatten

spark = SparkSession.builder.appName('pyspark-by-examples').getOrCreate()

arrayArrayData = [
  ("James",[["Java","Scala","C++"],["Spark","Java"]]),
  ("Michael",[["Spark","Java","C++"],["Spark","Java"]]),
  ("Robert",[["CSharp","VB"],["Spark","Python"]])
]

df = spark.createDataFrame(data=arrayArrayData, schema = ['name','subjects'])
# df.printSchema()
# df.show(truncate=False)
df2 = df.select(df.name, explode(flatten(df.subjects)))

# flatten function which converts array of array columns to a single array
df.select(df.name,flatten(df.subjects)).show(truncate=False)
'''
output: +-------+------------------------------+
        |name   |flatten(subjects)             |
        +-------+------------------------------+
        |James  |[Java, Scala, C++, Spark, Java]|
        |Michael|[Spark, Java, C++, Spark, Java]|
        |Robert |[CSharp, VB, Spark, Python]    |
        +-------+------------------------------+
'''
```

Groupby Function of DataFrame in Pyspark

1.Syntax & Method

```
def groupby(col1:str, *) --> GroupedData:
    pass

# aggregate functions of GroupedData object
# count()
# mean()
# max()
# min()
# sum()
# avg()
# agg()
# pivot()
```

```
# find the sum of salary for each department
df.groupBy("department").sum("salary").show(truncate=False)
```

2.GroupBy and Aggregate on Multiple Columns

```
df.groupBy("ColumnA","ColumnB").sum("ColumnC","ColumnD").show(False)
```

3.Running more aggregates at a time

```
df.groupBy("ColumnA").agg(
    sum("ColumnB").alias("sum_ColumnB"),
    avg("ColumnC").alias("avg_ColumnC")
    ).show()
```

4.Using filter on aggregate data

```
df.groupBy("ColumnA")\
    .agg(
        sum("ColumnB").alias("sum_ColumnB"),
        avg("ColumnC").alias("avg_ColumnC"))\
    .where(col('sum_ColumnB') >= 100)\
    .show()
# similar to HAVING in SQL
```

PySpark Aggregate Functions

```
import pyspark

simpleData = [("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)
  ]
schema = ["employee_name", "department", "salary"]
df = spark.createDataFrame(data=simpleData, schema = schema)
```

1.approx_count_distinct & countDistinct

```
# approx_count_distinct return count of distinct items in a group
df.select(approx_count_distinct('salary')).collect()[0][0]
# return type --> list[list[]] like

# countDistinct return the number of distinct items in a columns
df.select(countDistinct('salary')).collect()[0][0]
```

2.collect agg-function

```
# collect list
df.select(collect_list('salary'))

# collect set
df.select(collect_set('salary'))
```

3.other general agg-function

```
# avg
df.select(avg('salary')).collect()[0][0]

# first & last
# max & min
# mean
```

4.kurtosis & skewness

```
# kurtosis --> 峰度
# skewness --> 偏度
df.select(kurtosis('salary'))
```

5.stddev & stddev_samp & stddev_pop

```
# stddev = stddev_samp --> 样本标准差
# stddev_pop --> 总体标准差
df.select(stddev("salary"), stddev_samp("salary"), stddev_pop("salary"))

# also wo have avr-function sames as std
# variance(), var_samp(), var_pop()
```

Join Two DataFrames

```
# syntax : df1.join(df1, condition, join_type)
# eg :
df1.join(df2, df1('A') = df2('B'), 'inner')

df = df1.join(df2, ['id'], 'inner')
# this will work if id column is there in both df1 and df2.
```

1.Join Types

type	explain
inner	eq to INNER JOIN in SQL
outer, full, fullouter, full_outer	eq to FULL OUTER JOIN in SQL
left, leftouter, left_outer	eq to LEFT JOIN in SQL
right, rightouter, right_outer	eq to RIGHT JOIN in SQL
semi, leftsemi, left_semi	similar to LEFT JOIN but returns only columns in left dateset matching conditions
anti, leftanti, left_anti	similar to LEFT JOIN but returns only columns in left dateset not matching conditions

2.Using SQL Expression

```
df1.createOrReplaceTempView('temp1')
df2.createOrReplaceTempView('temp2')

joinDF = spark.sql("select * from temp1 e, temp2 d where e.id == d.id")
```

Pyspark RDD

Creating RDD

1.initialize SparkSession

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

```
        .appName("CreateSparkRDD")\
        .getOrCreate()
```

2.create RDD using sparkContext.parallelize()

```
data = [1, 2, 3, 4, 5, 6]
rdd = spark.sparkContext.parallelize(data)
```

3.create RDD using sparkContext.textFiles()

```
rdd = spark.sparkContext.textFiles("./path/textFile.txt")
'''
    key --> row number
    value --> row content
'''
```

4.create RDD using sparkContext.wholeTextFiles()

```
rdd = spark.sparkContext.wholeTextFiles("./path/textFile.txt")
'''
    key --> file path
    value --> file content
    can alse read data from other types of files.
'''
```

4.create empty RDD

```
# create empty RDD with no partition
rdd = spark.sparkContext.emptyRDD

# create empty RDD with partition
rdd = spark.sparkContext.parallelize([], 10)
# par 10 means creating 10 partition
```

RDD Transformations

Main transformations in pyspark:

- `flatMap()`
- `map()`
- `reduceByKey()`
- `filter()`

- sortByKey()

all above transformations return new RDD object but not execute until you call an action.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder()
      .master("local[1]")
      .appName("PySparkRDDTransformtions")
      .getOrCreate()
rdd = spark.sparkContext.textFile("/tmp/test.txt")
```

flatMap

```
rdd2 = rdd.flatMap(lambda x: x.split(" "))
'''
    key --> rdd1.key
    value --> array[str]
'''
```

map

```
rdd3 = rdd2.map(lambda x: (x, 1))
'''
    key --> x?
    value --> int?
'''
```

filter

```
rdd4 = rdd3.filter(lambda x: x[0].stratsWith('a'))
```

reduceByKey

```
rdd5 = rdd4.reduceByKey(lambda a, b : a + b)
# eq to agg-function sum()
```

sortByKey

```
rdd6 = rdd5.map(lambda x: (x[1], x[0])
'''
    key --> int
```

```
    value --> str
'''
rdd6.sortByKey()
print(rdd6.collect())
```

RDD Ations

```
# count() --> eq to Series.count()/len(df)
rdd.count()
# first()/max()/reduce()/top()
# take() --> siamilar to pd.head()
# fodl() --> agg the elements of all the partitions
# foreach() --> apply iter-function to each value
```