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.

| 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) |
| --- |

On the above table, record with employer name James 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.

## **Get Distinct Rows (By Comparing All Columns)**

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

dropDuplicates() function which returns a new DataFrame after removing duplicate rows.

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

## **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.

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

# **orderBy() and sort()**

ou can use either sort() or orderBy() function of PySpark DataFrame to sort DataFrame by ascending or descending order based on single or 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) \  ]  columns= ["employee\_name","department","state","salary","age","bonus"]  df = spark.createDataFrame(data = simpleData, schema = columns)  df.printSchema()  df.show(truncate=False) |
| --- |

## **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.

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

## **sorting using orderBy() function**

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

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

## **Sort by Ascending (ASC)**

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)

## **Sort by Descending (DESC)**

| 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) |
| --- |

## **Using Raw SQL**

| df.createOrReplaceTempView("EMP")  spark.sql("select employee\_name,department,state,salary,age,bonus from EMP ORDER BY department asc").show(truncate=False) |
| --- |

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

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.

| 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) |
| --- |

## **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) |
| --- |

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

| df.groupBy("department").count().show()  df.groupBy("department").min("salary").show()  df.groupBy("department").max("salary").show()  df.groupBy("department").avg( "salary").show()  df.groupBy("department").mean( "salary") .show() |
| --- |

## **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() |
| --- |

agg() - Using agg() function, we can calculate more than one aggregate at a time.

| 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) |
| --- |

## **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.

**Join** is used to combine two DataFrames and by chaining these you can join multiple DataFrames

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) |
| --- |

## **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) |
| --- |

## **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.

## **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(truncate=False)  empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id,"leftouter  ") \  .show(truncate=False) |
| --- |

## **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) |
| --- |