**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 **SQL** background, **both** these **functions** **operate** **exactly** the **same**.

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType, StructField, StringType, ArrayType

from pyspark.sql.functions import col

if \_\_name\_\_ == '\_\_main\_\_':

spark = SparkSession.builder \

.master("local[\*]") \

.appName("where filter Examples") \

.getOrCreate()

arrayStructureData = [

(("Saif", "", "Shaikh"), ["Java", "Scala", "C++"], "OH", "M"),

(("Ram", "Sachin", ""), ["Spark", "Java", "C++"], "NY", "F"),

(("Aniket", "", "Mishra"), ["CSharp", "VB"], "OH", "F"),

(("Mitali", "Sahil", "Kashiv"), ["CSharp", "VB"], "NY", "M"),

(("Zaid", "Riyaz", "Shaikh"), ["CSharp", "VB"], "NY", "M"),

(("Sufi", "Alim", "Shaikh"), ["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)

**1) 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**

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

**#OR**

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

**2) 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.

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

**3) 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 **&** condition, you can **extend** this with **OR (|)**, and **NOT (!)** conditional expressions as needed.

**#multiple condition:**

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

**#OR**

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

**4) 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 **array\_contains ( )** from Pyspark SQL functions which **checks** if a **value** **contains** in an **array** if **present** it **returns** **true** otherwise **false**.

**#Array Column Filter:**

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

**5) 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(col("name.lastname") == "Shaikh").show(truncate=False)

df.filter(df.name.lastname == "Shaikh").show(truncate=False)

**PySpark orderby ( ), sort ( ) & groupBy ( ):**

* You can **use** either **sort ( )** or **orderby ( )** **functions** 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**

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, sum, avg, max, round

if \_\_name\_\_ == '\_\_main\_\_':

spark = SparkSession.builder \

.master("local[3]") \

.appName("orderBy-sortBy-groupBy Examples") \

.getOrCreate()

orderData = [("Saif", "Sales", "NY", 90000, 34, 10000),

("Ram", "Sales", "NY", 86000, 56, 20000),

("Aniket", "Sales", "CA", 81000, 30, 23000),

("Saima", "Finance", "CA", 90000, 24, 23000),

("Sufiyan", "Finance", "CA", 99000, 40, 24000),

("Alim", "Finance", "NY", 83000, 36, 19000),

("Mitali", "Finance", "NY", 79000, 53, 15000),

("Neha", "Marketing", "CA", 80000, 25, 18000),

("Kajal", "Marketing", "NY", 91000, 50, 21000)]

columns = ["employee\_name", "department", "state", "salary", "age", "bonus"]

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

df.printSchema()

df.show(truncate=False)

**1) DataFrame sorting using the sort ( ) & orderby ( ) 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)

**Note:**

* The above **two** examples **return** the **same** 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**.

**2) DataFrame 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)

**3) Sort by Ascending (ASC):**

If you want to specify the **ascending order/sort explicitly** on **DataFrame**, you can use the **asc** **method** of the **Column** function.

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)

**4) Sort by Descending (DESC):**

If you want to specify the **sorting by descending order** on **DataFrame**, you can use the **desc** **method** of the **Column** function. for example.

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)

**2) groupBy ( ):**

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**.

**1) 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)

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

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

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

🡺 **min, max, avg**

**3) PySpark groupBy and aggregate on multiple columns:**

Similarly, we can also run **groupBy** and **aggregate** on **two** or **more** **DataFrame** **columns**.

Let’s group by on **department**, **state** and do **sum ( )** on **salary** and **bonus** columns.

**#GroupBy on multiple columns:**

df.groupBy("department", "state") \

.sum("salary", "bonus") \

.show(truncate=False)

**4) 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 ( )** etc. In **order** to **use** these, we should **import** from pyspark.sql.functions

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)

**5) 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)

**withColumn:**

PySpark **withColumn ( )** is a **transformation** **function** of **DataFrame** which is **used** to **change** or **update** the **value**, **convert** the **datatype** of an **existing** **DataFrame** **column**, **add**/**create** a **new** **column**.

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, lit

if \_\_name\_\_ == '\_\_main\_\_':

spark = SparkSession.builder \

.master("local[\*]") \

.appName("withColumn") \

.getOrCreate

data = [('Saif', '', 'Shaikh', '1991-04-01', 'M', 3000),

('Ram', 'Sachin', '', '2000-05-19', 'M', 4000),

('Aniket', '', 'Mishra', '1978-09-05', 'M', 4000),

('Mitali', 'Sahil', 'Kashiv', '1967-12-01', 'F', 4000),

('Nahid', 'Alim', 'Shaikh', '1980-02-17', 'F', -1)]

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

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

**1) Change column 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 ( )**.
* Below statement **changes** the **datatype** from **String** to **Integer** for “**salary**” **column**.

df2 = df.withColumn("salary", col("salary").cast("Integer"))

df2.printSchema()

**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 **value** to be **assigned** as a **second** **argument** to the **withColumn ( )** function.

**Note:** The **second** **argument** **should** be of **Column** **type**.

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

df3.show(5)

**3) Create a new 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.

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

df4.show()

**4) Add a new column using withColumn ( ):**

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

**# Add one column:**

df5 = df.withColumn("Country", lit("IND"))

df5.show()

**# Add multiple columns:**

df6 = df.withColumn("Country", lit("IND")) \

.withColumn("State", lit("MH"))

df6.show()

**5) Rename column name:**

* Though you **cannot** **rename** a **column** using **withColumn**, still **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 a column from PySpark DataFrame:**

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

df6.drop("anotherColumn").show(truncate=False)

**withColumnRenamed:**

* Use PySpark **withColumnRenamed ( )** to **rename** a **DataFrame** **column.**
* We often **need** to **rename** **one** **column** or **multiple** **columns** on PySpark **DataFrame**.

from pyspark.sql import SparkSession

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

from pyspark.sql.functions import col

if \_\_name\_\_ == '\_\_main\_\_':

spark = SparkSession.builder \

.master("local[3]") \

.appName("withColumnRenamed Examples") \

.getOrCreate()

dataDF = [(('Saif', '', 'Shaikh'), '1991-04-01', 'M', 3000),

(('Ram', 'Sachin', ''), '2000-05-19', 'M', 4000),

(('Aniket', '', 'Mishra'), '1978-09-05', 'M', 4000),

(('Mitali', 'Sahil', 'Kashiv'), '1967-12-01', 'F', 4000),

(('Nahid', 'Alim', 'Shaikh'), '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()

**1) To rename single column name:**

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

withColumnRenamed(existingName, newName)

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

**2) To rename multiple columns:**

To change **multiple** **column** **names**, we should **chain** **withColumnRenamed** **function**.

df2 = df.withColumnRenamed("dob", "DateOfBirth") \

.withColumnRenamed("salary", "salary\_amount")

df2.printSchema()

**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**.

schema2 = StructType([

StructField("fname", StringType()),

StructField("middlename", StringType()),

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

df.select(col("name").cast(schema2),

col("dob"),

col("gender"),

col("salary")) \

.printSchema()

**Note:**

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

**#Select struct columns:**

df4 = df.withColumn("fname", col("name.firstname")) \

.withColumn("mname", col("name.middlename")) \

.withColumn("lname", col("name.lastname"))

df4.printSchema()

**#Then drop the existing nested structure:**

df5 = df.withColumn("fname", col("name.firstname")) \

.withColumn("mname", col("name.middlename")) \

.withColumn("lname", col("name.lastname")) \

.drop("name")

df5.printSchema()

**4) Using Select to rename nested elements:**

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

df.select(col("name.firstname").alias("fname"),

col("name.middlename").alias("mname"),

col("name.lastname").alias("lname"),

col("dob"), col("gender"), col("salary")) \

.printSchema()

**PySpark Union and UnionAll:**

PySpark **union ( )** and **unionAll ( )** transformations are **used** to **merge** **two** or **more** **DataFrame’s** of the **same** **schema** or **structure**.

* **union ( ) method** of the **DataFrame** is used to **merge** **two** **DF** of the **same** **structure**/**schema**. If **schemas** are **not** the **same** it **returns** an **error**.
* **unionAll ( )** is **deprecated** since **Spark “2.0.0” version** and **replaced** with **union ( )**.

**Note:** In **SQL** languages, **Union** **eliminates** the **duplicates** but **UnionAll** **merges** **two** **datasets** including **duplicate** records.

But, in PySpark **both** **behave** the **same** and **recommend** using DF **duplicate ( )** function to **remove** **duplicate** rows.

from pyspark.sql import SparkSession

if \_\_name\_\_ == '\_\_main\_\_':

spark = SparkSession.builder \

.master("local[3]") \

.appName("union-unionAll Examples") \

.getOrCreate()

simpleData1 = [("Saif", "Sales", "NY", 90000, 34, 10000),

("Aniket", "Sales", "NY", 86000, 56, 20000),

("Ram", "Sales", "CA", 81000, 30, 23000),

("Mitali", "Finance", "CA", 90000, 24, 23000)]

columns = ["employee\_name", "department", "state", "salary", "age", "bonus"]

df = spark.createDataFrame(data=simpleData1, schema=columns)

df.printSchema()

df.show(truncate=False)

**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 = [("Saif", "Sales", "NY", 90000, 34, 10000),

("Mitali", "Finance", "CA", 90000, 24, 23000),

("Sufiyan", "Finance", "NY", 79000, 53, 15000),

("Alim", "Marketing", "CA", 80000, 25, 18000),

("Amit", "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)

**1) Merge two or more DataFrames using union:**

DF **union ( )** method **merges** **two** **DF** and **returns** the **new** **DF** with **all** **rows** from **two** **DF** **regardless** of **duplicate** data.

unionDF = df.union(df2)

unionDF.show(truncate=False)

**2) Merge DataFrames using unionAll:**

DF **unionAll ( )** method is **deprecated** since **Spark “2.0.0” version** and **recommends** **using** the **union ( )** method.

unionAllDF = df.unionAll(df2)

unionAllDF.show(truncate=False)

**Note**: **Returns** the **same** **output** as **above**.

**3) 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)

**drop – dropDuplicates:**

PySpark **distinct ( )** function is **used** to **drop** the **duplicate** **rows** (**all columns**) from **DF** and **dropDuplicates ( )** is **used** to **drop** selected (**one or multiple**) columns.

from pyspark.sql import SparkSession

from pyspark.sql.functions import expr

if \_\_name\_\_ == '\_\_main\_\_':

spark = SparkSession.builder\

.appName("drop/dropDuplicates Examples")\

.getOrCreate()

data = [("Saif", "Sales", 3000),

("Mitali", "Sales", 4600),

("Manas", "Sales", 4100),

("Kajal", "Finance", 3000),

("Neha", "Sales", 3000),

("Ram", "Finance", 3300),

("Aniket", "Finance", 3900),

("Shravan", "Marketing", 3000),

("Pramod", "Marketing", 2000),

("Vivek", "Sales", 4100)]

columns= ["employee\_name", "department", "salary"]

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

df.printSchema()

df.show(truncate=False)

**1) Get distinct all columns:**

Above **DF**, we have a **total** of **10 rows** with **2 rows** having **all values duplicated**, **performing** **distinct** on this **DF** should get us **9 rows**.

distinctDF = df.distinct()

print("Distinct count: " + str(distinctDF.count()))

distinctDF.show(truncate=False)

Alternatively, you can also run **dropDuplicates ( )** function which **return** a **new** DF with **duplicate** **rows** **removed**.

df2 = df.dropDuplicates()

print("Distinct count: "+str(df2.count()))

df2.show(truncate=False)

**2) PySpark Distinct of multiple columns:**

* PySpark **doesn’t** have a **distinct** **method** which **takes** **columns** that should **run** **distinct** on (**drop duplicate rows on selected columns**)
* However, it **provides** **another** **signature** of **dropDuplicates ( )** function which takes **multiple** **columns** to **eliminate** **duplicates**.

**Note:** Calling **dropDuplicates ( )** on DF **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)

**3) Drop columns:**

**#Single Column:**

drop\_scol = df.drop("salary")

drop\_scol.show(truncate=False)

**#Multiple Columns:**

drop\_mcol = df.drop("department", "salary")

drop\_mcol.show(truncate=False)

**case – when – others:**

* In PySpark DataFrame, “**when otherwise**” is used **derive** a **column** or **update** an **existing** **column** based on **some** **conditions** from **existing** **columns** data.
* **when ( ) is a SQL function** with a **return** type **Column** and **other ( )** is a **function** in **sql**.**Column** **class**.

Like **SQL** "**case when**" statement and “Switch", "**if then else**" statement also supports similar syntax using “**when otherwise**” or using “**case when**” statement.

from pyspark.sql import SparkSession

if \_\_name\_\_ == '\_\_main\_\_':

spark = SparkSession.builder \

.master("local[3]") \

.appName("case-when-others") \

.getOrCreate()

data = [("Saif","","Shaikh","36636","M",60000),

("Ram","Shirali","","40288","M",70000),

("Aniket","","Mishra","42114","",400000),

("Mitali","Sahil","Kashiv","39192","F",500000),

("Nahid","Alim","Shaikh","","F",0)]

columns = ["first\_name","middle\_name","last\_name","dob","gender","salary"]

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

df.show()

**1) Using “when otherwise” on PySpark DataFrame:**

**when( )** is a PySpark **SQL** **function**, so to **use** it **first** we should **import**

**from pyspark.sql.functions import when**.

Let’s **replace** the **value** of **gender** with new **derived** **value**, **when** **value** **not** **qualified** with the **condition**, we are **assigning** “**Unknown**” as **value**.

df2 = df.withColumn("new\_gender",

when(col("gender") == "M", "Male")

.when(col("gender") == "F", "Female")

.otherwise("Unknown"))

df2.show(truncate=False)

**#OR**

**SQL Like Syntax:**

df22 = df.select(col("\*"), when(col("gender") == "M", "Male")

.when(col("gender") == "F", "Female")

.otherwise("Unknown").alias("new\_gender"))

df2.show(truncate=False)

**2) Using “case when” on PySpark DataFrame:**

Similarly, we could use “**case when**” with expression **expr ( )** and **withColumn ( )**.

from pyspark.sql.functions import expr

exprDf1 = df.withColumn("new\_gender",

expr("case when gender = 'M' then 'Male' when gender = 'F' then 'Female' else 'Unknown' end"))

**#OR**

exprDf1 = df.withColumn("new\_gender", expr("case when gender = 'M' then 'Male' " +

"when gender = 'F' then 'Female' " +

"else 'Unknown' end"))

exprDf1.show(truncate=False)

**#case with select:**

df4 = df.select(col("\*"), expr("case when gender = 'M' then 'Male' " +

"when gender = 'F' then 'Female' " +

"else 'Unknown' end").alias("new\_gender"))

df4.show(truncate=False)

**3) Using & and | operator:**

We can also use **and (&) or (|)** within **when** function. Let’s create a **new set of data** to **make** it **simple**.

orAndDf = df.withColumn("New\_Gender",

when((col("gender") == "M" ) | (col("gender") == "F"), "Available")

.otherwise("Not Available"))

orAndDf.show()

**String Functions:**

**pyspark.sql.functions** provides **two** functions **concat ( )** and **concat\_ws ( )** to **concatenate** DF **multiple** columns into a **single** column.

from pyspark.sql import SparkSession

if \_\_name\_\_ == '\_\_main\_\_':

spark = SparkSession.builder \

.master("local[3]") \

.appName("functions Examples") \

.getOrCreate()

data = [("Saif" ,"M" ,"Shaikh" ,"2018" ,"M" ,3000),

("Ram" ,"S" ,"Shirali" ,"2010" ,"M" ,4000),

("Mitali" ,"S" ,"Kashiv" ,"2010" ,"M" ,4000),

("Anup" ,"B" ,"Garje" ,"2005" ,"F" ,4000),

("Sagar" ,"S" ,"Shinde" ,"2010" ,"" ,-1)]

columns = ["fname" ,"mname" ,"lname" ,"dob\_year" ,"gender" ,"salary"]

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

df.show(truncate=False)

**1) PySpark concatenate using concat ( ):**

* **concat ( )** function of **Pyspark SQL** is used to **concatenate** **multiple** **DF** **columns** into a **single** **column**.

**Syntax:** pyspark.sql.functions.concat(\*cols)

df2=df.select(concat("fname", lit (","), "mname", lit (","), "lname")

.alias("FullName"),"dob\_year","gender","salary")

df2.show(truncate=False)

**2) PySpark concat\_ws ( ) Usage:**

**concat\_ws ( )** function of Pyspark **concatenates** **multiple** **string** **columns** into a **single** **column** with a given **separator** or **delimiter**.

**Syntax:** pyspark.sql.functions.concat\_ws(**sep**,\*cols)

df3 = df.select(concat\_ws('\_', col("fname"), col("fname"), col("lname"))

.alias("FullName"), "dob\_year", "gender", "salary")

df3.show(truncate=False)

**Date/Time Functions:**

In PySpark, you can do **almost** **all** the **date** **operations** you can think of using **in-built** **functions**.

**Create a dataframe with sample date values:**

df\_1 = spark.createDataFrame([('2021-01-15', '2021-02-15',)], ['start\_dt', 'end\_dt'])

df\_1.show()

df\_1.printSchema()

Now the **problem** I see here is that columns **start**\_dt & **end**\_dt are of **type** **string** and **not** **date**. So let’s **quickly** **convert** it into **date**.

df\_2 = df\_1.select(df\_1.start\_dt.cast('date'), df\_1.end\_dt.cast('date'))

df\_2.printSchema()

Now we are **good**. We have a **DF** with 2 columns **start\_dt** & **end\_dt**. Both the **columns** are of datatype ‘**date’**. Let’s do some Date operations on this.

**1) Change Date Format:**

df\_2.select("start\_dt", "end\_dt", date\_format("start\_dt", 'dd/MM/yyyy').alias("dt\_format")).show()

**2) Fetch Current Date:**

df\_2.select("start\_dt", "end\_dt", current\_date().alias("cur\_dt")).show()

**3) Add Days to date:**

df\_2.select("start\_dt", "end\_dt", date\_add("start\_dt", 2).alias("add\_2\_days")).show()

**4) Subtract days from date:**

df\_2.select("start\_dt", "end\_dt", date\_sub("start\_dt", 2).alias("add\_2\_days")).show()

**5) Subtract 2 dates:**

df\_2.select("start\_dt", "end\_dt", datediff("end\_dt", "start\_dt").alias("sub\_2\_dates")).show()

**6) Add Months to date:**

df\_2.select("start\_dt", "end\_dt", add\_months("start\_dt", 2).alias("add\_2\_months")).show()

**7) Add Years to date:**

df\_2.select("start\_dt", "end\_dt", add\_months("start\_dt", 2 \* 12).alias("add\_2\_Yrs")).show()

**8) Extract Year, Month, Day, WeekofYear, DayofWeek, DayofYear from Date:**

df\_2.select("start\_dt", "end\_dt", year("start\_dt").alias("Year")

, month("start\_dt").alias("Month")

, dayofmonth("start\_dt").alias("Day")

, weekofyear("start\_dt").alias("Week\_of\_Year")

, dayofweek("start\_dt").alias("Day\_of\_Week")

, dayofyear("start\_dt").alias("Day\_of\_Year")).show()

**9) Last Day of Month:**

df\_2.select("start\_dt", "end\_dt", last\_day("start\_dt").alias("Last\_Day")).show()

**10) Determine how many months between 2 Dates:**

df\_2.select("start\_dt", "end\_dt",

months\_between("end\_dt", "start\_dt").alias("Months\_Betwn")).show()

**#Months between in whole no:**

df\_2.select("start\_dt", "end\_dt",

round(months\_between("end\_dt", "start\_dt")).cast('int').alias("Months\_Betwn")).show()

**11) Identify Next Day:**

**Monday:**

df\_2.select("start\_dt", "end\_dt", next\_day("start\_dt", "Mon").alias("Next\_Monday")).show()

**Tuesday:**

df\_2.select("start\_dt", "end\_dt", next\_day("start\_dt", "Tue").alias("Next\_Tuesday")).show()

**11) Fetch quarter of the year:**

df\_2.select("start\_dt", "end\_dt", quarter("start\_dt").alias("Quarter\_of\_Year")).show()

**12) Truncate Date to Year, Month:**

df\_2.select("start\_dt", "end\_dt", trunc("start\_dt", "year").alias("Trunc\_Year"),

trunc("end\_dt", "month").alias("Trunc\_Month")).show()

**Aggregate Functions:**

* PySpark provides **built-in** standard **Aggregate** **functions** defined in **DF.**
* **Aggregate** functions **operate** on a **group** of **rows** and **calculate** a **single** **return** **value** for **every** **group**.
* All these **aggregate** **functions** accept input as **column** **type** or **column** **name** in a **string** and **several** **other** **arguments** based on the **function** and **return** column type.
* **Aggregate** **functions** are little bit more **compile-time safety**, **handles null** and **perform** **better** when **compared** to **UDF’s**.
* If your **application** is **critical** on **performance** try to **avoid** using **custom** **UDF**.
* UDF’s **does not** **guarantee** on **performance**.

simpleData = [("Saif", "Sales", 3000),

("Ram", "Sales", 4600),

("Aniket", "Sales", 4100),

("Mitali", "Finance", 3000),

("Saif", "Sales", 3000),

("Sandeep", "Finance", 3300),

("John", "Finance", 3900),

("Jeff", "Marketing", 3000),

("Sagar", "Marketing", 2000),

("Swaroop", "Sales", 4100)]

schema = ["employee\_name", "department", "salary"]

agg\_df = spark.createDataFrame(data=simpleData, schema=schema)

agg\_df.printSchema()

agg\_df.show(truncate=False)

**1) approx\_count\_distinct:**

**approx\_count\_distinct ( )** function **returns** the **count** of **distinct** **items** in a **group**.

approxDistinctCount = agg\_df.select(approx\_count\_distinct("salary"))

approxDistinctCount.show()

**#OR:**

print("approx\_count\_distinct: " + str(agg\_df.select(approx\_count\_distinct("salary")).collect()[0][0]))

**2) avg (average):**

**avg( )** function returns the average of values in the input column.

avgSal = agg\_df.select(avg("salary")) 🡪 sum, min, max

avgSal.show()

**#OR:**

print("avg: " + str(agg\_df.select(avg("salary")).collect()[0][0]))

**3) collect\_list:**

**collect\_list ( )** function **returns** **all** **values** from an **input** **column** with **duplicates**.

agg\_df.select(collect\_list("salary")).show(truncate=False)

**4) collect\_set:**

**collect\_set ( )** function **returns** **all** **values** from an **input** **column** with **NO** **duplicate** **values**.

agg\_df.select(collect\_set("salary")).show(truncate=False)

**5) countDistinct:**

**countDistinct ( )** function **returns** the **number** of **distinct** **elements** in a **columns**.

df2 = agg\_df.select(countDistinct("department", "salary"))

df2.show(truncate=False)

**#OR:**

print("Distinct Count of Department & Salary: " + str(df2.collect()[0][0]))

**6) count function:**

**count ( )** function **returns** **number** of **elements** in a **column**.

cnt = agg\_df.count()

print(cnt)

**# OR:**

print("count: " + str(agg\_df.select(count("salary")).collect()[0]))

**7) first/last function:**

first() function returns the first/last element in a column when ignoreNulls is set to true, it returns the first non-null element.

agg\_df.select(first("salary")).show(truncate=False)

agg\_df.select(last("salary")).show(truncate=False)

**9) sumDistinct function:**

**sumDistinct ( )** function **returns** the **sum** of **all** **distinct** **values** in a **column**.

agg\_df.select(sumDistinct("salary")).show(truncate=False)

**Window Functions:**

* PySpark **Window** functions **operate** on a **group** of **rows** (**like frame, partition**) and **return** a **single** **value** for **every** **input** **row**.
* To **perform** an **operation** on a **group** **first** we **need** to **partition** the **data** **using** **Window.partitionBy ( )**
* For **row** **number** and **rank** **function** we **need** to **additionally** **order** **by** on **partition** **data** using **orderBy** clause.

**1)** ranking functions

**2)** analytic functions

**3)** aggregate functions

from pyspark.sql.window import Window

from pyspark.sql.functions import row\_number

simpleData = [("Saif", "Sales", 3000),

("Ram", "Sales", 4600),

("Aniket", "Sales", 4100),

("Mitali", "Finance", 3000),

("Saif", "Sales", 3000),

("Sandeep", "Finance", 3300),

("John", "Finance", 3900),

("Jeff", "Marketing", 3000),

("Sagar", "Marketing", 2000),

("Swaroop", "Sales", 4100)]

columns = ["employee\_name", "department", "salary"]

win\_df = spark.createDataFrame(data=simpleData, schema=columns)

win\_df.printSchema()

win\_df.show(truncate=False)

windowSpec = Window.partitionBy("department").orderBy("salary")

**1) row\_number:**

**row\_number ( )** window function is **used** to **give** the **sequential** **row** **number** starting **from 1** to the **result** of each **window** **partition**.

win\_df.withColumn("row\_number", row\_number().over(windowSpec))

.show(truncate=False)

**2) rank:**

**rank( )** window function is **used** to **provide** a **rank** to the **result** **within** a **window** **partition**. This function **leaves** **gaps** in **rank** when **there** are **ties**.

win\_df.withColumn("rank", rank().over(windowSpec)).show()

**3) dense\_rank:**

* **dense\_rank ( )** window function is **used** to get the **result** with **rank** of **rows** **within** a **window** **partition** **without** **any** **gaps**.
* This is similar to **rank ( )** function **difference** being **rank** function **leaves** **gaps** in rank **when** **there** are **ties** but **dense\_rank** **does not leave gaps**.

win\_df.withColumn("dense\_rank", dense\_rank().over(windowSpec)).show()

**4) ntile:**

**ntile ( )** window function **returns** the **relative** **rank** of **result** **rows** **within** a **window** **partition**.

If we **provide 2** as an **argument** to **ntile** it **returns** **ranking** **between 2 values** (1 and 2).

win\_df.withColumn("ntile", ntile(2).over(windowSpec)).show()

**5) lag:**

This is the same as the **LAG** function in **SQL**.

win\_df.withColumn("lag", lag("salary", 2).over(windowSpec)).show()

**6) lead:**

This is the same as the **LEAD** function in **SQL**.

win\_df.withColumn("lead", lead("salary", 2).over(windowSpec)).show()

**7) Window Aggregate Functions:**

* Let’s see how to **calculate** **sum**, **min**, **max** for **each** **department** using **PySpark SQL** **Aggregate** **window** **functions** and **WindowSpec**.
* When **working** with **Aggregate** **functions** we **don’t need** to **use order by** clause.

windowSpecAgg = Window.partitionBy("department")

win\_df.withColumn("row", row\_number().over(windowSpec)) \

.withColumn("avg", avg(col("salary")).over(windowSpecAgg)) \

.withColumn("sum", sum(col("salary")).over(windowSpecAgg)) \

.withColumn("min", min(col("salary")).over(windowSpecAgg)) \

.withColumn("max", max(col("salary")).over(windowSpecAgg)) \

.where(col("row") == 1).select("department", "avg", "sum", "min", "max") \

.show()

**8) Explode Function:**

* PySpark **explodes** **array** and **map** **columns** to **rows**.
* **PySpark function explode (e: Column)** is **used** to **explode** or **create** **array** or **map** **columns** to **rows**.
* When an **array** is **passed** to this **function**, it **creates** a **new** **default** **column** “**col1**” and it **contains all array** elements.
* When a **map** is **passed**, it **creates** **two** **new** **columns** **one for key** and **one** for **value** and **each** **element** in **map** **split** **into** the **rows**.

explodeData = [('Saif', ['Java', 'Scala'], {'hair': 'black', 'eye': 'brown'}),

('Mitali', ['Spark', 'Java', None], {'hair': 'brown', 'eye': None}),

('Ram', ['CSharp', ''], {'hair': 'red', 'eye': ''}),

('Wilma', None, None),

('Jatin', ['1', '2'], {})]

array\_df = spark.createDataFrame(data=explodeData,

schema=['name', 'knownLanguages', 'properties'])

array\_df.printSchema()

array\_df.show(truncate=False)

**Note:**

This will **ignore** **elements** that have **null** or **empty**.

**Wilma** and **Jatin** have **null** or **empty** **values** in **array** and **map** hence the following snippet **does not contain** these **rows**.

**1) explode array:**

df2 = array\_df.select("name", explode("knownLanguages"))

df2.printSchema()

df2.show()

**2) explode map:**

df3 = array\_df.select("name", explode("properties"))

df3.printSchema()

df3.show()

**Joins:**

* **Joins** is **used** to **combine** **two** **DF**.
* It **supports** **all** basic **join** **operations** **available** in **traditional** **SQL**.
* **Spark Joins** are **wider** **transformations** that **involve** **data** **shuffling** **across** the **network**.
* **Spark SQL Joins** comes with **more** **optimization** by **default** (thanks to DataFrames).

**Syntax:** join (self, other, on=None, how=None)

* **join** operation **takes** **parameters** as **below** and **returns** **DataFrame**.
* **other** **Right** **side** of the **join**
* **on** a **string** for the **join** **column** **name**
* **how** **default** **inner**.

**Must** be **one** of **inner**, **cross**, **outer**, **full**, **full\_outer**, **left**, **left\_outer**, **right**, **right\_outer**, **left\_semi** and **left\_anti**.

from pyspark.sql import SparkSession

if \_\_name\_\_ == '\_\_main\_\_':

spark = SparkSession.builder \

.master("local[3]") \

.appName("joins") \

.getOrCreate()

emp = [(1, "Saif", -1, "2018", "10", "M", 3000),

(2, "Ram", 1, "2010", "20", "M", 4000),

(3, "Aniket", 1, "2010", "10", "M", 1000),

(4, "Mitali", 2, "2005", "10", "F", 2000),

(5, "Nahid", 2, "2010", "40", "", -1),

(6, "Sufiyan", 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)

**1) Inner Join:**

* **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)

**Note:** When we apply **Inner join** on our **datasets**, it **drops** “**emp\_dept\_id**” **60** from “**emp**” and “**dept\_id**” **30** from “**dept**” **datasets**.

**2) 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)

**#OR:**

empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "full") \

.show(truncate=False)

**#OR:**

empDF.join(deptDF, empDF.emp\_dept\_id == deptDF.dept\_id, "fullouter") \

.show(truncate=False)

**Note:** From our “**emp**” dataset’s “**emp\_dept\_id**” with **value 60 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**.

**3) 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(False)

**Note:** From our dataset, “**emp\_dept\_id**” **60** **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**.

**4) 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 **match** **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)

**Note:** 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**” **60 dropped** as a **match not found** on **left**.

**4) 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)

**5) 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)

**6) 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** **DF** to **itself**.

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)

**Note:** Here, we are joining **emp** dataset **with itself** to find out **superior emp\_id** and **name** for **all employees**.

**7) Using SQL Expression:**

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

empDF.createOrReplaceTempView("EMP")

deptDF.createOrReplaceTempView("DEPT")

print("\*\*\*\*\*\*\*\*\*\*Inner ANSI-89\*\*\*\*\*\*\*\*\*\*")

joinDF1 = spark.sql("select \* from EMP e, DEPT d where e.emp\_dept\_id == d.dept\_id") \

.show(truncate=False)

print("\*\*\*\*\*\*\*\*\*\*Inner ANSI-92\*\*\*\*\*\*\*\*\*\*")

joinDF2 = spark.sql("select \* from EMP e INNER JOIN DEPT d

ON e.emp\_dept\_id == d.dept\_id") \

.show(truncate=False)

**8) PySpark SQL Join on multiple DataFrame’s:**

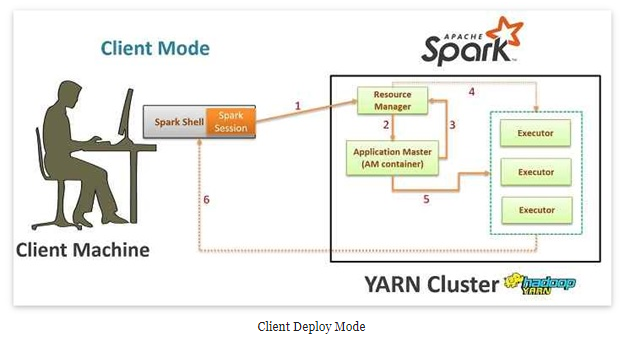
When you **need to join** more than **two tables** you either use **SQL expression** after **creating** a **temporary view** on the **DF** or **use** the **result** of **join operation** to **join** with **another** **DF** like **chaining** **them**.s

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

.join(df3,df1.id1 == df3.id3,"inner")

**1) YARN in Client Mode:**

* In Client mode, driver starts on the local machine and as soon as the driver creates a SparkSession a request goes to YARN Resource Manager to create a YARN application.
* The YARN Resource Manager starts an Application Master.
* For client mode AM acts as an Executor Launcher.
* So the YARN Application Master will reach out to YARN Resource Manager and request for further containers.
* The Resource Manager will allocate new containers and then Application Master starts executor in each container.
* After initial setup these executors directly communicate with the driver.



**2) YARN in Cluster Mode:**

* In Cluster mode, spark-submit utility will send a YARN Application Master request to the YARN Resource Manager.
* The YARN Resource Manager starts an Application Master and then the driver starts in the Application Master Container. That's where the client mode and cluster mode differs.
* In client mode, YARN AM acts as an executor launcher and the driver resides on your local machine but in cluster mode the YARN AM starts the driver and you don't have any dependency on your local computer.
* Once started, the driver will reach out to Resource Manager with a request for Containers.
* The Resource Manager will allocate new Containers and the driver starts an executor in each Container.
* After initial setup these executors directly communicate with the driver.

