databricks Pyspark practice

(https://databricks.com)

Union and UnionByName transformation

Concept :-
union works when the columns of both DataFrames being joined are **in** the same order. It can give surprisingly wrong results when the schemas aren't the same.

unionByName works when both DataFrames have the same columns, but **in** a different order.

union **and** unionByName transformation are used to merge two **or** more dataframes of the same schema **and** structure.

unionByName() functions takes param allowMissingColumns **with** the value **True**

Syntax of unionByName()

unionByName(df, allowMissingColumns = **True**)

Union() function **in** pyspark
The PySpark union() function **is** used to combine two **or** more data frames having the same structure **or** schema. This function returns an error **if** the schema of data frames differs **from** each other.

Syntax: data_frame1.union(data_frame2)

Where,

data_frame1 **and** data_frame2 are the dataframes.

Example 1

```
americans = spark.createDataFrame(
    [("bob", 42), ("lisa", 59)], ["first_name", "age"]
)
colombians = spark.createDataFrame(
    [("maria", 20), ("camilo", 31)], ["first_name", "age"]
)
res = americans.union(colombians)
res.show()
```

```
+----------+---+
|first_name|age|
+----------+---+
|       bob| 42|
|      lisa| 59|
|     maria| 20|
|    camilo| 31|
+----------+---+
```

```
details = [(1, 'Krishna', 'IT', 'male')]
column = ['id', 'name', 'department', 'gender']
details1 = [(1, 'Krishna', 'IT', 10000)]
column = ['id', 'name', 'department', 'salary']
df1 = spark.createDataFrame(details, column)
df2 = spark.createDataFrame(details1, column)
df1.show()
df2.show()
```

```
+---+-------+----------+------+
| id|   name|department|salary|
+---+-------+----------+------+
|  1|Krishna|        IT|  male|
+---+-------+----------+------+

+---+-------+----------+------+
| id|   name|department|salary|
+---+-------+----------+------+
|  1|Krishna|        IT| 10000|
+---+-------+----------+------+
```

```
df1.union(df2).show()
```

```
+---+-------+----------+------+
| id|   name|department|salary|
+---+-------+----------+------+
|  1|Krishna|        IT|  male|
|  1|Krishna|        IT| 10000|
+---+-------+----------+------+
```

```
df1.unionByName(df2, allowMissingColumns=True).show()

# if columns will miss then also it will union.  like here was a difference gender and
salary.
```

```
+---+-------+----------+------+
| id|   name|department|salary|
+---+-------+----------+------+
|  1|Krishna|        IT|  male|
|  1|Krishna|        IT| 10000|
+---+-------+----------+------+
```

```
Remember - If number will mismatch(if different column name or different number of column)
then we will use unionByName with True.
```

Example 2

```
# union

data_frame1 = spark.createDataFrame(
    [("Nitya", 82.98), ("Abhishek", 80.31)],
    ["Student Name", "Overall Percentage"]
)

# Creating another dataframe
data_frame2 = spark.createDataFrame(
    [("Sandeep", 91.123), ("Rakesh", 90.51)],
    ["Student Name", "Overall Percentage"]
)

# union()
UnionEXP = data_frame1.union(data_frame2)


UnionEXP.show()
```

```
+------------+------------------+
|Student Name|Overall Percentage|
+------------+------------------+
|       Nitya|             82.98|
|    Abhishek|             80.31|
|     Sandeep|            91.123|
|      Rakesh|             90.51|
+------------+------------------+
```

UnionByName() function in pyspark
The PySpark unionByName() function is also used to combine two or more data frames but it might be used to combine dataframes having different schema. This is because it combines data frames by the name of the column and not the order of the columns.

Syntax: data_frame1.unionByName(data_frame2)

Where, data_frame1 and data_frame2 are the dataframes

```
data_frame1 = spark.createDataFrame(
    [("Nitya", 82.98), ("Abhishek", 80.31)],
    ["Student Name", "Overall Percentage"]
)

# Creating another data frame
data_frame2 = spark.createDataFrame(
    [(91.123, "Naveen"), (90.51, "Sandeep"), (87.67, "Rakesh")],
    ["Overall Percentage", "Student Name"]
)

# Union both the dataframes using unionByName() method
byName = data_frame1.unionByName(data_frame2)


byName.show()
```

see in this example , data_frame1 and data_frame2 are of different schema but the output is the desired one.

```
+------------+------------------+
|Student Name|Overall Percentage|
+------------+------------------+
|       Nitya|             82.98|
|    Abhishek|             80.31|
|      Naveen|            91.123|
|     Sandeep|             90.51|
|      Rakesh|             87.67|
+------------+------------------+
```

```
data_frame1 = spark.createDataFrame(
    [("Bhuwanesh", 82.98, "Computer Science"), ("Harshit", 80.31, "Information Technology")],
["Student Name", "Overall Percentage", "Department"]
)

# Creating another dataframe
data_frame2 = spark.createDataFrame( [("Naveen", 91.123), ("Piyush", 90.51)], ["Student
Name", "Overall Percentage"] )

# Union both the dataframes using unionByName() method
column_name_morein1df = data_frame1.unionByName(data_frame2, allowMissingColumns=True)


column_name_morein1df.show()
```

in this example we have more columnname in 1st dataframe but less in 2nd dataframe.
but we are able to do our desired result with the help of unionByName().

```
+------------+------------------+--------------------+
|Student Name|Overall Percentage|          Department|
+------------+------------------+--------------------+
|   Bhuwanesh|             82.98|    Computer Science|
```

```
|     Harshit|             80.31|Information Techn...|
|      Naveen|            91.123|                null|
|      Piyush|             90.51|                null|
+------------+------------------+--------------------+
```

pySpark Window Ranking Functions

Definitions:

Window function:
pySpark window functions are useful when you want to examine relationships within group of data rather than between group of data. It performs statictical operations like below explained.

PySpark Window function performs statistical operations such as rank, row number, etc. on a group, frame, or collection of rows and returns results for each row individually. It is also popularly growing to perform data transformations.

There are mainly three types of Window function:

Analytical Function
Ranking Function
Aggregate Function

Analytical functions:

An analytic function is a function that returns a result after operating on data or a finite set of rows partitioned by a SELECT clause or in the ORDER BY clause. It returns a result in the same number of rows as the number of input rows. E.g. lead(), lag(), cume_dist().

```
from pyspark.sql.window import Window
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("pyspark_window").getOrCreate()
sampleData = (("Nitya", 28, "Sales", 3000),
              ("Abhishek", 33, "Sales", 4600),
              ("Sandeep", 40, "Sales", 4100),
              ("Rakesh", 25, "Finance", 3000),
              ("Ram", 28, "Sales", 3000),
              ("Srishti", 46, "Management", 3300),
              ("Arbind", 26, "Finance", 3900),
              ("Hitesh", 30, "Marketing", 3000),
              ("Kailash", 29, "Marketing", 2000),
              ("Sushma", 39, "Sales", 4100)
              )
# column names
columns = ["Employee_Name", "Age",
           "Department", "Salary"]

# creating the dataframe df
df = spark.createDataFrame(data=sampleData,schema=columns)
windowPartition = Window.partitionBy("Department").orderBy("Age")
df.printSchema()
df.show()
display(df)
```

```
root
 |-- Employee_Name: string (nullable = true)
 |-- Age: long (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)

+-------------+---+----------+------+
|Employee_Name|Age|Department|Salary|
+-------------+---+----------+------+
|        Nitya| 28|     Sales|  3000|
|     Abhishek| 33|     Sales|  4600|
|      Sandeep| 40|     Sales|  4100|
|       Rakesh| 25|   Finance|  3000|
|          Ram| 28|     Sales|  3000|
|      Srishti| 46|Management|  3300|
|       Arbind| 26|   Finance|  3900|
|       Hitesh| 30| Marketing|  3000|
|      Kailash| 29| Marketing|  2000|
|       Sushma| 39|     Sales|  4100|
+-------------+---+----------+------+
```

Table

	Employee_Name	Age	Department	Salary
1	Nitya	28	Sales	3000
2	Abhishek	33	Sales	4600
3	Sandeep	40	Sales	4100

4	Rakesh	25	Finance	3000
5	Ram	28	Sales	3000
6	Srishti	46	Management	3300
7	Arbind	26	Finance	3900
8	Hitesh	30	Marketing	3000
9	Kailash	29	Marketing	2000
10	Sushma	39	Sales	4100

10 rows

Using cume_dist():
cume_dist() window function is used to get the cumulative distribution within a window partition.

```
from pyspark.sql.functions import cume_dist
df.withColumn("cume_dist", cume_dist().over(windowPartition)).display()
```

Table

	Employee_Name	Age	Department	Salary	cume_dist
1	Rakesh	25	Finance	3000	0.5
2	Arbind	26	Finance	3900	1
3	Srishti	46	Management	3300	1
4	Kailash	29	Marketing	2000	0.5
5	Hitesh	30	Marketing	3000	1
6	Nitya	28	Sales	3000	0.4
7	Ram	28	Sales	3000	0.4

10 rows

Using lag()
A lag() function is used to access previous rows' data as per the defined offset value in the function.

```
from pyspark.sql.functions import lag
df.withColumn("Lag", lag("Salary", 2).over(windowPartition)) \
    .display()
```

Table

	Employee_Name	Age	Department	Salary	Lag
1	Rakesh	25	Finance	3000	null
2	Arbind	26	Finance	3900	null
3	Srishti	46	Management	3300	null
4	Kailash	29	Marketing	2000	null
5	Hitesh	30	Marketing	3000	null
6	Nitya	28	Sales	2000	null

6	Nitya	28	Sales	3000	null
7	Ram	28	Sales	3000	null

10 rows

Using lead()
A lead() function is used to access next rows data as per the defined offset value in the function.

```
from pyspark.sql.functions import lead
df.withColumn("Lead", lead("salary", 2).over(windowPartition)) \
    .display()
```

Table

	Employee_Name	Age	Department	Salary	Lead
1	Rakesh	25	Finance	3000	null
2	Arbind	26	Finance	3900	null
3	Srishti	46	Management	3300	null
4	Kailash	29	Marketing	2000	null
5	Hitesh	30	Marketing	3000	null
6	Nitya	28	Sales	3000	4600
7	Ram	28	Sales	3000	4100

10 rows

Ranking function

Ranking Function
The function returns the statistical rank of a given value for each row in a partition or group. The goal of this function is to provide consecutive numbering of the rows in the resultant column, set by the order selected in the Window.partition for each partition specified in the OVER clause. E.g. row_number(), rank(), dense_rank(), etc.

A. row_number() : row_number() window functions is used to give the sequential row number starting from 1 to the result of each
window partition.

B. rank():
rank window functions is used to provide a rank to the result within a window partition. this functions leaves gaps in rank when there are ties.
Example :- 1 1 1 4 this is rank

C. dense_rank(): dense_rank() window function is used to get the result with rank of rows within a window partition without a gaps.
This is similar to rank function difference being rank function leaves the gaps in rank when there are ties.
Example : - 1 1 1 2 this is dense rank.

Syntax for Window Funtions :

```
DataFrame.withColumn("col_name", Window_function().over(Window_partition))
```

```
from pyspark.sql.window import Window
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("pyspark_window").getOrCreate()
sampleData = ((101, "Ram", "Biology", 80),
              (103, "Sita", "Social Science", 78),
              (104, "Lakshman", "Sanskrit", 58),
              (102, "Kunal", "Phisycs", 89),
              (101, "Ram", "Biology", 80),
              (106, "Srishti", "Maths", 70),
              (108, "Sandeep", "Physics", 75),
              (107, "Hitesh", "Maths", 88),
              (109, "Kailash", "Maths", 90),
              (105, "Abhishek", "Social Science", 84)
              )
columns = ["Roll_No", "Student_Name", "Subject", "Marks"]
df2 = spark.createDataFrame(data=sampleData,
                            schema=columns)
windowPartition = Window.partitionBy("Subject").orderBy("Marks")
df2.printSchema()
df2.display()
```

```
root
 |-- Roll_No: long (nullable = true)
```

```
|-- Student_Name: string (nullable = true)
|-- Subject: string (nullable = true)
|-- Marks: long (nullable = true)
```

Table

	Roll_No	Student_Name	Subject	Marks
1	101	Ram	Biology	80
2	103	Sita	Social Science	78
3	104	Lakshman	Sanskrit	58
4	102	Kunal	Phisycs	89
5	101	Ram	Biology	80
6	106	Srishti	Maths	70
7	108	Sandeep	Physics	75
8	107	Hitesh	Maths	88
9	109	Kailash	Maths	90
10	105	Abhishek	Social Science	84

10 rows

Using row_number().
row_number() function is used to gives a sequential number to each row present in the table.

```
from pyspark.sql.functions import row_number
df2.withColumn("row_number", row_number().over(windowPartition)).display()
```

Table

	Roll_No	Student_Name	Subject	Marks	row_number
1	101	Ram	Biology	80	1
2	101	Ram	Biology	80	2
3	106	Srishti	Maths	70	1
4	107	Hitesh	Maths	88	2
5	109	Kailash	Maths	90	3
6	102	Kunal	Phisycs	89	1
7	108	Sandeep	Physics	75	1
8	104	Lakshman	Sanskrit	58	1
9	103	Sita	Social Science	78	1
10	105	Abhishek	Social Science	84	2

10 rows

Using rank()
The rank function is used to give ranks to rows specified in the window partition. This function leaves gaps in rank if there are ties.

```
from pyspark.sql.functions import rank
df2.withColumn("rank", rank().over(windowPartition)) \
    .display()
```

Table

	Roll_No	Student_Name	Subject	Marks	rank
1	101	Ram	Biology	80	1
2	101	Ram	Biology	80	1
3	106	Srishti	Maths	70	1
4	107	Hitesh	Maths	88	2
5	109	Kailash	Maths	90	3
6	102	Kunal	Phisycs	89	1
7	108	Sandeep	Physics	75	1

10 rows

Using percent_rank()
This function is similar to rank() function. It also provides rank to rows but in a percentile format.

```
from pyspark.sql.functions import percent_rank
df2.withColumn("percent_rank",percent_rank().over(windowPartition)).display()
```

Table

	Roll_No	Student_Name	Subject	Marks	percent_rank
1	101	Ram	Biology	80	0
2	101	Ram	Biology	80	0
3	106	Srishti	Maths	70	0
4	107	Hitesh	Maths	88	0.5
5	109	Kailash	Maths	90	1
6	102	Kunal	Phisycs	89	0
7	108	Sandeep	Physics	75	0

10 rows

Using dense_rank()
This function is used to get the rank of each row in the form of row numbers. This is similar to rank() function, there is only one difference the rank function leaves gaps in rank when there are ties.

```
from pyspark.sql.functions import dense_rank
df2.withColumn("dense_rank",dense_rank().over(windowPartition)).display()
```

Table

	Roll_No	Student_Name	Subject	Marks	dense_rank
1	101	Ram	Biology	80	1
2	101	Ram	Biology	80	1
3	106	Srishti	Maths	70	1
4	107	Hitesh	Maths	88	2
5	109	Kailash	Maths	90	3
6	102	Kunal	Phisycs	89	1
7	108	Sandeep	Physics	75	1

10 rows

Aggregate functions

```
Aggregate function
An aggregate function or aggregation function is a function where the values of multiple rows
are grouped to form a single summary value. The definition of the groups of rows on which
they operate is done by using the SQL GROUP BY clause. E.g. AVERAGE, SUM, MIN, MAX, etc.
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("pyspark_window").getOrCreate()
sampleData = (("Ram", "Sales", 3000),
              ("Meena", "Sales", 4600),
              ("Abhishek", "Sales", 4100),
              ("Kunal", "Finance", 3000),
              ("Ram", "Sales", 3000),
              ("Srishti", "Management", 3300),
              ("Sandeep", "Finance", 3900),
              ("Hitesh", "Marketing", 3000),
              ("Kailash", "Marketing", 2000),
              ("Shyam", "Sales", 4100)
              )
columns = ["Employee_Name", "Department", "Salary"]
df3 = spark.createDataFrame(data=sampleData,schema=columns)
df3.printSchema()
df3.display()
```

```
root
 |-- Employee_Name: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)
```

Table

	Employee_Name	Department	Salary
1	Ram	Sales	3000
2	Meena	Sales	4600
3	Abhishek	Sales	4100
4	Kunal	Finance	3000
5	Ram	Sales	3000
6	Srishti	Management	3300
7	Sandeep	Finance	3900
8	Hitesh	Marketing	3000
9	Kailash	Marketing	2000
10	Shyam	Sales	4100

10 rows

```
from pyspark.sql.window import Window
from pyspark.sql.functions import col,avg,sum,min,max,row_number
windowPartitionAgg  = Window.partitionBy("Department")
df3.withColumn("Avg",avg(col("salary")).over(windowPartitionAgg)).show()
    # sum()
df3.withColumn("Sum",sum(col("salary")).over(windowPartitionAgg)).show()
    #min()
df3.withColumn("Min",min(col("salary")).over(windowPartitionAgg)).show()
df3.withColumn("Max",max(col("salary")).over(windowPartitionAgg)).show()
```

```
+-------------+----------+------+------+
|Employee_Name|Department|Salary|   Avg|
+-------------+----------+------+------+
|        Kunal|   Finance|  3000|3450.0|
|      Sandeep|   Finance|  3900|3450.0|
|      Srishti|Management|  3300|3300.0|
|       Hitesh| Marketing|  3000|2500.0|
|      Kailash| Marketing|  2000|2500.0|
|          Ram|     Sales|  3000|3760.0|
|        Meena|     Sales|  4600|3760.0|
|     Abhishek|     Sales|  4100|3760.0|
|          Ram|     Sales|  3000|3760.0|
|        Shyam|     Sales|  4100|3760.0|
+-------------+----------+------+------+

+-------------+----------+------+-----+
|Employee_Name|Department|Salary|  Sum|
+-------------+----------+------+-----+
|        Kunal|   Finance|  3000| 6900|
|      Sandeep|   Finance|  3900| 6900|
|      Srishti|Management|  3300| 3300|
```

date and timestamp functions in timestamp

```
from pyspark.sql.functions import current_timestamp, to_timestamp
from pyspark.sql.functions import *
from pyspark.sql import SparkSession
from pyspark.sql.types import IntegerType, DateType, BooleanType

spark = SparkSession.builder \
    .master("local[*]") \
    .appName("timestamp") \
    .getOrCreate()
df = spark.createDataFrame([["1", "2019-07-01 12:01:19.000"], ["2", "2019-06-24
12:01:19.000"]], ["id", "input_timestamp"])
df.printSchema()
df.display()
```

```
root
 |-- id: string (nullable = true)
 |-- input_timestamp: string (nullable = true)
```

Table

	id	input_timestamp
1	1	2019-07-01 12:01:19.000
2	2	2019-06-24 12:01:19.000

2 rows

```
# Converting string datatype to timestamp

df1 = df.withColumn("timestamptype", to_timestamp("input_timestamp"))
df1.display()
```

Table

	id	input_timestamp	timestamptype
1	1	2019-07-01 12:01:19.000	2019-07-01T12:01:19.000+0000
2	2	2019-06-24 12:01:19.000	2019-06-24T12:01:19.000+0000

2 rows

```
#  selecting only necessary column and renaming
df2 = df1.select("id", "timestamptype").withColumnRenamed("timestamptype", "input_timestamp")
df2.display()
```

Table

	id	input_timestamp
1	1	2019-07-01T12:01:19.000+0000

2	2	2019-06-24T12:01:19.000+0000

2 rows

```
# using cast to convert timestamp to DataType

df3=df2.select(col("id"), col("input_timestamp").cast('string'))
df3.display()
```

Table

	id	input_timestamp
1	1	2019-07-01 12:01:19
2	2	2019-06-24 12:01:19

2 rows

```
# timestamp type to datetype
df4 = df2.select(col("id"), to_date(col("input_timestamp")))
df4.display()
```

Table

	id	to_date(input_timestamp)
1	1	2019-07-01
2	2	2019-06-24

2 rows

Select top N rows from each group

```
from pyspark.sql import SparkSession, Window
from pyspark.sql.functions import *
spark = SparkSession \
    .builder \
    .appName("TopN") \
    .master("local[*]") \
    .getOrCreate()
sampledata = (("Nitya", "Sales", 3000), \
    ("Abhi", "Sales", 4600), \
        ("Rakesh", "Sales", 4100), \
            ("Sandeep", "finance", 3000), \
                ("Abhishek", "Sales", 3000), \
                    ("Shyan", "finance", 3300), \
                        ("Madan", "finance", 3900), \
                            ("Jarin", "marketing", 3000), \
                                ("kumar", "marketing", 2000))

columns = ["employee_name", "department", "Salary"]
df = spark.createDataFrame(data = sampledata, schema = columns)
df.display()
```

Table

	employee_name	department	Salary
1	Nitya	Sales	3000
2	Abhi	Sales	4600
3	Rakesh	Sales	4100
4	Sandeep	finance	3000
5	Abhishek	Sales	3000
6	Shyan	finance	3300
7	Madan	finance	3900
8	Jarin	marketing	3000
9	kumar	marketing	2000

9 rows

```
windowSpec = Window.partitionBy("department").orderBy("salary")
df1 = df.withColumn("row", row_number().over(windowSpec))    # applying row_number
df1.display()
```

Table

	employee_name	department	Salary	row
1	Nitya	Sales	3000	1
2	Abhishek	Sales	3000	2
3	Rakesh	Sales	4100	3
4	Abhi	Sales	4600	4

5	Sandeep	finance	3000	1
6	Shyan	finance	3300	2
7	Madan	finance	3900	3
8	kumar	marketing	2000	1
9	Jarin	marketing	3000	2

9 rows

```
df2 = df1.filter(col("row") <3)
df2.display()
```

Table

	employee_name	department	Salary	row
1	Nitya	Sales	3000	1
2	Abhishek	Sales	3000	2
3	Sandeep	finance	3000	1
4	Shyan	finance	3300	2
5	kumar	marketing	2000	1
6	Jarin	marketing	3000	2

6 rows

Drop dublicate record

```
# drop dublicate in emp dataframe using distinct or dropDublicates

from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark = SparkSession \
        .builder \
        .appName("droppingDublicates") \
        .master("local[*]") \
        .getOrCreate()

sample_data = ([1, "ramesh", 1000], [2, "Krishna", 2000], [3, "Shri", 3000], [4, "Pradip",
4000],
               [1, "ramesh", 1000], [2, "Krishna", 2000], [3, "Shri", 3000], [4, "Pradip",
4000])

columns = ["id", "name", "salary"]

df = spark.createDataFrame(data = sample_data, schema= columns)
df.display()
```

Table

	id	name	salary
1	1	ramesh	1000

2	2	Krishna	2000
3	3	Shri	3000
4	4	Pradip	4000
5	1	ramesh	1000
6	2	Krishna	2000
7	3	Shri	3000
8	4	Pradip	4000

8 rows

```
df1= df.distinct().show()
```

```
+---+-------+------+
| id|   name|salary|
+---+-------+------+
|  1| ramesh|  1000|
|  2|Krishna|  2000|
|  3|   Shri|  3000|
|  4| Pradip|  4000|
+---+-------+------+
```

```
df3 = df.dropDuplicates().show()
```

```
+---+-------+------+
| id|   name|salary|
+---+-------+------+
|  1| ramesh|  1000|
|  2|Krishna|  2000|
|  3|   Shri|  3000|
|  4| Pradip|  4000|
+---+-------+------+
```

```
df4 = df.select(["id", "name"]).distinct().show()
```

```
+---+-------+
| id|   name|
+---+-------+
|  1| ramesh|
|  2|Krishna|
|  3|   Shri|
|  4| Pradip|
+---+-------+
```

Explode nested array in pyspark

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, flatten
spark = SparkSession.builder.appName("explodeempdata").master("local[*]").getOrCreate()
arrayArrayData = [("Abhishek", [["Java", "scala", "perl"], ["spark","java"]]),
                  ("Nitya", [["spark", "java", "c++"], ["spark", "java"]]),
                  ("Sandeep", [["csharp", "vb"], ["spark", "python"]])]

df = spark.createDataFrame(data = arrayArrayData, schema = ['name', 'subjects'])

df.printSchema()
df.show()
df.select(df.name, explode(df.subjects)).show(truncate=False)
df.select(df.name, flatten(df.subjects)).show(truncate=False)
```

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

+--------+--------------------+
|    name|            subjects|
+--------+--------------------+
|Abhishek|[[Java, scala, pe...|
|   Nitya|[[spark, java, c+...|
| Sandeep|[[csharp, vb], [s...|
+--------+--------------------+

+--------+-------------------+
|name    |col                |
+--------+-------------------+
|Abhishek|[Java, scala, perl]|
|Abhishek|[spark, java]      |
|Nitya   |[spark, java, c++] |
|Nitya   |[spark, java]      |
```

```
df.select(df.name, explode(df.subjects)).show(truncate=False)
```

```
+--------+-------------------+
|name    |col                |
+--------+-------------------+
|Abhishek|[Java, scala, perl]|
|Abhishek|[spark, java]      |
|Nitya   |[spark, java, c++] |
|Nitya   |[spark, java]      |
|Sandeep |[csharp, vb]       |
|Sandeep |[spark, python]    |
+--------+-------------------+
```

```
df.select(df.name, flatten(df.subjects)).show(truncate=False)
```

```
+--------+------------------------------+
|name    |flatten(subjects)             |
+--------+------------------------------+
|Abhishek|[Java, scala, perl, spark, java]|
|Nitya   |[spark, java, c++, spark, java] |
|Sandeep |[csharp, vb, spark, python]     |
+--------+------------------------------+
```

Splitting multi delimiter row

```
data = [(1, "Abhishek", "10|30|40"),
        (2, "Krishna", "50|40|70"),
        (3, "rakesh", "20|70|90")]

df = spark.createDataFrame(data, schema=["id", "name", "marks"])
df.show()
```

```
+---+--------+--------+
| id|    name|   marks|
+---+--------+--------+
|  1|Abhishek|10|30|40|
|  2| Krishna|50|40|70|
|  3|  rakesh|20|70|90|
+---+--------+--------+
```

```
from pyspark.sql.functions import split, col
df_s = df.withColumn("mark_details", split(col("marks"), "[|]")) \
    .withColumn("maths", col("mark_details")[0]) \
    .withColumn("physics", col("mark_details")[1]) \
    .withColumn("chemistry", col("mark_details")[2])
display(df_s)
```

Table

	id	name	marks	mark_details	maths	physics	chemistry
1	1	Abhishek	10|30|40	▸ ["10", "30", "40"]	10	30	40
2	2	Krishna	50|40|70	▸ ["50", "40", "70"]	50	40	70

3 rows

```
from pyspark.sql.functions import split, col
df_s = df.withColumn("mark_details", split(col("marks"), "[|]")) \
    .withColumn("maths", col("mark_details")[0]) \
    .withColumn("physics", col("mark_details")[1]) \
    .withColumn("chemistry", col("mark_details")[2]).drop("mark_details", "marks")
display(df_s)
```

Table

	id	name	maths	physics	chemistry
1	1	Abhishek	10	30	40
2	2	Krishna	50	40	70
3	3	rakesh	20	70	90

3 rows

How to check required column is exists or not

```
datacolumn = ['empid', 'empname']
data = [(10, 'Krishna'), (20, 'mahesh'), (30, 'Rakesh')]
df = spark.createDataFrame(data = data, schema= datacolumn)
df.show()
```

```
+-----+-------+
|empid|empname|
+-----+-------+
|   10|Krishna|
|   20| mahesh|
|   30| Rakesh|
+-----+-------+
```

```
print(df.schema.fieldNames())
```

```
['empid', 'empname']
```

```
columns = df.schema.fieldNames()
if columns.count('empid')>0:
    print('empid exists in the dataframe')
else:
    print('not exists')
```

```
empid exists in the dataframe
```

pySpark Join

Join is used to combine two or more dataframes based on columns in the dataframe.

Syntax: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”type”)

where-

dataframe1 is the first dataframe
dataframe2 is the second dataframe
column_name is the column which are matching in both the dataframes
type is the join type we have to join

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "Saroj", "company 1"],
        ["2", "Nitya", "company 1"],
        ["3", "Abhishek", "company 2"],
        ["4", "Sandeep", "company 1"],
        ["5", "Rakesh", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
dataframe.display()
```

Table

	ID	NAME	Company
1	1	Saroj	company 1
2	2	Nitya	company 1
3	3	Abhishek	company 2
4	4	Sandeep	company 1
5	5	Rakesh	company 1

5 rows

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']
dataframe1 = spark.createDataFrame(data1, columns)
dataframe1.display()
```

Table

	ID	salary	department
1	1	45000	IT

2	2	145000	Manager
3	6	45000	HR
4	5	34000	Sales

4 rows

Inner Join

```
Inner join
This will join the two PySpark dataframes on key columns, which are common in both
dataframes.

Syntax: dataframe1.join(dataframe2,dataframe1.column_name ==  dataframe2.column_name,”inner”)
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "sravan", "company 1"],
        ["2", "ojaswi", "company 1"],
        ["3", "rohith", "company 2"],
        ["4", "sridevi", "company 1"],
        ["5", "bobby", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']
dataframe1 = spark.createDataFrame(data1, columns)
dataframe.join(dataframe1,dataframe.ID == dataframe1.ID,"inner").display()
```

Table

	ID	NAME	Company	ID	salary	department
1	1	sravan	company 1	1	45000	IT
2	2	ojaswi	company 1	2	145000	Manager
3	5	bobby	company 1	5	34000	Sales

3 rows

Outer join

```
Full Outer Join
This join joins the two dataframes with all matching and non-matching rows, we can perform
this join in three ways

Syntax:

outer: dataframe1.join(dataframe2,dataframe1.column_name ==  dataframe2.column_name,”outer”)
full: dataframe1.join(dataframe2,dataframe1.column_name ==  dataframe2.column_name,”full”)
fullouter: dataframe1.join(dataframe2,dataframe1.column_name ==
dataframe2.column_name,”fullouter”)


#Using outer keyword

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "Nitya", "company 1"],
        ["2", "Ramesh", "company 1"],
        ["3", "Abhishek", "company 2"],
        ["4", "Sandeep", "company 1"],
        ["5", "Manisha", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']
dataframe1 = spark.createDataFrame(data1, columns)
dataframe.join(dataframe1, dataframe.ID == dataframe1.ID,"outer").display()
```

Table

	ID	NAME	Company	ID	salary	department
1	1	Nitya	company 1	1	45000	IT
2	2	Ramesh	company 1	2	145000	Manager
3	3	Abhishek	company 2	null	null	null
4	4	Sandeep	company 1	null	null	null
5	5	Manisha	company 1	5	34000	Sales
6	null	null	null	6	45000	HR

6 rows

```
# using full keyword

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "Nitya", "company 1"],
        ["2", "Rakesh", "company 1"],
        ["3", "Abhishek", "company 2"],
        ["4", "Anjali", "company 1"],
        ["5", "Saviya", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']
dataframe1 = spark.createDataFrame(data1, columns)
dataframe.join(dataframe1, dataframe.ID == dataframe1.ID,"full").display()
```

Table

	ID	NAME	Company	ID	salary	department
1	1	Nitya	company 1	1	45000	IT
2	2	Rakesh	company 1	2	145000	Manager
3	3	Abhishek	company 2	null	null	null
4	4	Anjali	company 1	null	null	null
5	5	Saviya	company 1	5	34000	Sales
6	null	null	null	6	45000	HR

6 rows

Left Join
Here this join joins the dataframe by returning all rows from the first dataframe and only matched rows from the second dataframe with respect to the first dataframe. We can perform this type of join using left and leftouter.

Syntax:
left: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”left”)
leftouter: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”leftouter”)

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "Amit", "company 1"],
        ["2", "Rakesh", "company 1"],
        ["3", "Abhishek", "company 2"],
        ["4", "Sri", "company 1"],
        ["5", "Sachin", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']
dataframe1 = spark.createDataFrame(data1, columns)
dataframe.join(dataframe1,dataframe.ID == dataframe1.ID,"left").display()
```

Table

	ID	NAME	Company	ID	salary	department
1	1	Amit	company 1	1	45000	IT
2	2	Rakesh	company 1	2	145000	Manager
3	3	Abhishek	company 2	null	null	null
4	4	Sri	company 1	null	null	null
5	5	Sachin	company 1	5	34000	Sales

5 rows

Right Join
Here this join joins the dataframe by returning all rows from the second dataframe and only matched rows from the first dataframe with respect to the second dataframe. We can perform this type of join using right and rightouter.

Syntax:

right: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”right”)
rightouter: dataframe1.join(dataframe2,dataframe1.column_name ==
dataframe2.column_name,”rightouter”)

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "Manisha", "company 1"],
        ["2", "Aarti", "company 1"],
        ["3", "Rohit", "company 2"],
        ["4", "Bhuvi", "company 1"],
        ["5", "Virat", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']
dataframe1 = spark.createDataFrame(data1, columns)

# right join on two dataframes
dataframe.join(dataframe1,dataframe.ID == dataframe1.ID,"right").display()
```

Table

	ID	NAME	Company	ID	salary	department
1	1	Manisha	company 1	1	45000	IT
2	2	Aarti	company 1	2	145000	Manager
3	null	null	null	6	45000	HR
4	5	Virat	company 1	5	34000	Sales

4 rows

Leftsemi join
This join will all rows from the first dataframe and return only matched rows from the second dataframe

Syntax: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”leftsemi”)

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "Mitchell", "company 1"],
        ["2", "Rachin", "company 1"],
        ["3", "Kuldeep", "company 2"],
        ["4", "Rahul", "company 1"],
        ["5", "Thomas", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']
dataframe1 = spark.createDataFrame(data1, columns)

# leftsemi join on two dataframes
dataframe.join(dataframe1,dataframe.ID == dataframe1.ID,"leftsemi").display()
```

Table

	ID	NAME	Company
1	1	Mitchell	company 1
2	2	Rachin	company 1
3	5	Thomas	company 1

3 rows

LeftAnti join
This join returns only columns from the first dataframe for non-matched records of the second dataframe

Syntax: dataframe1.join(dataframe2,dataframe1.column_name == dataframe2.column_name,”leftanti”)

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "Manisha", "company 1"],
        ["2", "Mohan", "company 1"],
        ["3", "Rohit", "company 2"],
        ["4", "Srini", "company 1"],
        ["5", "boWilliamsonbby", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']
dataframe1 = spark.createDataFrame(data1, columns)
dataframe.join(dataframe1,dataframe.ID == dataframe1.ID,"leftanti").display()
```

Table

	ID	NAME	Company
1	3	Rohit	company 2
2	4	Srini	company 1

2 rows

SQL Expression
We can perform all types of the above joins using an SQL expression, we have to mention the type of join in this expression. To do this, we have to create a temporary view.

Syntax: dataframe.createOrReplaceTempView(“name”)

where

dataframe is the input dataframe
name is the view name

Syntax: spark.sql(“select * from dataframe1, dataframe2 where dataframe1.column_name == dataframe2.column_name “)

where,

dataframe1 is the first view dataframe
dataframe2 is the second view dataframe
column_name is the column to be joined

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "Manoj", "company 1"],
        ["2", "Manisha", "company 1"],
        ["3", "RAmisha", "company 2"],
        ["4", "Sri", "company 1"],
        ["5", "RAkesh", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']

# creating a dataframe from the lists of data
dataframe1 = spark.createDataFrame(data1, columns)

# create a view for dataframe named student
dataframe.createOrReplaceTempView("student")

# create a view for dataframe1 named department
dataframe1.createOrReplaceTempView("department")

#use sql expression to select ID column
spark.sql("select * from student, department where student.ID == department.ID").display()
```

Table

	ID	NAME	Company	ID	salary	department
1	1	Manoj	company 1	1	45000	IT
2	2	Manisha	company 1	2	145000	Manager
3	5	RAkesh	company 1	5	34000	Sales

3 rows

Syntax: spark.sql(“select * from dataframe1 JOIN_TYPE dataframe2 ON dataframe1.column_name == dataframe2.column_name “)

where, JOIN_TYPE refers to above all types of joins

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "Ramesh", "company 1"],
        ["2", "Rakesh", "company 1"],
        ["3", "Rohan", "company 2"],
        ["4", "Rachin", "company 1"],
        ["5", "Ravindra", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)
data1 = [["1", "45000", "IT"],
         ["2", "145000", "Manager"],
         ["6", "45000", "HR"],
         ["5", "34000", "Sales"]]
columns = ['ID', 'salary', 'department']
dataframe1 = spark.createDataFrame(data1, columns)

# create a view for dataframe named student
dataframe.createOrReplaceTempView("student")

# create a view for dataframe1 named department
dataframe1.createOrReplaceTempView("department")

# inner join on id column using sql expression
spark.sql("select * from student INNER JOIN department on student.ID ==
department.ID").show()
```

```
+---+--------+---------+---+------+----------+
| ID|    NAME|  Company| ID|salary|department|
+---+--------+---------+---+------+----------+
|  1|  Ramesh|company 1|  1| 45000|        IT|
|  2|  Rakesh|company 1|  2|145000|   Manager|
|  5|Ravindra|company 1|  5| 34000|     Sales|
+---+--------+---------+---+------+----------+
```

Using functools
Functools module provides functions for working with other functions and callable objects to use or extend them without completely rewriting them.

Syntax:

```
functools.reduce(lambda df1, df2: df1.union(df2.select(df1.columns)), dfs)
```

where,

df1 is the first dataframe
df2 is the second dataframe

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('pyspark - example join').getOrCreate()
data = [(('Ram'), '1991-04-01', 'M', 3000),
        (('Mike'), '2000-05-19', 'M', 4000),
        (('Rohini'), '1978-09-05', 'M', 4000),
        (('Maria'), '1967-12-01', 'F', 4000),
        (('Jenis'), '1980-02-17', 'F', 1200)]
columns = ["Name", "DOB", "Gender", "salary"]
df1 = spark.createDataFrame(data=data, schema=columns)
df1.show()
```

```
+------+----------+------+------+
|  Name|       DOB|Gender|salary|
+------+----------+------+------+
|   Ram|1991-04-01|     M|  3000|
|  Mike|2000-05-19|     M|  4000|
|Rohini|1978-09-05|     M|  4000|
| Maria|1967-12-01|     F|  4000|
| Jenis|1980-02-17|     F|  1200|
+------+----------+------+------+
```

```
data2 = [(('Mohi'), '1991-04-01', 'M', 3000),
         (('Ani'), '2000-05-19', 'F', 4300),
         (('Shipta'), '1978-09-05', 'F', 4200),
         (('Jessy'), '1967-12-01', 'F', 4010),
         (('kanne'), '1980-02-17', 'F', 1200)]
columns = ["Name", "DOB", "Gender", "salary"]
df2 = spark.createDataFrame(data=data, schema=columns)
df2.show()
```

```
+------+----------+------+------+
|  Name|       DOB|Gender|salary|
+------+----------+------+------+
|   Ram|1991-04-01|     M|  3000|
|  Mike|2000-05-19|     M|  4000|
|Rohini|1978-09-05|     M|  4000|
| Maria|1967-12-01|     F|  4000|
| Jenis|1980-02-17|     F|  1200|
+------+----------+------+------+
```

```
import functools
def unionAll(dfs):
    return functools.reduce(lambda df1, df2: df1.union(
      df2.select(df1.columns)), dfs)
result3 = unionAll([df1, df2])
result3.display()
```

Table

	Name	DOB	Gender	salary
1	Ram	1991-04-01	M	3000

2	Mike	2000-05-19	M	4000
3	Rohini	1978-09-05	M	4000
4	Maria	1967-12-01	F	4000
5	Jenis	1980-02-17	F	1200
6	Ram	1991-04-01	M	3000
7	Mike	2000-05-19	M	4000
8	Rohini	1978-09-05	M	4000
9	Maria	1967-12-01	F	4000
10	Jenis	1980-02-17	F	1200

10 rows

Filter Columns with None or Null Values

the dataframes contains many NULL/None values in columns, in many of the cases before performing any of the operations of the dataframe firstly we have to handle the NULL/None values in order to get the desired result or output, we have to filter those NULL values from the dataframe.

df.filter(condition) : This function returns the new dataframe with the values which satisfies the given condition.
df.column_name.isNotNull() : This function is used to filter the rows that are not NULL/None in the dataframe column.

drop all columns with null values

PySpark drop() Syntax
The drop() method in PySpark has three optional arguments that may be used to eliminate NULL values from single, any, all, or numerous DataFrame columns. Because drop() is a transformation method, it produces a new DataFrame after removing rows/records from the current Dataframe.

drop(how='any', thresh=None, subset=None)
All of these settings are optional.

how - This accepts any or all values. Drop a row if it includes NULLs in any column by using the ‘any’ operator. Drop a row only if all columns contain NULL values if you use the ‘all’ option. The default value is ‘any’.
thresh - This is an int quantity; rows with less than thresh hold non-null values are dropped. ‘None’ is the default.
subset - This is used to select the columns that contain NULL values. ‘None’ is the default.

```
pip install findspark
```

```
Python interpreter will be restarted.
Collecting findspark
  Using cached findspark-2.0.1-py2.py3-none-any.whl (4.4 kB)
Installing collected packages: findspark
Successfully installed findspark-2.0.1
Python interpreter will be restarted.
```

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, FloatType
from pyspark.sql import SparkSession
import findspark
findspark.init('_path-to-spark_')
data2 = [("Pulkit", 12, "CS32", 82, "Programming"),
         ("Ritika", 20, "CS32", 94, "Writing"),
         ("Atirikt", 4, "BB21", 78, None),
         ("Reshav", 18, None, 56, None)
         ]
spark = SparkSession.builder.appName("Student_Info").getOrCreate()
schema = StructType([
    StructField("Name", StringType(), True),
    StructField("Roll Number", IntegerType(), True),
    StructField("Class ID", StringType(), True),
    StructField("Marks", IntegerType(), True),
    StructField("Extracurricular", StringType(), True)
])
df = spark.createDataFrame(data=data2, schema=schema)

# drop None Values
df.na.drop(how="any").show(truncate=False)

# stop spark session
spark.stop()
```

```
  The spark context has stopped and the driver is restarting. Your notebook will be automati
cally reattached.
```

Remove all columns where the entire column is null

```
pyspark.sql.SparkSession.createDataFrame()

Parameters:

dataRDD: An RDD of any kind of SQL data representation(e.g. Row, tuple, int, boolean, etc.),
or list, or pandas.DataFrame.
schema: A datatype string or a list of column names, default is None.
samplingRatio: The sample ratio of rows used for inferring
verifySchema: Verify data types of every row against schema. Enabled by default.
Returns: Dataframe
```

```
from pyspark.sql import SparkSession
import pyspark.sql.types as T


spark = SparkSession.builder.appName('My App').getOrCreate()

actor_data = [
    ("James", None, "Bond", "M", 6000),
    ("Michael", None, None, "M", 4000),
    ("Robert", None, "Pattinson", "M", 4000),
    ("Natalie", None, "Portman", "F", 4000),
    ("Julia", None, "Roberts", "F", 1000)
]
actor_schema = T.StructType([
    T.StructField("firstname", T.StringType(), True),
    T.StructField("middlename", T.StringType(), True),
    T.StructField("lastname", T.StringType(), True),
    T.StructField("gender", T.StringType(), True),
    T.StructField("salary", T.IntegerType(), True)
])

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

```
+---------+----------+---------+------+------+
|firstname|middlename|lastname |gender|salary|
+---------+----------+---------+------+------+
|James    |null      |Bond     |M     |6000  |
|Michael  |null      |null     |M     |4000  |
|Robert   |null      |Pattinson|M     |4000  |
|Natalie  |null      |Portman  |F     |4000  |
|Julia    |null      |Roberts  |F     |1000  |
+---------+----------+---------+------+------+
```

```
import pyspark.sql.functions as F
null_counts = df.select([F.count(F.when(F.col(c).isNull(), c)).alias(
    c) for c in df.columns]).collect()[0].asDict()
print(null_counts)
df_size = df.count()
to_drop = [k for k, v in null_counts.items() if v == df_size]
print(to_drop)
output_df = df.drop(*to_drop)

output_df.show(truncate=False)
```

```
{'firstname': 0, 'middlename': 5, 'lastname': 1, 'gender': 0, 'salary': 0}
['middlename']
+---------+---------+------+------+
|firstname|lastname |gender|salary|
+---------+---------+------+------+
|James    |Bond     |M     |6000  |
|Michael  |null     |M     |4000  |
|Robert   |Pattinson|M     |4000  |
```

```
|Natalie  |Portman  |F     |4000  |
|Julia    |Roberts  |F     |1000  |
+---------+---------+------+------+
```

Filtering rows based on column values

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data = [["1", "sravan", "company 1"],
        ["2", "ojaswi", "company 1"],
        ["3", "rohith", "company 2"],
        ["4", "sridevi", "company 1"],
        ["1", "sravan", "company 1"],
        ["4", "sridevi", "company 1"]]
columns = ['ID', 'NAME', 'Company']
dataframe = spark.createDataFrame(data, columns)

dataframe.show()
```

```
+---+-------+---------+
| ID|   NAME|  Company|
+---+-------+---------+
|  1| sravan|company 1|
|  2| ojaswi|company 1|
|  3| rohith|company 2|
|  4|sridevi|company 1|
|  1| sravan|company 1|
|  4|sridevi|company 1|
+---+-------+---------+
```

Using where() function

Syntax: dataframe.where(condition)

filter rows in dataframe where ID =1

```
dataframe.where(dataframe.ID=='1').show()
```

```
+---+------+---------+
| ID|  NAME|  Company|
+---+------+---------+
|  1|sravan|company 1|
|  1|sravan|company 1|
+---+------+---------+
```

```
dataframe.where(dataframe.NAME != 'sravan').show()
```

```
+---+-------+---------+
| ID|   NAME|  Company|
+---+-------+---------+
|  2| ojaswi|company 1|
```

```
|  3| rohith|company 2|
|  4|sridevi|company 1|
|  4|sridevi|company 1|
+---+-------+---------+
```

```
dataframe.filter(dataframe.ID>'3').show()
```

```
+---+-------+---------+
| ID|   NAME|  Company|
+---+-------+---------+
|  4|sridevi|company 1|
|  4|sridevi|company 1|
+---+-------+---------+
```

Count rows based on condition in Pyspark

```
Using where() function.
Using filter() function.
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sparkdf').getOrCreate()
data =[["1","Nitya","vignan"],
       ["2","Nitesh","vvit"],
       ["3","Neha","vvit"],
       ["4","Neerak","vignan"],
       ["1","Neekung","vignan"],
       ["5","Neelam","iit"]]
columns = ['ID','NAME','college']
dataframe = spark.createDataFrame(data,columns)
dataframe.show()
```

```
+---+-------+-------+
| ID|   NAME|college|
+---+-------+-------+
|  1|  Nitya| vignan|
|  2| Nitesh|   vvit|
|  3|   Neha|   vvit|
|  4| Neerak| vignan|
|  1|Neekung| vignan|
|  5| Neelam|    iit|
+---+-------+-------+
```

```
dataframe.count()
```

```
Out[8]: 6
```

```
# dataframe.where(condition)

print(dataframe.where(dataframe.ID == '1').count())

print('They are  ')
dataframe.where(dataframe.ID == '1').show()
```

```
2
They are
+---+-------+-------+
| ID|   NAME|college|
+---+-------+-------+
|  1|  Nitya| vignan|
|  1|Neekung| vignan|
+---+-------+-------+
```

```
print(dataframe.where(dataframe.ID != '1').count())

print(dataframe.where(dataframe.college == 'vignan').count())

print(dataframe.where(dataframe.ID > 2).count())
```

```
4
3
3
```

Scenario based question

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
spark = SparkSession.builder.appName("StorageVolumne").getOrCreate()

storage_houseData = [("Room1", 1, 1),
                     ("Room1", 2, 10),
                     ("Room1", 3, 5),
                     ("Room2", 1,2),
                     ("Room2", 2,2),
                     ("Room3", 4,1)
                     ]
storage_house_df = spark.createDataFrame(storage_houseData, ["name", "product_id", "units"])

product_data = [
    (1, "Mencollection", 5, 50, 40),
    (2, "Girlcollection", 5,5,5),
    (3, "Childcollectgion", 2,10,10),
    (4, "Womencollection", 4, 10, 20)
]

product_df = spark.createDataFrame(product_data, ["product_id", "product_name", "Width",
"Length", "Height"])

# calculatinh the volumne for each item in storage house

result_df= storage_house_df.join(product_df, on = "product_id", how = "inner") \
    .withColumn("volume", col("Width") * col("Length") * col("Height")) \
    .groupBy("name") \
    .agg({"volume":"sum"}) \
    .withColumnRenamed("sum(volumn)", "volume")

result_df.show()
```

```
+-----+-----------+
| name|sum(volume)|
+-----+-----------+
|Room3|        800|
|Room2|      10125|
|Room1|      10325|
+-----+-----------+
```