**Madiha Aimon Tappal**

[madihaaimon@gmail.com](mailto:madihaaimon@gmail.com)

**Data Engineering Batch – 1**

**Day – 17 Assignment**

**PySpark**

**GroupBy and Aggregate function: -**

The theoretical concepts behind the **groupBy** and **agg** functions in PySpark.

**1. groupBy Function:**

The **groupBy** operation is a fundamental concept in data processing and analytics. In the context of PySpark:

* **Grouping Rows:** The **groupBy** function is used to group rows of a DataFrame based on one or more specified columns.
* **Data Partitioning:** It divides the data into groups based on the unique values in the specified columns, forming subsets or partitions of the DataFrame.
* **GroupedData Object:** The result of a **groupBy** operation is a **GroupedData** object, which acts as an intermediary for performing aggregate operations on the grouped data.
* **Parallel Processing:** Under the hood, PySpark performs this grouping in a distributed and parallelized manner, which is essential for handling large-scale datasets across a cluster of machines.

**2. agg (Aggregate) Function:**

The **agg** function is used to perform aggregation operations on grouped data. Key theoretical aspects include:

* **Aggregate Functions:** Aggregation involves applying functions that summarize or compute statistics on the values within each group.
* **Dictionary Syntax:** The **agg** function typically takes a dictionary where keys are column names, and values are the aggregate functions to be applied to those columns.
* **Column Aliases:** It allows renaming the resulting columns by providing aliases, making the output more readable and user-friendly.
* **Flexible Aggregations:** PySpark supports a wide range of built-in and user-defined aggregate functions, providing flexibility in computing various statistics.
* **Distributed Computation:** Similar to the **groupBy** operation, the **agg** function operates in a distributed fashion, allowing it to efficiently handle large datasets across a cluster.

In summary, **groupBy** and **agg** functions in PySpark enable efficient and parallelized data grouping and aggregation, which is crucial for processing and analyzing massive datasets in distributed computing environments. They form the foundation for performing complex analytics and computations on Big Data using the PySpark framework.

Similar to SQL GROUP BY clause, PySpark groupBy() function is used to collect the identical data into groups on DataFrame and perform count, sum, avg, min, and max functions on the grouped data.

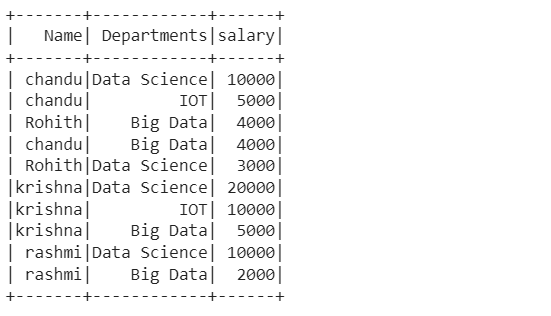
Before starting, let's create a simple DataFrame to work with. The CSV file used can be found here.

from pyspark.sql import SparkSession

spark =SparkSession.builder.appName("Practice").getOrCreate()

df\_pyspark= spark.read.csv("test2.csv",header=True,inferSchema=True)  
df\_pyspark.show()

df\_pyspark.groupBy("Departments").sum("salary").show()

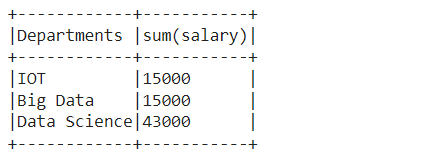


groupBy():

groupBy() on Departments column of DataFrame and then find the sum of salary for each department using sum() function.

#group by Departments which gives summation of salaries

df\_pyspark.groupBy("Departments").sum("salary").show()



Similarly, we can perform min, max, mean, avg, and count using the groupBy function.

df\_pyspark.groupBy("Departments").min("salary").show()  
df\_pyspark.groupBy("Departments").max("salary").show()  
df\_pyspark.groupBy("Departments").avg("salary").show()  
df\_pyspark.groupBy("Departments").mean("salary").show()

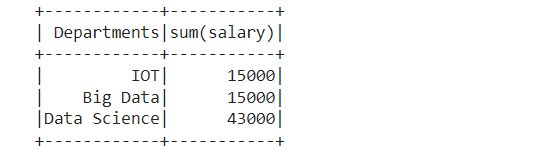
df\_pyspark.groupBy("Departments").count().show() #count of number of people in each Department

groupBy() using multiple columns

df\_pyspark.groupBy("Name","Departments").sum("salary").show()

groupBy() and agg() function

df\_pyspark.groupBy("Departments").agg(({"salary":"sum"})).show()

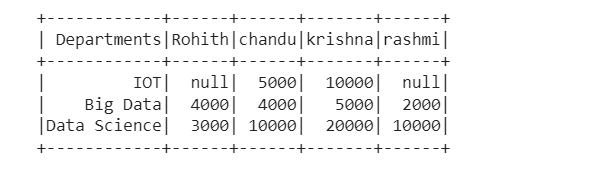


We can also perform agg() function on entire DataFrame without groupBy()

df\_pyspark.agg(({"salary":"sum"})).show()+-----------+   
|sum(salary)|   
+-----------+   
| 73000|   
+-----------+

Using Pivot/ UnPivot — Spark SQL provides pivot() function to rotate the data from one column into multiple columns (transpose row to column). It is an aggregation where one of the grouping column values is transposed into individual columns with distinct data. Similary UnPivot can be used.

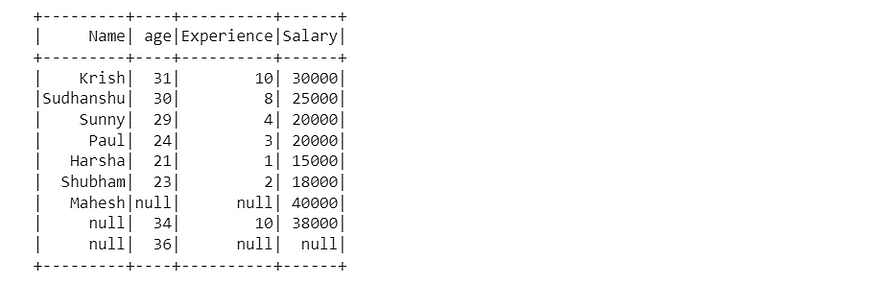
df\_pyspark.groupBy("Departments").pivot("Name").sum("salary").show()



Handling Missing Values Pyspark

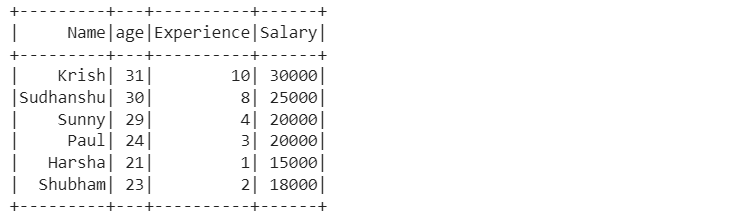
For this task, we will be using this CSV file

df\_pyspark1=spark.read.csv("test3.csv",header=True,inferSchema=True)  
df\_pyspark1.show()



Dropping rows based on null values

df\_pyspark1.na.drop().show()



drop() has the following parameters — how, thresh, and subset

1. df\_pyspark1.na.drop(how="all").show()

# if all values in rows are null then drop # default any

1. df\_pyspark1.na.drop(how="any",thresh=2).show()

#atleast 2 non null values should be present. df\_pyspark1.na.drop(how="any",subset=["salary"]).show()

# only in that column rows get deleted

Filling missing values — Single Value

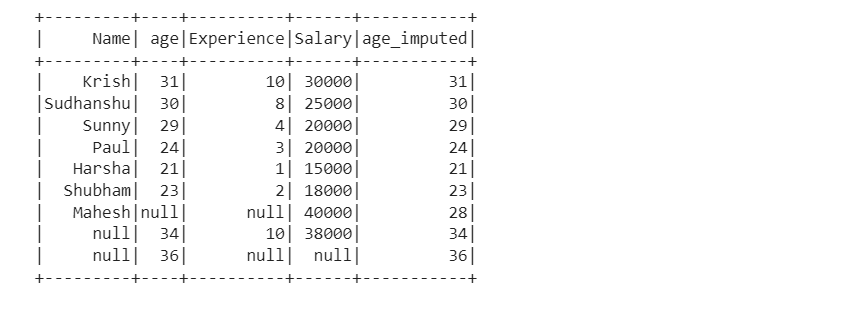
df\_pyspark1.na.fill('Missing Values').show() #string values will get replaced as string is given as inputdf\_pyspark1.na.fill(0).show() #integer values will get replaced as integer is given as input

Filling missing values using Mean, Median, or Mode with help of the Imputer function

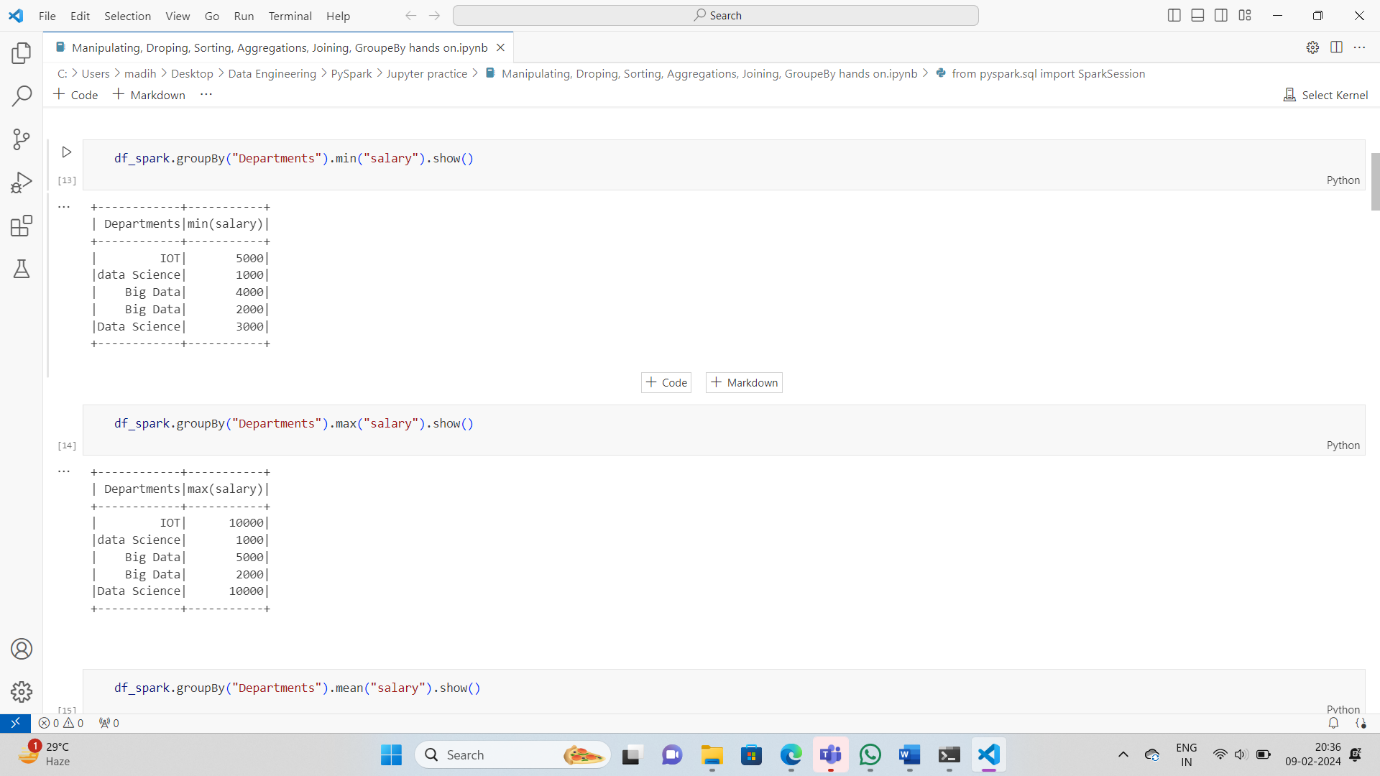
#filling with meanfrom pyspark.ml.feature import Imputerimputer = Imputer(inputCols=["age"],outputCols=["age\_imputed"]).setStrategy("mean")

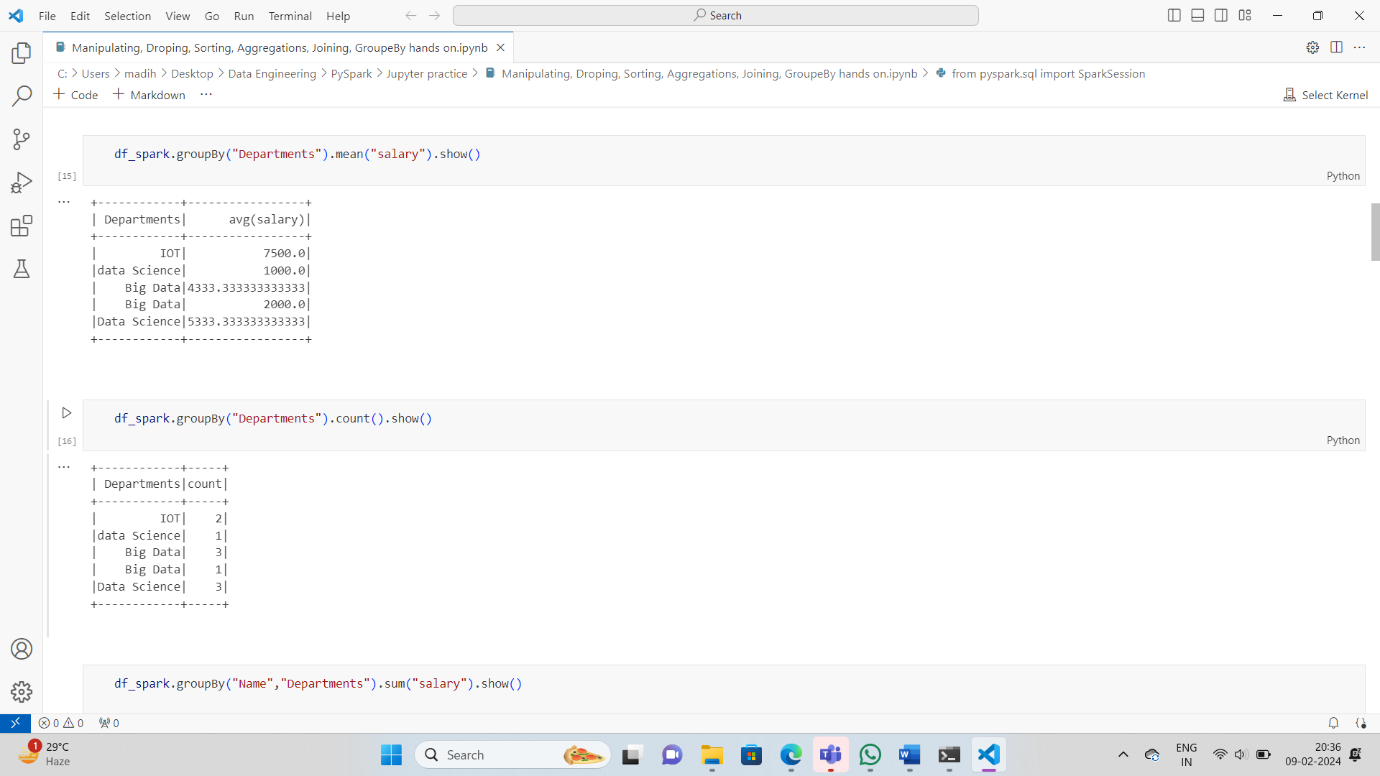
In setStrategy we can use mean, median, or mode.

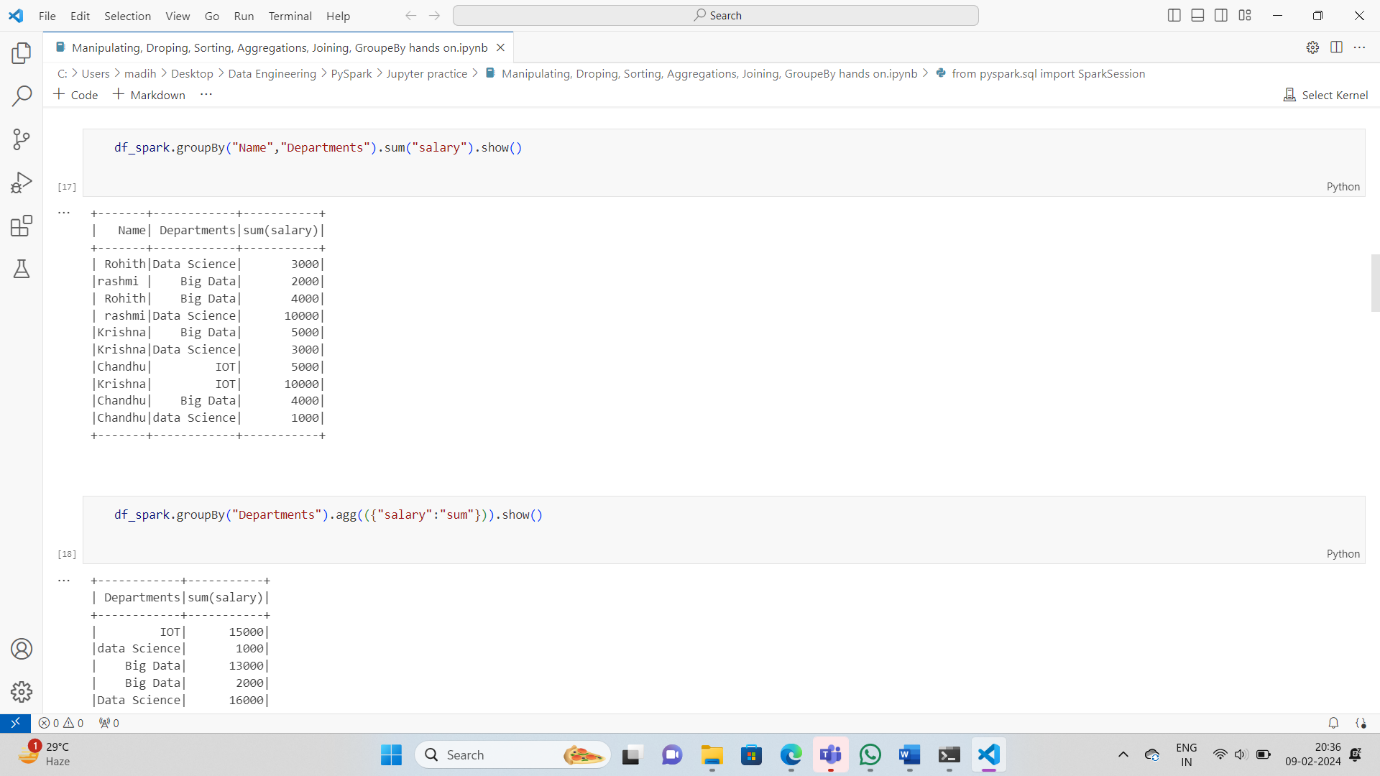
imputer.fit(df\_pyspark1).transform(df\_pyspark1).show()

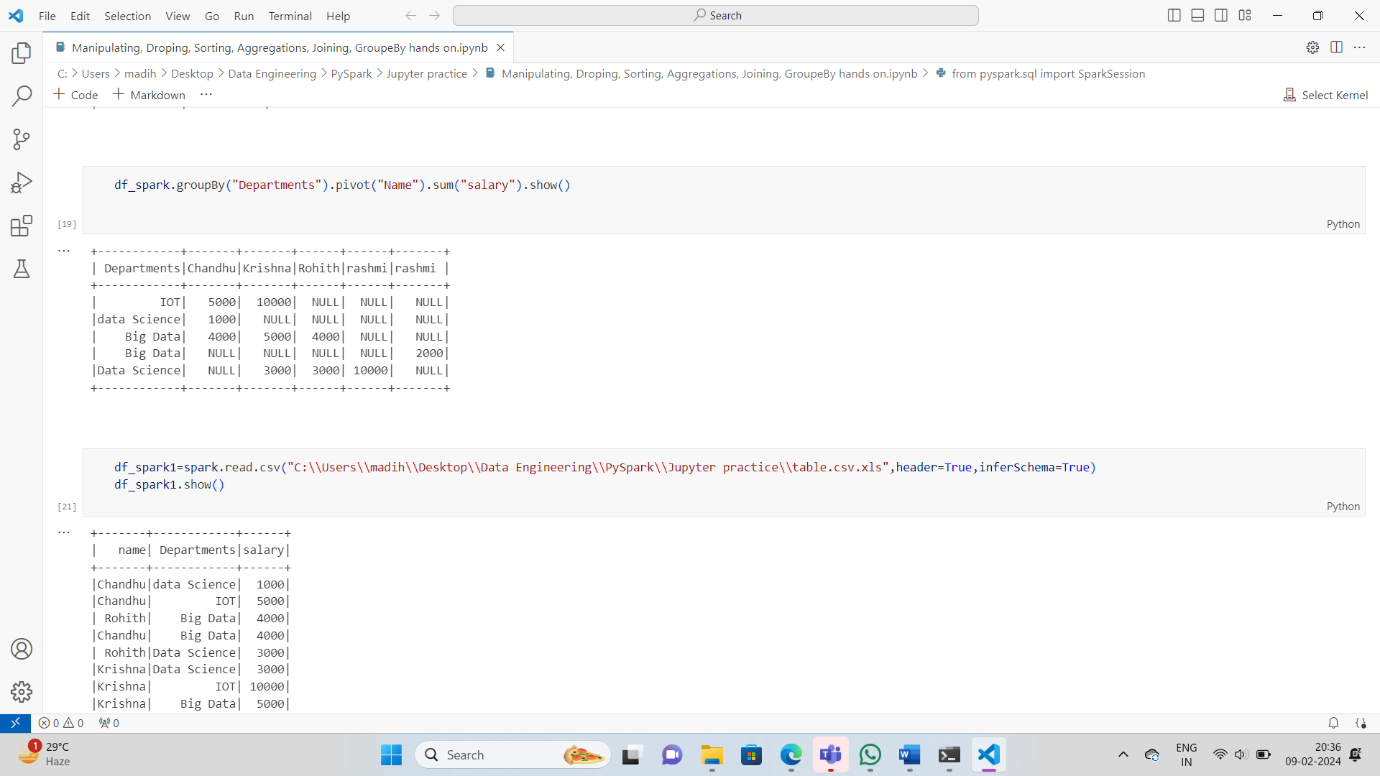


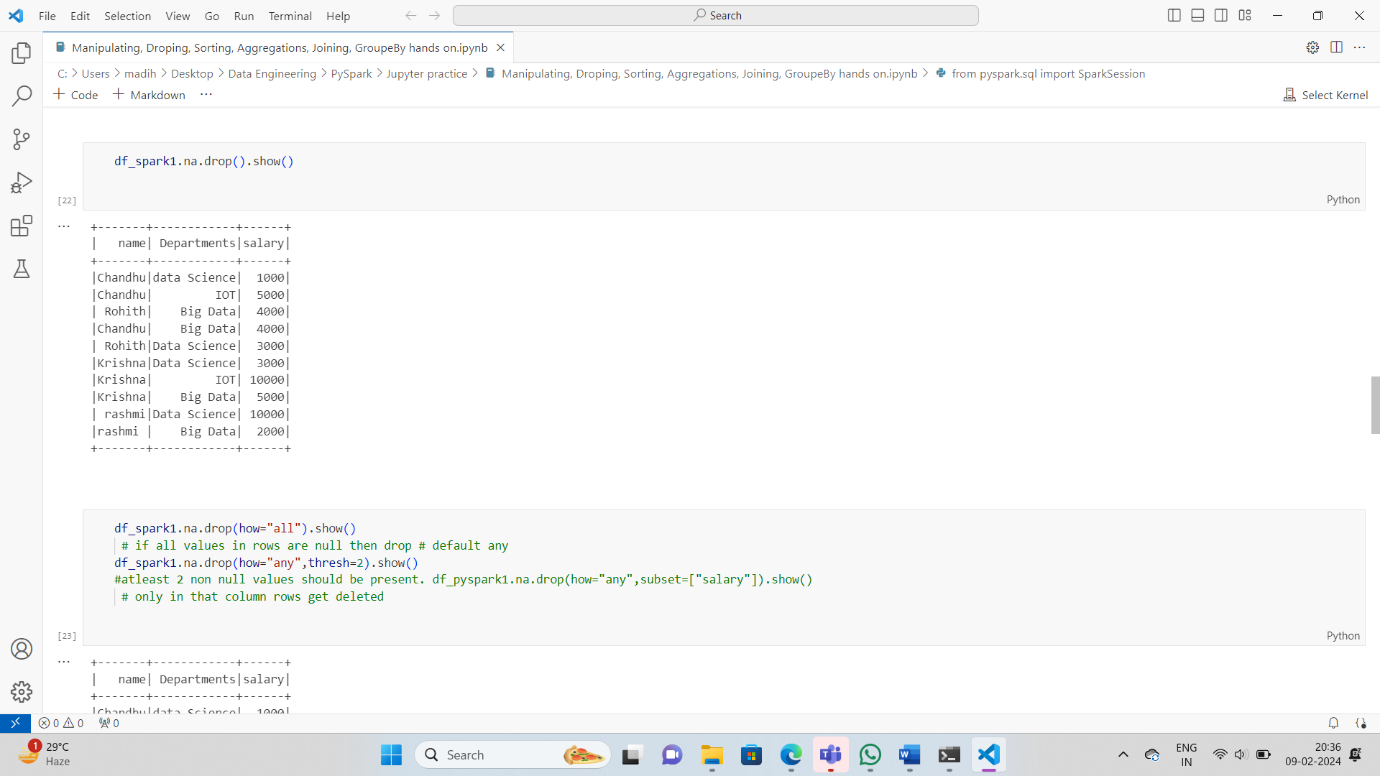
****

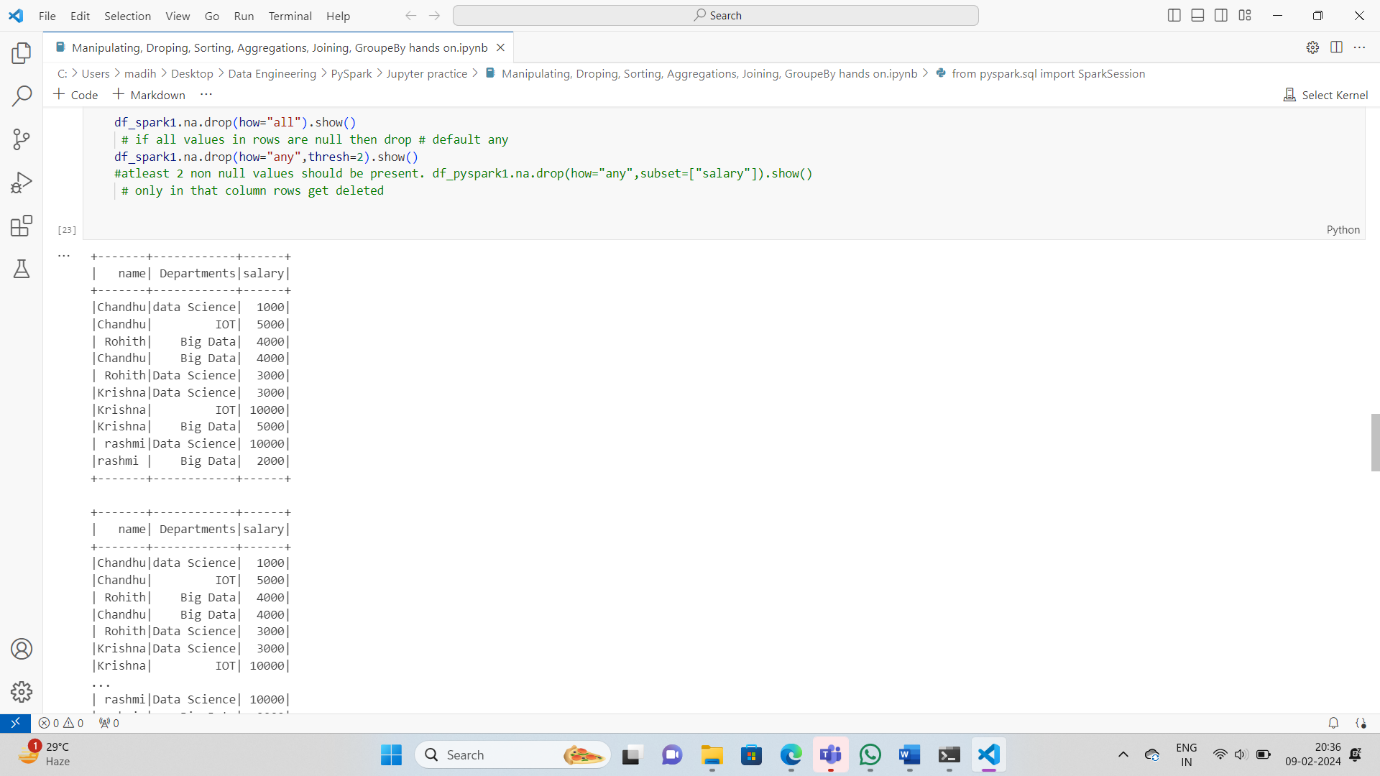
****

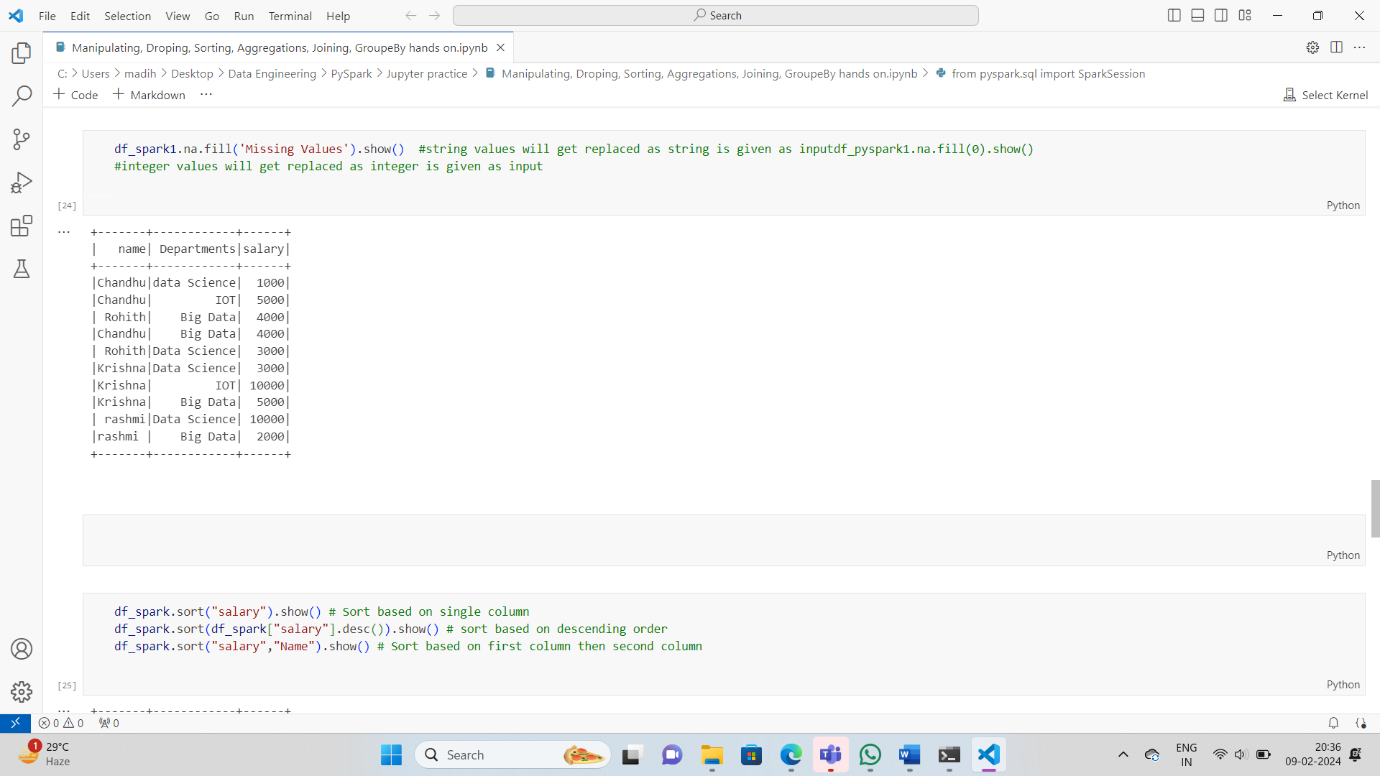
****

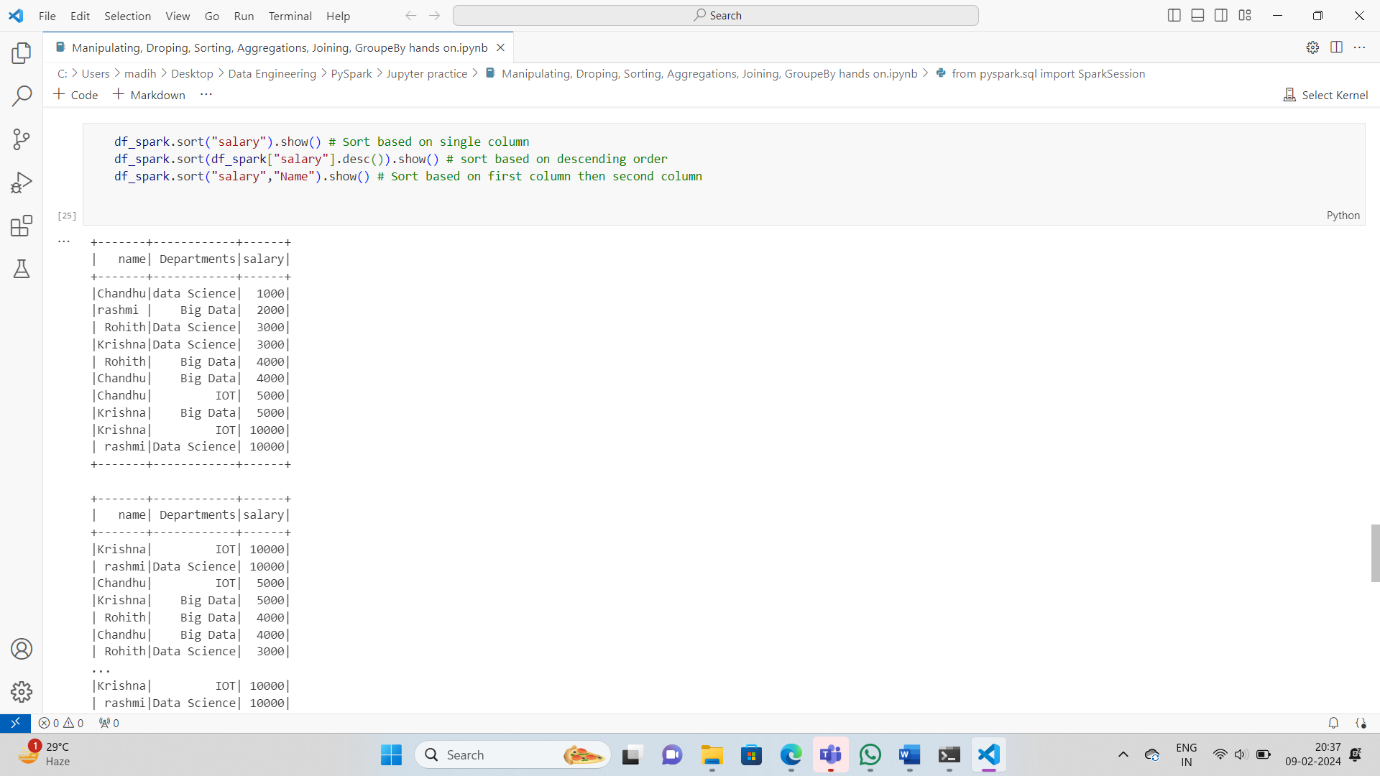
****

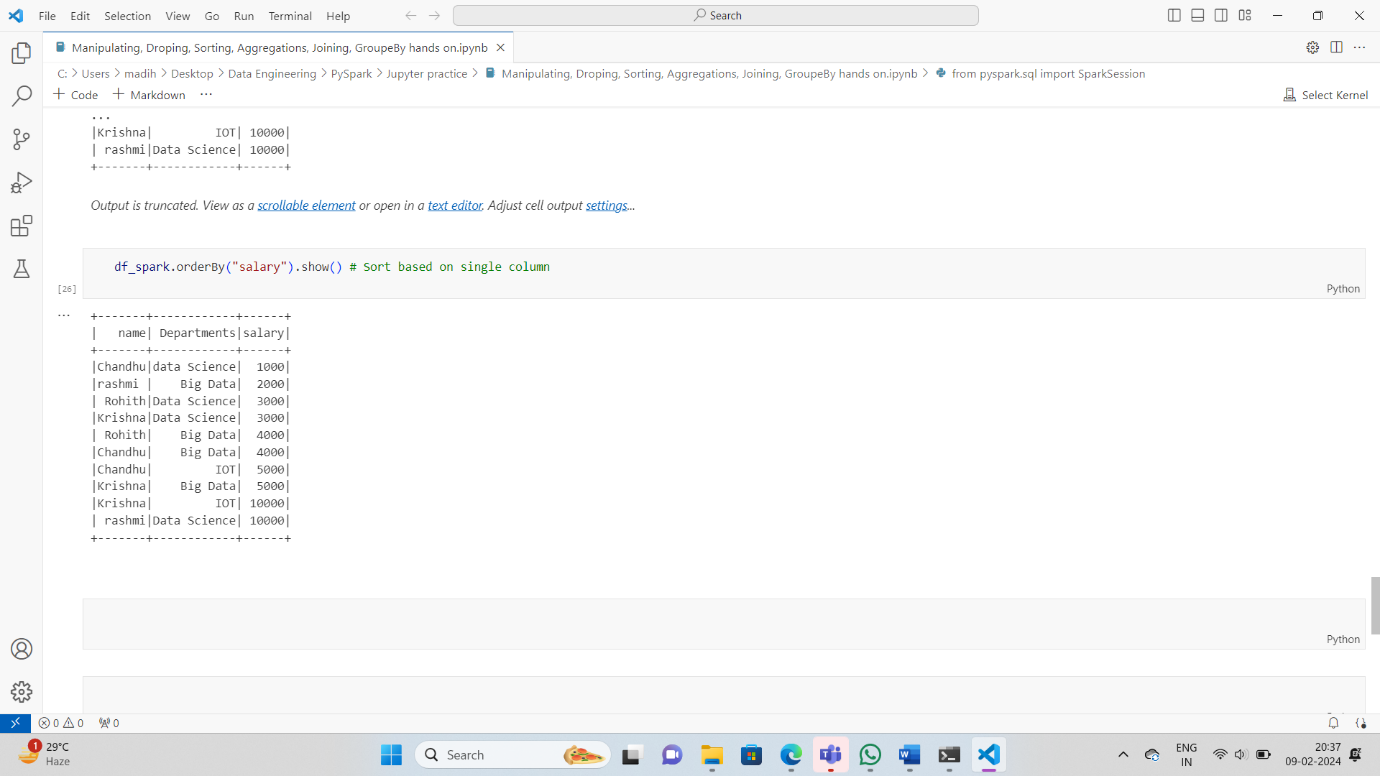
****

****

****

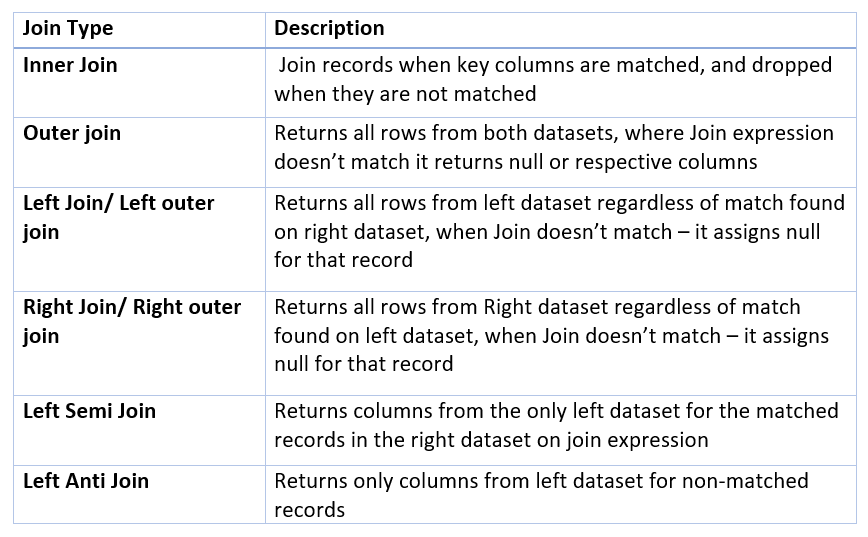
****

****

****

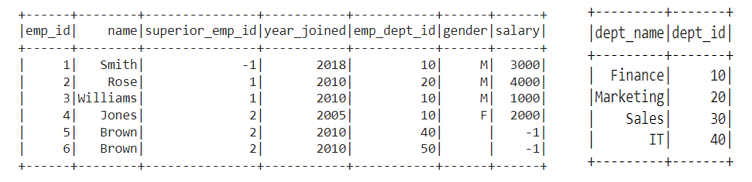
**Join() using PySpark:**

PySpark Join is used to combine two DataFrames and by chaining these you can join multiple DataFrames; it supports all basic join type operations available in traditional SQL like INNER, LEFT OUTER, RIGHT OUTER, LEFT ANTI, LEFT SEMI, CROSS, SELF join.

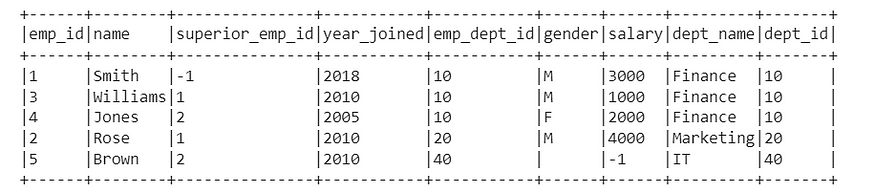


Lets start by creating two dataframes:

emp = [(1,"Smith",-1,"2018","10","M",3000),(2, "Rose",1 , "2010", "20","M", 4000),(3,"Williams",1,"2010","10","M",1000),(4, "Jones",2 ,"2005","10","F",2000),(5,"Brown",2,"2010","40","",-1),(6, "Brown", 2, "2010","50","",-1)]empColumns = ["emp\_id","name","superior\_emp\_id","year\_joined", "emp\_dept\_id","gender","salary"]  
  
empDF = spark.createDataFrame(data=emp, schema = empColumns)  
empDF.printSchema()  
empDF.show()  
  
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()

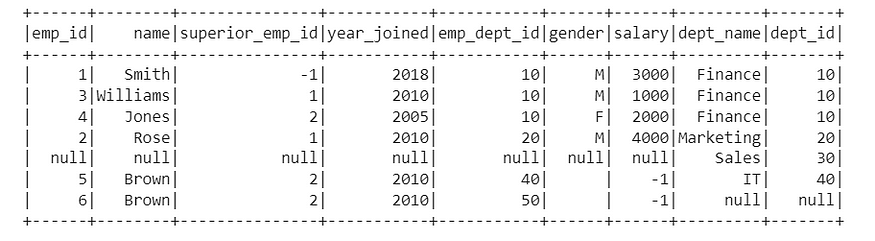


empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id,"inner") .show()



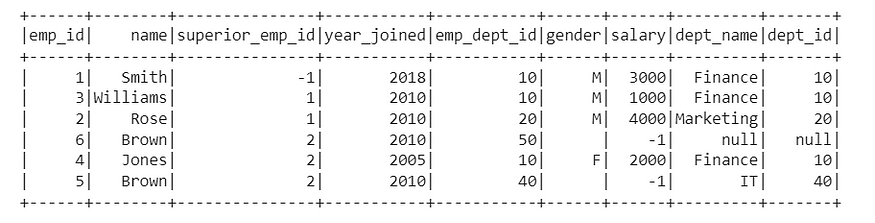
Inner Join

empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id,"outer").show()  
#Or  
empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id,"full").show()  
#Or  
empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id,"fullouter").show()



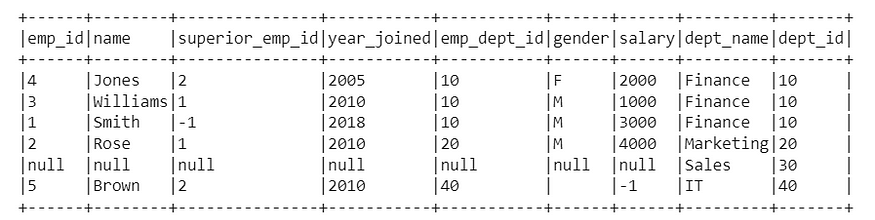
Outer Join

empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id,"left").show()  
#Or  
empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id,"leftouter").show()



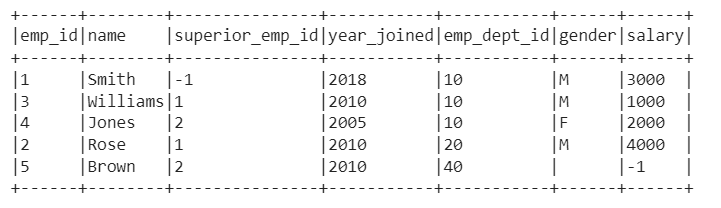
Left Join

empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id,"right").show()  
#Or  
empDF.join(deptDF,empDF.emp\_dept\_id == deptDF.dept\_id,"rightouter").show()



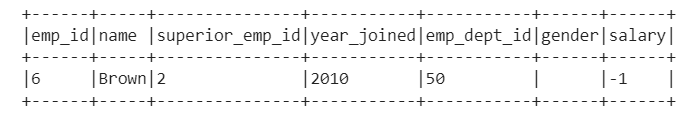
Right Join

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



Leftsemi Join

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



Left Anti Join

There is no self join available in Pyspark DataFrame, but it can be done using any of the available methods above.

Union() using pyspark

To merge two or more dataframes of same schema or structure union() is used.

unionDF = df.union(df2)

To merge without duplicates distinct() is used along with union()

disDF = df.union(df2).distinct()

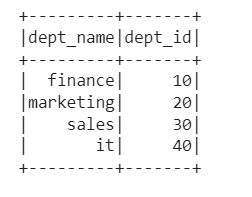
User Defined Functions(UDF):

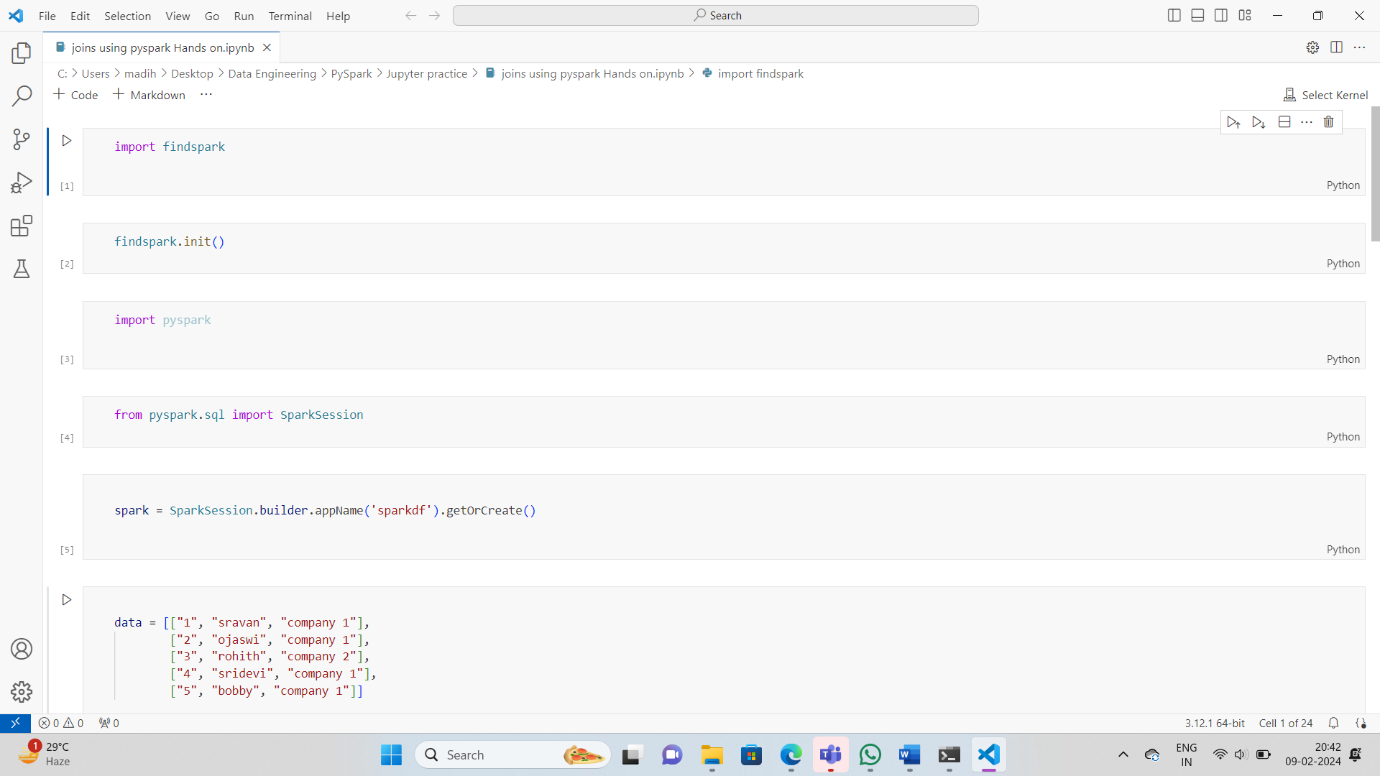
UDF’s are used to extend the functions of the framework and re-use these functions on multiple DataFrame’s. For example, if we want to convert every first letter of a word in a name string to a Lowercase. PySpark build-in features don’t have this function hence we can create a UDF and reuse this as needed on many Data Frames. UDF’s are once created they can be re-used on several DataFrame’s and SQL expressions.

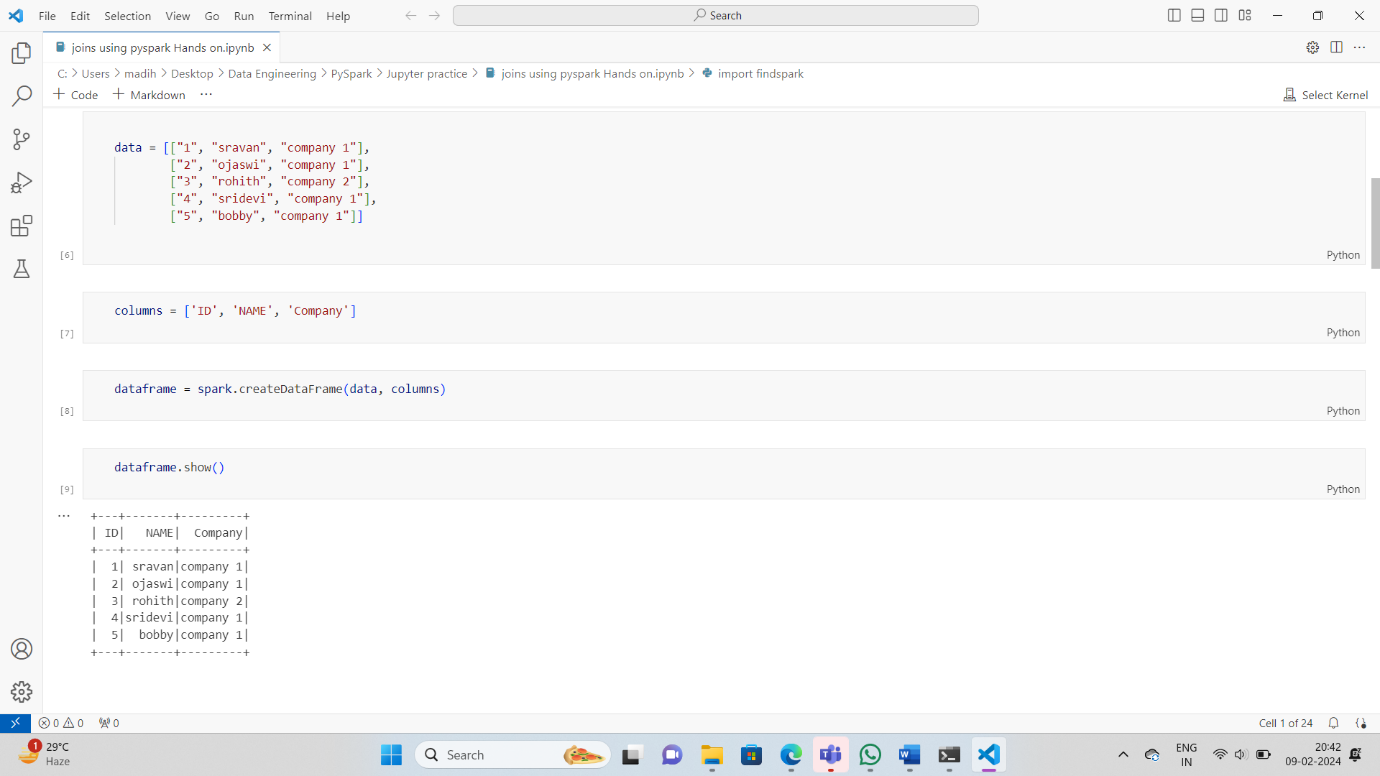
def lowerCase(str):  
 return str.lower()

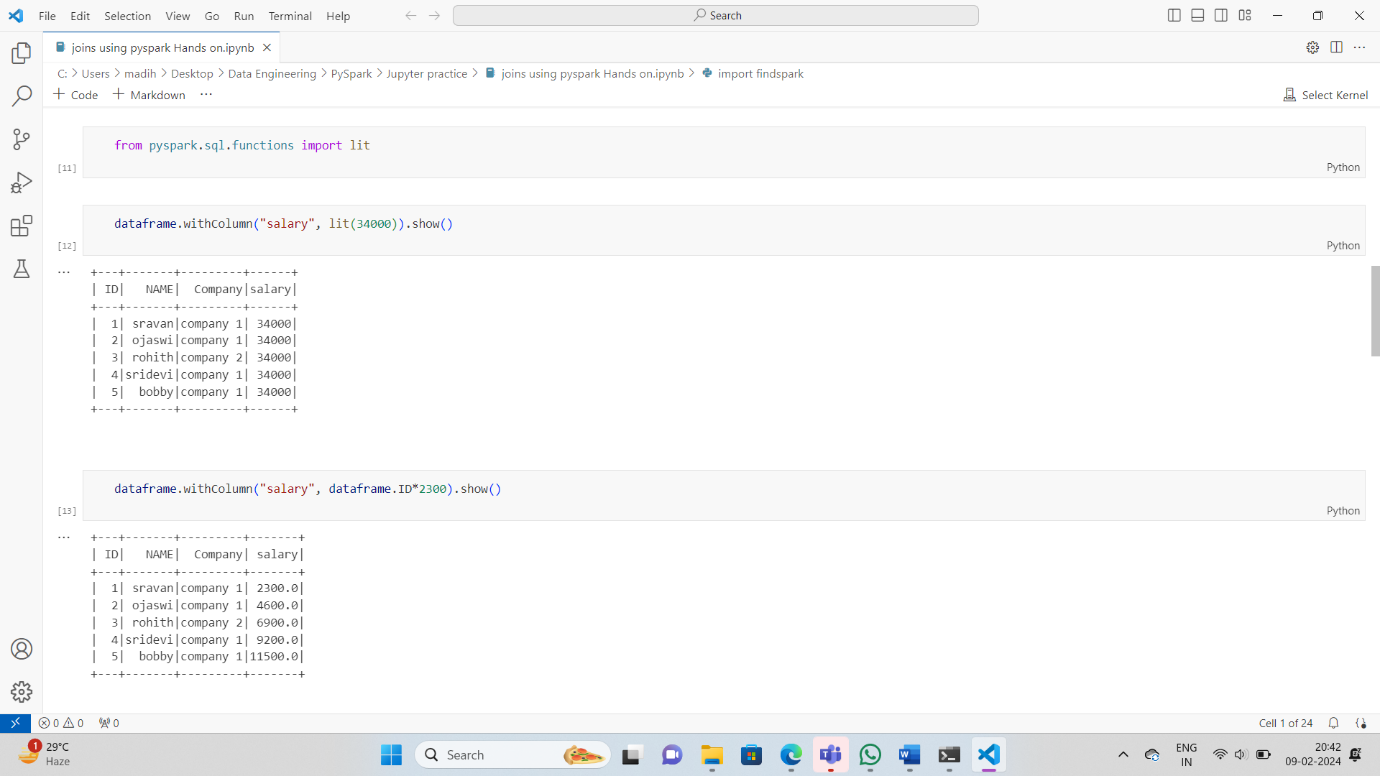
As function lowerCase() is created, let’s dwell into UDF code.

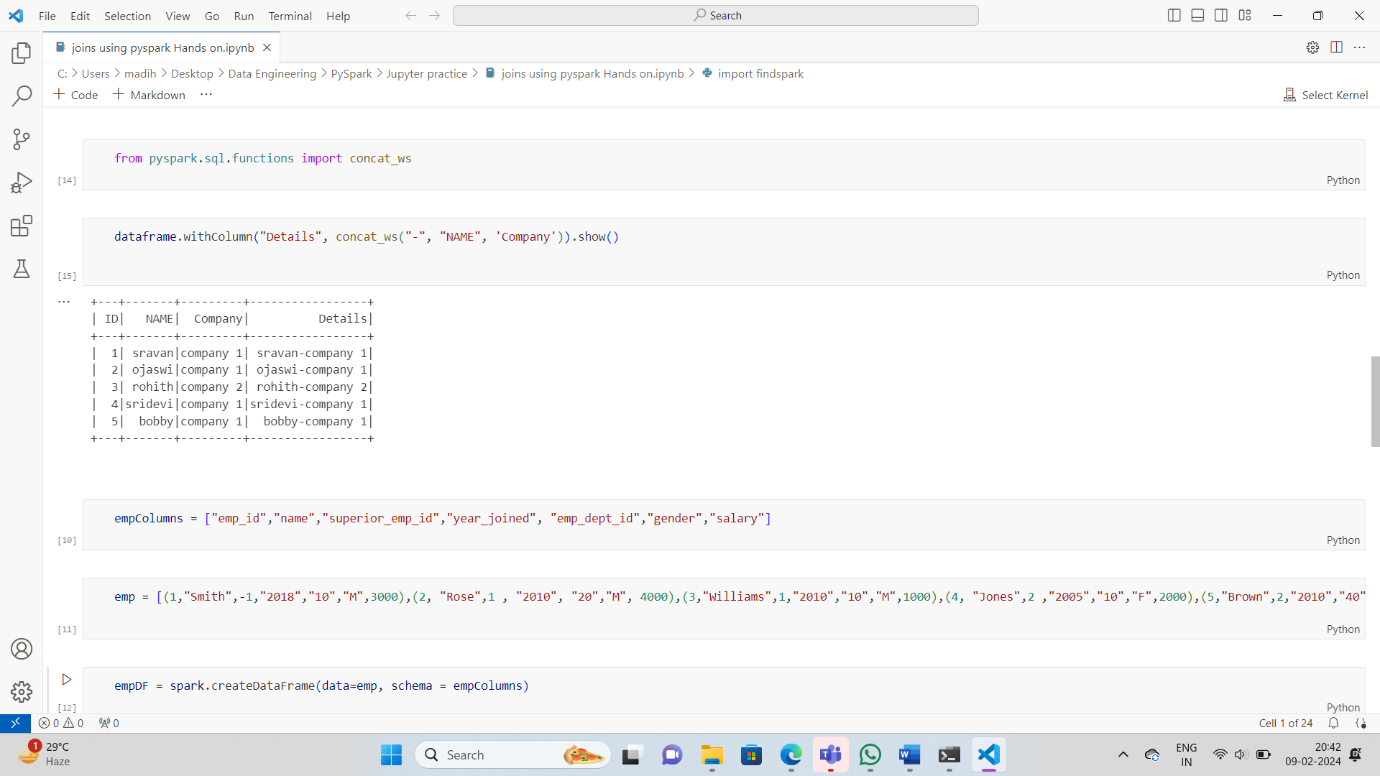
from pyspark.sql.functions import udfupperCaseUDF = udf(lambda z:upperCase(z))deptDF.withColumn("dept\_name", ulowerCaseUDF(deptDF["dept\_name"])).show()

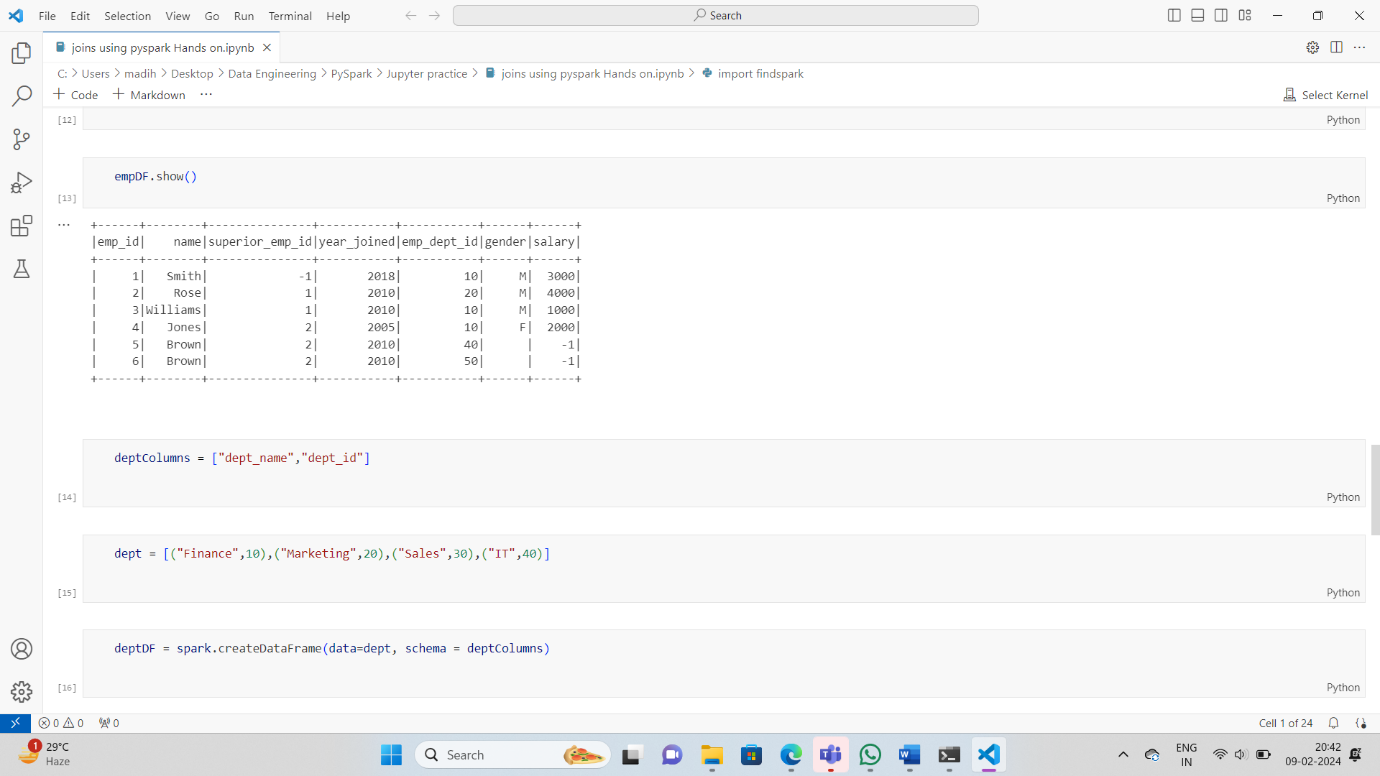


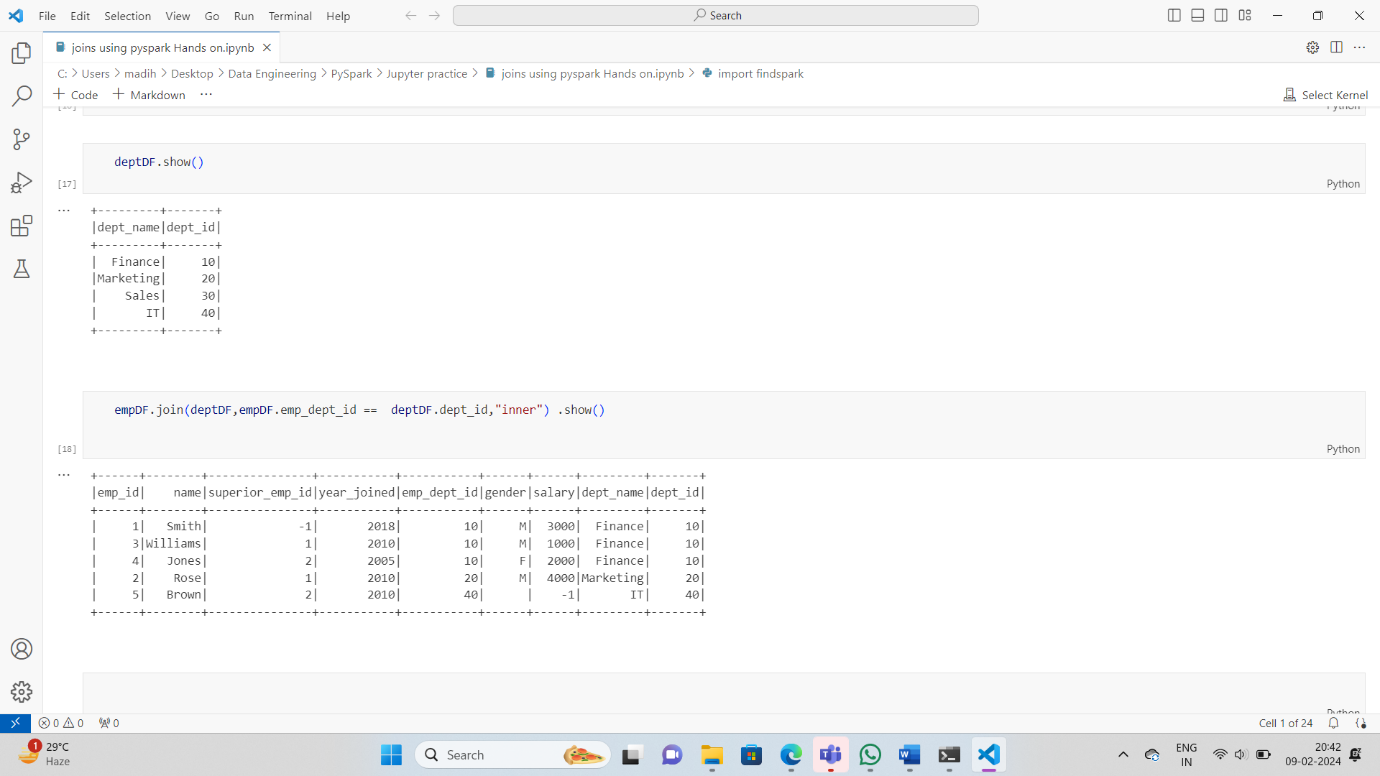
****

****

****

****

****

****

**Read Text file into PySpark Dataframe: -**

**There are three ways to read text files into PySpark DataFrame.**

* Using spark.read.text()
* Using spark.read.csv()
* Using spark.read.format().load()

Using these we can read a single text file, multiple files, and all files from a directory into Spark DataFrame and Dataset.

## **Method 1: Using spark.read.text()**

It is used to load text files into DataFrame whose schema starts with a string column. Each line in the text file is a new row in the resulting DataFrame. Using this method we can also read multiple files at a time.

***Syntax:****spark.read.text(paths)*

***Parameters:****This method accepts the following parameter as mentioned above and described below.*

* ***paths:****It is a string, or list of strings, for input path(s).*

***Returns:****DataFrame*

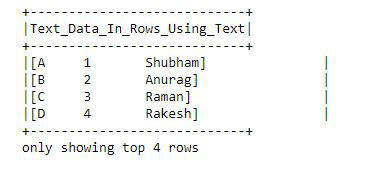
**Example : Read text file using spark.read.text().**

Here we will import the module and create a spark session and then read the file with **spark.read.text()**then create columns and split the data from the txt file show into a dataframe.

* Python3

|  |
| --- |
| **from** pyspark.sql **import** SparkSession    spark **=** SparkSession.builder.appName("DataFrame").getOrCreate()    df **=** spark.read.text("output.txt")    df.selectExpr("split(value, ' ') as\  Text\_Data\_In\_Rows\_Using\_Text").show(4,False) |

**Output:**



## **Method 2:** **Using spark.read.csv()**

It is used to load text files into DataFrame. Using this method we will go through the input once to determine the input schema if inferSchema is enabled. To avoid going through the entire data once, disable inferSchema option or specify the schema explicitly using the schema.

***Syntax:****spark.read.csv(path)*

***Returns:****DataFrame*

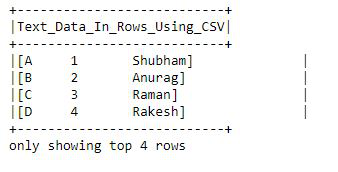
**Example: Read text file using spark.read.csv().**

First, import the modules and create a spark session and then read the file with spark.read.csv(), then create columns and split the data from the txt file show into a dataframe.

* Python3

|  |
| --- |
| **from** pyspark.sql **import** SparkSession    spark **=** SparkSession.builder.getOrCreate()    df **=** spark.read.csv("output.txt")    df.selectExpr("split(\_c0, ' ')\  as Text\_Data\_In\_Rows\_Using\_CSV").show(4,False) |

**Output:**



## **Method 3: Using spark.read.format()**

It is used to load text files into DataFrame. The **.format()**specifies the input data source format as “text”. The **.load()** loads data from a data source and returns DataFrame.

***Syntax:****spark.read.format(“text”).load(path=None, format=None, schema=None, \*\*options)*

***Parameters:****This method accepts the following parameter as mentioned above and described below.*

* ***paths :****It is a string, or list of strings, for input path(s).*
* ***format :****It is an optional string for format of the data source. Default to ‘parquet’.*
* ***schema :****It is an optional pyspark.sql.types.StructType for the input schema.*
* ***options :****all other string options*

***Returns:****DataFrame*

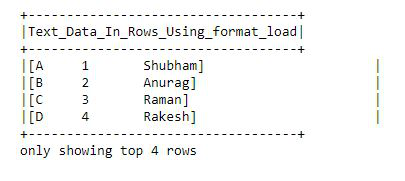
**Example: Read text file using spark.read.format().**

First, import the modules and create a spark session and then read the file with **spark.read.format()**, then create columns and split the data from the txt file show into a dataframe.

* Python3

|  |
| --- |
| **from** pyspark.sql **import** SparkSession    spark **=** SparkSession.builder.getOrCreate()    df **=** spark.read.format("text").load("output.txt")    df.selectExpr("split(value, ' ')\  as Text\_Data\_In\_Rows\_Using\_format\_load").show(4,False) |

**Output:**



# How to add a new column to a PySpark DataFrame ?

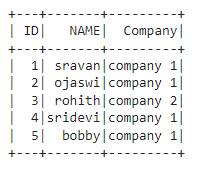
**Create the first data frame for demonstration:**

Here, we will be creating the sample data frame which we will be used further to demonstrate the approach purpose.

* Python3

|  |
| --- |
| # importing module  **import** pyspark    # importing sparksession from pyspark.sql module  **from** pyspark.sql **import** SparkSession    # creating sparksession and giving an app name  spark **=** SparkSession.builder.appName('sparkdf').getOrCreate()    # list  of employee data  data **=** [["1", "sravan", "company 1"],          ["2", "ojaswi", "company 1"],          ["3", "rohith", "company 2"],          ["4", "sridevi", "company 1"],          ["5", "bobby", "company 1"]]    # specify column names  columns **=** ['ID', 'NAME', 'Company']    # creating a dataframe from the lists of data  dataframe **=** spark.createDataFrame(data, columns)    dataframe.show() |

**Output:**



## Method 1: Add New Column With Constant Value

In this approach to add a new column with constant values, the user needs to call the lit() function parameter of the withColumn() function and pass the required parameters into these functions. Here, the lit() is available in pyspark.sql. Functions module.

**Syntax**:

dataframe.withColumn("column\_name", lit(value))

**where,**

* dataframe is the pyspark input dataframe
* column\_name is the new column to be added
* value is the constant value to be assigned to this column

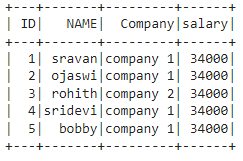
**Example:**

In this example, we add a column named salary with a value of 34000 to the above dataframe using the withColumn() function with the lit() function as its parameter in the python programming language.

* Python3

|  |
| --- |
| # importing module  **import** pyspark    # import lit function  **from** pyspark.sql.functions **import** lit    # importing sparksession from pyspark.sql module  **from** pyspark.sql **import** SparkSession    # creating sparksession and giving an app name  spark **=** SparkSession.builder.appName('sparkdf').getOrCreate()    # list  of employee data  data **=** [["1", "sravan", "company 1"],          ["2", "ojaswi", "company 1"],          ["3", "rohith", "company 2"],          ["4", "sridevi", "company 1"],          ["5", "bobby", "company 1"]]    # specify column names  columns **=** ['ID', 'NAME', 'Company']    # creating a dataframe from the lists of data  dataframe **=** spark.createDataFrame(data, columns)    # Add a column named salary with value as 34000  dataframe.withColumn("salary", lit(34000)).show() |

**Output:**



## Method 2: Add Column Based on Another Column of DataFrame

Under this approach, the user can add a new column based on an existing column in the given dataframe.

### **Example 1: Using withColumn() method**

Here, under this example, the user needs to specify the existing column using the withColumn() function with the required parameters passed in the python programming language.

**Syntax**:

dataframe.withColumn("column\_name", dataframe.existing\_column)

**where,**

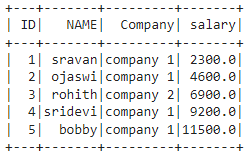
* dataframe is the input dataframe
* column\_name is the new column
* existing\_column is the column which is existed

In this example, we are adding a column named salary from the ID column with multiply of 2300 using the withColumn() method in the python language,

* Python3

|  |
| --- |
| # importing module  **import** pyspark    # import lit function  **from** pyspark.sql.functions **import** lit    # importing sparksession from pyspark.sql module  **from** pyspark.sql **import** SparkSession    # creating sparksession and giving an app name  spark **=** SparkSession.builder.appName('sparkdf').getOrCreate()    # list  of employee data  data **=** [["1", "sravan", "company 1"],          ["2", "ojaswi", "company 1"],          ["3", "rohith", "company 2"],          ["4", "sridevi", "company 1"],          ["5", "bobby", "company 1"]]    # specify column names  columns **=** ['ID', 'NAME', 'Company']    # creating a dataframe from the lists of data  dataframe **=** spark.createDataFrame(data, columns)    # Add a column named salary from ID column with multiply of 2300  dataframe.withColumn("salary", dataframe.ID**\***2300).show() |

**Output:**



### **Example**2 : Using concat\_ws()

Under this example, the user has to concat the two existing columns and make them as a new column by importing this method from pyspark.sql.functions module.

**Syntax**:

*dataframe.withColumn(“column\_name”, concat\_ws(“Separator”,”existing\_column1″,’existing\_column2′))*

where,

* dataframe is the input dataframe
* column\_name is the new column name
* existing\_column1 and existing\_column2 are the two columns to be added with Separator to make values to the new column
* Separator is like the operator between values with two columns

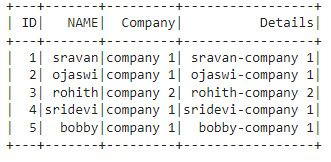
**Example:**

In this example, we add a column named Details from Name and Company columns separated by “-” in the python language.

* Python3

|  |
| --- |
| # importing module  **import** pyspark    # import concat\_ws function  **from** pyspark.sql.functions **import** concat\_ws    # importing sparksession from pyspark.sql module  **from** pyspark.sql **import** SparkSession    # creating sparksession and giving an app name  spark **=** SparkSession.builder.appName('sparkdf').getOrCreate()    # list  of employee data  data **=** [["1", "sravan", "company 1"],          ["2", "ojaswi", "company 1"],          ["3", "rohith", "company 2"],          ["4", "sridevi", "company 1"],          ["5", "bobby", "company 1"]]    # specify column names  columns **=** ['ID', 'NAME', 'Company']    # creating a dataframe from the lists of data  dataframe **=** spark.createDataFrame(data, columns)    # Add a column named Details from Name and Company columns separated by -  dataframe.withColumn("Details", concat\_ws("-", "NAME", 'Company')).show() |

**Output:**



## Method 3: Add Column When not Exists on DataFrame

In this method, the user can add a column when it is not existed by adding a column with the lit() function and checking using if the condition.

**Syntax:**

if 'column\_name' not in dataframe.columns:

dataframe.withColumn("column\_name",lit(value))

**where,**

* dataframe. columns are used to get the column names

**Example:**

In thisexample, we add a column of the salary to 34000 using the if condition with the withColumn() and the lit() function.

* Python3

|  |
| --- |
| # importing module  **import** pyspark    # import concat\_ws and lit function  **from** pyspark.sql.functions **import** concat\_ws, lit    # importing sparksession from pyspark.sql module  **from** pyspark.sql **import** SparkSession    # creating sparksession and giving an app name  spark **=** SparkSession.builder.appName('sparkdf').getOrCreate()    # list  of employee data  data **=** [["1", "sravan", "company 1"],          ["2", "ojaswi", "company 1"],          ["3", "rohith", "company 2"],          ["4", "sridevi", "company 1"],          ["5", "bobby", "company 1"]]    # specify column names  columns **=** ['ID', 'NAME', 'Company']    # creating a dataframe from the lists of data  dataframe **=** spark.createDataFrame(data, columns)    # add salary column by checking its existence  **if** 'salary' **not** **in** dataframe.columns:      dataframe.withColumn("salary", lit(34000)).show() |

