**PYSPARK**

**Prepared By : S. Lochani Vilehya**

**Emp ID : 43317**

**INDEX:**

1. What is PySpark?

# Create Dataframe manually with hard coded

# Read CSV file in to Dataframe

# Write DataFrame into CSV file

# Read json file into DataFrame

# Write DataFrame into json file

# show() : to display Dataframe contents in Table

# withColumn() : Add new column or Change existing column data or type in DataFrame

# withColumnRenamed() usage

# StructType() & StructField()

# ArrayType()

# explode(), split(), array() & array\_contains()

# MapType Column

# map\_keys(), map\_values() & explode() functions to work with MapType Columns

# Row() class

# Column class

# when() & otherwise()

# alias(), asc(), desc(), cast() & like() functions on Columns of dataframe

# filter() & where()

# distinct() & dropDuplicates()

# orderBy() & sort()

# union() & unionAll()

# groupBy()

# GroupBy agg() function

# unionByName()

# select()

# join() | inner, left, right, full Joins, Left semi, Left anti & self-join

# pivot(), unpivot()

# fill() & fillna()

# sample()

# collect()

# DataFrame.transform() function

# pyspark.sql.functions.transform()

# createOrReplaceTempView()

# createOrReplaceGlobalTempView()

# UDF(user defined function)

# Convert RDD to Dataframe

# map() transformation

# flatMap() transformation

# partitionBy()

# from\_json() function to convert json string in to MapType and StructType

# to\_json()

# json\_tuple()

# get\_ json\_object()

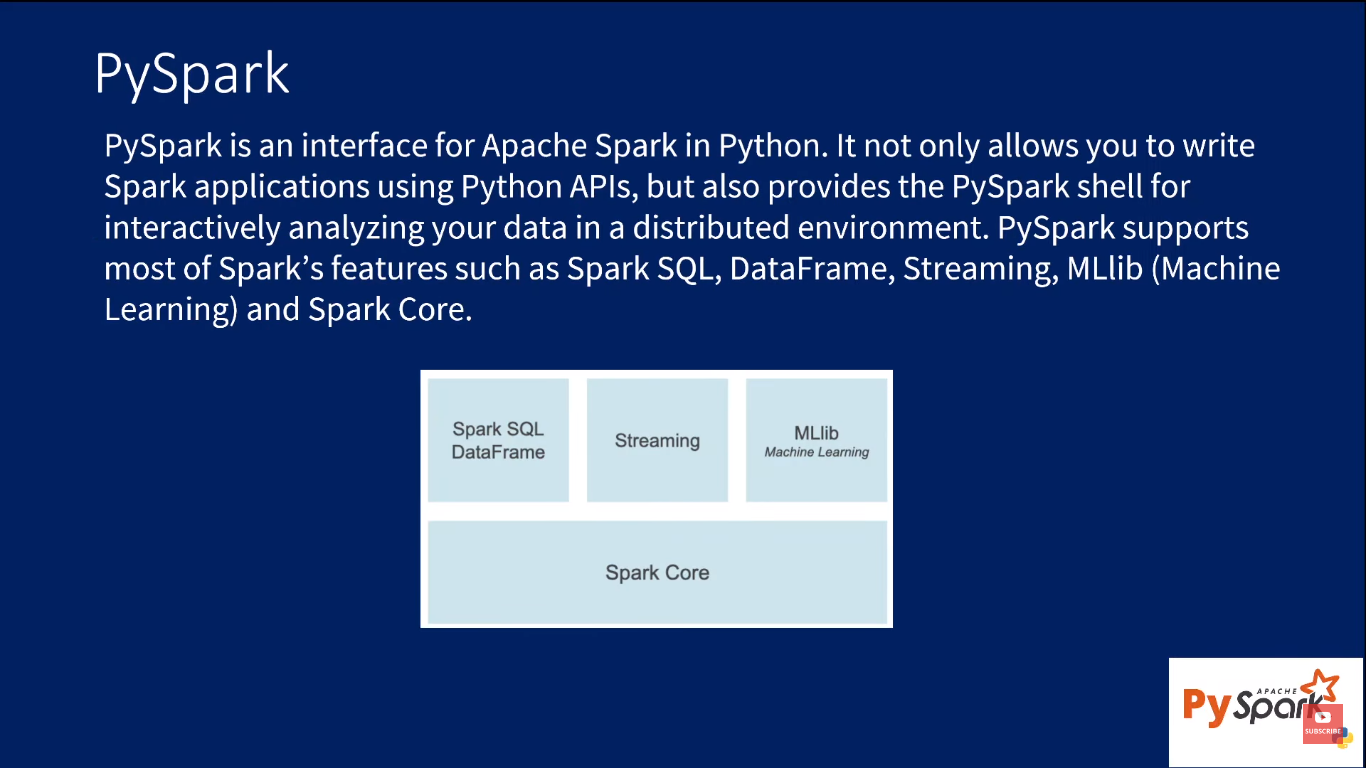
# Date functions | current\_date(), to\_date(), date\_format() , datediff(), months\_between(), add\_months(), date\_add(), month(), year() functions

# Timestamp Functions

# approx\_count\_distinct(), avg(), collect\_list(), collect\_set(), countDistinct(), count()

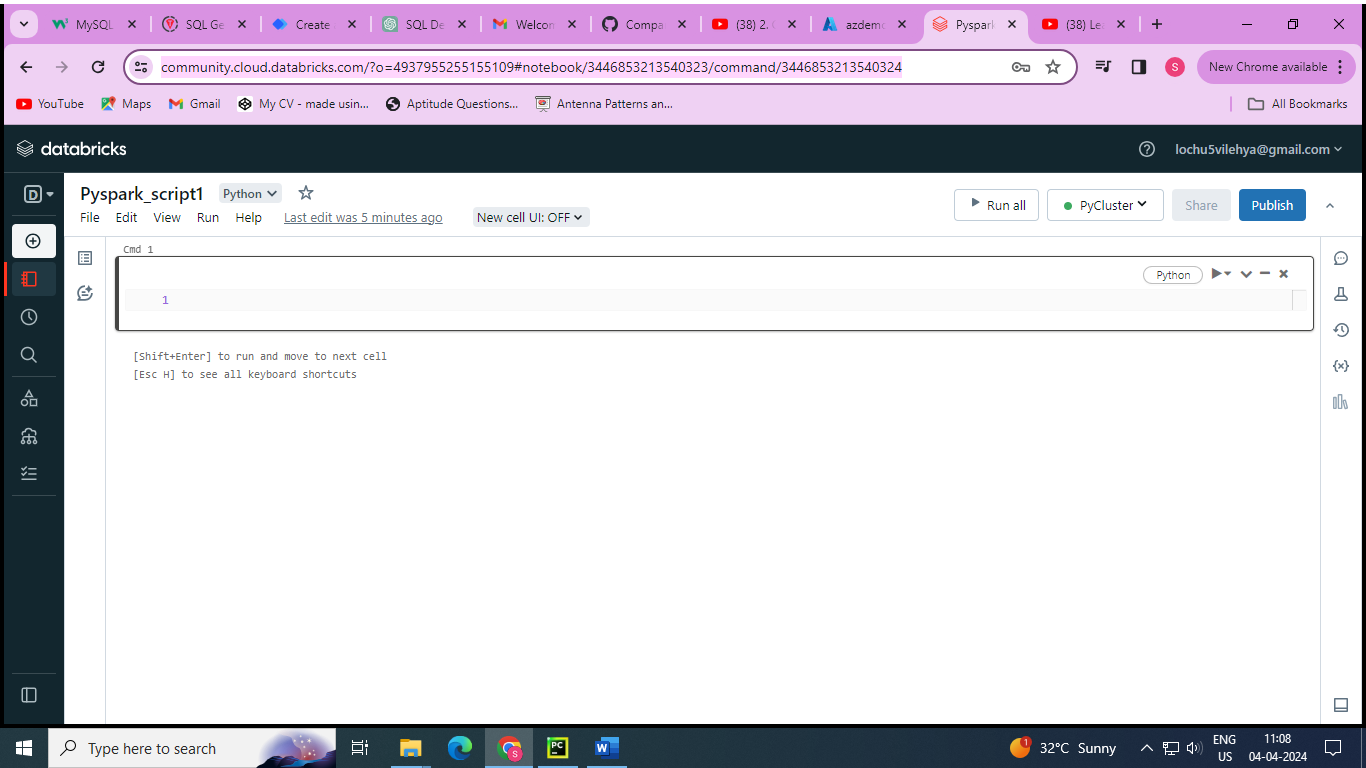
# row\_number(), rank(), dense\_rank() functions

# What is PySpark?



# Create Dataframe manually with hard coded values in PySpark

* Use the databricks community edition for free cost using the link [databricks community edition](https://community.cloud.databricks.com/?o=4937955255155109)
* Create a cluster using the option create compute in COMPUTE column
* Now using the +Create option create a notebook under this cluster



**type(spark)**

Out[1]: pyspark.sql.session.SparkSession

Command took 2.76 seconds -- by lochu5vilehya@gmail.com at 4/4/2024, 11:12:07 AM on PyCluster

**dir(spark)**

Out[2]: ['Builder', '\_\_annotations\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_enter\_\_', '\_\_eq\_\_', '\_\_exit\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_activeSession', '\_convert\_from\_pandas', '\_createFromLocal', '\_createFromLocalTrusted', '\_createFromRDD', '\_create\_dataframe', '\_create\_from\_pandas\_with\_arrow', '\_create\_rdd\_from\_local\_trusted', '\_create\_shell\_session', '\_getActiveSessionOrCreate', '\_get\_numpy\_record\_dtype', '\_inferSchema', '\_inferSchemaFromList', '\_instantiatedSession', '\_jconf', '\_jsc', '\_jsparkSession', '\_jvm', '\_repr\_html\_', '\_sc', '\_wrap\_data\_schema', '\_write\_to\_trusted\_path', 'builder', 'catalog', 'conf', 'createDataFrame', 'getActiveSession', 'newSession', 'range', 'read', 'readStream', 'sparkContext', 'sql', 'stop', 'streams', 'table', 'udf', 'version']

Command took 0.07 seconds -- by lochu5vilehya@gmail.com at 4/4/2024, 11:13:05 AM on PyCluster

**help(spark.createDataFrame)**

* To know about the function “createDataFrame” in spark

**d = [(1,'veena'),(2,'lochu')]**

**df = spark.createDataFrame(data = d)**

**df.show()**

\_1:long

\_2:string

+---+-----+

| \_1| \_2 |

+---+-----+

| 1 |veena|

| 2 |lochu|

+---+-----+

Command took 11.65 seconds -- by lochu5vilehya@gmail.com at 4/4/2024, 11:22:48 AM on PyCluster

**d = [(1,'veena'),(2,'lochu')]**

**df = spark.createDataFrame(data = d,schema=['id','name'])**

**df.show()**

**df.printSchema()**

id:long

name:string

+---+-----+

| id| name|

+---+-----+

| 1 |veena|

| 2 |lochu|

+---+-----+

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

Command took 1.03 seconds -- by lochu5vilehya@gmail.com at 4/4/2024, 11:26:03 AM on PyCluster

* To create a structure variable using pyspark

from pyspark.sql.types import \*

schema = StructType([StructField(name='id',dataType=IntegerType()),

StructField(name='name',dataType=StringType())])

type(schema)

Out[4]: pyspark.sql.types.StructType

* To set the data type of the column in table

from pyspark.sql.types import \*

d = [(1,'veena'),(2,'lochu')]

**schema = StructType([StructField(name='id',dataType=IntegerType()),**

**StructField(name='name',dataType=StringType())])**

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

df.show()

df.printSchema()

id:integer

name:string

+---+-----+

| id| name|

+---+-----+

| 1 |veena|

| 2 |lochu|

+---+-----+

root

|-- id: integer (nullable = true)

|-- name: string (nullable = true)

**schema = StructType([StructField(name='id',dataType=IntegerType()),**

**StructField(name='name',dataType=StringType())])**

**type(schema)**

Out[4]: pyspark.sql.types.StructType

**from pyspark.sql.types import \***

**d = [{'id':1,'name':'veena'},{'id':2,'name':'lochu'}]**

**df = spark.createDataFrame(data = d)**

**df.show()**

**df.printSchema()**

df:pyspark.sql.dataframe.DataFrame = [id: long, name: string]

+---+-----+

| id| name|

+---+-----+

| 1 |veena|

| 2 |lochu|

+---+-----+

root |-- id: long (nullable = true)

|-- name: string (nullable = true)

# Read CSV file in to Dataframe using PySpark

**Salary.csv File:**

**id,name,salary**

**1,veena,76000**

**2,lochu,55000**

**3,jyosthna,29090**

**4,vara,null**

**df = spark.read.csv(path='dbfs:/FileStore/Salary\_slip.csv')**

**df.display()**

**df.printSchema()**

\_c0 \_c1 \_c2

id name salary

1 veena 76000

2 lochu 55000

3 jyosthna 29090

4 vara null

root

|-- \_c0: string (nullable = true)

|-- \_c1: string (nullable = true)

|-- \_c2: string (nullable = true)

**df = spark.read.csv(path='dbfs:/FileStore/Salary\_slip.csv',header=True)**

**df.display()**

**df.printSchema()**

id name salary

1 veena 76000

2 lochu 55000

3 jyosthna 29090

4 vara null

root

|-- id: string (nullable = true)

|-- name: string (nullable = true)

|-- salary: string (nullable = true)

**df = spark.read.format('csv').option(key='header',value=True).load(path='dbfs:/FileStore/Salary\_slip.csv')**

**display(df)**

**df.printSchema()**

id name salary

1 veena 76000

2 lochu 55000

3 jyosthna 29090

4 vara null

root |-- id: string (nullable = true) |-- name: string (nullable = true) |-- salary: string (nullable = true)

* To read more than one csv file in the same folder

**df = spark.read.csv(path='dbfs:/FileStore/',header=True)**

**df.display()**

**df.printSchema()**

* To read to files from 2 different folders (filestrore,filestore1)

**df = spark.read.csv(path=['dbfs:/FileStore/Salary\_slip.csv', 'dbfs:/FileStore1/Salary\_slip1.csv'])**

**df.display()**

**df.printSchema()**

* Apply schema pf struct type to mention the datatypes of the fields for the loaded csv files

**from pyspark.sql.types import \***

**struct = StructType().add(field = 'id',data\_type=IntegerType())\**

**.add(field='name',data\_type=StringType())\**

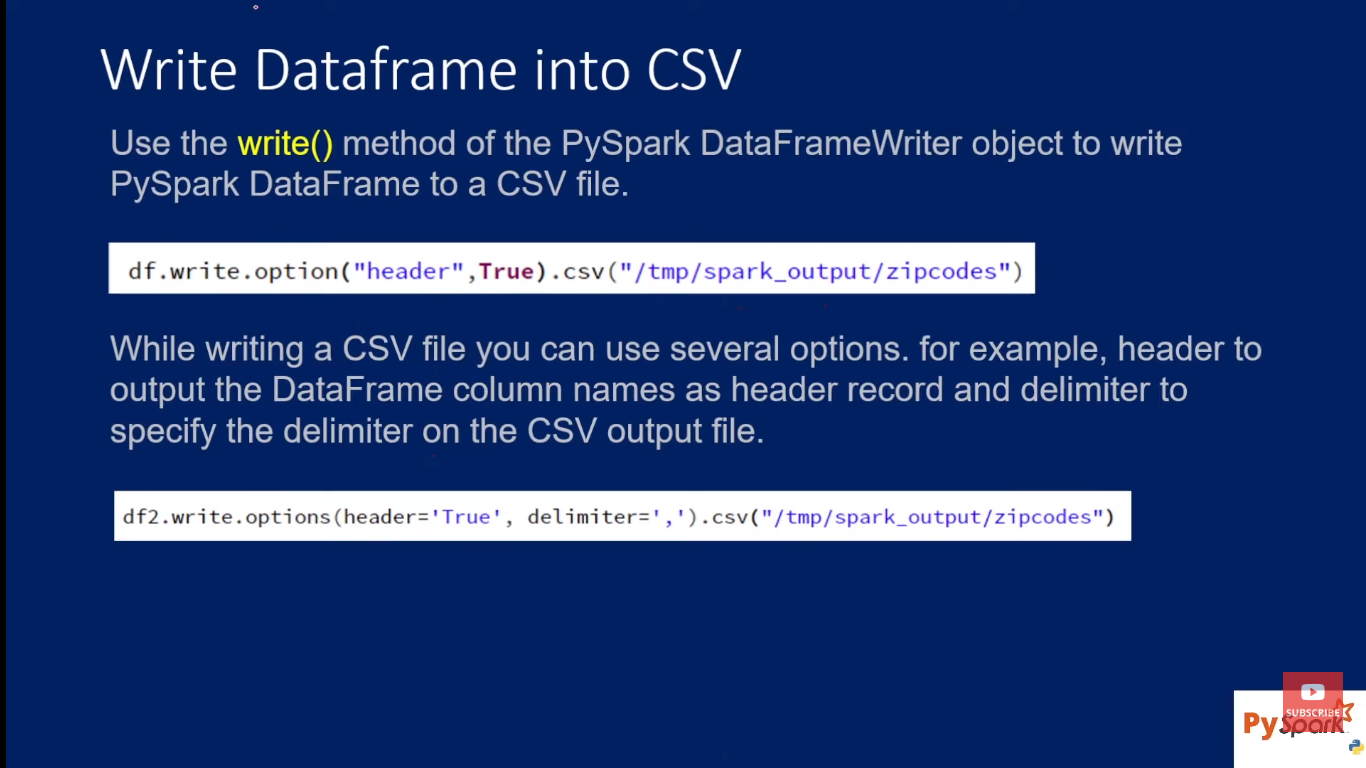
**.add(field='salary',data\_type=IntegerType())**

**df= spark.read.csv(path='dbfs:/FileStore/Salary\_slip.csv',schema=struct,header=True)**

**df.display()**

**df.printSchema()**

# Write DataFrame into CSV file using PySpark

****

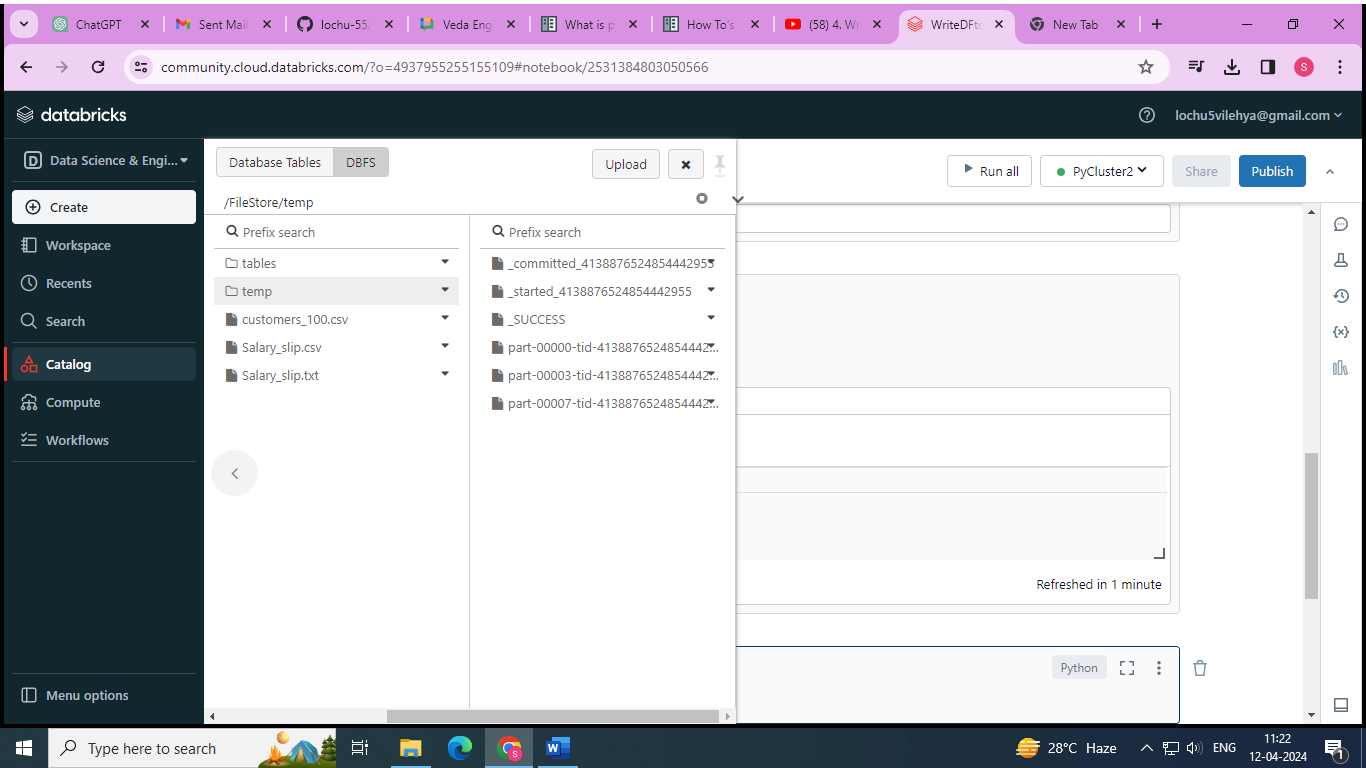
**d = [(1,'veena'),(2,'lochu')]**

**schema = ['id','name']**

**df = spark.createDataFrame(data = d,schema=schema)**

**df.display()**

**df.write.csv(path='dbfs:/FileStore/temp',header=True)**



* The files are generated ad part files and extra logs files are also generated

**df = spark.read.csv(path='dbfs:/FileStore/temp',header=True)**

**display(df)**

id name

1 veena

2 lochu

**help(df.write.csv)**

Help on method csv in module pyspark.sql.readwriter: csv(path: str, mode: Optional[str] = None, compression: Optional[str] = None, sep: Optional[str] = None, quote: Optional[str] = None, escape: Optional[str] = None, header: Union[bool, str, NoneType] = None, nullValue: Optional[str] = None, escapeQuotes: Union[bool, str, NoneType] = None, quoteAll: Union[bool, str, NoneType] = None, dateFormat: Optional[str] = None, timestampFormat: Optional[str] = None, ignoreLeadingWhiteSpace: Union[bool, str, NoneType] = None, ignoreTrailingWhiteSpace: Union[bool, str, NoneType] = None, charToEscapeQuoteEscaping: Optional[str] = None, encoding: Optional[str] = None, emptyValue: Optional[str] = None, lineSep: Optional[str] = None) -> None method of pyspark.sql.readwriter.DataFrameWriter instance

Saves the content of the :class:`DataFrame` in CSV format at the specified path.

.. versionadded:: 2.0.0

.. versionchanged:: 3.4.0

Support Spark Connect.

Parameters

----------

path : str the path in any Hadoop supported file system

mode : str, optional specifies the behavior of the save operation when data already exists.

\* ``append``: Append contents of this :class:`DataFrame` to existing data.

\* ``overwrite``: Overwrite existing data.

\* ``ignore``: Silently ignore this operation if data already exists.

\* ``error`` or ``errorifexists`` (default case): Throw an exception if data already \ exists.

Other Parameters

----------------

Extra options For the extra options, refer to `Data Source Option <https://spark.apache.org/docs/latest/sql-data-sources-csv.html#data-source-option>`\_ for the version you use. .. # noqa Examples -------- Write a DataFrame into a CSV file and read it back.

>>> import tempfile

>>> with tempfile.TemporaryDirectory() as d:

... # Write a DataFrame into a CSV file

... df = spark.createDataFrame([{"age": 100, "name": "Hyukjin Kwon"}])

... df.write.csv(d, mode="overwrite")

... ... # Read the CSV file as a DataFrame with 'nullValue' option set to 'Hyukjin Kwon'.

... spark.read.schema(df.schema).format("csv").option(

... "nullValue", "Hyukjin Kwon").load(d).show()

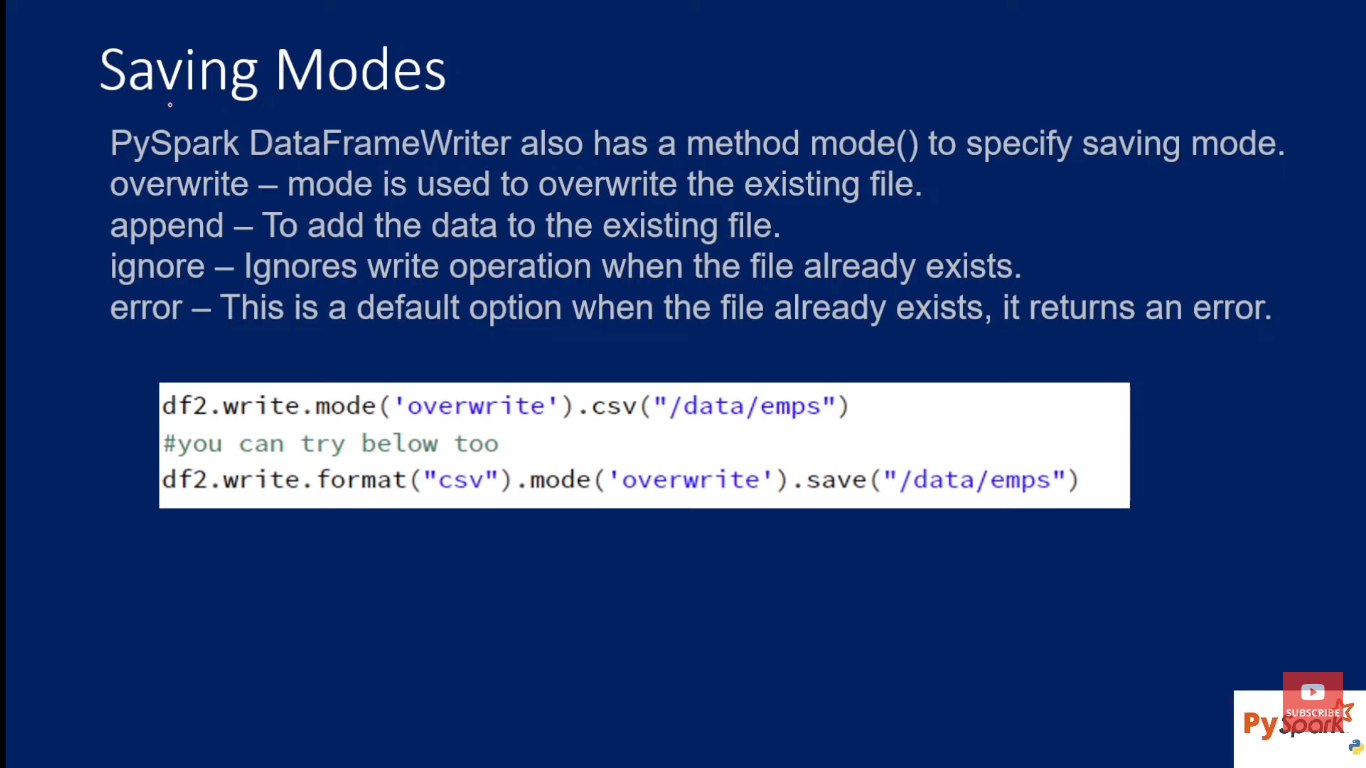
+---+----+

|age|name|

+---+----+

|100|null|

+---+----+

****

**df.write.csv(path='dbfs:/FileStore/temp',header=True,mode='append')**

**df = spark.read.csv(path='dbfs:/FileStore/temp',header=True)**

**display(df)**

id name

1 veena

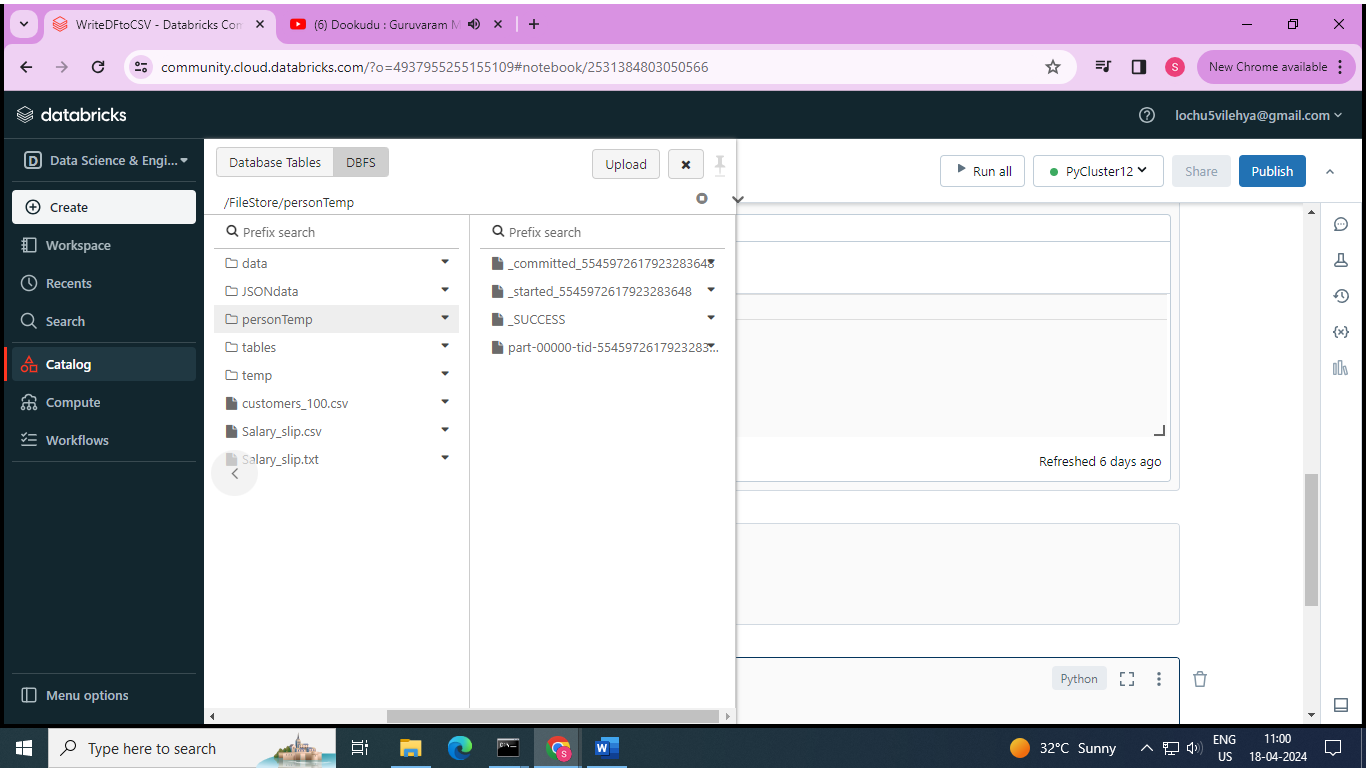
2 lochu

1 veena

2 lochu

* To save all part files as single .csv file

**df.coalesce(1).write.csv('FileStore/personTemp',header=True) #it creates one part file**



**fn = dbutils.fs.ls('FileStore/personTemp')**

**print(fn)**

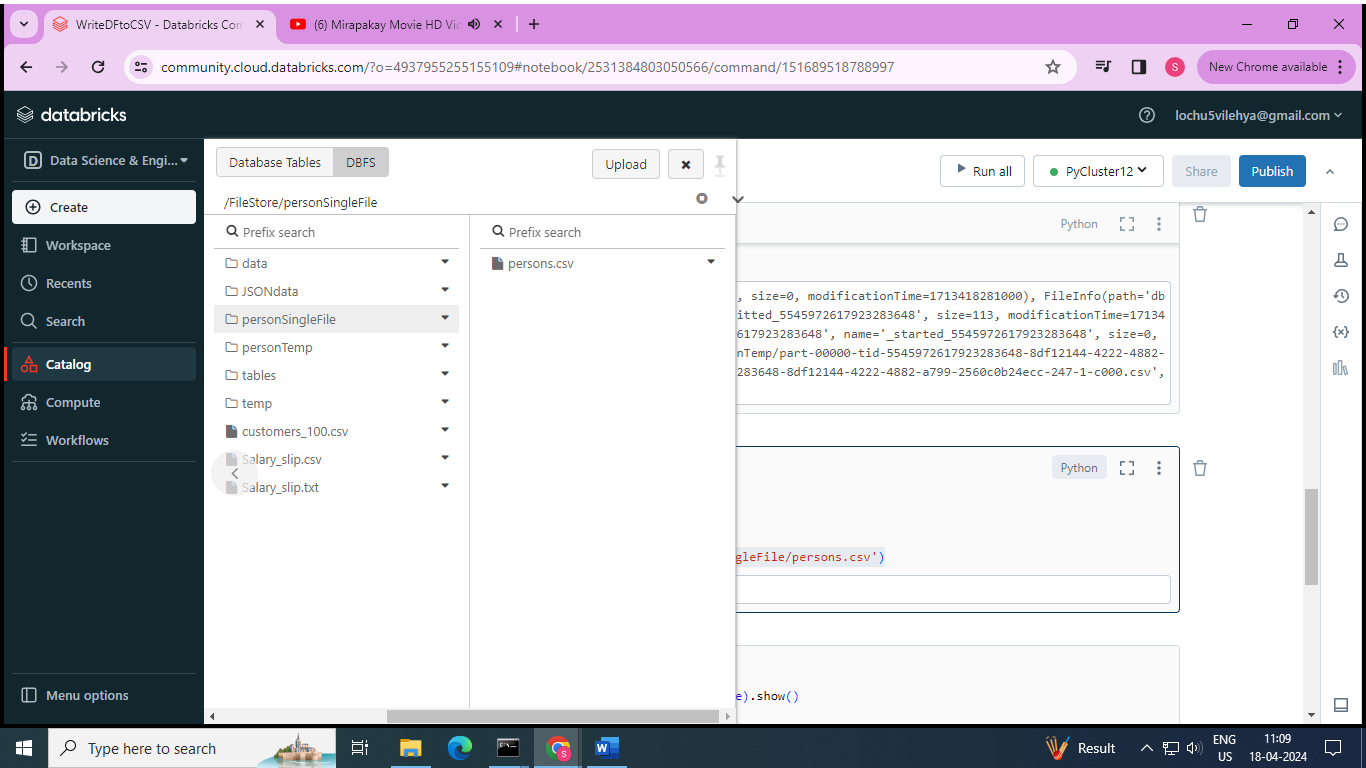
[FileInfo(path='dbfs:/FileStore/personTemp/\_SUCCESS', name='\_SUCCESS', size=0, modificationTime=1713418281000), FileInfo(path='dbfs:/FileStore/personTemp/\_committed\_5545972617923283648', name='\_committed\_5545972617923283648', size=113, modificationTime=1713418280000), FileInfo(path='dbfs:/FileStore/personTemp/\_started\_5545972617923283648', name='\_started\_5545972617923283648', size=0, modificationTime=1713418279000), FileInfo(path='dbfs:/FileStore/personTemp/part-00000-tid-5545972617923283648-8df12144-4222-4882-a799-2560c0b24ecc-247-1-c000.csv', name='part-00000-tid-5545972617923283648-8df12144-4222-4882-a799-2560c0b24ecc-247-1-c000.csv', size=24, modificationTime=1713418280000)]

**for filename in fn:**

**if filename.name.endswith('.csv'):**

**name = filename.name**

**dbutils.fs.cp('/FileStore/personTemp/' + name, 'FileStore/personSingleFile/persons.csv')**



**spark.read.csv('/FileStore/personSingleFile/persons.csv',header=True).show()**

+---+-----+

| id| name|

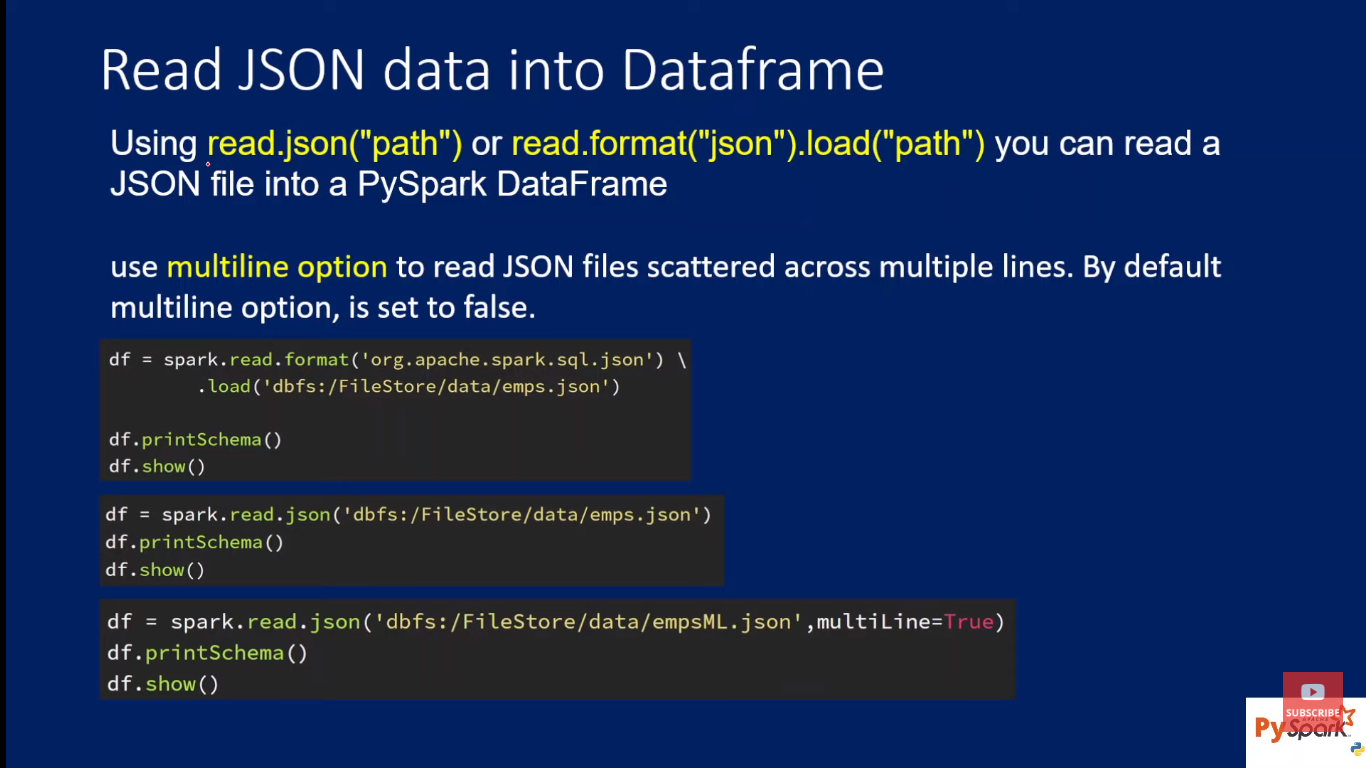
+---+-----+

| 1|veena|

| 2|lochu|

+---+-----+

# Read json file into DataFrame using Pyspark



**🡪Reading a single line json**

**Emp.json:**

{'id':1,'name':'veena','salary':2300}

{'id':2,'name':'lochu','salary':4300}

{'id':3,'name':'vara','salary':6300}

**df = spark.read.json(path='dbfs:/FileStore/tables/emp.json')**

**display(df)**

**df.printSchema()**

id name salary

1 veena 2300

2 lochu 4300

3 vara 6300

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

|-- salary: long (nullable = true)

* **Reading a multiline json**

**empML.json:**

[

{

'id':1,

'name':'veena',

'salary':2300

},

{

'id':2,

'name':'lochu',

'salary':4300

},

{

'id':3,

'name':'vara',

'salary':6300

}

]

**df = spark.read.json(path='dbfs:/FileStore/tables/empML.json',multiLine=True)**

**display(df)**

**df.printSchema()**

id name salary

1 veena 2300

2 lochu 4300

3 vara 6300

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

|-- salary: long (nullable = true)



**from pyspark.sql.types import \***

**struct = StructType().add(field = 'id',data\_type=IntegerType())\**

**.add(field='name',data\_type=StringType())\**

**.add(field='salary',data\_type=IntegerType())**

**df = spark.read.json(path='dbfs:/FileStore/tables/emp.json',schema= struct)**

**display(df)**

**df.printSchema()**

id name salary

1 veena 2300

2 lochu 4300

3 vara 6300

root

|-- id: integer (nullable = true)

|-- name: string (nullable = true)

|-- salary: integer (nullable = true)

# Write DataFrame into json file using PySpark

**d = [(1,'veena'),(2,'lochu')]**

**schema = ['id','name']**

**df = spark.createDataFrame(data = d,schema=schema)**

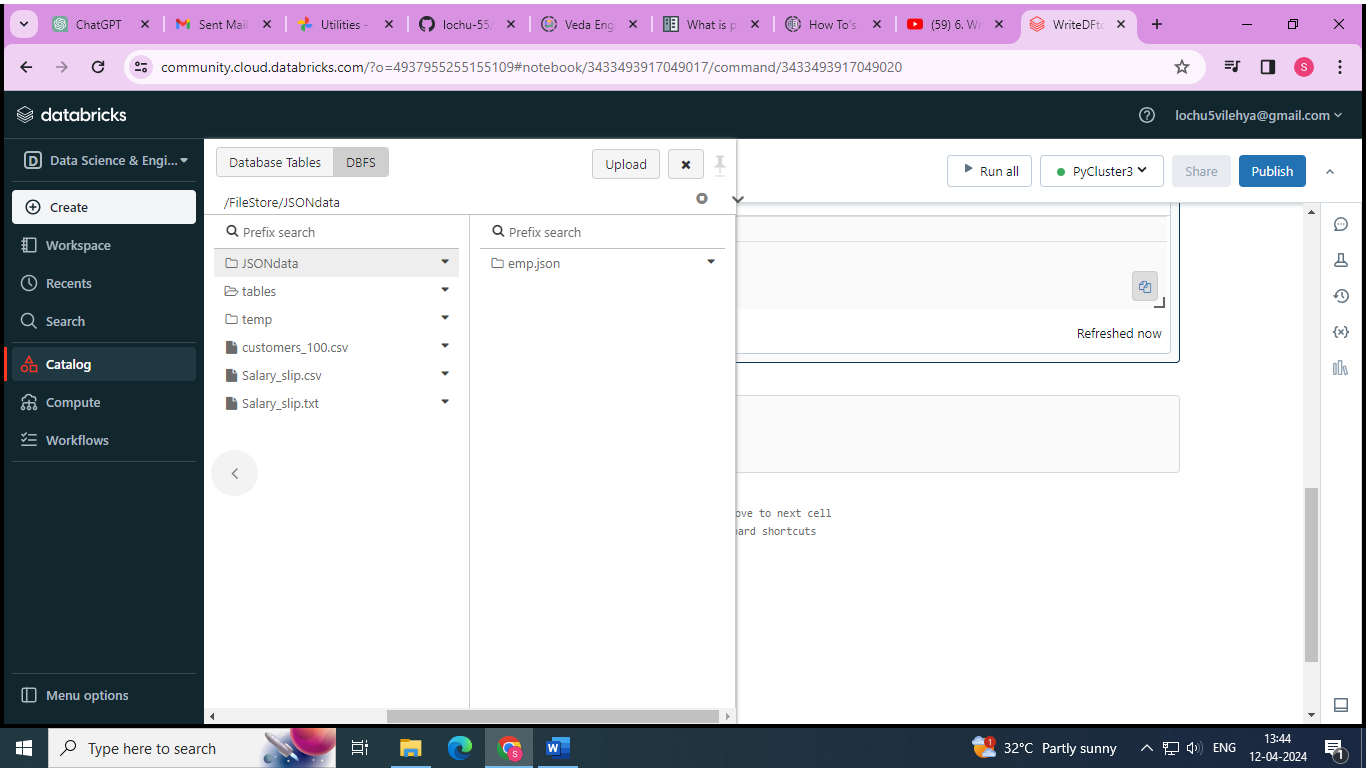
**df.display()**

id name

1 veena

2 lochu

**df.write.json(path='dbfs:/FileStore/JSONdata/emp.json')**

****

**data = spark.read.json(path='dbfs:/FileStore/JSONdata/emp.json')**

**display(data)**

id name

1 veena

2 lochu

# show() in Pyspark to display Dataframe contents in Table

**d=[(1,'jhvfuguydjhfvsdhvfdherewewfrfgfvjh'),(2,'bdhjbjfbjhbfjrhfgjergewdewdwed'),(3,'kwjdjnefnekjbefgrfguyfguyrfgufgruyfgerufg')]**

**schema = ['id','name']**

**df = spark.createDataFrame(data = d,schema = schema)**

**df.show()**

+---+--------------------+

| id| name|

+---+--------------------+

| 1 |jhvfuguydjhfvsdhv...|

| 2 |bdhjbjfbjhbfjrhfg...|

| 3 |kwjdjnefnekjbefgr...|

+---+--------------------+

**df.show(truncate=False)**

+---+-----------------------------------------+

|id | name |

+---+-----------------------------------------+

|1 |jhvfuguydjhfvsdhvfdherewewfrfgfvjh |

|2 |bdhjbjfbjhbfjrhfgjergewdewdwed |

|3 |kwjdjnefnekjbefgrfguyfguyrfgufgruyfgerufg|

+---+-----------------------------------------+

**df.show(truncate = 8)**

+---+--------+

| id| name|

+---+--------+

| 1 |jhvfu...|

| 2 |bdhjb...|

| 3 |kwjdj...|

+---+--------+

**df.show(n=2,truncate = False)**

+---+----------------------------------+

|id | name |

+---+----------------------------------+

|1 |jhvfuguydjhfvsdhvfdherewewfrfgfvjh|

|2 | bdhjbjfbjhbfjrhfgjergewdewdwed |

+---+----------------------------------+

only showing top 2 rows

**df.show(truncate = False,vertical=True)**

-RECORD 0-----------------------------------------

id | 1 name | jhvfuguydjhfvsdhvfdherewewfrfgfvjh

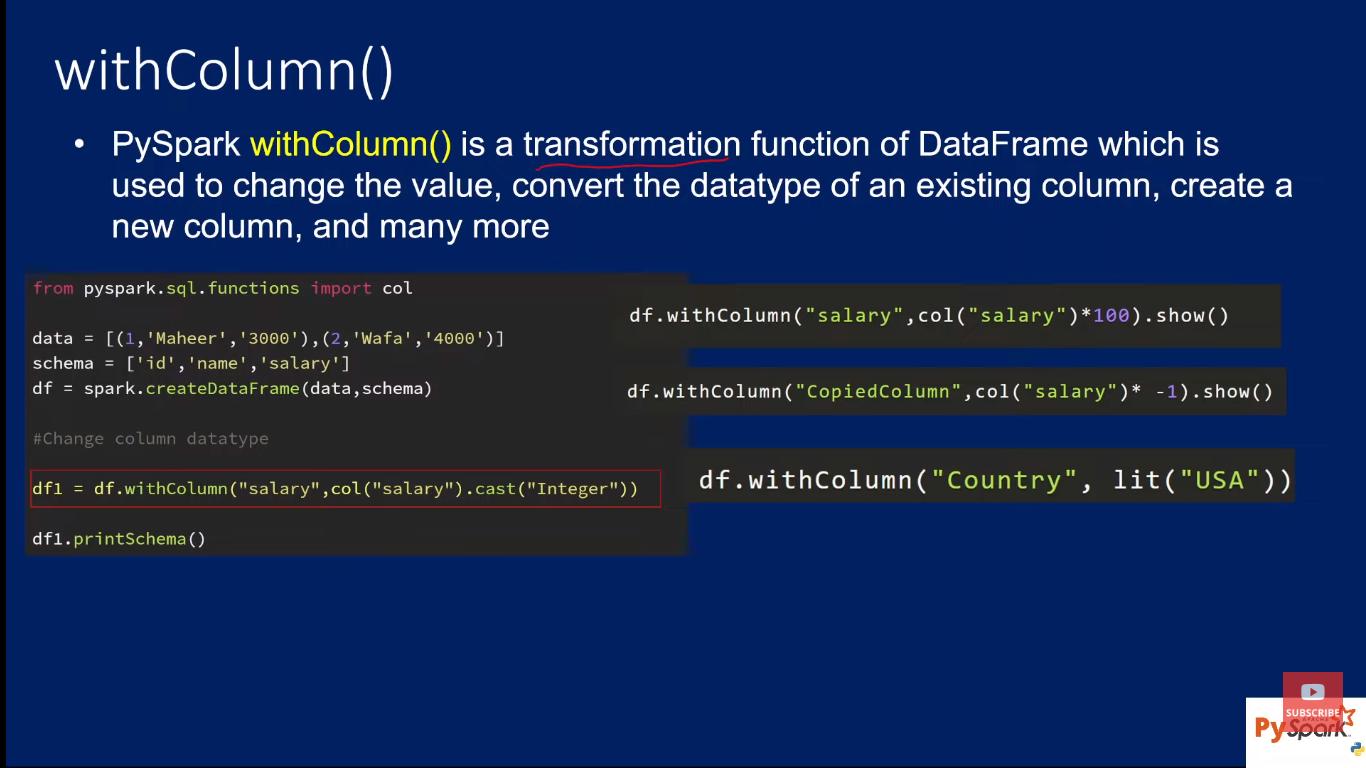
-RECORD 1-----------------------------------------

id | 2 name | bdhjbjfbjhbfjrhfgjergewdewdwed

-RECORD 2-----------------------------------------

id | 3 name | kwjdjnefnekjbefgrfguyfguyrfgufgruyfgerufg

# withColumn() in PySpark | Add new column or Change existing column data or type in DataFrame



**from pyspark.sql.functions import col**

**d = [(1,'veena',353455),(2,'lochu',234234)]**

**schema = ['id','name','salary']**

**df = spark.createDataFrame(data = d,schema=schema)**

**df1 = df.withColumn(colName='salary',col=col('salary').cast('Integer'))**

**df1.printSchema()**

**df1.show()**

**df2 = df1.withColumn('salary',col('salary')+5)**

**df2.show()**

**df3 = df2.withColumn('country',lit('india'))**

**df3.show()**

**df4 = df3.withColumn('copiedSalary',col('salary'))**

**df4.show()**

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

|-- salary: integer (nullable = true)

+---+-----+------+

| id| name|salary|

+---+-----+------+

| 1|veena |353455|

| 2|lochu |234234|

+---+-----+------+

+---+-----+------+

| id| name|salary|

+---+-----+------+

| 1|veena |353460|

| 2|lochu |234239|

+---+-----+------+

+---+-----+------+-------+

| id| name|salary|country|

+---+-----+------+-------+

| 1|veena|353460 | india |

| 2|lochu|234239 | india |

+---+-----+------+-------+

+---+-----+------+-------+------------+

| id| name|salary|country|copiedSalary|

+---+-----+------+-------+------------+

| 1 |veena|353460| india | 353460|

| 2 |lochu|234239| india | 234239|

+---+-----+------+-------+------------+

# withColumnRenamed() usage in PySpark

**df5 = df4.withColumnRenamed('salary','salary\_amt')**

**df5.show()**

+---+-----+----------+-------+------------+

| id| name|salary\_amt|country|copiedSalary|

+---+-----+----------+-------+------------+

| 1 |veena| 353460| india | 353460|

| 2 |lochu| 234239| india | 234239|

+---+-----+----------+-------+------------+

# StructType() & StructField() in PySpark

**from pyspark.sql.functions import col**

**from pyspark.sql.types import \***

**d = [(1,('veena','tammina'),353455),(2,('lochu','surisetti'),234234)]**

**structName = StructType([\**

**StructField('firstName',StringType()),\**

**StructField('lastName',StringType())])**

**schema = StructType([\**

**StructField('id',IntegerType()),\**

**StructField('name',structName),\**

**StructField('salary',IntegerType())])**

**df = spark.createDataFrame(d ,schema)**

**df.display()**

**df.printSchema()**

id name salary

1 {"firstName":"veena","lastName":"tammina"} 353455

2 {"firstName":"lochu","lastName":"surisetti"} 234234

root

|-- id: integer (nullable = true)

|-- name: struct (nullable = true)

| |-- firstName: string (nullable = true)

| |-- lastName: string (nullable = true)

|-- salary: integer (nullable = true)

# ArrayType Columns in PySpark

**from pyspark.sql.types import \***

**d = [('veena',[35,34,55]),('lochu',[23,42,34])]**

**structName = StructType([\**

**StructField('firstName',StringType()),\**

**StructField('Numbers',ArrayType(IntegerType()))])**

**df = spark.createDataFrame(data=d ,schema=structName)**

**df.display()**

**df.printSchema()**

**df.withColumn('firstNumber',df.Numbers[0]).show()**

firstName Numbers

veena [35,34,55]

lochu [23,42,34]

root

|-- firstName: string (nullable = true)

|-- Numbers: array (nullable = true)

| |-- element: integer (containsNull = true)

+---------+------------+-----------+

|firstName| Numbers |firstNumber|

+---------+------------+-----------+

| veena |[35, 34, 55]| 35|

| lochu |[23, 42, 34]| 23|

+---------+------------+-----------+

# explode(), split(), array() & array\_contains() functions in PySpark

* explode() function used to create a new row for each element in the given array column

**d= [(1,'maheer',['dotnet','azure']),(2,'lochu',['python','java'])]**

**schema = ['id','name','skills']**

**df=spark.createDataFrame(d,schema)**

**df.display()**

**df.printSchema()**

id name skills

1 maheer ["dotnet","azure"]

2 lochu ["python","java"]

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

|-- skills: array (nullable = true)

| |-- element: string (containsNull = true)

**from pyspark.sql.functions import col,explode**

**df.show()**

**df1 = df.withColumn('skill',explode(col('skills')))**

**df1.show()**

+---+------+---------------+

| id| name| skills|

+---+------+---------------+

| 1|maheer |[dotnet, azure]|

| 2| lochu | [python, java]|

+---+------+---------------+

+---+------+---------------+------+

| id| name| skills| skill|

+---+------+---------------+------+

| 1| maheer|[dotnet, azure]|dotnet|

| 1| maheer|[dotnet, azure]| azure|

| 2| lochu| [python, java]|python|

| 2| lochu| [python, java]| java|

+---+------+---------------+------+

* split() function returns an array type after splitting the string column by delimiter

**d= [(1,'maheer','dotnet,azure'),(2,'lochu','python,java')]**

**schema = ['id','name','skills']**

**df=spark.createDataFrame(d,schema)**

**df.display()**

**df.printSchema()**

id name skills

1 maheer dotnet,azure

2 lochu python,java

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

|-- skills: string (nullable = true)

**from pyspark.sql.functions import col,split**

**df.show()**

**df1 = df.withColumn('skillarray',split('skills',','))**

**df1.show()**

+---+------+------------+

| id| name| skills|

+---+------+------------+

| 1| maheer|dotnet,azure|

| 2| lochu| python,java|

+---+------+------------+

+---+------+------------+---------------+

| id| name| skills| skillarray|

+---+------+------------+---------------+

| 1| maheer|dotnet,azure|[dotnet, azure]|

| 2| lochu| python,java| [python, java]|

+---+------+------------+---------------+

* array() function is used to create a new array column by merging the data from multiple columns

**d= [(1,'maheer','dotnet','azure'),(2,'lochu','python','java')]**

**schema = ['id','name','primaryskill','secondaryskill']**

**df=spark.createDataFrame(d,schema)**

**df.display()**

**df.printSchema()**

id name primaryskill secondaryskill

1 maheer dotnet azure

2 lochu python java

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

|-- primaryskill: string (nullable = true)

|-- secondaryskill: string (nullable = true)

**from pyspark.sql.functions import col,array**

**df.show()**

**df1 = df.withColumn('skills',array('primaryskill','secondaryskill'))**

**df1.show()**

+---+------+------------+--------------+

| id| name|primaryskill|secondaryskill|

+---+------+------------+--------------+

| 1| maheer| dotnet| azure|

| 2| lochu| python| java|

+---+------+------------+--------------+

+---+------+------------+--------------+---------------+

| id| name|primaryskill|secondaryskill| skills|

+---+------+------------+--------------+---------------+

| 1| maheer| dotnet| azure|[dotnet, azure]|

| 2| lochu| python| java| [python, java]|

+---+------+------------+--------------+---------------+

* **array\_contains()** sql function is used to check of array column contains a value.Returns null if the array is null,true if the array contains the value and false otherwise

**d= [(1,'maheer',['dotnet','azure']),(2,'lochu',['python','java'])]**

**schema = ['id','name','primaryskill']**

**df=spark.createDataFrame(d,schema)**

**df.display()**

**df.printSchema()**

id name primaryskill

1 maheer ["dotnet","azure"]

2 lochu ["python","java"]

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

|-- primaryskill: array (nullable = true)

| |-- element: string (containsNull = true)

**from pyspark.sql.functions import col,array\_contains**

**df.show()**

**df = df.withColumn('skilltest',array\_contains('primaryskill','azure'))**

**df.show()**

+---+------+---------------+

| id| name| primaryskill|

+---+------+---------------+

| 1|maheer| [dotnet, azure]|

| 2| lochu| [python, java]|

+---+------+---------------+

+---+------+---------------+---------+

| id| name| primaryskill| skilltest|

+---+------+---------------+---------+

| 1| maheer|[dotnet, azure]| true|

| 2| lochu| [python, java]| false|

+---+------+---------------+---------+

# MapType Column in PySpark

* PySpark MapType is used to represent map-key value pair similar to python dictionary(dict)

**from pyspark.sql.types import \***

**data = [('lochu',{'hair':'black','eye':'brown'}),('veena',{'hair':'brown','eye':'blue'})]**

**schema = StructType([\**

**StructField('name',StringType()),\**

**StructField('properties',MapType(StringType(),StringType()))])**

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

**df.show(truncate = False)**

**df.printSchema()**

+-----+-----------------------------+

|name | properties |

+-----+-----------------------------+

|lochu|{eye -> brown, hair -> black}|

|veena|{eye -> blue, hair -> brown} |

+-----+-----------------------------+

root

|-- name: string (nullable = true)

|-- properties: map (nullable = true)

| |-- key: string

| |-- value: string (valueContainsNull = true)

**df1 = df.withColumn('hair',df.properties['hair'])**

**df1.show(truncate=False)**

+-----+-----------------------------+-----+

|name | properties |hair |

+-----+-----------------------------+-----+

|lochu|{eye -> brown, hair -> black}|black|

|veena|{eye -> blue, hair -> brown} |brown|

+-----+-----------------------------+-----+

**df2 = df1.withColumn('eye',df.properties.getItem('eye'))**

**df2.show(truncate=False)**

+-----+-----------------------------+-----+-----+

|name |properties |hair | eye |

+-----+-----------------------------+-----+-----+

|lochu|{eye -> brown, hair -> black}|black|brown|

|veena|{eye -> blue, hair -> brown} |brown|blue |

+-----+-----------------------------+-----+-----+

# map\_keys(), map\_values() & explode() functions to work with MapType Columns in PySpark

* **explode()**

**from pyspark.sql.types import \***

**from pyspark.sql.functions import explode**

**data = [('lochu',{'hair':'black','eye':'brown'}),('veena',{'hair':'brown','eye':'blue'})]**

**schema = StructType([\**

**StructField('name',StringType()),\**

**StructField('properties',MapType(StringType(),StringType()))])**

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

**df.select('name','properties',explode(df.properties)).show(truncate=False)**

+-----+-----------------------------+----+-----+

|name | properties |key |value|

+-----+-----------------------------+----+-----+

|lochu|{eye -> brown, hair -> black}|eye |brown|

|lochu|{eye -> brown, hair -> black}|hair|black|

|veena|{eye -> blue, hair -> brown} |eye |blue |

|veena|{eye -> blue, hair -> brown} |hair|brown|

+-----+-----------------------------+----+-----+

🡪**map\_keys(),map\_values()**

**from pyspark.sql.functions import map\_keys,map\_values**

**df1 = df.withColumn('keys',map\_keys(df.properties))**

**df2 = df1.withColumn('values',map\_values(df.properties))**

**df2.show(truncate=False)**

+-----+-----------------------------+-----------+--------------+

|name | properties | keys | values |

+-----+-----------------------------+-----------+--------------+

|lochu|{eye -> brown, hair -> black}|[eye, hair]|[brown, black]|

|veena|{eye -> blue, hair -> brown} |[eye, hair]|[blue, brown] |

+-----+-----------------------------+-----------+--------------+

# Row() class in PySpark

* Pyspark.sql.Row which is represented as a record/row in DataFrame, one can create a Row object by using named argumnets or create a custom row like class

**from pyspark.sql import Row**

**row = Row(name='lochu',salary=20000)**

**print(row[0] + ' ' + str(row[1]))**

lochu 20000

**from pyspark.sql import Row**

**row = Row(name='lochu',salary=20000)**

**print(row.name + ' ' + str(row.salary))**

lochu 20000

**from pyspark.sql import Row**

**row1 = Row(name='lochu',salary=20000)**

**row2 = Row(name='veena',salary=80000)**

**data = [row1,row2]**

**df = spark.createDataFrame(data)**

**df.show()**

**df.printSchema()**

+-----+------+

| name|salary|

+-----+------+

|lochu| 20000|

|veena| 80000|

+-----+------+

root

|-- name: string (nullable = true)

|-- salary: long (nullable = true)

**Person = Row('name','age')**

**person1 = Person('lochu',22)**

**person2 = Person('veena',21)**

**print(person1.name,person1.age)**

lochu 22

**Person = Row('name','age')**

**person1 = Person('lochu',22)**

**person2 = Person('veena',21)**

**df = spark.createDataFrame([person1,person2])**

**df.show()**

+-----+---+

| name|age|

+-----+---+

|lochu| 22|

|veena| 21|

+-----+---+

**data = [Row(name='lochu',prop=Row(age=20,gender='female')),Row(name='chandu',prop=Row(age=22,gender='male'))]**

**df = spark.createDataFrame(data)**

**df.show()**

**df.printSchema()**

+------+------------+

| name| prop|

+------+------------+

| lochu|{20, female}|

|chandu| {22, male}|

+------+------------+

root

|-- name: string (nullable = true)

|-- prop: struct (nullable = true)

| |-- age: long (nullable = true)

| |-- gender: string (nullable = true)

# Column class in PySpark

* Pyspark column class represents a single column in a dataframe
* Pyspark.sql.column class provides several functions to work with dataframe to manipulate the column values, evaluate the Boolean expression to filter rows, retrieve a value or part of a value from a dataframe column
* One of the simplest ways to create a column class object is using Pyspark lit() SQL function

**from pyspark.sql.functions import lit**

**col1 = lit('abcd')**

**print(type(col1))**

<class 'pyspark.sql.column.Column'>

**from pyspark.sql.functions import lit**

**data = [('lochu','female',22),('veena','female',21)]**

**schema = ['name','gender','age']**

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

**df1 = df.withColumn('newcol',lit('newColVal'))**

**df1.show()**

**df1.printSchema()**

+-----+------+---+---------+

| name|gender|age| newcol|

+-----+------+---+---------+

|lochu|female| 22|newColVal|

|veena|female| 21|newColVal|

+-----+------+---+---------+

root

|-- name: string (nullable = true)

|-- gender: string (nullable = true)

|-- age: long (nullable = true)

|-- newcol: string (nullable = false)

df1.select(df1.name).show()

(or)

df1.select(df1['name']).show()

(or)

from pyspark.sql.functions import col

df1.select(col('name')).show()

+-----+

| name|

+-----+

|lochu|

|veena|

+-----+

**from pyspark.sql.functions import lit**

**from pyspark.sql.types import \***

**data = [('lochu','female',22,('black','brown')),('veena','female',21,('black','brown'))]**

**propstype = StructType([\**

**StructField('hair',StringType()),\**

**StructField('eye',StringType())])**

**schema = StructType([**

**StructField('name',StringType()),\**

**StructField('gender',StringType()),\**

**StructField('age',IntegerType()),\**

**StructField('props',propstype)])**

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

**df.show()**

**df.printSchema()**

+-----+------+---+--------------+

| name|gender|age| props|

+-----+------+---+--------------+

|lochu|female| 22|{black, brown}|

|veena|female| 21|{black, brown}|

+-----+------+---+--------------+

root

|-- name: string (nullable = true)

|-- gender: string (nullable = true)

|-- age: integer (nullable = true)

|-- props: struct (nullable = true)

| |-- hair: string (nullable = true)

| |-- eye: string (nullable = true)

**df.select(df.props.hair).show()**

+----------+

|props.hair|

+----------+

| black|

| black|

+----------+

**df.select(df['props.hair']).show()**

+----------+

| hair|

+----------+

| black|

| black|

+----------+

**from pyspark.sql.functions import col**

**df.select(col('props.eye')).show()**

+-----+

| eye|

+-----+

|brown|

|brown|

+-----+

# when() & otherwise() functions in PySpark

* It is similar to SQL Case When,executes sequence of expressions until matches the condition and returns a value when match

**data = [(1,'lochu','F',2000),(2,'chandu','M',5600),(3,'abcd','',6780)]**

**schema = ['id','name','gender','salary']**

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

**df.show()**

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

| 3| abcd| | 6780|

+---+------+------+------+

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

**df1 = df.select(df.id,df.name,when(df.gender=='M','male').when(df.gender=='F','female').otherwise('unknown'))**

**df1.show()**

+---+------+-------------------------------------------------------------------------------+

| id| name|CASE WHEN (gender = M) THEN male WHEN (gender = F) THEN female ELSE unknown END|

+---+------+-------------------------------------------------------------------------------+

| 1| lochu| female|

| 2|chandu| male|

| 3| abcd| unknown|

+---+------+-------------------------------------------------------------------------------+

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

**df1 = df.select(df.id,\**

**df.name,\**

**when(df.gender=='M','male').\**

**when(df.gender=='F','female').\**

**otherwise('unknown').\**

**alias('gender'))**

**df1.show()**

+---+------+-------+

| id| name| gender|

+---+------+-------+

| 1| lochu| female|

| 2|chandu| male|

| 3| abcd|unknown|

+---+------+-------+

# alias(), asc(), desc(), cast() & like() functions on Columns of dataframe in PySpark

* alias() : provides alias to the column

**data = [(1,'lochu',2000),(2,'chandu',5600),(3,'veena',6780)]**

**schema = ['id','name','salary']**

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

**df.select(df.id.alias('emp\_id'),df.name.alias('emp\_name'),df.salary.alias('emp\_salary')).show()**

+------+--------+----------+

|emp\_id|emp\_name|emp\_salary|

+------+--------+----------+

| 1| lochu| 2000|

| 2| chandu| 5600|

| 3| veena| 6780|

+------+--------+----------+

* asc() , desc() : sorts the columns in ascending or descending order

**df.sort(df.name.asc()).show()**

**df.sort(df.salary.desc()).show()**

+---+------+------+

| id| name|salary|

+---+------+------+

| 2 |chandu| 5600|

| 1 | lochu| 2000|

| 3 | veena| 6780|

+---+------+------+

+---+------+------+

| id| name| salary|

+---+------+------+

| 3| veena| 6780|

| 2|chandu| 5600|

| 1| lochu| 2000|

+---+------+------+

* cast() : convert the datatype

**df1 = df.select(df.name,df.salary.cast('int'))**

**df1.show()**

**df1.printSchema()**

+------+------+

| name|salary|

+------+------+

| lochu| 2000|

|chandu| 5600|

| veena| 6780|

+------+------+

root

|-- name: string (nullable = true)

|-- salary: integer (nullable = true)

* like(): similar to SQL LIKE expression

**data = [(1,'lochu',2000),(2,'chandu',5600),(3,'veena',6780),(4,'lakshmi',2321)]**

**schema = ['id','name','salary']**

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

**df.filter(df.name.like('l%')).show()**

+---+-------+------+

| id| name|salary|

+---+-------+------+

| 1| lochu| 2000|

| 4|lakshmi| 2321|

+---+-------+------+

# filter() & where() in PySpark

* PySpark filter() function is used to filter the rows from database based the given condition or SQL expression
* We can also use where() clause instead of the filter If you are coming from SQL background, both these functions operate exactly same

**data = [(1,'lochu','F',2000),(2,'chandu','M',5600),(3,'veena','F',6780),(4,'lakshmi','F',2321)]**

**schema = ['id','name','gender','salary']**

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

**df.where(df.gender=='F').show()**

+---+-------+------+------+

| id| name|gender|salary|

+---+-------+------+------+

| 1| lochu| F| 2000|

| 3| veena| F| 6780|

| 4|lakshmi| F| 2321|

+---+-------+------+------+

**df.filter(df.gender=='M').show()**

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 2|chandu| M| 5600|

+---+------+------+------+

# distinct() & dropDuplicates() in PySpark

* PySpark distincht() function is used to remove the diplicates rows(all columns)
* dropDuplicates() is used to drop rows based on selected (one or multiple) columns
* So basically , using these functions we can get distinct rows

**data = [(1,'lochu','F',2000),(2,'chandu','M',5600),(3,'veena','F',6780),(1,'lochu','F',2000)]**

**schema = ['id','name','gender','salary']**

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

**df.show()**

**df.distinct().show()**

**df.distinct().count()**

**df.dropDuplicates().show()**

**df.dropDuplicates(['gender']).show()**

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

| 3| veena| F| 6780|

| 1| lochu| F| 2000|

+---+------+------+------+

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

| 3| veena| F| 6780|

+---+------+------+------+

Out[13]: 3

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

| 3| veena| F| 6780|

+---+------+------+------+

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

+---+------+------+------+

# orderBy() & sort() in PySpark

* sort() or orderBy(0 function of Pyspark DataFrame to sort dataframe by ascending or descending order based on single or multiple columns
* By default , sorting will happenin ascending order.We can explicitly mention ascending or descending using asc() , desc() functions

**data = [(1,'lochu','F',2000),(2,'chandu','M',5600),(3,'veena','F',6780),(4,'lakshmi','F',2000)]**

**schema = ['id','name','gender','salary']**

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

**df.sort('salary','name').show()**

+---+-------+------+------+

| id| name|gender|salary|

+---+-------+------+------+

| 4|lakshmi| F| 2000|

| 1| lochu| F| 2000|

| 2| chandu| M| 5600|

| 3| veena| F| 6780|

+---+-------+------+------+

**df.sort(df.salary,df.name.desc()).show()**

+---+-------+------+------+

| id| name|gender|salary|

+---+-------+------+------+

| 1| lochu| F| 2000|

| 4|lakshmi| F| 2000|

| 2| chandu| M| 5600|

| 3| veena| F| 6780|

+---+-------+------+------+

**df.sort(df.salary.desc(),df.name.desc()).show()**

+---+-------+------+------+

| id| name|gender|salary|

+---+-------+------+------+

| 3| veena| F| 6780|

| 2| chandu| M| 5600|

| 1| lochu| F| 2000|

| 4|lakshmi| F| 2000|

+---+-------+------+------+

**df.orderBy(df.salary.asc(),df.name.desc()).show()**

+---+-------+------+------+

| id| name|gender|salary|

+---+-------+------+------+

| 1| lochu| F| 2000|

| 4|lakshmi| F| 2000|

| 2| chandu| M| 5600|

| 3| veena| F| 6780|

+---+-------+------+------+

# union() & unionAll() in PySpark

* union() and unionAll() transformations are used to merge two or more dataframes of the same schema or structure
* these methods merges two dataframes and returns the new dataframe with all rows from two dataframes regardless of duplicate data
* to remove duplicates use distinct() function

**d1 = [(1,'lochu','F',2000),(2,'chandu','M',5600)]**

**d2 =[(3,'veena','F',6780),(1,'lochu','F',2000)]**

**schema = ['id','name','gender','salary']**

**df1=spark.createDataFrame(d1,schema)**

**df2=spark.createDataFrame(d2,schema)**

**df1.show()**

**df2.show()**

**newdf = df1.union(df2) (or) df1.unionAll(df2)**

**newdf.show()**

**newdf.distinct().show()**

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

+---+------+------+------+

+---+-----+------+------+

| id| name|gender|salary|

+---+-----+------+------+

| 3|veena| F| 6780|

| 1|lochu| F| 2000|

+---+-----+------+------+

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

| 3| veena| F| 6780|

| 1| lochu| F| 2000|

+---+------+------+------+

+---+------+------+------+

| id | name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

| 3| veena| F| 6780|

+---+------+------+------+

# groupBy() in PySpark

* 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, max functions on grouped data

**d1 = [(1,'lochu','F',2000,'HR'),(2,'chandu','M',5600,'IT'),\**

**(3,'veena','F',6780,'HR'),(1,'lochu','F',2000,'IT'),\**

**(5,'kalyan','M',2312,'IT')]**

**schema = ['id','name','gender','salary','dept']**

**df = spark.createDataFrame(d1,schema)**

**df2 = df.groupBy('dept').count()**

**df2.show()**

+----+-----+

|dept|count|

+----+-----+

| HR| 2|

| IT| 3|

+----+-----+

**df2 = df.groupBy('dept').max('salary')**

**df2.show()**

+----+-----------+

|dept|max(salary)|

+----+-----------+

| HR| 6780|

| IT| 5600|

+----+-----------+

**df2 = df.groupBy('dept').min('salary')**

**df2.show()**

+----+-----------+

|dept|min(salary)|

+----+-----------+

| HR| 2000|

| IT| 2000|

+----+-----------+

**df2 = df.groupBy('dept').avg('salary')**

**df2.show()**

+----+-----------+

|dept|avg(salary)|

+----+-----------+

| HR| 4390.0|

| IT| 3304.0|

+----+-----------+

**df2 = df.groupBy('dept','gender').count()**

**df2.show()**

+----+------+-----+

|dept|gender|count|

+----+------+-----+

| HR| F| 2|

| IT| M| 2|

| IT| F| 1|

+----+------+-----+

**df2 = df.groupBy('gender').sum('salary')**

**df2.show()**

+------+-----------+

|gender|sum(salary)|

+------+-----------+

| F| 10780|

| M| 7912|

+------+-----------+

# GroupBy agg() function in PySpark

* groupBy agg() is used to calculate more than one aggregate at a time on grouped dataframe

**from pyspark.sql.functions import min,max,count**

**d1 = [(1,'lochu','F',2000,'HR'),(2,'chandu','M',5600,'IT'),\**

**(3,'veena','F',6780,'HR'),(1,'lochu','F',2000,'IT'),\**

**(5,'kalyan','M',2312,'IT')]**

**schema = ['id','name','gender','salary','dept']**

**df = spark.createDataFrame(d1,schema)**

**df2 = df.groupBy('dept').agg(count('\*').alias('countOfEmps'),min('salary').alias('min\_salary'),max('salary').alias('max\_salary'))**

**df2.show()**

+----+-----------+----------+----------+

|dept|countOfEmps|min\_salary|max\_salary|

+----+-----------+----------+----------+

| HR| 2| 2000| 6780|

| IT| 3| 2000| 5600|

+----+-----------+----------+----------+

# unionByName() function in PySpark

* unionByName() lets you to merge/union two DataFrames with a different number of columns (different schema) by passing allowMissingColumns with value true.

**d1 = [(1,'lochu','F',2000),(2,'chandu','M',5600)]**

**d2 =[(3,'veena','F',22),(1,'lochu','F',20)]**

**schema1 = ['id','name','gender','salary']**

**schema2 = ['id','name','gender','age']**

**df1=spark.createDataFrame(d1,schema1)**

**df2=spark.createDataFrame(d2,schema2)**

**df1.show()**

**df2.show()**

**newdf = df1.unionAll(df2)**

**newdf.show()**

**newdf.distinct().show()**

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

+---+------+------+------+

+---+-----+------+---+

| id| name|gender|age|

+---+-----+------+---+

| 3|veena| F| 22|

| 1|lochu| F| 20|

+---+-----+------+---+

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

| 3| veena| F| 22|

| 1| lochu| F| 20|

+---+------+------+------+

+---+------+------+------+

| id| name|gender|salary|

+---+------+------+------+

| 1| lochu| F| 2000|

| 2|chandu| M| 5600|

| 3| veena| F| 22|

| 1| lochu| F| 20|

+---+------+------+------+

**df1.unionByName(allowMissingColumns=True,other=df2).show()**

+---+------+------+------+----+

| id| name|gender|salary| age|

+---+------+------+------+----+

| 1| lochu| F| 2000|null|

| 2|chandu| M| 5600|null|

| 3| veena| F| null| 22|

| 1| lochu| F| null| 20|

+---+------+------+------+----+

# select() function in PySpark

* select() function is used to select single, multiple, column by index, all columns from the list and the nested columns from a dataframe

**d1 = [(1,'lochu','F',2000,'HR'),(2,'chandu','M',5600,'IT'),\**

**(3,'veena','F',6780,'HR'),(1,'lochu','F',2000,'IT'),\**

**(5,'kalyan','M',2312,'IT')]**

**schema = ['id','name','gender','salary','dept']**

**df = spark.createDataFrame(d1,schema)**

**df.select(df.id,df.name,df.dept).show()**

**df.select(df['id'],df['name']).show()**

**df.select('id','name','salary').show()**

+---+------+

| id| name|

+---+------+

| 1| lochu|

| 2|chandu|

| 3| veena|

| 1| lochu|

| 5|kalyan|

+---+------+

+---+------+------+

| id| name|salary|

+---+------+------+

| 1| lochu| 2000|

| 2|chandu| 5600|

| 3| veena| 6780|

| 1| lochu| 2000|

| 5|kalyan| 2312|

+---+------+------+

**from pyspark.sql.functions import col**

**df.select(col('id'),col('name')).show()**

+---+------+

| id | name|

+---+------+

| 1| lochu|

| 2|chandu|

| 3| veena|

| 1| lochu|

| 5|kalyan|

+---+------+

**df.select('\*').show()**

+---+------+------+------+----+

| id| name|gender|salary|dept|

+---+------+------+------+----+

| 1| lochu| F| 2000| HR|

| 2|chandu| M| 5600| IT|

| 3| veena| F| 6780| HR|

| 1| lochu| F| 2000| IT|

| 5|kalyan| M| 2312| IT|

+---+------+------+------+----+

df.select([col for col in df.columns]).show()

+---+------+------+------+----+

| id| name|gender|salary|dept|

+---+------+------+------+----+

| 1| lochu| F| 2000| HR|

| 2|chandu| M| 5600| IT|

| 3| veena| F| 6780| HR|

| 1| lochu| F| 2000| IT|

| 5|kalyan| M| 2312| IT|

+---+------+------+------+----+

# join() function in PySpark | inner, left, right, full Join,Left semi, Left anti & self join

* join() is like SQL JOIN. We can combine columns from different dataframes based on the condition. It supports all basic join types as INNER, LEFT, OUTER,RIGHT OUTER, LEFT ANTI, LEFT SEMI, CROSS,SELF

**d1 = [(1,'lochu',2000,2),(2,'veena',2001,1),(3,'vara',3000,4)]**

**s1 = ['id','name','salary','dep']**

**d2 = [(1,'IT'),(2,'HR'),(3,'Payroll')]**

**s2 = ['depId','depName']**

**empdf = spark.createDataFrame(d1,s1)**

**depdf = spark.createDataFrame(d2,s2)**

**empdf.show()**

**depdf.show()**

+---+-----+------+---+

| id| name|salary|dep|

+---+-----+------+---+

| 1|lochu| 2000| 2|

| 2|veena| 2001| 1|

| 3| vara| 3000| 4|

+---+-----+------+---+

+-----+-------+

|depId|depName|

+-----+-------+

| 1| IT|

| 2| HR|

| 3|Payroll|

+-----+-------+

**empdf.join(depdf,empdf.dep == depdf.depId,'inner').show()**

+---+-----+------+---+-----+-------+

| id| name|salary|dep|depId|depName|

+---+-----+------+---+-----+-------+

| 2|veena| 2001| 1| 1| IT|

| 1|lochu| 2000| 2| 2| HR|

+---+-----+------+---+-----+-------+

**empdf.join(depdf,empdf.dep == depdf.depId,'left').show()**

+---+-----+------+---+-----+-------+

| id| name|salary|dep|depId|depName|

+---+-----+------+---+-----+-------+

| 1|lochu| 2000| 2| 2| HR|

| 2|veena| 2001| 1| 1| IT|

| 3| vara| 3000| 4| null| null|

+---+-----+------+---+-----+-------+

**empdf.join(depdf,empdf.dep == depdf.depId,'right').show()**

+----+-----+------+----+-----+-------+

| id| name|salary| dep|depId|depName|

+----+-----+------+----+-----+-------+

| 2|veena| 2001| 1| 1| IT|

| 1|lochu| 2000| 2| 2| HR|

|null| null| null|null| 3|Payroll|

+----+-----+------+----+-----+-------+

**empdf.join(depdf,empdf.dep == depdf.depId,'full').show()**

+----+-----+------+----+-----+-------+

| id| name|salary| dep|depId|depName|

+----+-----+------+----+-----+-------+

| 2|veena| 2001| 1| 1| IT|

| 1|lochu| 2000| 2| 2| HR|

|null| null| null| 3|Payroll|

| 3| vara| 3000| 4| null| null|

+----+-----+------+----+-----+-------+

* leftsemi() join similar to inner join but get columns only from left dataframe for matching rows
* leftanti() opposite to leftsemi(), it gets not matching rows from left dataframe
* self-join, joins data with same dataframe

empdf.join(depdf,empdf.dep == depdf.depId,'leftsemi').show()

+---+-----+------+---+

| id| name|salary|dep|

+---+-----+------+---+

| 2|veena| 2001| 1|

| 1|lochu| 2000| 2|

+---+-----+------+---+

empdf.join(depdf,empdf.dep == depdf.depId,'leftanti').show()

+---+----+------+---+

| id|name|salary|dep|

+---+----+------+---+

| 3|vara| 3000| 4|

+---+----+------+---+

from pyspark.sql.functions import col

df = spark.createDataFrame(d1,s1)

df.alias('empData').join(df.alias('mgrData'),\

    col('empData.dep')==col('mgrData.id'),\

        'left').show()

+---+-----+------+---+----+-----+------+----+

| id| name|salary|dep| id| name|salary| dep|

+---+-----+------+---+----+-----+------+----+

| 1|lochu| 2000| 2| 2|veena| 2001| 1|

| 2|veena| 2001| 1| 1|lochu| 2000| 2|

| 3| vara| 3000| 4|null| null| null|null|

+---+-----+------+---+----+-----+------+----+

from pyspark.sql.functions import col

df.alias('empData').join(df.alias('mgrData'),\

    col('empData.dep')==col('mgrData.id'),\

        'left').select(col('empData.name').alias('empName'),col('mgrData.name').alias('mgrName')).show()

+-------+-------+

|empName|mgrName|

+-------+-------+

| lochu| veena|

| veena| lochu|

| vara| null|

+-------+-------+

# 28. pivot(),unpivot() function in PySpark

* it’s used to rotate data in one column into multiple columns
* it is an aggregation where one of the grouping column values will be converted in individual columns

**d1 = [\**

**(1,'lochu','F','HR'),\**

**(2,'veena','F','IT'),\**

**(3,'chandu','M','HR'),\**

**(4,'kalyan','M','IT'),\**

**(5,'vara','F','IT'),\**

**(6,'sai','M','HR'),\**

**(7,'jyo','F','IT')]**

**schema = ['id','name','gender','dept']**

**df = spark.createDataFrame(d1,schema)**

**df.show()**

+---+------+------+----+

| id| name|gender|dept|

+---+------+------+----+

| 1| lochu| F| HR|

| 2| veena| F| IT|

| 3|chandu| M| HR|

| 4|kalyan| M| IT|

| 5| vara| F| IT|

| 6| sai| M| HR|

| 7| jyo| F| IT|

+---+------+------+----+

**df.groupBy('dept','gender').count().show()**

+----+------+-----+

|dept|gender|count|

+----+------+-----+

| HR| F| 1|

| IT| F| 3|

| HR| M| 2|

| IT| M| 1|

+----+------+-----+

**df.groupBy('dept').pivot('gender').count().show()**

+----+---+---+

|dept| F| M|

+----+---+---+

| HR| 1| 2|

| IT| 3| 1|

+----+---+---+

**df.groupBy('dept').pivot('gender',['M']).count().show()**

+----+---+

|dept| M|

+----+---+

| HR| 2|

| IT| 1|

+----+---+

**df.groupBy('dept').pivot('gender',['M','F']).count().show()**

+----+---+---+

|dept| M| F|

+----+---+---+

| HR| 2| 1|

| IT| 1| 3|

+----+---+---+

* unpivot is rotating columns into rows. Pyspark SQL doesn’t have unpivot function hence will use the stack() function

**d1 = [\**

**(1,'lochu','F','HR'),\**

**(2,'veena','F','IT'),\**

**(3,'chandu','M','HR'),\**

**(4,'kalyan','M','IT'),\**

**(5,'vara','F','IT'),\**

**(6,'sai','M','HR'),\**

**(7,'jyo','F','IT')]**

**schema = ['id','name','gender','dept']**

**df = spark.createDataFrame(d1,schema)**

**df2= df.groupBy('dept').pivot('gender',['M','F']).count()**

**df2.show()**

**from pyspark.sql.functions import expr**

**unpivotdf = df2.select('dept',expr("stack(2, 'M',M,'F',F)as (gender,count)"))**

**unpivotdf.show()**

+----+---+---+

|dept| M| F|

+----+---+---+

| HR| 2| 1|

| IT| 1| 3|

+----+---+---+

+----+------+-----+

|dept|gender|count|

+----+------+-----+

| HR| M| 2|

| HR| F| 1|

| IT| M| 1|

| IT| F| 3|

+----+------+-----+

# 29. fill() & fillna() functions in PySpark

**d1 = [\**

**(1,'lochu','F','HR'),\**

**(2,'veena','F',None),\**

**(3,'chandu','M','HR'),\**

**(None,'kalyan','M','IT'),\**

**(5,'vara','F','IT'),\**

**(6,'sai',None,'HR'),\**

**(7,'jyo','F','IT')]**

**schema = ['id','name','gender','dept']**

**df = spark.createDataFrame(d1,schema)**

**df.show()**

+----+------+------+----+

| id| name|gender|dept|

+----+------+------+----+

| 1| lochu| F| HR|

| 2| veena| F|null|

| 3|chandu| M| HR|

|null|kalyan| M| IT|

| 5| vara| F| IT|

| 6| sai| null| HR|

| 7| jyo| F| IT|

+----+------+------+----+

**df2 = df.na.fill('unknown',['gender','dept']).fillna('nil',['id'])**

**df2.show()**

+----+------+-------+-------+

| id| name| gender| dept|

+----+------+-------+-------+

| 1| lochu| F| HR|

| 2| veena| F|unknown|

| 3|chandu| M| HR|

|null|kalyan| M| IT|

| 5| vara| F| IT|

| 6| sai|unknown| HR|

| 7| jyo| F| IT|

+----+------+-------+-------+

# 30. sample() function in PySpark

* to get the random sampling subset from the large dataset
* use fraction to indicate what percentage of data to return and seed value t make sure every time to get same random sample

**df = spark.range(start=1,end=101)**

**df1 = df.sample(fraction=0.1,seed=123)**

**df2 = df.sample(fraction=0.1)**

**df3 = df.sample(fraction=0.1)**

**display(df1)**

**display(df2)**

**display(df3)**

id (fraction=0.1,seed=123)#to get fixed number of rows of numbers (8 rows)

36

39

42

46

72

85

88

100

Id (fraction = 0.1,13 rows)

5

7

10

12

15

16

40

48

68

77

79

81

85

Id (fraction = 0.1, 10 rows)

6

34

37

40

58

74

79

89

92

98

# 31. collect() function in PySpark

* collect() retrieves all the elements in a Dataframe as an array of row type to the driver node.
* collect() is an action hence it does not return a dataframe instead, it returns data in an array to the driver . Once the data is in an array, you can use python for loop to process it further.
* collect() use it with small dataframes. With big DataFrames it may result in out of memory error as its return entire data to single node(driver)

**d1 = [\**

**(1,'lochu','F','HR'),\**

**(2,'veena','F',None),\**

**(3,'chandu','M','HR'),\**

**(None,'kalyan','M','IT'),\**

**(5,'vara','F','IT'),\**

**(6,'sai',None,'HR'),\**

**(7,'jyo','F','IT')]**

**schema = ['id','name','gender','dept']**

**df = spark.createDataFrame(d1,schema)**

**listrow = df.collect()**

**print(listrow)**

**print(f"first row : {listrow[0]}")**

**print(f"dept : {listrow[0][3]}")**

[Row(id=1, name='lochu', gender='F', dept='HR'), Row(id=2, name='veena', gender='F', dept=None), Row(id=3, name='chandu', gender='M', dept='HR'), Row(id=None, name='kalyan', gender='M', dept='IT'), Row(id=5, name='vara', gender='F', dept='IT'), Row(id=6, name='sai', gender=None, dept='HR'), Row(id=7, name='jyo', gender='F', dept='IT')]

first row : Row(id=1, name='lochu', gender='F', dept='HR')

dept : HR

# 32. DataFrame.transform() function in PySpark

* it is used to chain the custom transformations and this function returns the new DataFrame after applying the specified transformations

**d = [(1,'veena',2000),(2,'lochu',3000)]**

**schema = ['id','name','salary']**

**df = spark.createDataFrame(d,schema)**

**df.show()**

+---+-----+------+

| id| name|salary|

+---+-----+------+

| 1|veena| 2000|

| 2|lochu| 3000|

+---+-----+------+

**from pyspark.sql.functions import upper**

**df.withColumn('name',upper(df.name)).show()**

+---+-----+------+

| id| name|salary|

+---+-----+------+

| 1|VEENA| 2000|

| 2|LOCHU| 3000|

+---+-----+------+

**from pyspark.sql.functions import upper**

**def convertToUpper(df):**

**return df.withColumn('name',upper(df.name))**

**df1 = df.transform(convertToUpper)**

**df1.show()**

+---+-----+------+

| id| name|salary|

+---+-----+------+

| 1|VEENA| 2000|

| 2|LOCHU| 3000|

+---+-----+------+

**def doubleSalary(df):**

**return df.withColumn('salary',df.salary\*2)**

**df1 = df.transform(doubleSalary)**

**df1.show()**

+---+-----+------+

| id| name|salary|

+---+-----+------+

| 1|veena| 4000|

| 2|lochu| 6000|

+---+-----+------+

**df1 = df.transform(doubleSalary).transform(convertToUpper)**

**df1.show()**

+---+-----+------+

| id| name|salary|

+---+-----+------+

| 1|VEENA| 4000|

| 2|LOCHU| 6000|

+---+-----+------+

# 33. pyspark.sql.functions.transform()

* It is used to apply the transformation on a column of type Array. This function applies the specified transformation on every element of the array and returns an object of ArrayType.

**data = [(1,'veena',['dotnet','python']),(2,'lochu',['java','aws'])]**

**schema = ['id','name','skills']**

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

**df.show()**

**df.printSchema()**

+---+-----+----------------+

| id| name| skills|

+---+-----+----------------+

| 1|veena|[dotnet, python]|

| 2|lochu| [java, aws]|

+---+-----+----------------+

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

|-- skills: array (nullable = true)

| |-- element: string (containsNull = true)

**from pyspark.sql.functions import transform**

**df.select('id','name',transform('skills',lambda x: upper(x)).alias('skills\_upper')).show()**

+---+-----+----------------+

| id| name| skills\_upper|

+---+-----+----------------+

| 1|veena|[DOTNET, PYTHON]|

| 2|lochu| [JAVA, AWS]|

+---+-----+----------------+

**from pyspark.sql.functions import transform**

**def convUpper(x):**

**return upper(x)**

**df.select(transform('skills',convUpper).alias('upper\_skills')).show()**

+----------------+

| upper\_skills|

+----------------+

|[DOTNET, PYTHON]|

| [JAVA, AWS]|

+----------------+

# 34. createOrReplaceTempView()

* Used to create temporary view on DataFrame to do selection and manipulation of data.
* Temporary views are session scoped and cannot be shared between the sessions

**d = [(1,'veena',2000),(2,'lochu',3000)]**

**schema = ['id','name','salary']**

**df = spark.createDataFrame(d,schema)**

**df.createOrReplaceTempView('emps')**

**df1=spark.sql('SELECT id,name FROM emps')**

**df1.show()**

+---+-----+

| id| name|

+---+-----+

| 1|veena|

| 2|lochu|

+---+-----+

**%sql**

**SELECT id,upper(name) as NAME FROM emps**

id NAME

1 VEENA

1. LOCHU

# createOrReplaceGlobalTempView() function in PySpark

* It’s used to create temp views or tables globally, when can be accessed across the sessions with in spark application
* To query these tables, we need append global\_temp.<tablename>

**Notebook1:**

**%scala**

**spark**

res0: org.apache.spark.sql.SparkSession = [org.apache.spark.sql.SparkSession@279c4543](mailto:org.apache.spark.sql.SparkSession@279c4543)

**d = [(1,'veena',2000),(2,'lochu',3000)]**

**schema = ['id','name','salary']**

**df = spark.createDataFrame(d,schema)**

**df.createOrReplaceTempView('emps')**

**Notebook2:**

**%scala**

**spark**

res0: org.apache.spark.sql.SparkSession = [org.apache.spark.sql.SparkSession@**16612d78**](mailto:org.apache.spark.sql.SparkSession@16612d78)

**%sql**

**SELECT id,upper(name) as NAME FROM emps**

AnalysisException: [[TABLE\_OR\_VIEW\_NOT\_FOUND](https://docs.databricks.com/error-messages/error-classes.html#table_or_view_not_found)] The table or view `emps` cannot be found. Verify the spelling and correctness of the schema and catalog.

* If print the data in one notebook by selecting the required from data base created in the other notebook : it may give error “table not found”
* To overcome this problem we use createOrReplaceGlobalTempView()

**Notebook1:**

**%scala**

**spark**

res0: org.apache.spark.sql.SparkSession = [org.apache.spark.sql.SparkSession@279c4543](mailto:org.apache.spark.sql.SparkSession@279c4543)

**d = [(1,'veena',2000),(2,'lochu',3000)]**

**schema = ['id','name','salary']**

**df = spark.createDataFrame(d,schema)**

**df.createOrReplaceGlobalTempView('empGlobal')**

**spark.catalog.currentDatabase()**

Out[3]: 'default'

**spark.catalog.listTables('default')**

Out[4]: []

**spark.catalog.listTables('global\_temp')**

Out[5]: [Table(name='empGlobal', catalog=None, namespace=['global\_temp'], description=None, tableType='TEMPORARY', isTemporary=True)]

**Notebook2:**

**%scala**

**spark**

res0: org.apache.spark.sql.SparkSession = [org.apache.spark.sql.SparkSession@**16612d78**](mailto:org.apache.spark.sql.SparkSession@16612d78)

**%sql**

**SELECT id,upper(name) as NAME FROM global\_temp.empGlobal**

id NAME

1 VEENA

2 LOCHU

* spark.catalog.dropGlobalTempView(‘empGlobal’) or spark.catalog.dropTempView(‘emps’) used to drop the views created

# UDF(user defined function)

* These are similar to function sin SQL. We define some logic in functions and store them in database and use them in queries
* Similar to that we can write our custom logic in python function and register it with PySpark using udf() function.

**d = [(1,'lochu',3000,500),(2,'veena',2000,1000)]**

**schema = ['id','name','salary','bonus']**

**df = spark.createDataFrame(d,schema)**

**def total(s,b):**

**return s+b**

**from pyspark.sql.functions import udf**

**from pyspark.sql.types import IntegerType**

**TotalPay = udf(lambda s,b:total(s,b),IntegerType())**

**df.withColumn('totPay',TotalPay(df.salary,df.bonus)).show()**

+---+-----+------+-----+------+

| id| name|salary|bonus|totPay|

+---+-----+------+-----+------+

| 1|lochu| 3000| 500| 3500|

| 2|veena| 2000| 1000| 3000|

+---+-----+------+-----+------+

@udf(returnType=IntegerType())

def total(s,b):

    return s+b

df.select('\*',total(df.salary,df.bonus).alias('totPay')).show()

+---+-----+------+-----+------+

| id| name|salary|bonus|totPay|

+---+-----+------+-----+------+

| 1|lochu| 3000| 500| 3500|

| 2|veena| 2000| 1000| 3000|

+---+-----+------+-----+------+

**d = [(1,'lochu',3000,500),(2,'veena',2000,1000)]**

**schema = ['id','name','salary','bonus']**

**df = spark.createDataFrame(d,schema)**

**df.createOrReplaceTempView('emps')**

**def total(s,b):**

**return s+b**

**spark.udf.register(name='TotalPay',f=total,returnType=IntegerType())**

id:long

name:string

salary:long

bonus:long

Out[10]: <function \_\_main\_\_.total(s, b)>

**%sql**

**SELECT \*,TotalPay(salary,bonus) as totPay FROM emps**

id name salary bonus totPay

1 lochu 3000 500 3500

2 veena 2000 1000 3000

# Convert RDD to Dataframe

* RDD (Resilient Distributed Dataset)
* Its collection of objects similar to list in Python.Its immutable and in memory processing
* By using parallelize() function of SparkContext you create an RDD

**d = [(1,'veena'),(2,'lochu')]**

**print(type(d))**

**rdd = spark.sparkContext.parallelize(d)**

**print(type(rdd))**

**print(rdd.collect())**

<class 'list'>

<class 'pyspark.rdd.RDD'>

[(1, 'veena'), (2, 'lochu')]

**df = rdd.toDF(schema=['id','name'])**

**df.show()**

+---+-----+

| id| name|

+---+-----+

| 1|veena|

| 2|lochu|

+---+-----+

**df = spark.createDataFrame(rdd,schema=['id','name'])**

**df.show()**

+---+-----+

| id| name|

+---+-----+

| 1|veena|

| 2|lochu|

+---+-----+

# map() transformation

* It’s RDD transformation used to apply function(lambda) on every element of RDD and returns new RDD
* DataFrame doesn’t have map() transformation to use with DataFrame you need to generate RDD first

**d=[('lochani','vilehya'),('veenanjali','tammina')]**

**rdd = spark.sparkContext.parallelize(d)**

**rdd1 = rdd.map(lambda x: x + (x[0]+' '+x[1],))**

**print(rdd1.collect())**

[('lochani', 'vilehya', 'lochani vilehya'), ('veenanjali', 'tammina', 'veenanjali tammina')]

**d=[('lochani','vilehya'),('veenanjali','tammina')]**

**df = spark.createDataFrame(d,['fn','ln'])**

**rdd1 = df.rdd.map(lambda x: x + (x[0]+' '+x[1],))**

**df1 = rdd1.toDF(['fn','ln','fullname'])**

**df1.show()**

+----------+-------+------------------+

| fn| ln| fullname|

+----------+-------+------------------+

| lochani|vilehya| lochani vilehya|

|veenanjali|tammina|veenanjali tammina|

+----------+-------+------------------+

**def fullname(x):**

**x=x+(x[0]+' '+x[1],)**

**return x**

**d=[('lochani','vilehya'),('veenanjali','tammina')]**

**df = spark.createDataFrame(d,['fn','ln'])**

**rdd1 = df.rdd.map(lambda x: fullname(x))**

**df1 = rdd1.toDF(['fn','ln','fullname'])**

**df1.show()**

+----------+-------+------------------+

| fn| ln| fullname|

+----------+-------+------------------+

| lochani|vilehya| lochani vilehya|

|veenanjali|tammina|veenanjali tammina|

+----------+-------+------------------+

# flatMap() transformation

* flatMap() is a transformation operation that flattens the RDD (array/map DataFrame columns) after applying the function on every element and returns a new PySpark RDD.
* Its not available in dataframes.Explode() functions can be used in dataframes to flatten arrays

**d=[('lochani vilehya'),('veenanjali tammina')]**

**rdd = spark.sparkContext.parallelize(d)**

**print("rdd - fullnames")**

**for i in rdd.collect():**

**print(i)**

**print('\n')**

**rdd1 = rdd.flatMap(lambda x: x.split(' '))**

**print("rdd - splitnames")**

**for i in rdd1.collect():**

**print(i)**

rdd - fullnames

lochani vilehya

veenanjali tammina

rdd - splitnames

lochani

vilehya

veenanjali

# partitionBy function in PySpark

* It’s used to partition large dataset into smaller files based on one or multiple columns

**d = [(1,'lochu','F','IT'),(2,'veena','F','HR'),(3,'chandu','M','IT')]**

**schema = ['id','name','gender','dept']**

**df = spark.createDataFrame(d,schema)**

**df.write.parquet(path='/FileStore/data/Optemp/',mode='overwrite',partitionBy='gender')**

**df.write.parquet(path='/FileStore/data/Optemp1/',mode='overwrite',partitionBy=['dept','gender'])**

**spark.read.parquet('/FileStore/data/Optemp/').show()**

+---+------+----+------+

| id| name|dept|gender|

+---+------+----+------+

| 3|chandu| IT| M|

| 1| lochu| IT| F|

| 2| veena| HR| F|

+---+------+----+------+

**spark.read.parquet('/FileStore/data/Optemp1/').show()**

+---+------+----+------+

| id| name|dept|gender|

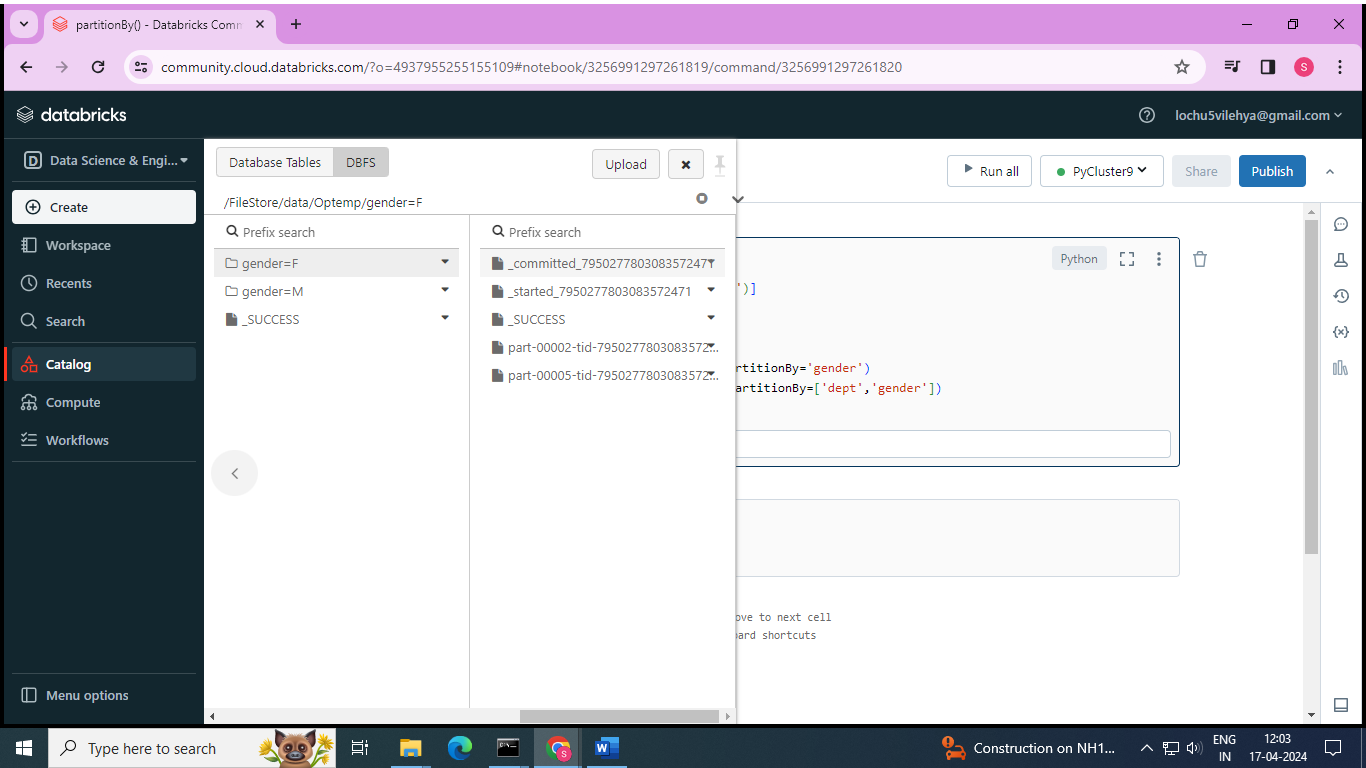
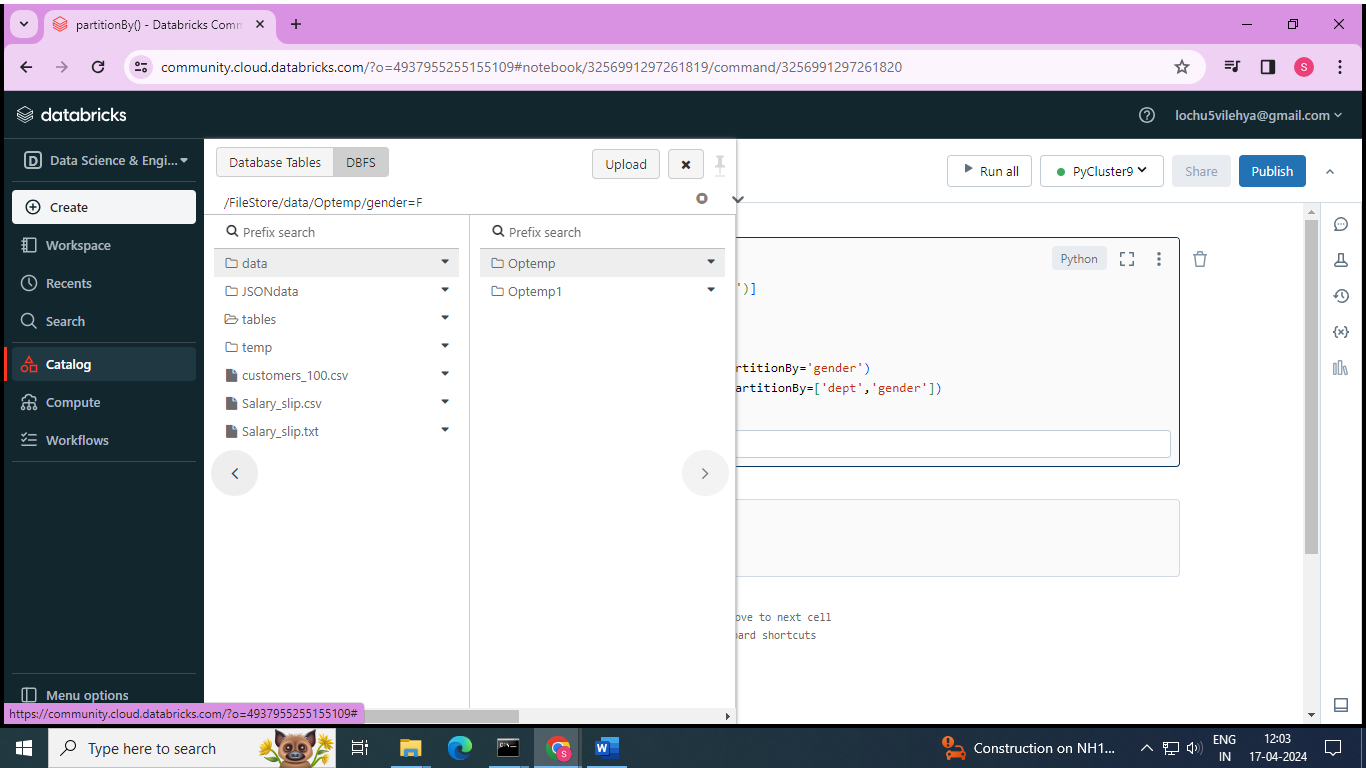
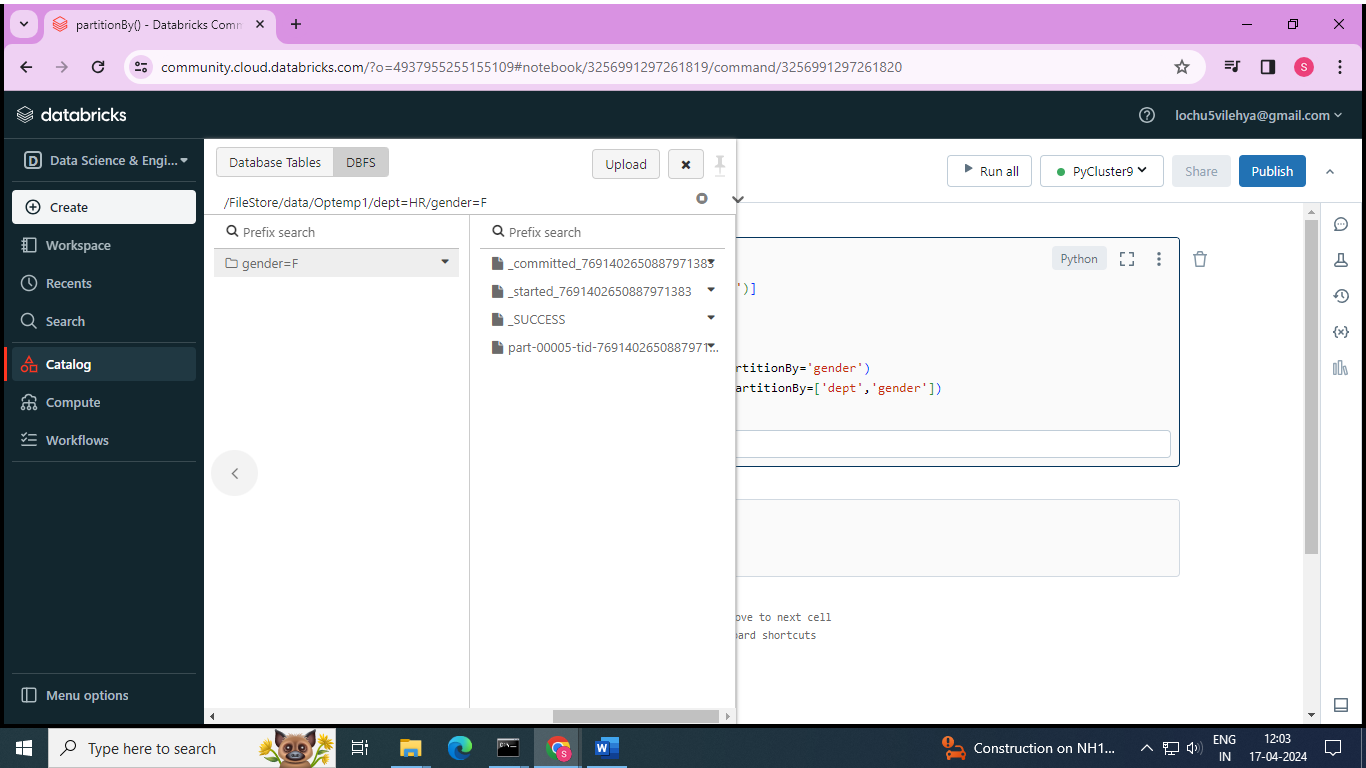
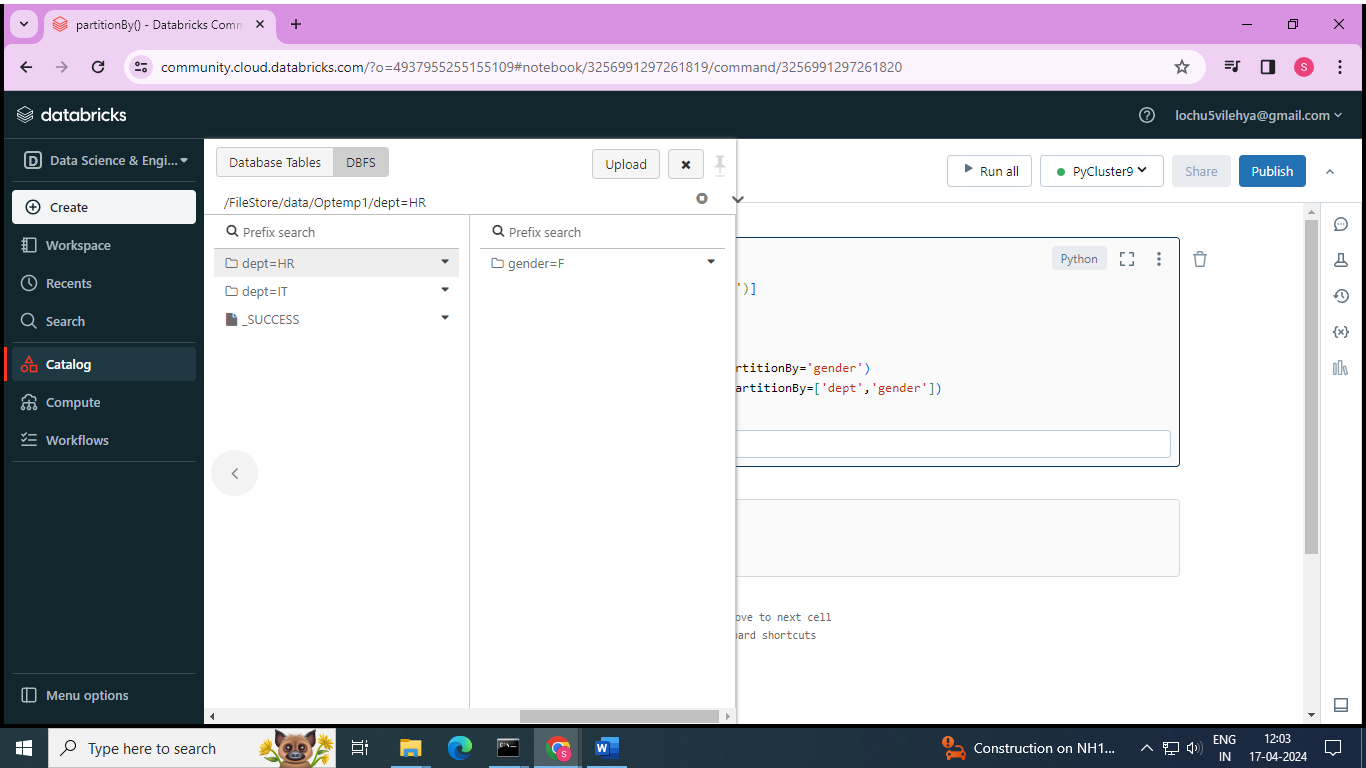
+---+------+----+------+

| 3|chandu| IT| M|

| 2| veena| HR| F|

| 1| lochu| IT| F|

+---+------+----+------+



# from\_json() function to convert json string into MapType and StructType

* its used to convert json string into MapType or structType

**d = [('veena','{"hair":"black","eye":"brown"}')]**

**schema = ['name','props']**

**df = spark.createDataFrame(d,schema)**

**df.show(truncate = False)**

**df.printSchema()**

name:string

props:string

+-----+------------------------------+

|name | props |

+-----+------------------------------+

|veena|{"hair":"black","eye":"brown"}|

+-----+------------------------------+

root

|-- name: string (nullable = true)

|-- props: string (nullable = true)

**from pyspark.sql.functions import from\_json**

**from pyspark.sql.types import MapType,StringType**

**mapSchema = MapType(StringType(),StringType())**

**df1 = df.withColumn('propsMap',from\_json(df.props,mapSchema))**

**df1.show(truncate=False)**

**df1.printSchema()**

+-----+------------------------------+-----------------------------+

|name | props | propsMap |

+-----+------------------------------+-----------------------------+ |veena|{"hair":"black","eye":"brown"}|{hair -> black, eye -> brown}|

+-----+------------------------------+-----------------------------+

root

|-- name: string (nullable = true)

|-- props: string (nullable = true)

|-- propsMap: map (nullable = true)

| |-- key: string

| |-- value: string (valueContainsNull = true)

**df2 = df1.withColumn('hair',df1.propsMap.hair)\**

**.withColumn('eye',df1.propsMap.eye)**

**df2.show(truncate=False)**

+-----+------------------------------+-----------------------------+-----+-----+ |name | props | propsMap |hair | eye |

+-----+------------------------------+-----------------------------+-----+-----+ |veena|{"hair":"black","eye":"brown"}|{hair -> black, eye -> brown}|black|brown| +-----+------------------------------+-----------------------------+-----+-----+

**from pyspark.sql.functions import from\_json**

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

**structSchema = StructType([\**

**StructField('hair',StringType()),\**

**StructField('eye',StringType())])**

**df1 = df.withColumn('propsStruct',from\_json(df.props,structSchema))**

**df1.show(truncate=False)**

**df1.printSchema()**

+-----+------------------------------+--------------+

|name | props | propsStruct |

+-----+------------------------------+--------------+ |veena|{"hair":"black","eye":"brown"}|{black, brown}|

+-----+------------------------------+--------------+

root

|-- name: string (nullable = true)

|-- props: string (nullable = true)

|-- propsStruct: struct (nullable = true)

| |-- hair: string (nullable = true)

| |-- eye: string (nullable = true)

**df2 = df1.withColumn('hair',df1.propsStruct.hair)\**

**.withColumn('eye',df1.propsStruct.eye)**

**df2.show(truncate=False)**

+-----+------------------------------+--------------+-----+-----+

|name | props | propsStruct |hair | eye |

+-----+------------------------------+--------------+-----+-----+ |veena|{"hair":"black","eye":"brown"}|{black, brown}|black|brown|

+-----+------------------------------+--------------+-----+-----+

# to\_json() function in PySpark

* to\_json() is used to convert DataFrame column MapType or StructType to JSON string

**from pyspark.sql.functions import to\_json**

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

**d = [('veena',{"hair":"black","eye":"brown"})]**

**schema = ['name','props']**

**df = spark.createDataFrame(d,schema)**

**df.show(truncate = False)**

**df.printSchema()**

+-----+-----------------------------+

|name | props |

+-----+-----------------------------+

|veena|{eye -> brown, hair -> black}|

+-----+-----------------------------+

root

|-- name: string (nullable = true)

|-- props: map (nullable = true)

| |-- key: string

| |-- value: string (valueContainsNull = true)

**df1 = df.withColumn('prop',to\_json(df.props))**

**df1.show(truncate=False)**

**df1.printSchema()**

+-----+-----------------------------+------------------------------+ |name |props |prop |

+-----+-----------------------------+------------------------------+ |veena|{eye -> brown, hair -> black}|{"eye":"brown","hair":"black"}| +-----+-----------------------------+------------------------------+

root

|-- name: string (nullable = true)

|-- props: map (nullable = true)

| |-- key: string

| |-- value: string (valueContainsNull = true)

|-- prop: string (nullable = true from pyspark.sql.functions import to\_json

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

**d = [('veena',("black","brown"))]**

**structSchema = StructType([\**

**StructField('name',StringType()),\**

**StructField('props',StructType([StructField('hair',StringType()),StructField('eye',StringType())]))])**

**df = spark.createDataFrame(d,structSchema)**

**df1 = df.withColumn('propsStruct',to\_json(df.props))**

**df1.show(truncate=False)**

**df1.printSchema()**

+-----+--------------+------------------------------+

|name |props |propsStruct |

+-----+--------------+------------------------------+

|veena|{black, brown}|{"hair":"black","eye":"brown"}|

+-----+--------------+------------------------------+

root

|-- name: string (nullable = true)

|-- props: struct (nullable = true)

| |-- hair: string (nullable = true)

| |-- eye: string (nullable = true)

|-- propsStruct: string (nullable = true)

# json\_tuple() function in PySpark

* json\_tuple() function is used to query or extract elements from json string column and create as new columns

**d = [('veena','{"hair":"black","eye":"brown","skin":"brown"}'),('lochu','{"hair":"brown","eye":"blue","skin":"white"}')]**

**schema = ['name','props']**

**df = spark.createDataFrame(d,schema)**

**df.show(truncate = False)**

**df.printSchema()**

+-----+---------------------------------------------+

|name |props |

+-----+---------------------------------------------+ |veena|{"hair":"black","eye":"brown","skin":"brown"}| |lochu|{"hair":"brown","eye":"blue","skin":"white"} |

+-----+---------------------------------------------+

root

|-- name: string (nullable = true)

|-- props: string (nullable = true)

**from pyspark.sql.functions import json\_tuple**

**df2 = df.select(df.name,json\_tuple(df.props,'hair','skin').alias('hair','skin'))**

**df2.show()**

+-----+-----+-----+

| name| hair| skin|

+-----+-----+-----+

|veena|black|brown|

|lochu|brown|white|

+-----+-----+-----+

# get\_ json\_object() function in PySpark

* its used to extract the json string based on path from JSON column
* here , $ means the root node

**d = [('veena','{"genes":{"hair":"black","eye":"brown","skin":"brown"},"gender":"F"}'),\**

**('lochu','{"genes":{"hair":"brown","eye":"blue","skin":"white"},"gender":"F"}')]**

**schema = ['name','props']**

**df = spark.createDataFrame(d,schema)**

**df.show(truncate = False)**

+-----+--------------------------------------------------------------------+

|name | props |

+-----+--------------------------------------------------------------------+ |veena|{"genes":{"hair":"black","eye":"brown","skin":"brown"},"gender":"F"}| |lochu|{"genes":{"hair":"brown","eye":"blue","skin":"white"},"gender":"F"} |

+-----+--------------------------------------------------------------------+

**from pyspark.sql.functions import get\_json\_object**

**df.select('name',get\_json\_object('props','$.gender').alias('gender')).show()**

+-----+------+

| name|gender|

+-----+------+

|veena| F|

|lochu| F|

+-----+------+

**from pyspark.sql.functions import get\_json\_object**

**df.select('name',get\_json\_object('props','$.genes.hair').alias('hair colour')).show()**

+-----+-----------+

| name|hair colour|

+-----+-----------+

|veena| black|

|lochu| brown|

+-----+-----------+

**from pyspark.sql.functions import get\_json\_object**

**df.select('name',get\_json\_object('props','$.genes.hair').alias('hair colour'),get\_json\_object('props','$.genes.skin').alias('skin colour')).show()**

+-----+-----------+-----------+

| name|hair colour|skin colour|

+-----+-----------+-----------+

|veena| black| brown|

|lochu| brown| white|

+-----+-----------+-----------+

# Date functions in PySpark | current\_date(), to\_date(), date\_format(), datediff(), months\_between(), add\_months(), date\_add(), month(), year() functions

* DateType default format is **yyyy-MM-dd**
* **current\_date()** get the current system date. By default , the data will be returned in yyyy-dd-MM format
* **date\_format()** to parses the date and converts from yyyy-MM-dd to specified format.
* **to\_date()** converts date string into datetype. We need to specify format of date in the string in the function

**from pyspark.sql.functions import current\_date,date\_format,lit,to\_date**

**df = spark.range(1)**

**df1 = df.withColumn('todays\_Date',current\_date())**

**df1.show()**

**df2 = df1.withColumn('newFormat',date\_format(lit(df1.todays\_Date),'MM.dd.yyyy'))**

**df2.show()**

**df3 = df2.withColumn('newDatecol',to\_date(lit(df2.newFormat),'MM.dd.yyyy'))**

**df3.show()**

**df3.printSchema()**

df:pyspark.sql.dataframe.DataFrame = [id: long]

df1:pyspark.sql.dataframe.DataFrame = [id: long, todays\_Date: date]

df2:pyspark.sql.dataframe.DataFrame = [id: long, todays\_Date: date ... 1 more field]

df3:pyspark.sql.dataframe.DataFrame = [id: long, todays\_Date: date ... 2 more fields]

+---+-----------+

| id|todays\_Date|

+---+-----------+

| 0| 2024-04-17|

+---+-----------+

+---+-----------+----------+

| id|todays\_Date| newFormat|

+---+-----------+----------+

| 0| 2024-04-17|04.17.2024|

+---+-----------+----------+

+---+-----------+----------+----------+

| id|todays\_Date| newFormat|newDatecol|

+---+-----------+----------+----------+

| 0| 2024-04-17|04.17.2024|2024-04-17|

+---+-----------+----------+----------+

root

|-- id: long (nullable = false)

|-- todays\_Date: date (nullable = false)

|-- newFormat: string (nullable = false)

|-- newDatecol: date (nullable = true)

**from pyspark.sql.functions import datediff,months\_between,add\_months,date\_add,year,month**

**df = spark.createDataFrame([('2015-04-18','2015-05-08')],['d1','d2'])**

**df1 = df.withColumn('diff',datediff(df.d2,df.d1))**

**df2 = df1.withColumn('monthsBetween',months\_between(df.d2,df.d1))**

**df3 = df2.withColumn('addmonth',add\_months(df.d2,4))**

**df4 = df3.withColumn('submonth',add\_months(df.d2,-4))**

**df4.show()**

+----------+----------+----+-------------+----------+----------+

| d1| d2|diff|monthsBetween| addmonth| submonth|

+----------+----------+----+-------------+----------+----------+

|2015-04-18|2015-05-08| 20| 0.67741935|2015-09-08|2015-01-08|

+----------+----------+----+-------------+----------+----------+

**df1 = df.withColumn('addDate',date\_add(df.d2,4))**

**df2 = df1.withColumn('subdate',date\_add(df.d2,-4))**

**df3 = df2.withColumn('year',year(df.d1))**

**df4 = df3.withColumn('month',month(df.d2))**

**df4.show()**

+----------+----------+----------+----------+----+-----+

| d1| d2| addDate| subdate|year|month|

+----------+----------+----------+----------+----+-----+

|2015-04-18|2015-05-08|2015-05-12|2015-05-04|2015| 5|

+----------+----------+----------+----------+----+-----+

# Timestamp Functions

* timestampType default format **is yyyy-MM-dd HH:mm:ss.SS**
* **current\_timestamp()** get the current timestamp . By default , the data will in default format
* **to\_timestamp()** converts timestamp string into Timestamptype . We need format of timestamp in the string the function
* **hour(),minute(),second()** functions

**from pyspark.sql.functions import current\_timestamp,to\_timestamp,lit,hour,minute,second**

**df = spark.range(1)**

**df1 = df.withColumn('timestamp',current\_timestamp())**

**df1.show(truncate=False)**

**df1.printSchema()**

**df2 = df1.withColumn('toTimestamp',lit('25.12.2022 06.10.13.45'))**

**df3 = df2.withColumn('toTimestamp',to\_timestamp(df2.toTimestamp,'dd.MM.yyyy HH.mm.ss.SSS'))**

**df3.show(truncate=False)**

**df3.printSchema()**

+---+-----------------------+

|id | timestamp |

+---+-----------------------+

| 0 |2024-04-17 11:33:24.026|

+---+-----------------------+

root

|-- id: long (nullable = false)

|-- timestamp: timestamp (nullable = false)

+---+-----------------------+----------------------+

|id | timestamp | toTimestamp |

+---+-----------------------+----------------------+

| 0 |2024-04-17 11:33:24.331|2022-12-25 06:10:13.45|

+---+-----------------------+----------------------+

root

|-- id: long (nullable = false)

|-- timestamp: timestamp (nullable = false)

|-- toTimestamp: timestamp (nullable = true)

**df1.select('\*',hour(df1.timestamp).alias('hour'),\**

**minute(df1.timestamp).alias('minute'),\**

**second(df1.timestamp).alias('second')).show(truncate=False)**

+---+-----------------------+----+------+------+

|id | timestamp |hour|minute|second|

+---+-----------------------+----+------+------+

| 0 |2024-04-17 11:34:57.659| 11 | 34 | 57 |

+---+-----------------------+----+------+------+

# Aggregate functions : approx\_count\_distinct(), avg(), collect\_list(), collect\_set(), countDistinct(), count()

* Aggregate functions operate on a group of rows and calculate a single return value for every group
* Approx\_count\_distinct() = returns the count of distinct items in a group of rpws
* Avg() = returns average of values in agroup of rows
* Collect\_list() = returns all values from input column as list with duplicates
* Collect\_set() = returns all values from input column as list without duplicates
* CountDistinct() = returns number od distinct elements in input column
* Count() = return number of elements in a column

**from pyspark.sql.functions import approx\_count\_distinct,avg**

**data = [('lochu','HR',1500),('veena','IT',3000),('hiii','HR',1500)]**

**schema = ['name','dept','salary']**

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

**df.show()**

**df.select(approx\_count\_distinct('salary')).show()**

**df.select(avg('salary')).show()**

+-----+----+------+

| name|dept|salary|

+-----+----+------+

|lochu| HR| 1500|

|veena| IT| 3000|

| hiii| HR| 1500|

+-----+----+------+

+-----------------------------+

|approx\_count\_distinct(salary)|

+-----------------------------+

| 2|

+-----------------------------+

+-----------+

|avg(salary)|

+-----------+

| 2000.0|

+-----------+

**from pyspark.sql.functions import \***

**df.select(collect\_list('salary')).show()**

**df.select(collect\_set('salary')).show()**

**df.select(countDistinct('dept')).show()**

**df.select(count('salary')).show()**

+--------------------+

|collect\_list(salary)|

+--------------------+

| [1500, 3000, 1500]|

+--------------------+

+-------------------+

|collect\_set(salary)|

+-------------------+

| [3000, 1500]|

+-------------------+

+--------------------+

|count(DISTINCT dept)|

+--------------------+

| 2|

+--------------------+

+-------------+

|count(salary)|

+-------------+

| 3|

+-------------+

# row\_number(), rank(), dense\_rank() functions

* we need to partition the data using Window. partitionBy() , and for row number and rank function we need to additionally order by on partition data using orderBy clause
* row\_number() window function is used to give the sequential row number starting from 1 to the result of each window partition
* rank() window function is used to provide a rank to the result within a window partition. This function leaves gaps in the rank when there are ties.
* 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

**from pyspark.sql.functions import row\_number,rank,dense\_rank**

**from pyspark.sql.window import Window**

**data = [('lochu','HR',1500),('veena','IT',3000),('hiii','HR',1500),('chandu','IT',8500),('vara','HR',4500),('sai','HR',5500),('jyo','IT',7500)]**

**schema = ['name','dept','salary']**

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

**df.show()**

+------+----+------+

| name|dept|salary|

+------+----+------+

| lochu| HR| 1500|

| veena| IT| 3000|

| hiii| HR| 1500|

|chandu| IT| 8500|

| vara| HR| 4500|

| sai| HR| 5500|

| jyo| IT| 7500|

+------+----+------+

**df.sort('dept').show()**

**win = Window.partitionBy('dept').orderBy('salary')**

**df.withColumn('rowno.',row\_number().over(win)).\**

**withColumn('rank',rank().over(win)).\**

**withColumn('denseRank',dense\_rank().over(win)).show()**

+------+----+------+

| name|dept|salary|

+------+----+------+

| vara| HR| 4500|

| lochu| HR| 1500|

| sai| HR| 5500|

| hiii| HR| 1500|

| veena| IT| 3000|

| jyo| IT| 7500|

|chandu| IT| 8500|

+------+----+------+

+------+----+------+------+----+---------+

| name|dept|salary|rowno.|rank|denseRank|

+------+----+------+------+----+---------+

| lochu| HR| 1500| 1| 1| 1|

| hiii| HR| 1500| 2| 1| 1|

| vara| HR| 4500| 3| 3| 2|

| sai| HR| 5500| 4| 4| 3|

| veena| IT| 3000| 1| 1| 1|

| jyo| IT| 7500| 2| 2| 2|

|chandu| IT| 8500| 3| 3| 3|

+------+----+------+------+----+---------+