SPARK (Venu Sir) *–By Nirupam*

Spark is a framework written in scala language.

Recommended Spark link : https://archive.apache.org/dist/spark/spark-3.1.2/

To Check if Spark is installed or not , in command prompt run > spark-shell

To install spark first download the file and unzip it. Now in the environment variable set SPARK_HOME and create two paths for spark.

First path is -> spark path\bin

Second path is -> spark path\sbin

- Now open anaconda prompt and check > python
  (By default it will show python 3.9, but we need python 3.8 [recomended])
- To Install python 3.8 run [in anaconda prompt] > conda create -n py38 python=3.8

**We can run our program also in command prompt after running spark-shell but code/data wont be saved here,so this prompt can be used for testing purpose but for saving the code we will go with pycharm.

- **Pycharm configuration for our spark project** :

  First open pycharm>click on file>settings>Project:[project name]>Project Interpreter (here by default it will show python 3.10)> in the right side click on settings(gear) sign>click on add>from the left side bar click on conda environment>and in the main screen choose existing environment>choose the path of Interpreter (C:\Users\pc\anaconda3\envs\py38\python.exe)>ok>Now python interpreter will be shown with python 3.8>now click on Project structure from the side bar>In the right side click on +Add content Root> Now choose the path of python 2 dependencies from spark folder(C:\bigdata\spark-3.1.2-bin-hadoop3.2\python\lib\py4j-0.10.9-src.zip),(C:\bigdata\spark-3.1.2-bin-hadoop3.2\python\lib\pyspark.zip)>Now click on ok

*It is also recommended to create a new environment variable named **PYSPARK_PYTHON** with variable value : **C:\Users\pc\anaconda3\envs\py38\python.exe**

Path : **C:\Users\pc\anaconda3\envs\py38\python.exe**

Now the configuration part is done. It is recommended to create a package first and inside the package create .py files.

Ex -> create a new package with name sparksql and now inside sparksql create a file test.py.

***To write any pyspark program it is mandatory to write these three lines mentioned below :**

from pyspark.sql import *

from pyspark.sql.functions import *

spark = SparkSession.builder.master("local[2]").appName("test").getOrCreate()

Everytime we open a file to write a pyspark code we need those first 3 line compulsorily

To get rid of this we can save these 3 lines as by default lines with the following steps :

File>settings>editor>file and code templates>python script>now in this blank part write these three lines and press ok button

from pyspark.sql import *

from pyspark.sql.functions import *

spark = SparkSession.builder.master("local[2]").appName("test").getOrCreate()

Q. Why Spark ?

Ans :

1. Hive,pig only process with historical/stored data, strom processes live data and neo4j/FraphFrames only process graphical data. But Spark is a unified platform, it works with both historical, live and graphical data. That's why it is highly used.

2. It does all the operations in memory, that's why it is very fast. Its execution is 100x faster than map-reduce job.

Some Important Definitations :

- CONTEXT = A nutshell to create different APIs.
- SparkContext=> It is used to create RDD APIs
- SqlContext => It is used to create dataframe APIs
- HiveContext=> It is used to connect hive.
- SparkStreaming Context => It is used to process streaming data.
- SparkSession Context=>It is used for unifying all contexts and all APIs.

NOTE :

Spark understands everything in the form of RDD only.

WHAT IS RDD (Resilent Distributed Datasets)?

Ans- Collection of JVM objects having 3 properties **immutable, fault tolerance, Laziness.**

Two ways To create RDDs:

1. sc.parallalize() -> Converts Java/scala/python variables or objects into RDD.
2. sc.textfile() -> Converts external data(eg – hdfs/local/s3/physical data) into RDD.

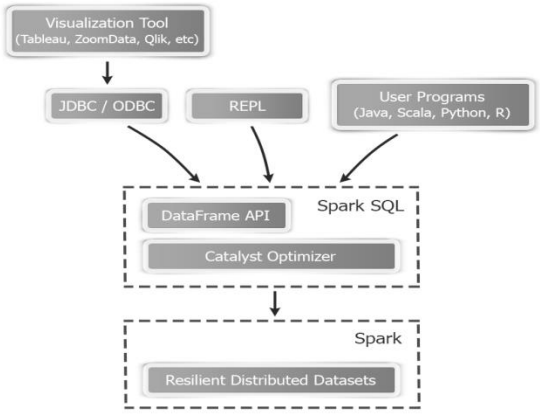
****By default RDD supports txt and csv files only**.

First pyspark program (to retrieve all the elements of a list using pyspark) :

```
from pyspark.sql import *

from pyspark.sql.functions import *

spark=SparkSession.builder.master("local[2]").appName("test").getOrCreate()

data=[1,2,3,4]

drdd=spark.sparkContext.parallelize(data)  #creating rdd from list

for i in drdd.collect():   #collect() method is used to show the data from rdd

   print(i)
```

Q. Can we say DataFrame is the replacement of RDD ??

Ans : No. Dataframe is just an extension to RDD, not replacement of RDD. DataFrame accepts data from different resources and submits to Catalyst Optimizer. This Catalyst Optimizer internally converts these data in the form of RDD. For better understanding observe the figure below -



Day 2

Some Important functions :

map() -> Runs operation on each element of any collections(list/array etc).

Scala> val nums=Array(1,2,3,4)

Scala> nums.map(x=>x*4) (each element gets multiplied by 4)

Output =>Array(4,8,12,16)

Filter() -> Runs on collections(list/array) and returns only filtered elements

(syntax supported for scala)

val nums =Array(1,2,3,4)

nums.filter(x=>x%2==0) [to get only even numbers]

output = Array(2,4)

In case of spark if we want to do any operation we must have to convert it into RDD first :

(suitable for pycharm)

```
Ex =>          lst=[1,2,3,4]
               nrdd=sc.parallelize(lst)    #converting the list into rdd
               res= nrdd.map(lambda x: x*4)
               for i in res.collect():
                  print(i)
```

**Here if we don't use collect() method all the operations will be done, but no output will be shown to us. This is known as laziness of RDD.

Transformations : A function/method feels lazy to process is called Transformation. It returns RDD. For better understanding we can compare this same as DDL of SQL.

Ex- map(), filter() ,distinct() etc

Action : A function/method which does not feel lazy to process is called Action. It returns values. We can compare it as DML in SQL. Ex- collect(),take(), first() etc

NOTE :

RDD is a programming model API. It does not support SQL.

In the other hand DataFrame is both programing and SQL model API.

Sample data : asl.csv

name,age,city

venu,32,hyd

anu,49,mas

jyo,12,blr

koti,29,blr

mastan,30,blr

hydera,99,mas

blru,11,hyd

Q. A spark program to filter data using Transformations and Actions

```
from pyspark.sql import *

from pyspark.sql.functions import *

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

sc=spark.sparkContext

#data=[12,32,34,4,54,26]

#drdd=spark.sparkContext.parallelize(data)

data="C:\\bigdata\\asl.csv"    #provide the path of your dataset with '\\' sign

aslrdd=sc.textFile(data)

res=aslrdd.map(lambda x :x.split(',')).filter(lambda x: "hyd" in x)

#filter by default apply a logic /filter on top of entire line

'''filter almost in sql ur using where condition to filter

results similarly ur using filter function to filter values.'''

for i in res.collect():

    print(i)
```

Q. Spark program rddusercase 2

```
from pyspark.sql import *

from pyspark.sql.functions import *

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

#sc=spark.sparkContext

data="C:\\bigdata\\asl.csv"

aslrdd=spark.sparkContext.textFile(data)

#res=aslrdd.map(lambda x:x.split(",")).filter(lambda x: "blr" in x[2])

res=aslrdd.filter(lambda x: "age" not in x).map(lambda x:x.split(",")).filter(lambda x: int(x[1])>=30)

for i in res.collect():

    print(i)
```

Q. Spark Program to retrive data using Dataframe concept

```
from pyspark.sql import *
from pyspark.sql.functions import *
spark = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
#sc=spark.sparkContext
data="C:\\bigdata\\asl.csv"
aslrdd=spark.sparkContext.textFile(data)
#res=aslrdd.map(lambda x:x.split(",")).filter(lambda x: "blr" in x[2])
res=aslrdd.filter(lambda x: "age" not in x).map(lambda x:x.split(",")).toDF(["name","age","city"])
#res.show()
res.createOrReplaceTempView("tab") #giving a title to the Dataframe
result=spark.sql("select * from tab where city='hyd'") #sql friendly
#result=res.where(col("city")=="hyd") #programing friendly
result.show()
```

Q. program to Handle datasets which are not in proper structure :

Sample Data : (file name chatlog.txt)

Nirupam Mondal (to Everyone): 07:03: nirupammondal@gmail.com

Nitin (to Everyone): 07:03: Nitin Naikwadi

Akesh Tonge (to Everyone): 07:03: Akesh Tonge

Karan (to Everyone): 07:03: KaranPatil@gmail.com

Himanshu Salunkhe (to Everyone): 07:03: HimanshuSalunkhe@gmail.com

Mayuri Nidhonkar (to Everyone): 07:03: Mayuri Nidhonkar

If we look carefullv into the above dataset .few members have email id and few of them don't have. Below is the spark program to retrieve name and email address of the members and ignore those who don't have mentioned email id.

```
from pyspark.sql import *

from pyspark.sql.functions import *

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

data="D:\\Spark Venu sir\\drivers\\chatlog.txt"

erdd=spark.sparkContext.textFile(data)

#res=erdd.flatMap(lambda x:x.split(" "))  #this method separates every delimited elemets into new line

#res=erdd.flatMap(lambda x:x.split(" ")).filter(lambda x: "@" in x)

res=erdd.filter(lambda x: "@" in x).map(lambda x:x.split(" ")).map(lambda x: (x[0],x[-1]))

for i in res.collect():

   print(i)
```

NOTE :

As the above dataset is not in proper structure/format it can't be handled directly with dataframe. We have to handle this type of of dataset with rdd first and then we can use dataframe on this.

DAY-3

reduceByKey() :

Spark RDD reduceByKey() transformation is used to merge the values of each key using an associative reduce function.

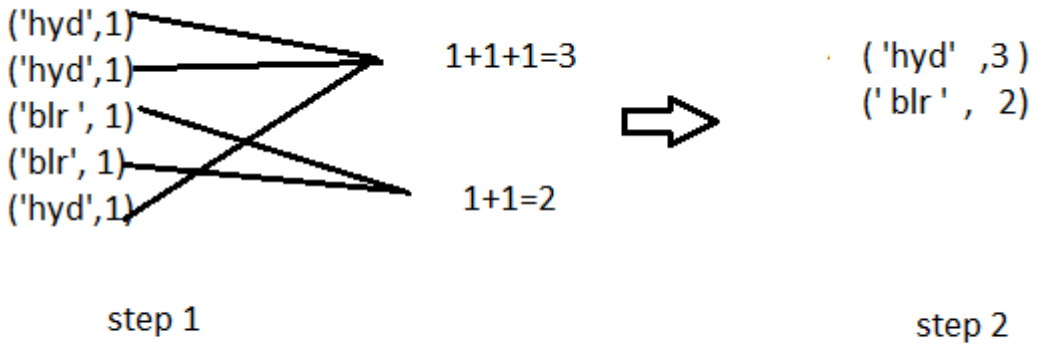
Syantax : reduceByKey(lambda x,y : x+y)

Here x is the 1st value of the column and y is the 2nd value of the column.

For better understanding see the image below :

By doing x+y each time 1 is added corresponding to that same grouped rows.

It is similar to group by in SQL.



Q. An example program using reduceByKey

```
from pyspark.sql import *
from pyspark.sql.functions import *

spark = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
sc=spark.sparkContext
data = "C:\\bigdata\\asl.csv"
aslrdd = sc.textFile(data)
#select * from tab where city='hyd'
#res=aslrdd.filter(lambda x: "age" not in x).map(lambda x:x.split(",")).filter(lambda x: "hyd" in x)
res=aslrdd.filter(lambda x: "age" not in x).map(lambda x:x.split(",")).map(lambda x:(x[2],1)).reduceByKey(lambda x,y:x+y)
for i in res.collect():
   print(i)
'''
group by you are using ... category based col and something aggregation mandatory
if u want to group the value first u must use reduceByKey ... its used to group the values.
reduceByKey (any function/method ends with Key, means data must be in (key, value) format
reduceByKey ... based on same key , data process the values ..
reduceByKey (x, y: x+y) ..4
'''
```

sortBY() -> This function is used to sort the records in ascending or descending order.

Syntax : sortBy(lambda x: x[1],ascending=True/False)

Dataset: donation.txt

```
name,donation
nirupam,10000
raju,2000
shyam,5000
nirupam,2000
shyam,5000
srinu,4000
```

Q. An example program to show sortBy() Transformation (Showing name and donation amt in ascending order)

```
from pyspark.sql import *
from pyspark.sql.functions import *
spark = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
sc = spark.sparkContext
data="D:\\Spark Venu sir\\drivers\\donation.txt"
drdd=sc.textFile(data)
pro = drdd.filter(lambda x: "donation" not in x).map(lambda x:x.split(",")).map(lambda
x:(x[0],int(x[1])))\
   .reduceByKey(lambda x,y:x+y).sortBy(lambda x:x[1],ascending=True)

for i in pro.collect():
   print(i)
```

distinct() -> This transformation is used to retrieve the unique/distinct records only.

Shuffling : A process of transfering data from one executor to another executor for further processing is called Shuffling

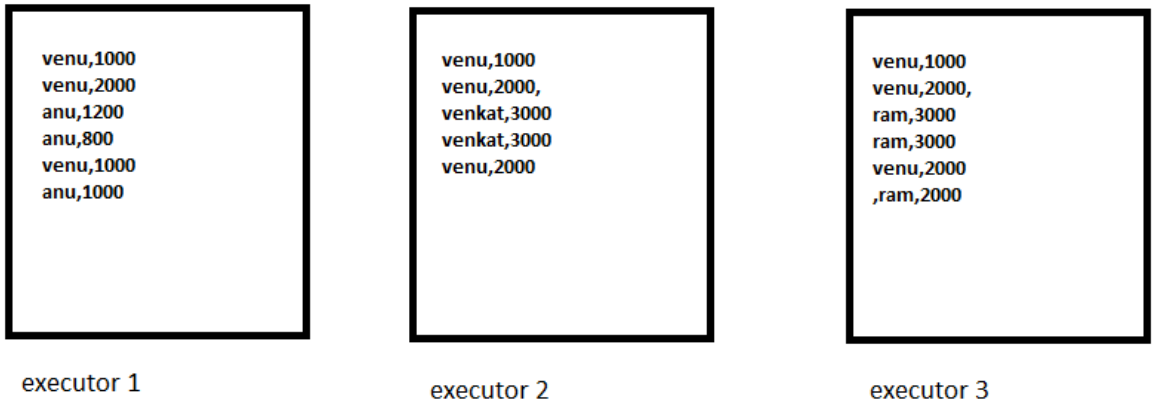
Narrow Transformation : A Transformation processes within same executor without shuffle is called Narrow Transformation.

Wide Transformation : A Transformation processes data from one executor to another executor is called Wide Transformation. Shuffling will be involved in case of Wide Transformation.

Q. Difference between reduceByKey() and groupByKey()

Both the reduceByKey() and groupByKey() transformations are used for grouping the dataset/records, but the main difference between them is reduceByKey() combines the data first inside each executor then the combined records are sent to another executors for shuffling. But in the other hand groupByKey() does not combine any records , rather it sends all the records among the executors for shuffling. So we can say reduceByKey() is more optimized way than groupByKey().

Lets understand the difference with an example :



Above there are three executors. In case of reduceByKey() in each executor data will be combined like -

executor 1 : venu,(1000+2000+1000)=4000
Anu,(1200+800+1000)=3000

Executor 2 : venu,5000
Venkat,6000

Executor 3: venu,5000
Ram,8000

After the records are combined in each executor , then they are sent for shuffling,so number of records are less while shuffling.

But In case of groupByKey() all the records are directly sent for shuffling from each executor, so number of records sent for shuffling are more compared to reduceByKey().

So we can conclude that reducedBYkey() is more optimized way than groupByKey().

DAY -4

- **Handling data with Dataframe making the First line as column names:**

Sample data : asl.csv

name,age,city

venu,32,hyd

anu,49,mas

jyo,12,blr

koti,29,blr

In this sample data if we normally load the data into dataframe then by default the first line is considered as a record, but if we want to make the first line as column names then the code will be like :

```
from pyspark.sql import *

from pyspark.sql.functions import *

# creating SparkSession object

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

data="D:\\Spark Venu sir\\drivers\\asl.csv"

df=spark.read.format("csv").option("header","true").load(data)

#if u mention header true, first line of data is considered as columns

df.show()
```

NB: Here **.option("header","true")** this function is making first line as columns of the dataframe. By default the value of header is false, for this kind of scenario we have to set the header value as true inside option() function.

- **Suppose I have data like first line is the description about the data and 2nd line is the header . In this case we have to skip the first the first line first and then we have to make the 2nd line as column names with the help of header=True**

Sample Data : sample.csv

this is the details about data
name,age,donation
nirupam,25,10000
raju,26,2000
shyam,27,5000
nirupam,25,2000
shyam,27,5000
srinu,29,4000

Code :

```
from pyspark.sql import *
from pyspark.sql.functions import *
spark = SparkSession.builder.master("local[*]").appName("sparkdf").getOrCreate()

#if u have any mal records like first line second line wrong clean that data using rdd or udf.
#skip first line second line onwards original data available.
data="D:\\Spark Venu sir\\drivers\\sample.csv"
rdd=spark.sparkContext.textFile(data)
skip=rdd.first()
odata= rdd.filter(lambda x:x!=skip)
df=spark.read.csv(odata,header=True,inferSchema=True)
df.printSchema()
#printing columns and its datatype in nice tree format.
df.show(5)
#display top 20 rows by default.if u want to display top 5 lines use show(5)
```

Elaboration :

skip=rdd.first() -> with the help of first() method we are storing the fisrt line into skip variable.
odata= rdd.filter(lambda x:x!=skip)-> Here we are filtering the rdd by eliminating skip variable. So in the variable odata first line is removed.
header=True -> Here we are making the first line of ordd as header/column name.
inferSchema -> InferSchema refers to the datatype of the columns, By delault it is False, and by default all the columns' datatypes are string. But If we set **inferSchema=True** then it will assign appropriate datatypes to all the column names.
df.printSchema() -> This method is used to print/show the schema of the dataframe in spark.
df.show(5) -> This method is used to show the results. **By default show() method prints 20 records only.** Here we mentioned show(5) that's why it will show top 5 rows only.

➢ Some Operations On data using spark Programming.

Sample data : bank-full.csv

```
"age";"job";"marital";"education";"default";"balance"
91;"management";"married";"tertiary";"no";2143
40;"technician";"single";"secondary";"no";29
33;"entrepreneur";"married";"secondary";"no";2
47;"blue-collar";"married";"unknown";"no";1506
33;"unknown";"single";"unknown";"no";1
```

CODE :

```
from pyspark.sql import *
from pyspark.sql.functions import *

# creating SparkSession object
spark = SparkSession.builder.master("local[*]").appName("sparkdf").getOrCreate()
data="D:\\Spark Venu sir\\drivers\\bank-full.csv"
df=spark.read.format("csv").option("header","true").option("sep",";").option("inferSchema","true").load(data)
#sep option used to specify delimiter
#by default  spark considers every field as string, but i want to change columns appropriate datatype like int,
double, string, use inforschema, true option
#if u do not mention like this 1000+1000 ... if int .. 2000 if string, u ll get 10001000

#data processing programming friendly
#res=df.where(col("age")>90)
#res=df.select(col("age"),col("marital"), col("balance")).where((col("age")>60) & (col("marital")!="married"))
#res=df.where((col("age")>60) | (col("marital")=="married") & (col("balance")>=40000))
#res=df.where(((col("age")>60) | (col("marital")=="married")) & (col("balance")>=40000))
#res=df.groupBy(col("marital")).agg(sum(col("balance")).alias("smb")).orderBy(col("smb").desc())
#res=df.groupBy(col("marital")).count()
res=df.groupBy(col("marital")).agg(count("*").alias("cnt"),sum(col("balance")).alias("smb"))
   #where(col("balance")>avg(col("balance")))


#process sql friendly
df.createOrReplaceTempView("tab")
#createOrReplaceTempView .. register this dataframe as a table. its very useful to run sql queries.
#res=spark.sql("select * from tab where age>60 and balance>50000")
#res=spark.sql("select marital, sum(balance) sumbal from tab group by(marital)")
#married, how much balance they have
#divorced , how much balance they have

res.show()
#res.printSchema()
```

Important Points :

option("sep",";") -> By default spark rdd accepts comma separated values, but if the delimiter is different then we have to use this method. Here in our dataset delimiter is " ; " that's why we have used **("sep" , ";")**

option("inferSchema","true") -> We need to set **inferSchema as true** because it chooses exact datatype for each column. If inferSchema is false(by default), in that case if we want to perform summation, records will be concatenated as strings. Ex- 10+20=1020

AND ,OR -> In programming friendly operation symbol of AND operator is **(&)** and symbol for OR operator is **(|)**

agg() -> This function is used to mention any aggregation operation on any dataset.
Ex -> agg(sum(col("salary")))

df.createOrReplaceTempView("tab") -> This createOrReplaceTempView() function registers the dataframe df in a table named tab. It is useful for running SQL queries.
To See particular columns Only -> df.select(col("col_name1"),col("col_name2"))

➢ Operations On Data in pyspark program :

```
from pyspark.sql import *
from pyspark.sql.functions import *

# creating SparkSession object
spark = SparkSession.builder.master("local[*]").appName("sparkdf").getOrCreate()
data=" D:\\Spark Venu sir\\drivers\\10000Records.csv"
df=spark.read.format("csv").option("header","true").option("sep",",").option("inferSchema","true").load(data)
import re
#num = int(df.count())
#df.show(num,truncate=True)
cols=[re.sub('[^a-zA-Z0-9]',"",c.lower()) for c in df.columns]
# re .. replace .. except all Small letters, capital letters and number excetopt those any other symbols if u have
replace/remove

ndf =df.toDF(*cols)
#toDF used to rename all cloumns , and convert rdd to dataframe ... at that time use toDF

ndf.show(21,truncate=True)
#by default show method showing top 20rows and if any field having more than 20 chars its truncated and shows
...
ndf.printSchema()
#dataframe column names and its data type display properly
#data processing programming friendly (datframe api)
res=ndf.groupBy(col("gender")).agg(count(col("*")).alias("cnt"))
res.show()
```

SOME IMPORTANT POINTS :

num = int(df.count())
df.show(num,truncate=True) ->In the above line we are counting total number of records in the dataframe and storing it into num variable.
And in the next line we are writing num variable into show() method. So all the records will be shown. (by default show() method shows only top 20 records).
truncate=True -> If we use truncate =True then if any column contains long data, it will only show first 20 characters, after 1st 20 characters it will show dots (….). By default the value of truncate is false.

cols=[re.sub('[^a-zA-Z0-9]',"",c.lower()) for c in df.columns] -> In this line with the help of regular expression we are removing all the characters except a-z, A-Z,0-9 and then converting the filtered data into lowercase.

Actually when the column names contain some space or some unwanted special characters , in that scenarios we will perform this kind of data cleansing techniques.

df.toDF(column names)-> This method is used to rename all the columns and convert RDD into DataFrame.

Day - 5

Q. Data cleansing-1 using some important functions :

Sample Data : us-500.csv

"first_name","last_name","company_name","address","city","county","state","zip","phone1","phone2","email","web"
"James","Butt","Benton, John B Jr","6649 N Blue Gum St","New Orleans","Orleans","LA",70116,"504-621-8927","504-845-1427","jbutt@gmail.com","http://www.bentonjohnbjr.com"
"Josephine","Darakjy","Chanay, Jeffrey A Esq","4 B Blue Ridge Blvd","Brighton","Livingston","MI",48116,"810-292-9388","810-374-9840","josephine_darakjy@darakjy.org","http://www.chanayjeffreyaesq.com"
"Art","Venere","Chemel, James L Cpa","8 W Cerritos Ave #54","Bridgeport","Gloucester","NJ","08014","856-636-8749","856-264-4130","art@venere.org","http://www.chemeljameslcpa.com"

Code :

```
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.types import *
spark = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
data="D:\\spark venu sir\\drivers\\us-500.csv"
df=spark.read.format("csv").option("header","true").option("inferSchema","true").load(data)
#ndf=df.groupBy(df.state).agg(count("*").alias("cnt")).orderBy(col("cnt").desc())
ndf=df.withColumn("fullname", concat_ws("_",df.first_name, df.last_name, df.state))\
   .withColumn("phone1",regexp_replace(col("phone1"),"-","").cast(LongType()))\
   .withColumn("phone2",regexp_replace(col("phone2"),"-","").cast(LongType()))\
   .drop("email","web","city","country","address")\
   .withColumnRenamed("first_name","fname").withColumnRenamed("last_name","lname")

#withColumnRenamed used to rename onecolumn at a time
#withColumn used to add a new column (if column not exists) or update (if already column exists)
#lit(value) used to add something dummy value
#drop (columns) ... delete unnecessary columns.
ndf.show()
ndf.printSchema()
```

IMPORTANT POINTS :

agg(count("*").alias("cnt")) -> Using count on all records we are here giving alias name here.

orderBy(col("cnt").desc()) -> For sorting that column in descending order

withColumn() -> This function is used to add a new column (If column does not exist) or update (if the column already exists)

concat_ws("_",df.first_name, df.last_name) ->concat_ws() method is used to concat multiple columns .Here we are concatenating first_name and last_name and the delimiter is (_).

lit() -> This method is used to add any dummy values. If we write **withcolumn("age",lit(18))** then a column named **age** will be added and all the rows will contain value 18.

withColumn("phone1",regexp_replace(col("phone1"),"-","").cast(LongType())) -> Here we are updating the column phone1 . **the regexp_replace()** function is removing the "-" sign . and then **cast() method** is used to change the datatype of the column phone1.

drop() -> This Method is used to ignore the unwanted columns. Suppose we have 100 columns but we want to show only 95 columns, in this case mention the 5 unwanted column names inside drop() method and they will not be shown.

withColumnRenamed("first_name","fname") -> withColumnRenamed() is used to rename one column name at a time. Here we are renaming column name first_name into fname.

****NOTE** -> inferSchema() is applicable once while we are reading the data..If we use more than one time after cleansing any data it is not of any use.

Q . Data Cleansing-2 using some spark functions :

Sample Data : us-500.csv

CODE :

```
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.types import *
spark = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
data="D:\\spark venu sir\\drivers\\us-500.csv"
df=spark.read.format("csv").option("header","true").option("inferSchema","true").load(data)
#ndf=df.groupBy(df.state).agg(count(col("city")).alias("cnt"),
collect_set(df.city).alias("names")).orderBy(col("cnt").desc())
#ndf=df.groupBy(df.state).agg(count("*").alias("cnt"),
collect_list(df.first_name).alias("names")).orderBy(col("cnt").desc())

#ndf=df.withColumn("state",when(col("state")=="NY","NewYork").when(col("state")=="CA","Cali").otherwise(col("s
tate")))
#ndf=df.withColumn("address1",
when(col("address").contains("#"),"*****").otherwise(col("address"))).withColumn("address2",regexp_replace(col("a
ddress"),"#","_"))
#ndf1=df.withColumn("substr",substring(col("email"),0,5)).withColumn("emails", substring_index(col("email"),"@",-
1)).withColumn("username", substring_index(col("email"),"@",1))#
#ndf=ndf1.groupBy(col("emails")).count().orderBy(col("count").desc())
df.createOrReplaceTempView("tab")
qry="""with tmp as (select *, concat_ws('_', first_name, last_name) fullname, substring_index(email,'@',-1) mail
from tab)
select mail, count(*) cnt from tmp group by mail order by cnt desc
"""
#ndf=spark.sql(qry)
#create ur own functions
def func(st):
    if(st=="NY"):
        return "30% off"
    elif(st=="CA"):
        return "40% off"
    elif(st=="OH"):
        return "50% off"
    else:
        return "500/- off"

#by default spark unable to understand python functions. so convert python/scala/java function to UDF (spark
able to understand udfs)
uf = udf(func)
spark.udf.register("offer",uf) #user define function convert to sql function
ndf=spark.sql("select *, offer(state) todayoffers from tab")
#ndf=df.withColumn("offer",uf(col("state")))
ndf.printSchema()

ndf.show(truncate=False)
```

Important Points :

collect_list() -> This function is used to list all the records after any groupBy statement and followed by aggregation function.

Ex= **df.groupBy(df.state).agg(count("*"), collect_list(df.first_name))** here after grouping the states we are counting the elements in each group and then , with the help of collect_list() we are listing all the names inside every group.

Note -> Note collect_list() may contain duplicate elements.

collect_set() -> This function is same like collect_list() but it will contain only unique values, duplicate values are ignored here.

ndf=df.withColumn("state",when(col("state")=="NY","NewYork").when(col("state")=="CA","Cali").otherwise(col("state")))

- ➔ The above query is something like case statement in sql query. If the state is NY then it will print NewYork, if the state is CA then it will print Calli, else it will print the state names as it is.

df.withColumn("address1",
when(col("address").contains("#"),"***").otherwise(col("address"))).withColumn("address2",regexp_replace(col("address"),"#","_"))**

- ➔ In this above statement's when part (1st part) , if address contains # symbol it will print *****, but in the regexp_replace part (2nd part) if the address contains # symbol, it will be replaced by _ symbol.
- ➔ So main difference between when() and regexp_replace() is first one will replace whole record and in the other hand the 2nd one will replace only particular mentioned character.

("substr",substring(col("email"),0,5)) -> It will print 0th character to 5th character of email column into the column named **substr.**

("emails", substring_index(col("email"),"@",-1))-> This will print all the characters **after** the @ symbol of email column into new column named **emails .**

("username", substring_index(col("email"),"@",1)) -> This will print all the characters **before** the @ symbol of email column into new column named **username .**

UDF : If any function is not directly available then we can create our own function with the help of python language. This function is called UDF or User Defined Function.

def func(st): Here we are defining our own function named func with parameter named st.

uf = udf(func) -> By default spark does not understand function written in another language, So here we are converting our python function into spark udf with the help of **udf()** function and storing into a variable named **uf . [remember , this uf function is for coding friendly environment, not for SQL friendly environment]**

spark.udf.register("offer",uf) -> udf function is being converted into sql friendly function here and a new name is given ie **offer() [Now It can be used in SQL friendly environment]**

Day -6

Q . Usages of date functions.

Sample Data : donation2.csv

```
name,dt,amount
venu,10-1-2021,10000
venu,11-3-2021,2000
anu,12-12-2021,5000
anu,12-12-2021,2000
venu,18-11-2021,5000
srinu,15-8-2021,4000
```

CODE :

```
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.types import *
spark = SparkSession.builder.master("local[2]").appName("test").config("spark.sql.session.timeZone",
"EST").getOrCreate()
data=" D:\\spark venu sir\\drivers \\donation2.csv"
df=spark.read.format("csv").option("header","true").option("inferSchema","true").load(data)
#spark by default able to understand 'yyyy-MM-dd' format only
#but in original data u hve dd-MM-yyyy so this date format convert to spark understandable format.
#to_date convert input date format to 'yyyy-MM-dd' format
#current_dat() used to get today date based on ur system time.
#config("spark.sql.session.timeZone", "EST") ... its very imp based on original client date all default time based on
us time only.at that time mention "EST"
#current_timestamp() u ll get seconds minutes as well.

#create udf to get expected date format. like 1yr, 2 months, 4 days ..

res=df.withColumn("dt",to_date(col("dt"),"d-M-yyyy"))\
   .withColumn("today",current_date())\
   .withColumn("ts",current_timestamp())\
   .withColumn("dtdiff",datediff(col("today"),col("dt")))\
   .withColumn("dtadd",date_add(col("dt"),100))\
   .withColumn("dtsub",date_sub(col("dt"),100))\
   .withColumn("lastdt",date_format(last_day(col("dt")),"yyyy-MM-dd-EEE"))\
   .withColumn("nxtday",next_day(col("dt"),"Friday"))\
   .withColumn("dtformat",date_format(col("dt"),"dd/MMMM/yy/EEEE/zzz"))\
   .withColumn("monLstFri",next_day(date_add(last_day(col("dt")),-7),"Fri"))\
   .withColumn("dayofweek", dayofweek(col("dt")))\
   .withColumn("dayofmon",dayofmonth(col("dt")))\
   .withColumn("dayofyr", dayofyear(col("dt")))\
   .withColumn("monbet" ,months_between(current_date(),col("dt")))\
   .withColumn("floor",floor(col("monbet")))\
   .withColumn("ceil",ceil(col("monbet")))\
   .withColumn("round",round(col("monbet")).cast(IntegerType()))\
   .withColumn("dttrunc",date_trunc("day",col("dt")).cast(DateType()))\
   .withColumn("weekofyear",weekofyear(col("dt")))

res.printSchema()
res.show(truncate=False)
#dtdiff .. 588 days .. i want to conver to 1yr-3mon-4days ..
#dayofweek ... from sun how many days completed.. if sun ..1, mon.2.tue..3..sat 7
```

```
#dayofmon .... from month 1 to how many days completed
#dayofyear .... from jan 1 to specified date, how many days completed.
#date_add(df.dt, -100) and date_sub(df.dt, 100) both are same.
#last_day ... it return month's last day.. let jan lastday jan 31.. feb lastday 28
#whats next sun, next mon, next wednesday from today ull get. next_day(dt, "sun")
#https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html
#date_format used to get ur desired format date. let eg: 20/April/21/Tuesday/ at that time use
.withColumn("dtformat",date_format(col("dt"),"dd/MMMM/yy/EEEE/"))

#tasks: i want udf get dtdiff conver to 3 years, 4 months 9 days
#every month 15th what day u ll get? (sun?or mon)
```

IMPORTANT POINTS :

config("spark.sql.session.timeZone", "EST") -> This configuration is for setting the timezone according to area/zone. Here **EST** refers to the timezone for US, for India the timezone will be **IST**

df.withColumn("dt",to_date(col("dt"),"d-M-yyyy")) -> Here we are converting the string format dt column into date format dt column. to_date() function is used to convert string into date . **d-M-yyyy** is the format of the stirng. After using this function it automatically turns into **yyyy-MM-dd** format .

current_date() -> This function is used to get the current date.

current_timestamp() -> To get the current timestamp.

datediff(col("today"),col("dt")) -> To get the differents of days from first date to final date mentioned in the function.

date_add(col("dt"),100)) -> To get the future date after 100 days from now.

date_sub(col("dt"),100)) -> To get the past date before 100 days from now.

NOTE : date_add(col("dt"),-100)) and **date_sub(col("dt"),100))** both are same, both of the functions will return past date before 100 days from dt column's dates.

last_day(col("dt")) -> To get the last date of of the the month of each row of dt column.

withColumn("nxtday",next_day(col("dt"),"Friday")) -> Here we are creating a new column nxtday and with the help of **next_day(col("dt"),"Friday"))** function we are returning the date of next Friday from the current date in dt column.

date_format(col("dt"),"dd/MMMM/yy/EEEE/zzz")) -> date_format() function is used to change the format of date of any existing column. Here we are changing the format of date of dt column into a new format **dd/MMMM/yy/EEEE/zzz** (eg- 01/January/22/Monday/-5.00)

For more datetime related we can visit this bleow link :

#https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html

next_day(date_add(last_day(col("dt")),-7),"Fri")) -> Here we are finding the date of the last Friday of each month. **last_day(col("dt")** -> This part is used to find the last date of the month, **date_add(last_day(col("dt")),-7)** -> This part is used to get the date of 7 days before the of the last date, **next_day(date_add(last_day(col("dt")),-7),"Fri")) ->** In this part we are finding the Friday in the last 7 days of the month.

dayofweek(col("dt"))) -> This function calculates the day of week as number.

Ex-> Sunday=1,Monday=2,Tuesday=3,Wednesday=4 etc

dayofmonth(col("dt"))) -> This function returns day number of the month of dt column.

Ex= 11 jan = 11, 23 feb=23, 30march= 30

dayofyear(col("dt"))) -> This function returns the day number of the date in this year

ex-> 10th Jan = 10th day of this year, 10th feb= 41th day of this year etc.

months_between(current_date(),col("dt")) -> This function is used to find the difference between two dates in terms of month. It will return values in floating points.
ex -> 13.34 months

floor() -> This function is used to print only the number before the floating point.

If the record is 19.23 then floor() function will return 19

ceil() -> This function returns the next integer value of any float number. Ex -> If the number is 23.31 then ceil() function will return 24

round() -> This function is used to to return the nearest round value of any float number. Suppose the number is 19.2 then it will return 19.00 but if the number is 19.7 it will return 20

date_trunc("day",col("dt")) ->

cast(DateType()) -> This method is used to change the datatype of any column. Here we are changing the data type into date.

NOTE -> To use cast() method we have to import **types** module.
from pyspark.sql.types import *

weekofyear(col("dt")) -> To see how many weeks completed of this year on the basis of each records in dt column.

Question : What is the difference between concat() and concat_ws() ??

Ans : In concat() method we need to mention the delimiter everytime after mentioning a column name.

Ex= concat("first_name", lit(" "),"middle_name",lit(" "), "last_name")

In the other hand for concat_ws() method we don't require to mention the delimiter every time . we just mention it once.

Ex = concat_ws(" ","first_name","middle_name","last_name")

DAY -7

Q. Loading rdbms data into spark and after cleaning writing them into rdbms staging table.

Code :

```
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.types import *
spark = SparkSession.builder.master("local").appName("testing").getOrCreate()
host="jdbc:mysql://sravanthidb.c7nqndsntouw.us-east-1.rds.amazonaws.com:3306/sravanthidb?useSSL=false"
df=spark.read.format("jdbc").option("url",host).option("user","myuser").option("password","mypassword")\
   .option("dbtable","emp").option("driver","com.mysql.jdbc.Driver").load()
#df.show()
#process data
res=df.na.fill(0,['comm','mgr']).withColumn("comm", col("comm").cast(IntegerType()))\
   .withColumn("hiredate", date_format(col("hiredate"),"yyyy/MMM/dd"))

res.write.mode("overwrite").format("jdbc").option("url",host).option("user","myuser").option("password","mypassw
ord")\
   .option("dbtable","empclean").option("driver","com.mysql.jdbc.Driver").save()



res.show()
res.printSchema()
#: java.lang.ClassNotFoundException: com.mysql.jdbc.Driver
#mysql dependency problem so pls add mysql jar and place in spark/jars folder
```

Important Points :

java.lang.ClassNotFoundException: com.mysql.jdbc.Driver -> This is a common error while we try to load data from RDBMS. It happens due to dependency file. By default spark does not contain any rdbms jar file.
To overcome this error open your spark folder , then you will find a jar folder inside it. Inside the jar folder place your rdbms server's jar file.
Here in this example we are using mysql , that's why we will place mysql jar file inside this folder.

host : Here we are storing the host details of our RDBMS.

?useSSL=false : We are setting useSSL=False at the end of host , it means low security level. If we set useSSL=True then security level will be high. As we are just practicing we don't need high security level.

**df=spark.read.format("jdbc").option("url",host).option("user","myuser").option("password","mypasswor
d").option("dbtable","emp").option("driver","com.mysql.jdbc.Driver").load()**

- ➔ This is the process of creating dataframe from RDBMS table, Here to find the driver name just google for it. Ex -> for mysql driver search **mysql jdbc driver class** and you will find **com.mysql.jdbc.Driver**

df.na.fill(0,['comm','mgr']) : To fill the null values of comm and mgr column with 0.
If we want to apply it on whole dataframe then syntax will be **df.na.fill(0)**

res.write.mode("overwrite").format("jdbc").option("url",host).option("user","myuser").option("password","mypassword").option("dbtable","empclean").option("driver","com.mysql.jdbc.Driver").save()

- ➔ This is the process to write the processed/cleaned data into staging table in RDBMS.
  mode("overwrite") is used to overwrite the records if the table already exists in RDBMS.
  option("dbtable","empclean") -> this is the staging table name.
  save() -> This method should be written at last to save the records into the staging table.

Q .Why can't we store data into the same table after processing ??

ANS : Because it is not possible to read and write in a table at a same time. This situation is called deadlock. If we try to do so, all the data will be deleted from the original table. we need a staging table to overcome it.

Q. Taking Input from csv file and Writing into RDBMBS Staging Table :

SAMPLE DATA FILE : 10000Records.csv

Code

```
from pyspark.sql import *
from pyspark.sql.functions import *
spark = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
sc = spark.sparkContext

data="D:\\spark venu sir\\drivers\\10000Records.csv"
df=spark.read.format("csv").option("header","True").option("inferSchema","True").option("sep",",").load(data)

import re
cols=[re.sub('[^a-zA-Z0-9]',"",c) for c in df.columns]  #Removing space and special char from column names
ndf=df.toDF(*cols)

ndf.show(21,truncate=False)
df.printSchema()

host="jdbc:mysql://sravanthidb.c7nqndsntouw.us-east-1.rds.amazonaws.com:3306/sravanthidb?useSSL=false"
uname="myuser"
pwd="mypassword"

ndf.write.mode("overwrite").format("jdbc").option("url",host)\
   .option("dbtable","empclean100").option("user",uname).option("password",pwd)\
   .option("driver","com.mysql.jdbc.Driver").save()
```

Explanation :

Here we are first took csv formatted data from local system, the processed the data and converted into Dataframe. Next stored the host, user,and password into variables and finally wrote the processed data into a staging table named empclean100.

Q. Processing the above Program in highly secured way. That is without showing data, host, username, password .

We are placing these credential data into a file named config.txt in our local system

Config.txt

```
[cred]
host =jdbc:mysql://sravanthidb.c7nqndsntouw.us-east-1.rds.amazonaws.com:3306/sravanthidb?useSSL=false
user=myuser
pass=mypassword

[input]
data=D:\spark venu sir\drivers \10000Records.csv
opdata= D:\spark venu sir\drivers \\\opdata

[pro]
qry1=select * from tab where
```

Code :

```
from pyspark.sql import *
from pyspark.sql.functions import *
import configparser
from configparser import ConfigParser
conf=ConfigParser()
conf.read(r" D:\\spark venu sir\\drivers \\config.txt")
host=conf.get("cred","host")
user=conf.get("cred","user")
pwd=conf.get("cred","pass")
data=conf.get("input","data")
spark = SparkSession.builder.master("local[*]").appName("testing").getOrCreate()
df = spark.read.format("csv").option("header","true").option("sep",",").option("inferschema","true").load(data)
import re

cols=[re.sub('[^a-zA-Z0-1]',"",c.lower()) for c in df.columns]
ndf = df.toDF(*cols)
ndf.show(4)
ndf.printSchema()

ndf.write.mode("overwrite").format("jdbc").option("url",host).option("user",user).option("password",pwd)\
.option("dbtable","testarjun").option("driver","com.mysql.jdbc.Driver").save()
```

Explanation :

Here we are storing all these credential data into a file named **config.txt** in our local system instead of directly writing them into code. This is a secured way.

from configparser import ConfigParser : To process the data in this way we are calling ConfigParser from configparser module.

conf.read(r" D:\\spark venu sir\\drivers \\config.txt") -> In this method we are reading the config.txt file from our local system.

host=conf.get("cred","host") -> get() method is used to access any credential. We are here accessing host details by this method and storing them into host variable.

.option("url",host) -> while writing into database ,we are simply writing all variable names. As host is our variable name , it is not written inside any quote.

Q . Reading all the tables from a RDBMS

```
from pyspark.sql import *
from pyspark.sql.functions import *

spark = SparkSession.builder.master("local").appName("testing").getOrCreate()
from configparser import ConfigParser
conf=ConfigParser()
conf.read(r" D:\\spark venu sir\\drivers\\config.txt")
host=conf.get("cred","host")
user=conf.get("cred","user")
pwd=conf.get("cred","pass")
data=conf.get("input","data")
#tabs=['dept','EMP','abc','banktab','DEPT']
qry="(select table_name from information_schema.tables where TABLE_SCHEMA='sravanthidb') aaa"
df1=spark.read.format("jdbc").option("url",host).option("user",user).option("password",pwd)\
   .option("dbtable",qry).option("driver","com.mysql.jdbc.Driver").load()
#converting into list
tabs=[x[0] for x in df1.collect()]
#host="jdbc:mysql://sravanthidb.c7nqndsntouw.us-east-1.rds.amazonaws.com:3306/sravanthidb?useSSL=false"
for i in tabs:
   df=spark.read.format("jdbc").option("url",host).option("user",user).option("password",pwd)\
   .option("dbtable",i).option("driver","com.mysql.jdbc.Driver").load()
   df.show()
```

Explanation :

Here also all the credentials are stored into config.txt file.

qry="(select table_name from information_schema.tables where TABLE_SCHEMA='sravanthidb') aaa"

- ➔ In this statement we are retrieving all the tables from database and storing into a variable named qry. **aaa** is a temporary name.

Q. Example of window functions.

Sample Data : us-500.csv

CODE :

```
from pyspark.sql import *
from pyspark.sql.functions import *

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

data = "D:\\spark venu sir\\drivers\\us-500.csv"
df=spark.read.format("csv").option("header","true").option("inferSchema","true").load(data)
res=df.withColumn("today",current_date()).drop("phone1","phone2","email","web","address","county","compa
ny_name","city")\
   .withColumnRenamed("zip","sal")
#withColumnRenamed (oldcol,newcol) used to rename one column at a time.
#if u want to rename all columns use toDF()
#res.show()
res.createOrReplaceTempView("tab")
#result=res.orderBy(col("sal").desc()).withColumn("rno",monotonically_increasing_id()+1).where(col("rno")<=
5)
#monotonically_increasing_id() ... get unique numbers starts from 0 ..
#result.show()
from pyspark.sql.window import Window
win=Window.partitionBy("state").orderBy(col("sal").desc())
fin=res.withColumn("rnk",rank().over(win)).withColumn("drnk",dense_rank().over(win))\
   .withColumn("rno",row_number().over(win)).withColumn("prank",percent_rank().over(win))\
   .withColumn("ntr",ntile(10).over(win)).withColumn("lead",lead(col("sal")).over(win))\
   .withColumn("diff",col("sal")-col("lead")).withColumn("lag",lag(col("sal")).over(win))\
   .withColumn("fst",first(col("sal")).over(win))
fin.show()
```

Important Points :

drop() -> This method is used to ignore one or multiple columns.

withColumnRenamed("zip","sal") - > with this method we are renaming the zip column into sal column.

withColumnRenamed (oldcol,newcol) used to rename one column at a time.
if u want to rename all columns use toDF()

monotonically_increasing_id() = This method is used to get unique numbers stsrting from 0, If we write +1 after this method then count will start from 1.

from pyspark.sql.window import Window -> To import Window functions.
percent_rank() - > To find out percentile rank from a group.
ntile(x) - > To find ranks according to the value of x. If we write ntile(10) then it will rank a group from 1 to 10.
lead() = To print the values from 2^{nd} values. It is mainly used for comparison between every two adjacent rows.

Day -8

Q . Running Pyspark codes in Production Environment (ex -1):

Filename : first.py

```
import os
import sys
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.window import Window
spark = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
sc = spark.sparkContext
sc.setLogLevel("ERROR")
win=Window.partitionBy('state').orderBy(col('zip').desc())
#data="s3://nirupam2022/us-500.csv"
data=sys.argv[1]

df=spark.read.format('csv').option('header','true').option('inferSchema','true').load(data)
ndf=df.withColumn('ntile _column',ntile(5).over(win))
ndf.show()
ndf1=df.withColumn('ntile_col',ntile(10).over(win)).where(col('ntile_col')==1)
ndf1.show()

#sql friendly
df.createOrReplaceTempView('tab')
ndf2=spark.sql("select * from tab where zip between 85260 and 85381")
ndf2.show()
```

Important Points :

setLogLevel("ERROR") = This property is used to hide the log details while running the code.

data=sys.argv[1] -> Here we are taking data/inputs in the form of argument.

** Generally production environment is linux platform.

<u>To run pyspark code in linux production env:</u>

1st Approach :

spark-submit --master local --deploy-mode client [YourProgramName].py

2nd Approach [with argv[1]] :

spark-submit --master local --deploy-mode client [YourProgramName].py [data-location-path]

If we are storing the path of data directly inside any variable inside the source code then we should go for 1st approach.

Ex :

spark-submit --master local --deploy-mode client first.py

But If we are taking data(variable) as arguments (sys.argv[1]) then we should go with 2nd approach, because we need to pass any value to the arguments.
Ex :
spark-submit --master local --deploy-mode client first.py s3://nirupam2022/us-500.csv

Imp -> Why Should we go with the 2nd approach ? And why is it most preffered ??

Ans : If we accept the data with the help of arguments then we don't need to modify the source code again and again which is a good practice. We just need to provide the path of data while running the spark job outside of that program.

Note :

If you have multiple spark job to execute then create a shell script (ex : abc.sh) and inside this .sh file just copy all the jobs one by one and run that script

Q. Pyspark Code in Production Environment (ex – 2 [Database connectivity]) :

Filename : second.py

```
#!/bin/bin
from pyspark.sql import *
from pyspark.sql.functions import *

spark = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
sc = spark.sparkContext
sc.setLogLevel("ERROR")
host="jdbc:mysql://mysqldb1.co7gi3agncec.ap-south-1.rds.amazonaws.com:3306/mysqldb"
uname='myuser'
pwd='password'
df=spark.read.format('jdbc').option('url',host).option('user',uname)\
   .option('password',pwd).option('driver','com.mysql.jdbc.Driver').option('dbtable','asltab').load()
df.show()
```

Note:
If we are connecting RDBMS with spark inside production cluster(here putty) we will face classNotFound error. Beacuse by default RDBMS server dependency jars are not available in spark.
To overcome this first place the jar file of your RDBMS server into /usr/lib/spark/jars (if you have your jar file in s3 then you can download it with either hdfs dfs -get command or with aws s3 command)

Q . Writing spark program in production env [writing into Hive table] (ex -3):
Filename : hiveimport.py

```
import os
import sys
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.window import Window
spark = SparkSession.builder.master("local[*]").appName("test").enableHiveSupport().getOrCreate()
sc = spark.sparkContext
sc.setLogLevel("ERROR")
#data="s3://nirupam2022/us-500.csv"
data=sys.argv[1]
tab=sys.argv[2]

df=spark.read.format('csv').option('header','true').option('inferSchema','true').load(data)
df.show()
df.write.format("hive").saveAsTable(tab)
```

Important Points :

enableHiveSupport() = This property is used to communicate directly with hive.

data=sys.argv[1]

tab=sys.argv[2]

- ➔ Here we have written first argument for accepting input data and second argument is written for writing data into hive. While running the spark job after writing the command first we need to provide the path of input data then we need to provide hive table name .

Ex :

```
spark-submit --master local --deploy-mode client hiveimport.py  s3://nirupam2022/us-500.csv  hivetab1
```

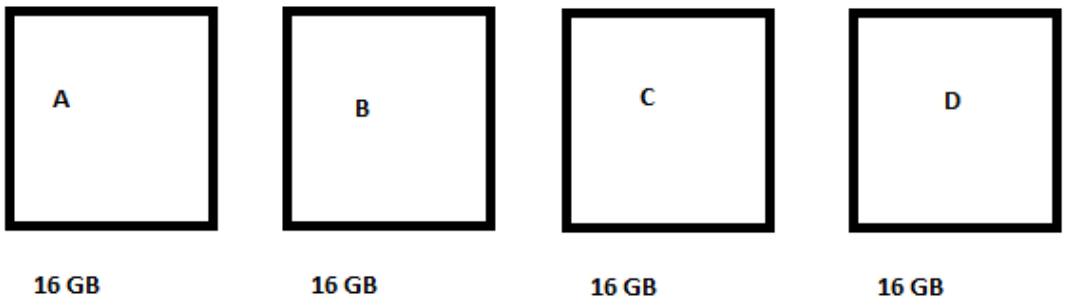
(Input data path) (output hive table name)

df.write.format("hive").saveAsTable(tab) : This is used to write in hive . we need to write this to directly write data into hive table .

After writing into hive table we can go to hive and see the data by running select * from hivetab1

Imp : Elaboration Of spark job command :

Command : spark-submit --master local/yarn --deploy-mode client/cluster first.py



Suppose you have a four node cluster , each of size 16 GB.

1) --master local : If you are working on local or single node cluster then use --master local .

2) **--master yarn :** If you are working on hdfs multi node cluster and you want to use all the recourses then you should use –master yarn. Here resources will be 16x4=64GB.

3) --deploy-mode client : If your spark code (Driver Program) is in your local system then use --deploy-mode client

4) **--deploy-mode cluster :** If your pyspark code is in hdfs/aws s3 then you should use --deploy-mode cluster

Note :

1) If we are using --deploy-mode cluster then we have to use --master yarn.

But if our program is in hdfs/s3 and we use master as local and deploy mode as client and provide full hdfs/s3 path of the pyspark code then also our spark job will be executed.

2) If we write --master spark://123:444:000; like this then spark use its own cluster manager.

***To run your spark job in pycharm , at the bottom of pycharm click on terminal and after opening terminal just write your spark job there and run.

Day -9

AWS GLUE :
AWS Glue is a serverless data integration service that makes it easy to discover, prepare, and combine data for analytics, machine learning, and application development.
Mianly we use apache spark ETL in glue.

WORKING WITH GLUE :
To work with glue we need a database or AWS s3 bucket. So first we can create a database In AWS RDS service.
Next search GLUE in aws and open it.
Now in the left hand side bar inside Data Catalog click on databases > Now inside it click on add databases> give a name to the database service and save.

Next click on connections(Here we connect with external daabases or RDS)>Click on add connections>connection name= connectmysql>connection type= choose any type from drop down box (we are using JDBC here)> Database Engine=Mysql (here we are using MySql)>click on next> JDBC URL =(copy paste the url from sqlworkbech[which contains hostname,port num and db name])> Now provide username and password in next two fields.> click on create connection [VPC part is optional]
(Note : Our RDS database's security should be 0.0.0.0/0 for easy access to all)

Now test your connection, If it isn't connecting then go to IAM ROLES and select GLUE then create a new role with amazonrdsfullaccess,amazons3fullaccess,amazonglueservicerole and amazonredshiftfullaccess and assign the role while testing the connection (If you are not choosing the VPC option then this IAM role is not necessary)
Crawler :
Crawler is a job defined in AWS Glue. It crawls databases and buckets in S3 and then creates tables in Amazon Glue together with their schema.Then, you can perform your data operations in Glue, like ETL. Remember crawler does not process any data itself.

In our s3 bucket place a data file in a folder. Ex path = s3://nirupam2022/folder1/asl.csv

Click on crawler in the left side> add crawler > crawler name =enter a name (ex – connects3)> Next> Fill all the next fields according to your need. And create the crawler.

➔ **MAIN DIFFERENCE BETWEEN ATHENA AND GLUE** :

Athena only takes data from s3 bucket , processes it and only stores the output to s3. But GLUE takes data from multiple stores and stores output into multiple destinations.

Day – 10

GLUE INTERNALS :

Glue is mainly an optimized layer on top of SparkContext. Inside GLUE there is GlueContext. GlueContext is converted into dataframe and then dataframe process the data and convert it into GlueContext again to show the output.

Let's Take Data From AWS S3 and process it using GLUE job and store the output into AWS s3 bucket again :

For our operation it is recommended to create a crawler, take data from s3 bucket and store into a staging table.
Steps : click on crawler> click on add crawler >Give a name to your crawler(ex=getmalcsv)>next>crawler source type = data stores >Repeat crawls of s3 data Stores=crawl all folders>Next> Choose a datastore=s3>(Ignore connection option)>Crawl data in =Specified path in My account>Include Path=Choose the path of s3 bucket's folder where your input data is located(ex=s3://nirupam2022/input/maldata)
[Give path upto the folder only] >Next>Add another data Store =No>next>click on existing IAM role and select your previously created IAM role(with the following accesses **amazonrdsfullaccess,amazons3fullaccess,amazonglueservicerole and amazonredshiftfullaccess**) from the dropdown box.>next>Scheduler Frequency =Run on demand.(you can choose the frequency according to your need also. But here we will go with Run on demand option only)>Next>database = choose a database from the dropdown box(If you don't have a database then click on add database>database name=[give a name]>click on create)[temporarily data gets stored in this database]>Next>click on Finish. Now your crawler is created . select the crawler and click on Run crawler to get the data from s3.

Now your data is temporarily stored into a staging table. To check click on tables option in the left and you will find a table in the name of your s3 folder name. Click on it to see about the table in detail.

Next We will create a Glue Job and process the data.
Click On Jobs>Choose the option Visual with a source and target and below source=amazon s3, Target=amazon s3> click on Create>Click on data source properties – s3> click on s3 loaction> inside s3 url choose / paste your data location(upto last directory)>choose data format (Here we are choosing CSV)> choose your delimiter present in the data(here we are choosing comma)>choose quote character(We are here choosing double quote)>click on inferSchema>click on advance and S3 Target location=[Enter your target data location and at last give a slash(/)](ex = s3://nirupam2022/output/malop/)
Now click on Job Details in the upper bar >name= Give A name(sparkpoc)>IAM role= (choose the previously create role)>Requested No. of Workers=2(recomended)>Advanced Properties>Script Filename=(Give a name)[ex- sparkpoc.py]>leave rest of the fields as they are.> now click on Save

Now if we click on Run button the data will be imported from s3 and will be saved at s3 target directory again. But we have not done any processing with the data.

To process the data click on the option Script . Here by default a script will be written according to our data. Now click on the option edit script in the right hand upper side. Inside the script we can write our own code to process the data.

Q. Example of data processing in GLUE :

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from awsglue.dynamicframe import DynamicFrame

from pyspark.sql.functions import *
from pyspark.sql.types import *
import re
args = getResolvedOptions(sys.argv, ["JOB_NAME"])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args["JOB_NAME"], args)
db_name="mydb"
tb_name="maldata"
data="s3://nirupam2022/input/maldata"
#get data from crawler/ table and create dynamic frame
datasource = glueContext.create_dynamic_frame.from_catalog(database=db_name,table_name=tb_name)

#dynamicframe convert to dataframe
df = datasource.toDF()

#data cleaning
#remove special characters from header/schema

cols=[re.sub('[^0-9a-zA-Z]',"",c)for c in df.columns]
ndf=df.toDF(*cols)
res=ndf.withColumn("DateofBirth",to_date(col("DateofBirth"),"M/d/yyyy")).withColumn("today",current_date()).withColu
mn("diff",datediff(col("DateofBirth"),col("today")))

#convert dataframe to gluecontext/dynamicframe

fres = DynamicFrame.fromDF(res, glueContext, 'results')
#store data in s3
s3_write_path = 's3://nirupam2022/output/maldata'

glueContext.write_dynamic_frame.from_options(frame=fres, connection_type="s3", connection_options={"path":
s3_write_path}, format="csv", transformation_ctx="datasink1")

job.commit()
```

Whenever we use glue some by default lines will be present there. Below are those lines

```
args = getResolvedOptions(sys.argv, ["JOB_NAME"])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args["JOB_NAME"], args)
```

Important points :

from awsglue.dynamicframe import DynamicFrame => We need to import the this module to work with glueContext dynamicframe.

db_name="mydb" => Database name in the crawler, where the table is stored temporarily
tb_name="maldata" => Table name in the crawler.

datasource = glueContext.create_dynamic_frame.from_catalog(database=db_name,table_name=tb_name)
In this line we are converting our data into dynamic frame inside glueContext and storing it inside a variable named datastore. It is necessary to create glueContext first while working with GLUE.

df = datasource.toDF() => Here we are converting this glueContext dynamic frame into spark dataframe, because in GLUE for processing the data it is mandatory to convert dynamic frame into dataframe.

fres = DynamicFrame.fromDF(res, glueContext, 'results') : After processing the data in dataframe we are again converting it into dynamic frame. It is necessary to convert dataframe into dynamic frame again once the data is processed.

s3_write_path = 's3://nirupam2020/output/maldata' => Here we want to store our processed data.

glueContext.write_dynamic_frame.from_options(frame=fres, connection_type="s3", connection_options={"path": s3_write_path}, format="csv", transformation_ctx="datasink1")
This is the method to write the finally processed data into target location.

job.commit() => To save the changes permanently to target location.

Q . Example Of loading Processed data into RDBMS

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from awsglue.dynamicframe import DynamicFrame

from pyspark.sql.functions import *
from pyspark.sql.types import *
import re
args = getResolvedOptions(sys.argv, ["JOB_NAME"])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args["JOB_NAME"], args)
db_name="mysqldb"
tb_name="input"
data="s3://nirupam2022/input"
#get data from crawler/ table and create dynamic frame
datasource = glueContext.create_dynamic_frame.from_catalog(database=db_name,table_name=tb_name)

#dynamicframe convert to dataframe
df = datasource.toDF()

#data cleaning
#remove special characters from header/schema

cols=[re.sub('[^0-9a-zA-Z]',"",c)for c in df.columns]
ndf=df.toDF(*cols)
#res=ndf.withColumnRenamed("DateofBirth","dob").withColumn("today", current_date()).where(col("Gender")=="F")
res=ndf.withColumn("DateofBirth",to_date(col("DateofBirth"),"M/d/yyyy")).withColumn("today",current_date()).withColu
mn("diff",datediff(col("DateofBirth"),col("today")))

#convert dataframe to gluecontext/dynamicframe

fres = DynamicFrame.fromDF(res, glueContext, 'mysql')
#store data in s3
s3_write_path = 's3://nirupam2022/output'
#glueContext.write_dynamic_frame.from_options(frame=fres, connection_type="s3", connection_options={"path":
s3_write_path}, format="csv", transformation_ctx="datasink1")
#store data in mysql
glueContext.write_dynamic_frame.from_options(frame = fres, connection_type = "mysql" ,  connection_options = {"url":
"jdbc:mysql://sravanthidb.c7nqndsntouw.us-east-1.rds.amazonaws.com:3306/sravanthidb", "user": "myuser",
"password": "mypassword", "dbtable": "myglue100"})
job.commit()
```

Important Points :

**** Before creating the job first create a crawler to import data

** Whenever we are creating glue job to deal with RDBMS it is compulsory to add the sql jar file into that job.
To add the jar first place the jar file into your aws s3 bucket. Then Go to Job Details option inside glue job, next go bleow and you will find python libaray path field and Dependency Field. In both of the fields paste your jar file's location (ex s3://nirupam2022/drivers/mysql jar name)

** All the data loading and processing parts are same as the previous example, as we are sotring our output into RDBMS just the storing part is different.

glueContext.write_dynamic_frame.from_options(frame = fres, connection_type = "mysql" , connection_options = {"url": "jdbc:mysql://sravanthidb.c7nqndsntouw.us-east-1.rds.amazonaws.com:3306/sravanthidb", "user": "myuser", "password": "mypassword", "dbtable": "myglue100"})

- ➔ Here we are storing our output into mysql that's why connection_type="mysql"
- ➔ Inside connection option just paste your host url, username,password and at last inside dbtable enter the table name where you want to store data (ex = myglue100)

Day -11

If we simply copy paste our pyspark code into putty EMR cluster for the first time and run, we will get an error saying no module found and if we are using RDBMS for reading/writing purpose then will get class not found error.

Solution :

First run the command in putty = pip install findspark

To overcome the first error we have to set the environment variables first in putty. To do this we have two approach :

1) Inside .bashrc file set spark_home and java_home envvironment variable with path

2) Inside code define the path of spark_home and java_home as following :

```
import sys
import os
import findspark
findspark.init()
os.environ["JAVA_HOME"]="/usr/lib/jvm/java"
os.environ["SPARK_HOME"]="/usr/lib/spark"
from pyspark.sql import *
from pyspark.sql.functions import *
```

To overcome The second error (classNotFound error) we have to put rdbms dependency jar file inside **/usr/lib/spark/jars/**

OOZIE :

Generally If we want to import/read data from any RDBMS then we need to write pyspark program to access the database and and using read method we and read it and then we can process the data and finally we can save it in any RDBMS database or in hive or in aws s3 bucket.

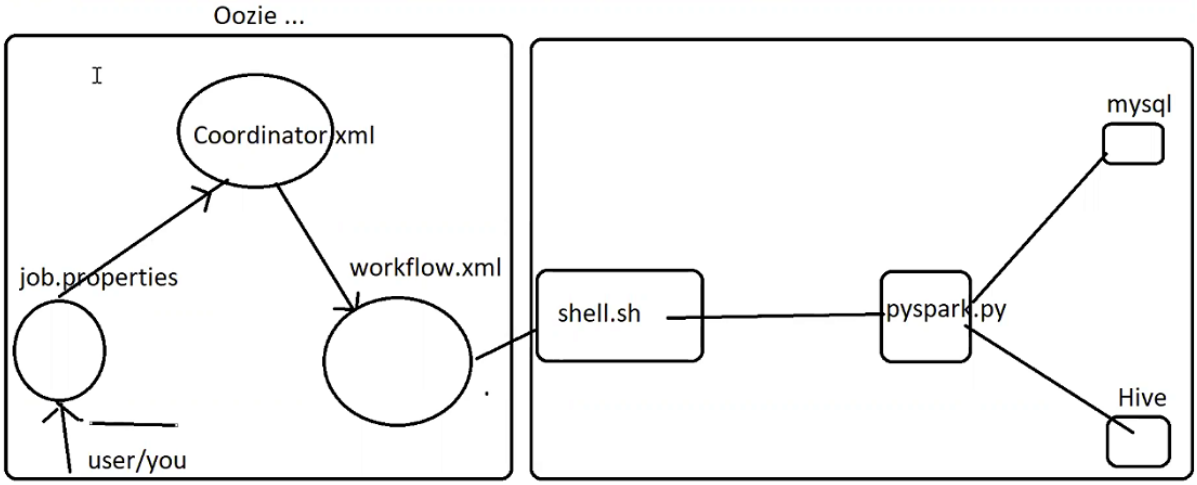
But if the data is increasing in RDBMS after a certain time, In this case we need to run the pyspark program again and again, which is not a good approach.

Here oozie comes into picture

Apache Oozie is a server-based workflow scheduling system to manage Hadoop jobs.

There are mainly three components of oozie

1. Job.properties
2. Coordinator.xml
3. Workflow.xml



In linux(putty) Inside /usr/lib/hadoop/etc/hadoop/ there is core-site.xml file
Just copy the ip address of name node from core-site.xml file and replace the ip address written in job.properties file,coordinator.properties file,workflow.xml file,notes.txt file of of your oozie folder with the copied ip from core-site.xml

Inside coordinator.xml file we have to set/change these properties

<coordinator-app name="${appName}" frequency="${coord:minutes(6)}" start="2020-03-12T13:00Z" end="2020-11-01T05:00Z" timezone="UTC" xmlns="uri:oozie:coordinator:0.2">

frequency="${coord:minutes(6)} => Here we are mentioning the frequency of oozie job . In this case the job will run in every 6 minutes. We can change it to hour,min,day,month according to our needs.
start="2020-03-12T13:00Z" end="2020-11-01T05:00Z" => These are the starting and ending dates of job.
timezone="UTC" => Time zone is set according to US, we can set it in IST also.

Sample Code : pyspark.py

```
import os
import sys
import findspark
findspark.init()
os.environ["JAVA_HOME"]="/usr/lib/jvm/java"
os.environ["SPARK_HOME"]="/usr/lib/spark"
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.window import Window
spark = SparkSession.builder.master("local[*]").appName("test").enableHiveSupport().getOrCreate()
sc = spark.sparkContext
sc.setLogLevel("ERROR")
host="jdbc:mysql://sravanthidb.c7nqndsntouw.us-east-1.rds.amazonaws.com:3306/sravanthidb?useSSL=false"
df=spark.read.format("jdbc").option("url",host).option("user","myuser").option("password","mypassword")\
    .option("dbtable","emp").option("driver","com.mysql.jdbc.Driver").load()
df.write.format("hive").saveAsTable(tab)
```

Here we are working on production cluster (ie Linux/putty). In this code pyspark.py we are reading data from mysql and writing it in hive, but if the data in RDBMS is increasing continuously then we have to run this code (spark-submit command) again and again.
We will work with oozie now onwards to overcome this. But problem is oozie does not support files from local source. **It only supports files from hdfs or s3**. So we will put our pyspark.py file into s3 bucket and write the command and spark-submit command into a file named shell.sh

Script file : shell.sh

hdfs dfs –put –f pyspark.py s3://nirupam2022/drivers/

spark-submit --master local --deploy-mode client s3://nirupam2022/drivers/pyspark.py

Next we will place our job.properties file,coordinator.properties file,workflow.xml file into putty or linux from windows with the help of winscp or aws s3 .

But as we know oozie does not support local files, so now we will place these files into hdfs
Ex = hdfs dfs -put job.properties /user/hadoop/ ooziepoc/
Same like this upload rest 2 files into hdfs.

To Run oozie job run this command below :

oozie job --oozie http://ip-172-31-23-206.us-east-2.compute.internal:11000/oozie --config job.properties –run

Note :
http://ip-172-31-23-206.us-east-2.compute.internal
This is the IP Address of namenode/your system. So we need to write our own IP address in this place.
11000 => This is the port number of oozie server

After running oozie job, we will get a job ID.
Ex : **14-094387837583783-oozie-tucu**

To know detailed Information about a job :
oozie job --oozie http://ip-172-31-23-206.us-east-2.compute.internal:11000/oozie --info 14-094387837583783-oozie-tucu

** **In oozie job minimum frequency of job is 5 min. We cannot set it below 5 min.**

Day-12

AWS LAMBDA :

AWS lambda is an event driven serverless computing service provided by amazon as a part of Amazon Web Services . It is a computing service that runs code in response to events and automatically manages the computing resources required by that code.

To work with lambda first create a glue job in spark editor. The process of creating glue job is same as discussed in glue chapter. Difference is here we will create a new glue role with the policies **amazonrdsfullaccess,amazons3fullaccess,amazonglueservicerole and CloudWatchFullAccess** and we will attach/use this role in our glue job.

Now go to script and write this code

Code : (job name =lambdaGluePOC)

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
args = getResolvedOptions(sys.argv, ["file","buck"])
file_name=args['file']
bucket_name=args['buck']
print("Bucket Name" , bucket_name)
print("File Name" , file_name)
input_file_path="s3a://{}/{}".format(bucket_name,file_name)
output="s3a://{}/{}".format(bucket_name,"/output/aslres")
print("Input File Path : ",input_file_path);

df = spark.read.format("csv").option("header", True).option("inferSchema", False).load(input_file_path)
df.coalesce(1).write.format("csv").option("header", "true").save(output)
```

In glue job we have created some arguments , and now we will create a lambda function to send values/data to those arguments.
Before creating a lambda function we will create a role for lambda named **lambdaglues3role with the policies ClouWatchFullAccess ,AmazonS3FullAccess, AWSGlueConsoleFullAccess**

Steps :
Go to lambda > click on create function >Author From Scratch>Function name =[Enter a name for your function](ex sparkPOC)>Runtime = Python 3.9>Change default execution role> choose existing role> select the created role (lambdaglues3role)>click on create function

Now inside we will find an option saying add trigger. Click on it> select a source =[here our source of data is s3, so we will choose s3] >bucket=[choose your bucket](ex = s3://nirupam2022) > Event Type=All object Create Event>prefix=choose your folder name from where you want data (ex=input/)>check on acknowledgement box > click on add.

Now click on code and write :

```
import json
import boto3

def lambda_handler(event, context):
  # Retrieve File Information
  bucket_name =   event['Records'][0]['s3']['bucket']['name']
  s3_file_name =  event['Records'][0]['s3']['object']['key']
  client = boto3.client('glue')
  print("bucket: ",bucket_name)
  print("file: ", s3_file_name)
  response = client.start_job_run(JobName = 'lambdaGluePOC', Arguments={"--buck":bucket_name,"--file":s3_file_name})
```

Day -13

Handling JSON DataTypes :

Here we will learn how to deal with complex datatypes like json.

First Example :

Sample data : zips.json

Code :

```
from pyspark.sql import *
from pyspark.sql.functions import *
spark = SparkSession.builder.master("local[2]").appName("test").getOrCreate()
data=" D:\\Spark Venu sir\\drivers\\zips.json"
df=spark.read.format("json").load(data)
#df.show(truncate=False)
#df.printSchema()
#ndf=df.withColumnRenamed("_id","id").withColumn("loc", explode(col("loc")))
ndf1=df.withColumnRenamed("_id","id").withColumn("lang", col("loc")[0]).withColumn("lati", col("loc")[1]).drop("loc")
ndf1.createOrReplaceTempView("tab")
ndf=spark.sql("select * from tab where state='CA'")
ndf.show()
ndf.printSchema()
op="E:\\bigdata\\datasets\\output\\resultjson"
ndf.write.mode("append").format("csv").option("header","true").save(op)
#simple datatypes: int, string, double, date etc
#complex datatypes: Array, Struct, Map
# Don't use special char in column. If you have column named "_id", then change it to "id"
#explode simple explode /unnest data means remove arrays elements
#url="jdbc:mysql://sravanthidb.c7nqndsntouw.us-east-1.rds.amazonaws.com:3306/sravanthidb"
#ndf.write.format("jdbc").option("url", url).option("dbtable",
"jsontomysql").option("user","myuser").option("driver","com.mysql.jdbc.Driver").option("password","mypassword").save(
)
```

Some Important Points :

withColumnRenamed("_id","id") => With this method we are renaming "_id" column into "id" column.

explode(col("loc"))) => Explode() method is used to retrive any array elements in vertical order.
Ex- > Suppose we have data like ['Ram','Raju']. After using explode data will look like –
Ram
Raju

withColumn("lang", col("loc")[0]).withColumn("lati", col("loc")[1])
In this above method's first part **withColumn("lang", col("loc")[0])** we are creating a new column with 0^{th} element from the array and in second method **withColumn("lati", col("loc")[1])** we are creating another column with the elements of the 1^{st} element of array

Q . 2nd Example of Handling Complex Data (JSON) :

Code:

```
from pyspark.sql import *
from pyspark.sql.functions import *

spark = SparkSession.builder.master("local[2]").appName("test").getOrCreate()
data=" D:\\Spark Venu sir\\drivers \\world_bank.json"
df=spark.read.format("json").load(data)
ndf=df.withColumn("majorsector_percent", explode(col("majorsector_percent")))\
   .withColumn("mjsector_namecode",explode(col("mjsector_namecode")))\
   .withColumn("mjtheme_namecode",explode(col("mjtheme_namecode")))\
   .withColumn("mjtheme",explode(col("mjtheme")))\
   .withColumn("majorsector_percent_name", col("majorsector_percent.Name"))\
   .withColumn("majorsector_percent_percent",col("majorsector_percent.Percent"))\
   .withColumn("mjsector_namecode_code", col("mjsector_namecode.code"))\
   .withColumn("mjsector_namecode_name",col("mjsector_namecode.name"))\
   .withColumn("project_abstract_cdata",col("project_abstract.cdata"))\
   .drop("majorsector_percent","mjsector_namecode","mjtheme_namecode","project_abstract")\
   .withColumn("projectdocs",explode(col("projectdocs")))\
   .withColumn("sector",explode(col("sector")))\
   .withColumn("sector_namecode", explode(col("sector_namecode")))\
   .withColumn("theme_namecode",explode(col("theme_namecode")))\
   .withColumn("theme_namecode_code",col("theme_namecode.code"))\
   .withColumn("theme_namecode_name",col("theme_namecode.name"))\
   .drop("theme_namecode")\
   .withColumn("theme1_name",col("theme1.Name"))\
   .withColumn("theme1_Percent",col("theme1.Percent"))\
   .drop("theme1")\
   .withColumn("sector_namecode_code",col("sector_namecode.code"))\
   .withColumn("sector_namecode_name",col("sector_namecode.name"))\
   .drop("sector_namecode")\
   .withColumn("sector4_Name",col("sector4.Name"))\
   .withColumn("sector4_Percent",col("sector4.Percent"))\
   .drop("sector4") \
   .withColumn("sector3_Name", col("sector3.Name")) \
   .withColumn("sector3_Percent", col("sector3.Percent")) \
   .drop("sector3") \
   .withColumn("sector2_Name", col("sector2.Name")) \
   .withColumn("sector2_Percent", col("sector2.Percent")) \
   .drop("sector2") \
   .withColumn("sector1_Name", col("sector1.Name")) \
   .withColumn("sector1_Percent", col("sector1.Percent")) \
   .drop("sector1") \
   .withColumn("sector_name",col("sector.Name")).drop("sector")\
   .withColumn("projectdocs_docdate",col("projectdocs.DocDate"))\
   .drop("projectdocs")\
   .withColumn("idoid",col("_id.$oid")).drop("_id")

ndf.createOrReplaceTempView("tab")
#res=spark.sql("")
#res=ndf.where(col("countrycode")!="ET")
res=ndf.groupBy(col("countrycode")).count().orderBy(col("count").desc())
res.show()
```

```
res.printSchema()
#explode what it does?
#if u have anywhere Array format remove arrays .. to remove array use explode
#let eg: majorsector_percent: array (nullable = true)
# |    |-- element: struct (containsNull = true)
# |    |    |-- Name: string (nullable = true)
# |    |    |-- Percent: long (nullable = true)
#above data remove array ... use explode at that time u ll get like this
#majorsector_percent: struct (nullable = true)
# |    |-- Name: string (nullable = true)
# |    |-- Percent: long (nullable = true)

#i want to solve struct value ... parent_col.child_column
#theme_namecode: struct (nullable = true)
# |    |-- code: string (nullable = true)
# |    |-- name: string (nullable = true)
#this data convert to theme_namecode_name and theme_namecode_code in this
#col("theme_namecode.code"), col("theme_namecode.name") .drop("theme_namecode")
```

Important Points :

Array datatype example : [“ram”,”rohit”,”raj”]

Struct Datatype example : { "Name" : "Tertiary education", "Percent" : 12 }

Struct Inside array : “student”:[{“name”:”raju”,”roll”:”23”},{“name”:”sinu”,”roll”:”32”}]

explode(col("majorsector_percent")) => explode() method is used to retrieve array elements in vertical representation.

withColumn("majorsector_percent_name", col("majorsector_percent.Name")) => Here we are making separate column from struct datatype. Here **majorsector_percent** is parent column name and **Name** is the child column name.

syntax : withColumn(“newcol”,col(parentColumn.ChildColumn))

Ex -3 Handling Json datatype in most recommended way (UDF)

```
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.functions import *
from pyspark.sql.types import *
import re
spark = SparkSession.builder.master("local[2]").appName("test").getOrCreate()
data="D:\\Spark Venu sir\\drivers\\companies.json"
df=spark.read.format("json").option("multiLine","true").load(data)


def read_nested_json(df):
   column_list = []
   for column_name in df.schema.names:
      if isinstance(df.schema[column_name].dataType, ArrayType):
         df = df.withColumn(column_name, explode(column_name))
         column_list.append(column_name)
      elif isinstance(df.schema[column_name].dataType, StructType):
         for field in df.schema[column_name].dataType.fields:
            column_list.append(col(column_name + "." + field.name).alias(column_name + "_" + field.name))
      else:
         column_list.append(column_name)
   df = df.select(column_list)
   return df;


def flatten(df):
   read_nested_json_flag = True
   while read_nested_json_flag:
      df = read_nested_json(df);
      read_nested_json_flag = False
      for column_name in df.schema.names:
         if isinstance(df.schema[column_name].dataType, ArrayType):
            read_nested_json_flag = True
         elif isinstance(df.schema[column_name].dataType, StructType):
            read_nested_json_flag = True;
   cols = [re.sub('[^a-zA-Z0-1]', "", c.lower()) for c in df.columns]
   return df.toDF(*cols);

ndf=flatten(df)
ndf.printSchema()
```

Reference Links for Study :

https://github.com/maroovi/aws_etl/blob/23d5af070f6c605852ff91fdf0f28423548f59e5/glue.py

#"def read_nested_json" pyspark

https://github.com/krish-17/CSCI5408_Assignments/blob/bf72c77b14ea6320c77accfe306e944fc6a3d4f5/lab_6/lab6.py

https://spark.apache.org/docs/3.1.3/api/java/org/apache/spark/sql/DataFrameReader.html#json-org.apache.spark.sql.Dataset-

Day – 14

Handling XML data in Spark :

To work with XML datatypes we need to import dependency file of xml into spark's jar file, or you will face ClassNotFound error.

First search **spark-xml_2.12** in google and download the .jar file and then go to spark> go to jar> paste that xml jar inside that file.

To Work with AVRO datatype we also need avro dependency jar file. For this go to **mvnrepository.com** and search spark avro. Here choose the latest release dated file (here we are taking 2022 version) and click on it. Next choose the version 3.1.2 as we are working with spark 3.1.2 version. And now download that jar file and place it inside spark's jar folder.