Loading files: (after loading files , data will be in Data frames)

 # TSV/CSV (Structued Data)

#get the schema from the data (Use inferSchema)

#User define Schema(use it based on schema else use InferSchema)

#JSON/PARQUET(SEMI STRUCTURED DATA)

#Write the dataset into file

```
df=spark.read.json(<"filepath.json">) # for reading json
df=spark.read.csv(<"filepath".csv>) # for reading csv file
df=spark.read.parquet(<"filepath".parquet>) # for reading parquet file
```

for reading tsv file (Tab seperated file) use CSV and define seperator using options and heade is used to display header

#infer schema converst the schema as per the data automatically-Kind of learning and siplayig the schema

df=spark.read.option("sep","\t").options("header",True).option("inferSchema",True).csv(<"filepath" >)

#General method to laod any file

df=Spark.read.format(<format of file>).load(<filepath>)

```
df=(spark.read.
        csv("/FileStore/tables/BroadcastLogs_2018_Q3_M8_sample.CSV",
        sep="|",
        header=True,
        inferSchema=True
        )
 )
```

 # Creating User define schema and using it while Loading the data

(This is not only used to define the Data type but also control the number of columns to be in DF)

 # Format is

```
StructType([
                StructFiled("Field Name1",Datatype,Nullable)
                StructFiled("Field Name2",Datatype,Nullable)
                StructFiled("Field Name3",Datatype,Nullable)
            ])
```

#Example:

```
from pyspark.sql.types import StructType,StructFiled,StringType,IntergerType

UserdefiedSchema=StructType([
                                        StructFiled("TimeStamp",StringType(),True)
                                        StructFiled("Site",StringType(),True)
                                        StructFiled("Requestid",IntergerType(),True)
                                    ])
```

#Using user defined schema while loading the file

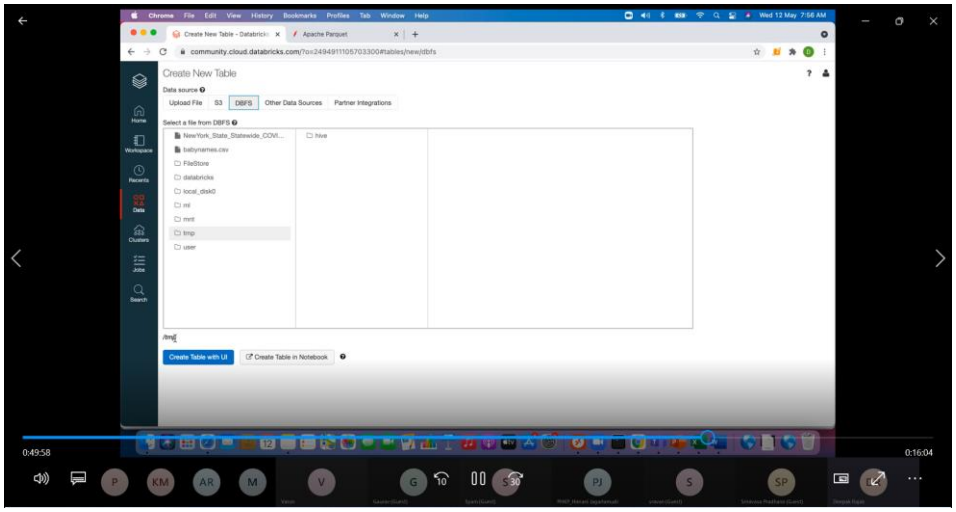
```
df=spark.read.option("sep","\t").option("header",True).schema(UserdefiedSchema).csv("filename.tsv")

df=spark.read.option("inferSchema",True).schema(UserdefiedSchema).parquet("filename.parquet")
```

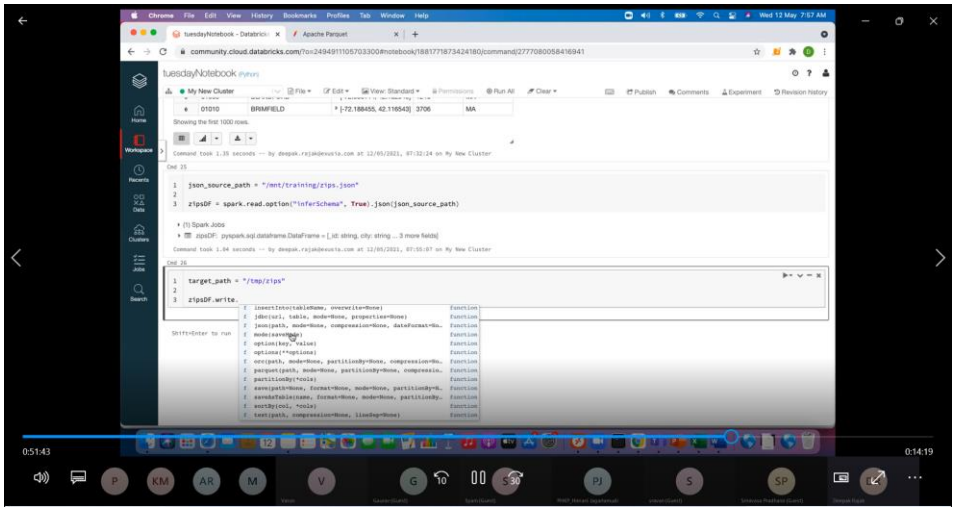
#DDL Formatting for creating schema

ddlschema=”stie string, request integer”

df=spark.read.schema(ddlschema).parquet(filename.parquet)

Writing dataset to file:





#defualt file format is parquet

Target_path=”/temp/zips”

df.write.save(Target_path) *is same as* **df.write.parquet(Target_path)** as the default is parquet.

df.write.format(<file format>).mode(“overwrite”).save(Target_path)

#Reas and write the files in one go:

Spark.read.option(“inferSchema”,True).json(json_source_path).write.mode(“overwrite”).save(Target_path).

Note:*if no mode is applied while writing then it will be error mode by default*

OutputModes*:*

overwrite , append , error , ignore

Deleting files

dbutils.fs.rm(<filepath>)

```
1  dbutils.fs.rm("/tmp/df1.parquet", True)
```

Day 4

#Read the source(file/kafka/Mango DB) -Extraction

#Transformation (Applying business Logic)

#Loading (Store into file/DB/Streamimg(kafka))

#Tranformation(select,filter,join,groupby ,drop) & Action(saving the data,displaying,counting,first n display)

#Tranformations are Lazy

#Actions are Eager

Agenda:

#How can we RENAME a COLUMN?

#How can we change the datatype of a column?

#How can we change the value of the existing column?

#How can we derive new column from an existing?

#How can we RENAME a COLUMN?

```
Sourcepath="/mnt/training/wikipedia/pageviews/pageviews_by_second.tsv"
df=(spark
    .read
    .option("sep","\t")
.option("header",True)
.csv(sourcePath)
)
```

timestamp -→ recordedTime is a ***Transformation and will not display and results***

site->Column2

```
newdf=df.withColumnRenamed("timestamp","recordedTime").withColumnRenamed("site","Column2") #chaning the columns
```

display(newdf) -→*is an action will display the resulst*

#To change all the column

anotherdf=df.toDF("c1","c2","c3")

display(anotherdf)

#How can we change the datatype of a column?

#requests(int)-→String

from pyspark.sql.functions import col

newdf=df.withColumn("requests",col("requests").cast("string"))

display(newdf)

#How can we change the value of the existing column?

from pyspark.sql.functions import col

*newdf=df.withColumn("requests",col("requests")*2) #Mutiplying requests column by 2*

#How can we derive new column from an existing?

#creating a new column newcolumnname and assigning it a value which is twice the value of requests column

from pyspark.sql.functions import col,lit

newdf=df.withColumn("newcolumnname",col("requests")*2)

#creating multiple columns

newdf=df.withColumn("newcolumnname",col("requests")*2).withColumn("country",lit("USA"))

***Note: lit** function is used for supplying Constant Values*

#Reading from source -file /table/Streaming (kafka)

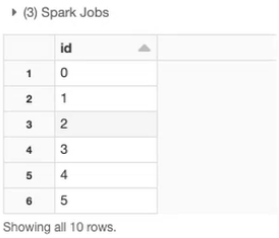
#range-id

#createDataframe -Value

#range(start(optional),number of records,increament(optional),number of partions(optional))

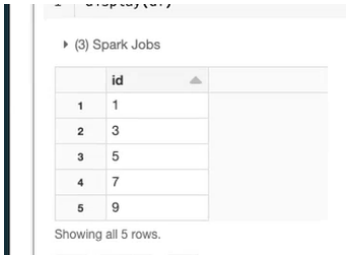
df=spark.range(10) -by default column name is ***id***

display(df)



df=spark.range(1,10,2)

display(df)



createDataFrame

#createDataFram()

from pyspark.sql.types import integerType

data=[1,2,3,4]

df=spark.createDataFrame(data,integerType()) #Default column name is ***value***

UNION:

myschema=StructType([

StructFiled("id",integerType(),True)

StructFiled("name",StringType(),True)

])

My_list=['deepak','leena','ankit','madhu']

Data=[[0,' deepak'],[1,'leena'],[2,'ankit'],[3,'madhu']]

#id should be generated -used enumerate

df=spark.createDataFrame(enumerate(my_list),schema=myschema)

or

df= spark.createDataFrame(Data ,schema=myschema)

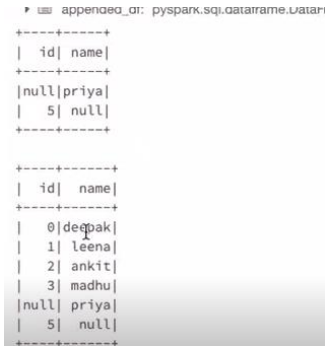
#create 2 records with nulls

nulldf=spark.createDataFrame([[None,'priya'], [5,None]],schema=myschema)

nulldf.show() *#in first record id is null and in second record name is null*

appended_df=df.union(nulldf)

appended_df.show()



FILTERS:

from pyspark.sql.functions import col

missing_name_df=appended_df.filter(col("name").isNull()) *#retrives only rows with missing names*

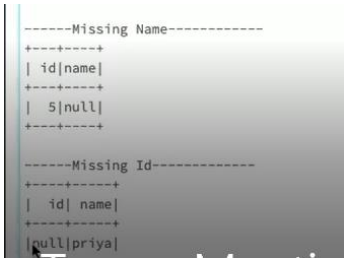
print("-------missing names------")

display(missing_name_df)

missing_id_df=appended_df.filter(col("id").isNull()) *#retrives only rows with missing id*

print("-------missing id------")

display(missing_id_df)



COUNT AND DISTINCT COUNT:

df

	date	delay	distance	origin	destination
1	1011245	6	602	ABE	ATL
2	1020600	-8	369	ABE	DTW
3	1021245	-2	602	ABE	ATL
4	1020605	-4	602	ABE	ATL
5	1031245	-4	602	ABE	ATL
6	1030605	0	602	ABE	ATL

Showing the first 1000 rows.

#Count total flights leaving from a particular source

```
newdf=(df
        .groupby("origin")
        .agg({'destination' : 'count'})
        .withColumnRenamed(count(destination"),"totalNoofFlights")
        )
```

(2) Spark Jobs

csvDF: pyspark.sql.dataframe.DataFrame

```
date: integer
delay: integer
distance: integer
origin: string
destination: string
```

newDF: pyspark.sql.dataframe.DataFrame

```
origin: string
totalNoofFlights: long
```

display(newdf)

	origin	totalNoofFlights
1	GEG	2044
2	BUR	5079
3	GRB	1109
4	GTF	425
5	GRR	2585
6	EUG	1271

```
---------Total Count-----------------
+------+-----------+
|origin|totalCounts|
+------+-----------+
|   GEG|       2044|
|   BUR|       5079|
|   GRB|       1109|
|   GTF|        425|
|   GRR|       2585|
+------+-----------+
only showing top 5 rows
```

#Count only distinct destination

```
from pyspark.sql.functions import countDistinct
uniquedf=(df
        .groupby("origin")
        .agg(countDistinct('destination') .alias('uniquedest'})
        )
Uniquedf.show()
```

```
---------Total Count Distinct----------------
+------+----------+
|origin|uniquedest|
+------+----------+
|   MSY|        35|
|   GEG|         9|
|   BUR|        11|
```

FILTERS AND WHERE WORK THE SAME WAY:

FILTER vs WHERE

```
1  from pyspark.sql.functions import col
2
3  parquetDF = spark.read.parquet("/mnt/training/City-Data.parquet") # Create DF via Parquet File
4
5  print("------------------Printing first 1 records of Parquet DF---------------")
6  parquetDF.show(1)
7
8  filteredDF = parquetDF.filter(col("state") == "New York") # filter() records from state = New York
9  print("------------------Printing first 1 records of Filtered DF---------------\n")
10 filteredDF.show(1)
11
12 filter_by_where_df = parquetDF.where((col("state") == "California") & (col("city") == "Los Angeles"))
13
14 print("------------------Printing first 1 records of Filtered DF by where()---------------\n")
15 filter_by_where_df.show() # The where() clause is equivalent to filter()
```

ORDERBY OR SORTBY.*BOTH ARE SAME*

Usage:

df.sort("state","city").show()

df.sort("state","city").ascn().show() #by default descending

df.orderby("state","city",ascending=0).show(4) #ascending =1 means ascending

GROUP BY AND ORDER BY:

GROUPBY ORDERBY

```
parquetDF = spark.read.parquet("/mnt/training/City-Data.parquet") # Create DF via Parquet File
print("------------------Printing first 5 records of Parquet DF---------------")
parquetDF.show(5)

# Apply GroupBy and Agg Operations and Orderby - state desc

groupedDF = (parquetDF
             .groupBy("state")
             .agg({"city": "count"})  # count(city)
             .withColumnRenamed("count(city)", "NoOfCities")
             .orderBy("state", ascending = 0)
            ) # option 0 for descending

print("------------------Printing first 5 records of Grouped / Aggregated / ordered DF---------------")

groupedDF.write.mode("overwrite").save("/tmp/city1/grouped.parquet")
```

AGGREGATION ON MULTIPLE COLUMNS

AGG on MULTIPLE COLUMNS

```
parquetDF = spark.read.parquet("/mnt/training/City-Data.parquet") # Create DF via Parquet FiLE

# # groupBy and aggregate on single column

(parquetDF
 .groupBy("state")
 .min("population2010")
 .withColumnRenamed("min(population2010)", "minPopulation")
 .orderBy("state", ascending = 0)
 .show(4)
)

# groupBy and aggregate on multiple columns

from pyspark.sql.functions import min, max, avg, sum, mean, count

(parquetDF
 .groupBy("state")
 .agg(min("population2010").alias("min_population")\
                           ,max("population2010").alias("max_population")\
                           ,sum("estPopulation2016").alias("est_population")\
                           ,count("city").alias("total_no_cities")\
                           ,mean("estPopulation2016").alias("mean_estPopulation2016"))\

 .orderBy("state", ascending = 1).show(5)
) # ascending True
```

DISTINCT:

distinctdf=df.distinct()

DROPDUPLICATES:

distinctdf=df.dropDuplicates()

SELECT:

from pyspark.sql.functions import col

ndf=df.select("origin","destination").filter(col("origin")=="ABE")--Narrow Dependency/Tranformation-

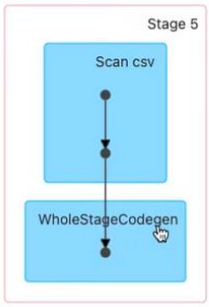
```
nDF: pyspark.sql.dataframe.DataFrame
  origin: string
  destination: string
```

display(ndf)

```
1  display(nDF)
```

(3) Spark Jobs
- Job 3 View (Stages: 1/1)
- Job 4 View (Stages: 1/1)
- Job 5 View (Stages: 1/1)

	origin	destination
1	ABE	ATL
2	ABE	DTW
3	ABE	ATL
4	ABE	ATL
5	ABE	ATL
6	ABE	ATL

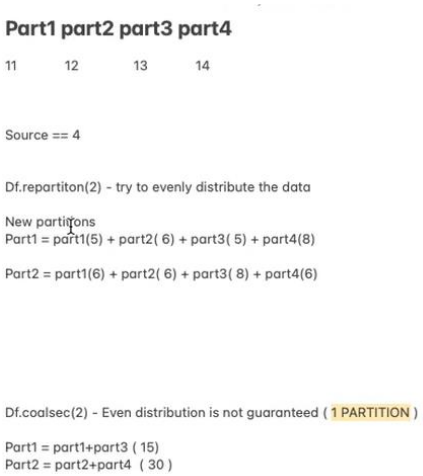


wdf=df.groupBy("origin").agg({"destination":"count"})---WIDE TRANFORMATION-SHUFFLING HAPPENS

```
wDF: pyspark.sql.dataframe.DataFrame
  origin: string
  count(destination): long
```

Command took 0.20 seconds -- by deepak.rajak@exusia.com at 17/05/2021, 07:18:37 on My New Cluster

REPARTITION AND COALESCE:



---

df.rdd.getNumPartitions() *#to get the number of partitions in Data frame*

df.repartition(2) *-Diritibutes the Data evenly*

df.coalesce(2) *-Even distribution is not guaranteed -so used when you need one partition*

form pyspark.sql import Row

form pyspark.sql.functions import corl,split

df_row=Row("id","name","city") #header

row1=df_row(1,"VAMSHIDHAR,Reddy,kanamantahreddy","T-Hyderabad")

row2= df_row(2,"SANDEEP,Reddy,E","GA-ATLANTA")

row3= df_row(3,"DHARMEDER,,A","GA-ATLANTA")

row4= df_row(4,"SRAVAN,Reddy,G","TX-FRISCO")

row5= df_row(5,"VARUN,Reddy,kanamantahreddy","KW-KUWAIT")

df_row_seq=[row1,row2,row3,row4,row5]

df=spark.createDataFrame(df_row_seq)

display(df)

▸ (3) Spark Jobs
▾ df: pyspark.sql.dataframe.DataFrame
id: long
name: string
city: string

	id	name	city
1	1	deepak,kumar,rajak	T-Hyderabad
2	2	virendra,singh,solanki	MP-Bhoapl
3	3	sarvesh,kumar,mishra	MH-Pune
4	4	vasavi,,devi	AP-Vizag
5	5	leena,,rajak	CG-Bilaspur

SPLITING

#Split name

```
Splitteddf=(df
        .withColumn("firstname",split(col("name"),",")[0]
        .withColumn("Middlename",split(col("name"),",")[1]
        .withColumn("lastname",split(col("name"),",")[2]
        .drop(col("name"))
        )
display(spliteddf)
```

	id	city	firstname	middlename	lastname
1	1	T-Hyderabad	deepak	kumar	rajak
2	2	MP-Bhoapl	virendra	singh	solanki
3	3	MH-Pune	sarvesh	kumar	mishra
4	4	AP-Vizag	vasavi		devi
5	5	CG-Bilaspur	leena		rajak

Showing all 5 rows.

IMPUTING NULL OR MISSING DATA

Imputing Null or Missing Data

```
# Creating a dataframe
corruptDF = spark.createDataFrame([ (11, "Deepak", 5), \
                                    (12, "Leena", None), \
                                    (13, "Deepa", 7), \
                                    (14, None, 9), \
                                    (15, "Preety", None)], \
                                   ["id", "name", "bday"])
# Handling NULLs
drop_nulls = corruptDF.na.drop() # Dropping NULL Values ( Only 2 Rows will be left)

drop_nulls.show()

fill_nulls_with_value = corruptDF.na.fill({"name" : "Unknown", "bday": 1}) # ( Filling NULL Values)

fill_nulls_with_value.show() # name we are filing as "Unknown", bday we are filling as "1"
```

```
▸ (6) Spark Jobs
▸ corruptDF:  pyspark.sql.dataframe.DataFrame = [id: long, name: string ... 1 more fields]
▸ drop_nulls:  pyspark.sql.dataframe.DataFrame = [id: long, name: string ... 1 more fields]
▸ fill_nulls_with_value:  pyspark.sql.dataframe.DataFrame = [id: long, name: string ... 1 more fields]
+---+------+----+
| id|  name|bday|
+---+------+----+
| 11|Deepak|   5|
| 13| Deepa|   7|
+---+------+----+

+---+-------+----+
| id|   name|bday|
+---+-------+----+
| 11| Deepak|   5|
| 12|  Leena|   1|
| 13|  Deepa|   7|
| 14|Unknown|   9|
| 15| Preety|   1|
+---+-------+----+
```

SPARK TABLES: 4 types

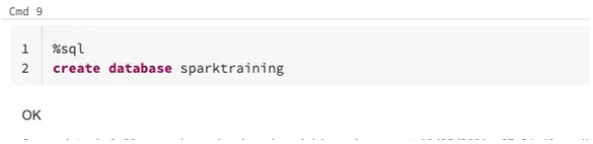
```
# Managed Table

# External Table

# Temporary View / Table

# Global Temp View / Table
```

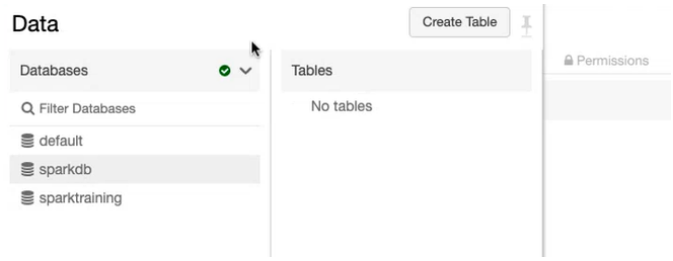
 # Managed Table-→Spark will control the Data + Metadata(Drop)

#External Table -→ Spark controls the metadata & user controls the Data(drop)

%sql →dentoed you are writing seq code below

Create database sparktraining





For using Datatbase

%sql

use sparkdb

df.write.saveAsTable("my_managed_table")

Creating Data frame from DB

query*="""" select * from my_managed_table """"*

df=spark.sql(query)

or

df=sql(query)

%sql

select * from my_mamged_table

(3) Spark Jobs

	id	name	city
1	2	virendra,singh,solanki	MP-Bhoapl
2	1	deepak,kumar,rajak	T-Hyderabad
3	3	sarvesh,kumar,mishra	MH-Pune
4	5	leena,,rajak	CG-Bilaspur
5	4	vasavi,,devi	AP-Vizag

Showing all 5 rows.

Other way:

tabledf=spark.read.table("my_managed_table")

or

tabledf= table("my_managed_table")

%sql

describe detail my_managed_table

describe extended detail my_managed_table

```
%sql
describe extended my_managed_table
```

	col_name	data_type	comment
1	id	bigint	
2	name	string	
3	city	string	
4			
5	# Partitioning		
6	Not partitioned		

Showing all 13 rows.

	col_name	data_type	comment
8	# Detailed Table Information		
9	Name	sparkdb.my_managed_table	
10	Location	dbfs:/user/hive/warehouse/sparkdb.db/my_managed_table	
11	Provider	delta	
12	Type	MANAGED	
13	Table Properties	[delta.minReaderVersion=1,delta.minWriterVersion=2]	

owing all 13 rows.

spark.catalog.listTables()

```
spark.catalog.listTables()
```

(1) Spark Jobs

```
Out[28]: [Table(name='my_managed_table', database='sparkdb', description=None, tableType='MANAGED', isTemporary=False)]
```

Command took 0.64 seconds -- by deepak.rajak@exusia.com at 18/05/2021, 08:02:18 on My New Cluster

Writing the data to table-EXTERNAL TABLE

df.write.option("path","/tmp/mypath_ext").saveAsTable("*MY_EXTERNAL_TABLE*")

```
df.write.option("path", "/tmp/mypath_ext").saveAsTable("MY_EXTERNAL_TABLE")
```

▸ (4) Spark Jobs

Command took 8.74 seconds -- by deepak.rajak@exusia.com at 18/05/2021, 08:05:17 on My New Cluster

md 46

```
%sql
describe extended my_external_table
```

	col_name	data_type	comment
1	id	bigint	
2	name	string	
3	city	string	
4			
5	# Partitioning		
6	Not partitioned		

Showing all 13 rows.

Command took 0.26 seconds -- by deepak.rajak@exusia.com at 18/05/2021, 08:05:38 on My New Cluster

	col_name	data_type	comment
8	# Detailed Table Information		
9	Name	sparkdb.my_external_table	
10	Location	dbfs:/tmp/mypath_ext	
11	Provider	delta	
12	Type	EXTERNAL	
13	Table Properties	[delta.minReaderVersion=1,delta.minWriterVersion=2]	

Showing all 13 rows.

LIST THE TABLES:

```
spark.catalog.listTables()
```

▸ (2) Spark Jobs

```
Out[30]: [Table(name='my_external_table', database='sparkdb', description=None, tableType='EXTERNAL', isTemporary=False),
 Table(name='my_managed_table', database='sparkdb', description=None, tableType='MANAGED', isTemporary=False)]
```

#Temporary View

```
df.cre
```

```
f createGlobalTempView(name)          function
f createOrReplaceGlobalTempView(name) function
f createOrReplaceTempView(name)       function
f createTempView(name)                function
```

Cmd 48

```
dbutils.fs.ls("/tmp/mypath_ext")
```

df.createOrReplaceTempView("temptable")

spark.catalog.lisTables()

```
1  spark.catalog.listTables()
```

▸ (3) Spark Jobs

```
Out[32]: [Table(name='my_external_table', database='sparkdb', description=None, tableType='EXTERNAL', isTemporary=False),
 Table(name='my_managed_table', database='sparkdb', description=None, tableType='MANAGED', isTemporary=False),
 Table(name='temptable', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]
```

%sql

Select * from temptable

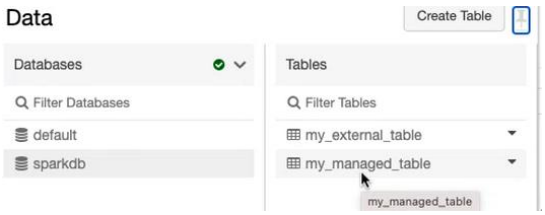
```
1  %sql
2  select * from temptable
```

▸ (3) Spark Jobs

	id	name	city
1	1	deepak,kumar,rajak	T-Hyderabad
2	2	virendra,singh,solanki	MP-Bhoapl
3	3	sarvesh,kumar,mishra	MH-Pune
4	4	vasavi,,devi	AP-Vizag
5	5	leena,,rajak	CG-Bilaspur

Showing all 5 rows.

Note that temp table is not available in spark



#Global tables

df.createOrReplaceGlobalTempView("gtemptable")

spark.catalog.listTables()

```
1  spark.catalog.listTables()
```

▸ (3) Spark Jobs

```
Out[34]: [Table(name='my_external_table', database='sparkdb', description=None, tableType='EXTERNAL', isTemporary=False),
 Table(name='my_managed_table', database='sparkdb', description=None, tableType='MANAGED', isTemporary=False),
 Table(name='temptable', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]
```

Command took 0.49 seconds -- by deepak.rajak@exusia.com at 18/05/2021, 08:11:53 on My New Cluster

%sql

select * from gtemptable—will give error

```
1  %sql
2  select * from gtemptable
```

```
Error in SQL statement: AnalysisException: Table or view not found: gtemptable; line 1 pos 14;
'Project [*]
+- 'UnresolvedRelation [gtemptable], [], false
```

Command took 0.07 seconds -- by deepak.rajak@exusia.com at 18/05/2021, 08:12:10 on My New Cluster

Note: GLOBAL TEMP TABLES WILL BE STRORED UNDER DB global_temp

select * from global_temp.gtemptable

```
1  %sql
2  select * from global_temp.gtemptable
```

▸ (3) Spark Jobs

	id	name	city
1	1	deepak,kumar,rajak	T-Hyderabad
2	2	virendra,singh,solanki	MP-Bhoapl
3	3	sarvesh,kumar,mishra	MH-Pune
4	4	vasavi,,devi	AP-Vizag
5	5	leena,,rajak	CG-Bilaspur

Showing all 5 rows.

```
1  spark.catalog.listTables("global_temp")
```

▸ (2) Spark Jobs

```
Out[35]: [Table(name='gtemptable', database='global_temp', description=None, tableType='TEMPORARY', isTemporary=True),
 Table(name='temptable', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]
```

Command took 0.36 seconds -- by deepak.rajak@exusia.com at 18/05/2021, 08:13:13 on My New Cluster

Note: we can access global temp tables from any where until cluster is running where as temp tables cannot be accessed in another jupitor book

Configure storage account key:

Spark.conf.set*("fs.azure.accoun.key.<storage-account-name>.dfs.core.windows.net",<storage-acount-key>*

Spark.conf.set*("fs.azure.createRemoteFileSystemDuringInitialization","true")*

Configure storage account key

```
1  spark.conf.set("fs.azure.account.key.<storage-account-name>.dfs.core.windows.net", <storage-account-key>)
2
3  spark.conf.set("fs.azure.createRemoteFileSystemDuringInitialization", "true")
4
5  dbutils.fs.ls("abfss://<file_system>@<storage-account-name>.dfs.core.windows.net/")
6
7  spark.conf.set("fs.azure.createRemoteFileSystemDuringInitialization", "false")
8
```

Cmd 26

```
#storage-account-name - storagedatabricks19
#storage-account-key - Bm49nF+HVC/NRlTuHBhuHY2y8yhm0Db1PCEs7F1EcQRlXL3XK9u7HkA5kBqajZkbjuI0uxgBOfOZ/ojTeaq1Nw==
```

Cmd 27

```
spark.conf.set("fs.azure.account.key.storagedatabricks19.dfs.core.windows.net",
"Bm49nF+HVC/NRlTuHBhuHY2y8yhm0Db1PCEs7F1EcQRlXL3XK9u7HkA5kBqajZkbjuI0uxgBOfOZ/ojTeaq1Nw==")
```

Command took 0.44 seconds -- by deepak.rajak@exusia.com at 19/05/2021, 08:06:17 on My New Cluster

Cmd 28

```
dbutils.fs.ls("abfss://first@storagedatabricks19.dfs.core.windows.net/")
```

```
Out[2]: [FileInfo(path='abfss://first@storagedatabricks19.dfs.core.windows.net/source/', name='source/', size=0),
 FileInfo(path='abfss://first@storagedatabricks19.dfs.core.windows.net/target/', name='target/', size=0)]
```

Command took 1.76 seconds -- by deepak.rajak@exusia.com at 19/05/2021, 08:08:18 on My New Cluster

Reading file from azure storage

```
# Reading from ADLS
csvDF = (spark
 .read
 .option("sep", "\t")
 .option("header", True)
 .option("inferschema", True)
 .csv("abfss://first@storagedatabricks19.dfs.core.windows.net/source/pageviews_by_second.tsv")
)
```

Writing to azure location:

```
csvDF.write.format("parquet").save("abfss://first@storagedatabricks19.dfs.core.windows.net/target/pageview.parquet")
```

Dr

mounting AWS S3 bucket - databricks1905 to dbfs location - /mnt/deepakS3_databricks19

```
ACCESS_KEY = "AKIAZ57C5I33LJZGZQ4F"

SECRET_KEY = "jEXxA9ni5Pkf3dIk4676bhzhU7OFkLzI++qMuVSe"

ENCODED_SECRET_KEY = SECRET_KEY.replace("/", "%2F")

AWS_BUCKET_NAME = "databricks1905"

MOUNT_NAME = "deepak_s3"


# Mounting
dbutils.fs.mount(f"s3a://{ACCESS_KEY}:{ENCODED_SECRET_KEY}@{AWS_BUCKET_NAME}", f"/mnt/{MOUNT_NAME}")
```

Transformation

```
# test_date - > date ( col("column").cast("date") )
# Drop - cumulative_number_of_positives, cumulative_number_of_tests
# Add a column - State (Newyork State)
# Add a column - load_date ( current_date)

from pyspark.sql.functions import to_date, lit, current_date

loadDF = (neWYorkCovidTestingDF
          .withColumn("test_date", to_date("test_date"))         # test_date - > date
          .drop("cumulative_number_of_positives", "cumulative_number_of_tests") # Drop - cumulative_number_of_positives, cumulative_number_of_tests
          .withColumn("state", lit("Newyork State")) # Add a column - State (Newyork State)
          .withColumn("load_date", current_date()) # Add a column - load_date ( current_date)
         )


display(loadDF)
```

Connecting to Database in Cloud server

Request connection string details from Admin

Load to SQL Database

```
jdbcUsername = "drajak" # dbuser
jdbcPassword = "Infy@123" # dbpass

driverClass = "com.microsoft.sqlserver.jdbc.SQLServerDriver"

jdbcHostname = "covid19server20.database.windows.net"
jdbcPort = 1433
jdbcDatabase = "covid19"

# Create the JDBC URL without passing in the user and password parameters.
jdbcUrl = "jdbc:sqlserver://{0}:{1};database={2}".format(jdbcHostname, jdbcPort, jdbcDatabase)

# Create a Properties() object to hold the parameters.
connectionProperties = {
  "user" : jdbcUsername,
  "password" : jdbcPassword,
  "driver" : driverClass
}

# Load in to Database - By Default single connecion / 4

loadDF.repartition(8).write.jdbc(url=jdbcUrl, table="covid6april", mode="overwrite" , properties=connectionProperties)
```

Load to SQL Datatable

Load to SQL Database

```
jdbcUsername = "drajak" # dbuser
jdbcPassword = "Infy@123" # dbpass

driverClass = "com.microsoft.sqlserver.jdbc.SQLServerDriver"

jdbcHostname = "covid19server20.database.windows.net"
jdbcPort = 1433
jdbcDatabase = "covid19"

# Create the JDBC URL without passing in the user and password parameters.
jdbcUrl = "jdbc:sqlserver://{0}:{1};database={2}".format(jdbcHostname, jdbcPort, jdbcDatabase)

# Create a Properties() object to hold the parameters.
connectionProperties = {
  "user" : jdbcUsername,
  "password" : jdbcPassword,
  "driver" : driverClass
}

# Load in to Database - By Default single connecion / 4

loadDF.write.jdbc(url=jdbcUrl, table="covid19", mode="overwrite" , properties=connectionProperties)

print("Table loaded successfully!!!\n")
```

Read from SQL Database:

df=spark.read.jdbc(url=<jdbcurl>,table="SalesLT.Product", properties=connectionProperties)

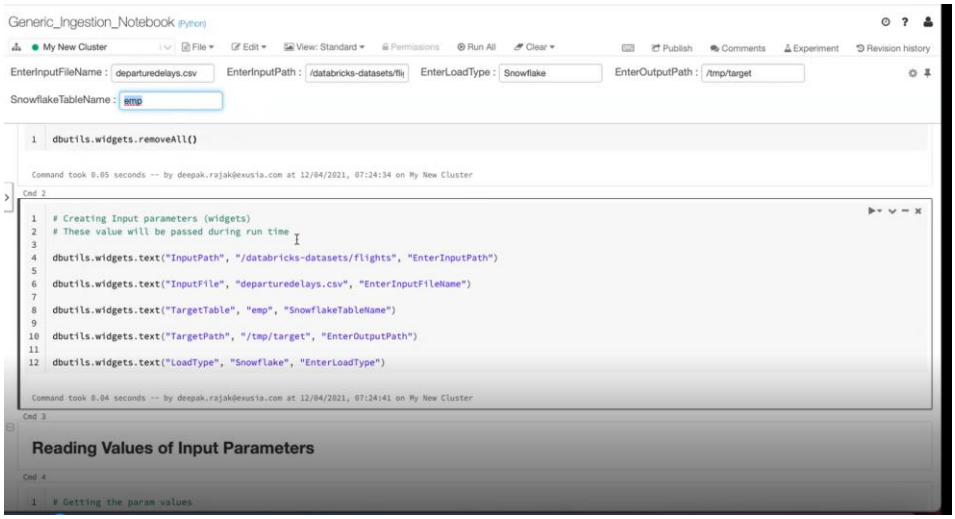
display(df)

Select only few columns

df=spark.read.jdbc(url=<jdbcurl>,table="SalesLT.Product", properties=connectionProperties).select("ProductID","Name")

#Creating Input parameters in Databricks:

dbutils.widgets.text("Inputpath","/databriks-datasets/flights","EnterInputPath")



#Reading Parameters

Source_File_Path=getArgument("InputPath")

Reading file format /extracting file format:

Method 1

import pathlib

path=getArgument("InputPath")+"/"+getArgument("InputFile")

File_Extension=pathlib.Path(path).suffix

Method 2

File_Extension=path.split("/")[-1].split(".")[-1] -Using python split function

Get the file based on file format:

Cmd 7

```
# Function to get the DataFrame based on the File Format

def getDf(path, File_Extension):
    if File_Extension == '.csv':
            inputDF = (spark.read.option("header",True).csv(path))
    elif File_Extension == '.parquet':
            inputDF = (spark.read.parquet(path))
    elif File_Extension == '.json':
            inputDF = (spark.read.json(path))
    else:
          print("File Format Not Supported")
    return inputDF
```

```
# Calling the function and getting the Dataframe
df = getDf(path, File_Extension)
```

SNOWFLAKE-SAAS

REDSHIFT-FROM AWS

SYNAPSE-

BIGQuery

Are all MPP-How do they spereate storage and compute

Running Notebook using %run

#Creating and Renaming the columns using Select and Alias

```
df1=(df
   .select("Duration",
          f.col("Duration").substr(1,2).cast("int").alias("dur_Hours"),
          f.col("Duration").substr(1,2).cast("int").alias("dur_Minutes"),
          f.col("Duration").substr(1,2).cast("int").alias("dur_Seconds"))
   .select("Duration",
          (f.col("dur_Hours")*60*60+
          f.col("dur_Minutes")*60+
          f.col("dur_Seconds")).
```

alias("Duration_in_sec")

)

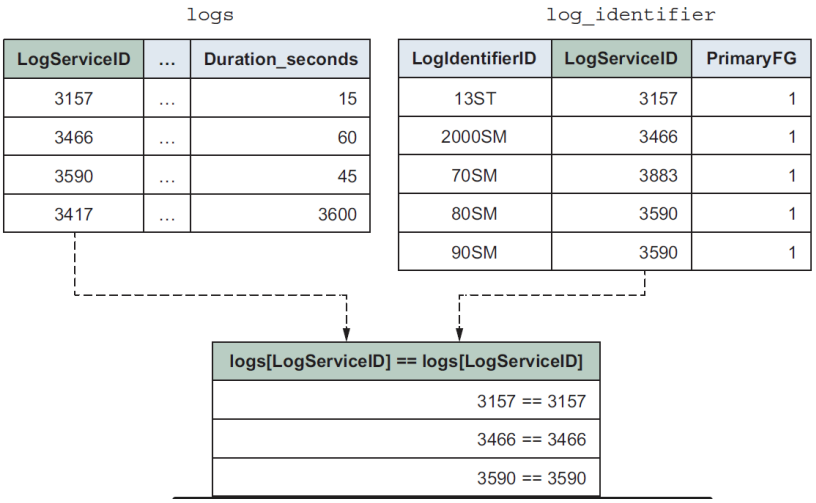
)

Joins:

LEFT and RIGTH ARE TWO DIFFERENT DATA FRAMES

[LEFT].join(
[RIGHT],
on=[PREDICATES]
how=[METHOD]
)

e.g.



logs_Channels_Verbose=logs.join(log_identifier,"LogServiceID")

#or

logs_Channels_Verbose=logs.join(log_identifier,logs["LogServiceID"]==log_identifier["LogServiceID"])

Iterating over each Columns and Renaming the

For col in df.columns:

df=df.withColumnRenamed(col,col.replace(“.”,””).replace(“ ”,”_”)

or too change the type

For col in df.columns:

df=df.select(F.col(col).cast(“int”).alias(col))

or

For c in df.columns:

df=df.withcolumn(c,col(c).cast(“int”))

same as

df.columns =[col(c).cast(“int”).alias(c) for c in df.columns]

User Defined Function:

E.G

lower=udf(lambda s:s.lower())

df=df.select(lower(df[“firstname

Print columns:

df.columns<-gives the list of columns avialable

--

When-otherwise

```
F.when(
F.trim(F.col("ProgramClassCD")).isin(
["COM", "PRC", "PGI", "PRO", "PSA", "MAG", "LOC", "SPO", "MER", "SOL"]
),
F.col("duration_seconds"),
).otherwise(0)
```

ExplODE AND Arrayzip

```
from pyspark.sql.functions import arrays_zip
df_arr = spark.createDataFrame([(([1, 2, 3], [2, 3, 4]))], ['vals1', 'vals2'])
display(df_arr)
```

▸ (3) Spark Jobs

	vals1	vals2
1	▸ [1, 2, 3]	▸ [2, 3, 4]

For converting the two lists into key pair ise array zip

```
df_zip=df_arr.select(arrays_zip(F.col("vals1"),F.col("vals2")).alias("zipped"))
display(df_zip)
```

▸ (3) Spark Jobs

	zipped
1	▸ [{"vals1": 1, "vals2": 2}, {"vals1": 2, "vals2": 3}, {"vals1": 3, "vals2": 4}]

Showing all 1 rows.

For making it into multiple columns:

Cmd 10

```
df_final=df_zip.select(F.explode("zipped").alias("EXPLODED"))
display(df_final)
```

▸ (3) Spark Jobs

	EXPLODED
1	▸ {"vals1": 1, "vals2": 2}
2	▸ {"vals1": 2, "vals2": 3}

Showing all 3 rows.

Reading JSON Files In Pyspark

```
shows = spark.read.json("./data/shows/shows-silicon-valley.json")
```

Reading Multiple Json files:-Make sure to have same schema

```
three_shows = spark.read.json("./data/shows/shows-*.json", multiLine=True)
```

Splitting arry columns into multiple columsn

Import pyspark.sql.functions as f

Consider below data framre as df

array_subset: DataFrame

schedule	genres
Silicon Valley	Comedy

Split genres columns inti multiple columns

```
df=(df
        .select("schedule",f.col("genres")[0].aslias("genres_1"))
)
```

Or

```
df=(df
        .select("schedule",f.col("genres").getItem(0).aslias("genres_1"))
)
```

Or

```
df=(df
        .select("schedule",df.genres[0].aslias("genres_1"))
)
```

Or

```
df=(df
        .select("schedule",df.genres.getItem[0].aslias("genres_1"))
)
```

Note:Do not used slicing operation like df.genres[1:10].

Crating arrays from Columns

```
array_subset_repeated = array_subset
select(
"name",
F.array("one", "two", "three").alias("Some_Genres"),
F.array_repeat("dot_and_index",
5).alias("Repeated_Genres"),# Duplicating the values five times within an array using array_repeat()
)
```

Computing the size of array:

```
df=(df
      .select ("name",f.size("Some_Genres"),
      f.size("Repeated_Genres"))
      )
```

o/p:

```
array_subset_repeated.select(
    "name", F.size("Some_Genres"), F.size("Repeated_Genres")
).show()

# +--------------+-----------------+---------------------+
# |          name|size(Some_Genres)|size(Repeated_Genres)|
# +--------------+-----------------+---------------------+
# |Silicon Valley|                3|                    5|
# +--------------+-----------------+---------------------+
```

Computing the number of elements into both arrays using the size() function

Removing the Duplicates form Arrays-Using array_distinct

```
df=df.select("name",f.array_disinct("Some_Genres"),
f.array_disinct("Repeated_Genres"))
```

```
array_subset_repeated.select(
    "name",
    F.array_distinct("Some_Genres"),
    F.array_distinct("Repeated_Genres"),
).show(1, False)
```

Removing duplicates into both arrays with the array_distinct() method. Since Some_Genres doesn't have any duplicates, the values within the array don't change.

```
# +--------------+--------------------------+------------------------------+
# |name          |array_distinct(Some_Genres)|array_distinct(Repeated_Genres)|
# +--------------+--------------------------+------------------------------+
# |Silicon Valley|[Comedy, Horror, Drama]    |[Comedy]                      |
# +--------------+--------------------------+------------------------------+
```

Finding common/matching values from Arrays-Using array_intersect

```
df=df.select("name",f.array_intersect("Some_Genres
"Repeated_Genres"))
```

o/p

```
# +--------------+--------+
# |          name|  Genres|
# +--------------+--------+
# |Silicon Valley|[Comedy]|
# +--------------+--------+
```

Finding position of an element in an array : using array_position

df=df.select("Genres",f.array_position("Genres","Comedy"))

```
# +--------+-------------------------------+
# |  Genres|array_position(Genres, Comedy)|
# +--------+-------------------------------+
# |[Comedy]|                              1|
# +--------+-------------------------------+
```

Creating Maps form Columns:

```
df_items=df_items.select(*df_items.columns)
df_items.show()
```

▸ (3) Spark Jobs

```
+------+--------+-----+
| Items|Quantity|Price|
+------+--------+-----+
|Banana|       2| 1.74|
| Apple|       4| 2.04|
|Carrot|       1| 1.09|
|  Cake|       1|10.99|
+------+--------+-----+
```

```
df1=df_items.select(f.array("Items","Quantity","Price").alias("values"),f.array([f.lit(c) for c  in df_items.columns]).alias("keys")).show(10)
# df.show()
```

▸ (3) Spark Jobs

```
+----------------+--------------------+
|          values|                keys|
+----------------+--------------------+
|[Banana, 2, 1.74]|[Items, Quantity,...|
| [Apple, 4, 2.04]|[Items, Quantity,...|
|[Carrot, 1, 1.09]|[Items, Quantity,...|
| [Cake, 1, 10.99]|[Items, Quantity,...|
+----------------+--------------------+
```

df2=df1.select(f.map_from_arrays("keys","values"))

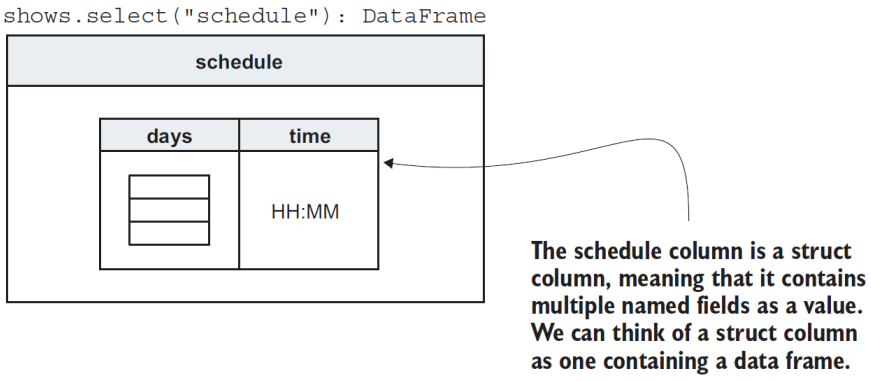
df2.show()

```
df2=df1.select(f.map_from_arrays("keys","values"))
df2.show()
```

▸ (3) Spark Jobs

```
+----------------------------+
|map_from_arrays(keys, values)|
+----------------------------+
|         {Items -> Banana,...|
|         {Items -> Apple, ...|
|         {Items -> Carrot,...|
|         {Items -> Cake, Q...|
+----------------------------+
```

Struct :



RDD N UDFS

sc = spark.sparkContext

collection = [1, "two", 3.0, ("four", 4), {"five": 5}]

collection_rdd = sc.parallelize(collection)

print(collection_rdd)

```
def add_one(value):
    value=value+1
```

map:Used for transformation .applies transformation function on entire data

collection_rdd = collection_rdd.map(add_one)

Print(collection_rdd)

Filters-Filter will return Truthy Values:

collection_rdd = collection_rdd.filter(

lambda elem: isinstance(elem, (float, int))

)

Reduce : *Similar to group by And aggaregare in DF*

```
from operator import add
collection_rdd = sc.parallelize([4, 7, 9, 1, 3])
print(collection_rdd.reduce(add))
```

UDF Definition

Userdefinded Name=udf(userdefiedfunction,returntype)

```
reduce_fraction = F.udf(py_reduce_fraction, SparkFrac)
```

Reading Corrupt data:

```
BY DEFAULT READ MODE IS PERMISSIVE
```

data="""{"a":1,"b":2,"c":3}|{"a":1,"b":2,"c":3}|{"a":1,"b","c":3}""".split("|")

rdd_data=sc.parallelize(data)

curreptdf=spark.read.json(rdd_data)

display(curreptdf)

```
1  data="""{"a":1,"b":2,"c":3}|{"a":1,"b":2,"c":3}|{"a":1,"b","c":3}""".split("|")
2  rdd_data=sc.parallelize(data)
3  print(rdddf.collect())
4  curreptdf=spark.read.json(rdd_data)
5  display(curreptdf)
```

▸ (5) Spark Jobs

```
['{"a":1,"b":2,"c":3}', '{"a":1,"b":2,"c":3}', '{"a":1,"b","c":3}']
```

	_corrupt_record	a	b	c
1	null	1	2	3
2	null	1	2	3
3	{"a":1,"b","c":3}	null	null	null

Types of reading files

DROPMALFORMED:

```
data="""{"a":1,"b":2,"c":3}|{"a":1,"b":2,"c":3}|{"a":1,"b","c":3}""".split("|")
rdd_data=sc.parallelize(data)
curreptdf=spark.read.option("mode","DROPMALFORMED").json(rdd_data)
display(curreptdf)
```

▸ (4) Spark Jobs

	a	b	c
1	1	2	3
2	1	2	3

FAILFAST

Python

```
data="""{"a":1,"b":2,"c":3}|{"a":1,"b":2,"c":3}|{"a":1,"b","c":3}""".split("|")
rdd_data=sc.parallelize(data)
curreptdf=spark.read.option("mode","FAILFAST").json(rdd_data)
display(curreptdf)
```

▸ (1) Spark Jobs

⊞org.apache.spark.SparkException: Job aborted due to stage failure: Task 7 in stage 20.0 failed 1 times, most recent failure: Lost task 7.0 in stage 20.0 (TID 103) (ip-10-172-166-243.us-west-2.compute.internal executor driver): org.apache.spark.SparkException: Malformed records are detected in schema inference. Parse Mode: FAILFAST.

Command took 0.85 seconds -- by vkanamanthareddy1@esu.edu at 3/20/2022, 9:31:07 AM on Demo

---

Read Streaming data:

```
# Define the schema using a DDL-formatted string.
dataSchema = "Recorded_At timestamp, Device string, Index long, Model string, User string, _corrupt_record String, gt string, x double, y double, z double"
```

Command took 0.04 seconds -- by deepak.rajak@exusia.com at 13/04/2021, 07:33:01 on Exusia

d 8

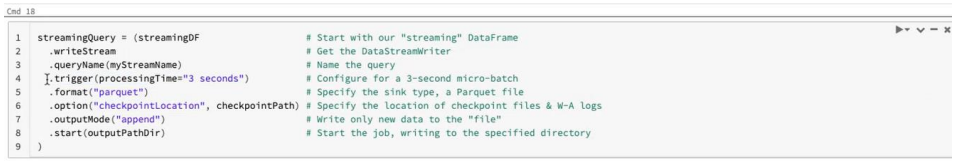
```
# Near Real time ( A little lag between receving the data & processing the data )
# continuous stream of data == microbatches1 + microbach2 + 3 + 4.....
# processing rate > input rate
# Why inferSchema cant be used ?
```

d 9

```
dataPath = "dbfs:/mnt/training/definitive-guide/data/activity-data-stream.json"

initialDF = (spark
  .readStream                           # Returns DataStreamReader
  .option("maxFilesPerTrigger", 1)      # Force processing of only 1 file per trigger
  .schema(dataSchema)                   # Required for all streaming DataFrames
  .json(dataPath)                       # The stream's source directory and file type
)
```

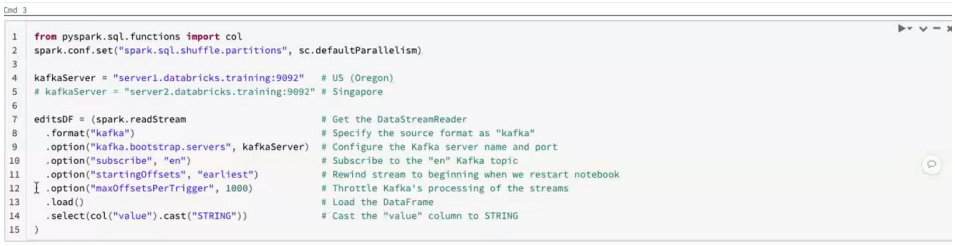
Streaming Query:

Cmd 18

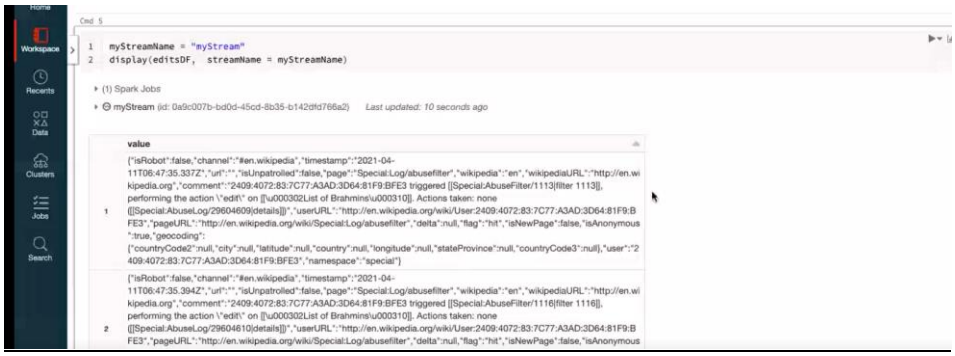


```
streamingQuery = (streamingDF                  # Start with our "streaming" DataFrame
  .writeStream                                 # Get the DataStreamWriter
  .queryName(myStreamName)                     # Name the query
  .trigger(processingTime="3 seconds")         # Configure for a 3-second micro-batch
  .format("parquet")                           # Specify the sink type, a Parquet file
  .option("checkpointLocation", checkpointPath) # Specify the location of checkpoint files & W-A logs
  .outputMode("append")                        # Write only new data to the "file"
  .start(outputPathDir)                        # Start the job, writing to the specified directory
)
```

Reading from KAFKA



```
Cmd 3

from pyspark.sql.functions import col
spark.conf.set("spark.sql.shuffle.partitions", sc.defaultParallelism)

kafkaServer = "server1.databricks.training:9092"   # US (Oregon)
# kafkaServer = "server2.databricks.training:9092" # Singapore

editsDF = (spark.readStream                        # Get the DataStreamReader
  .format("kafka")                                 # Specify the source format as "kafka"
  .option("kafka.bootstrap.servers", kafkaServer)  # Configure the Kafka server name and port
  .option("subscribe", "en")                       # Subscribe to the "en" Kafka topic
  .option("startingOffsets", "earliest")           # Rewind stream to beginning when we restart notebook
  .option("maxOffsetsPerTrigger", 1000)            # Throttle Kafka's processing of the streams
  .load()                                          # Load the DataFrame
  .select(col("value").cast("STRING"))             # Cast the "value" column to STRING
)
```



```
Cmd 5

myStreamName = "myStream"
display(editsDF,  streamName = myStreamName)
```

▸ (1) Spark Jobs

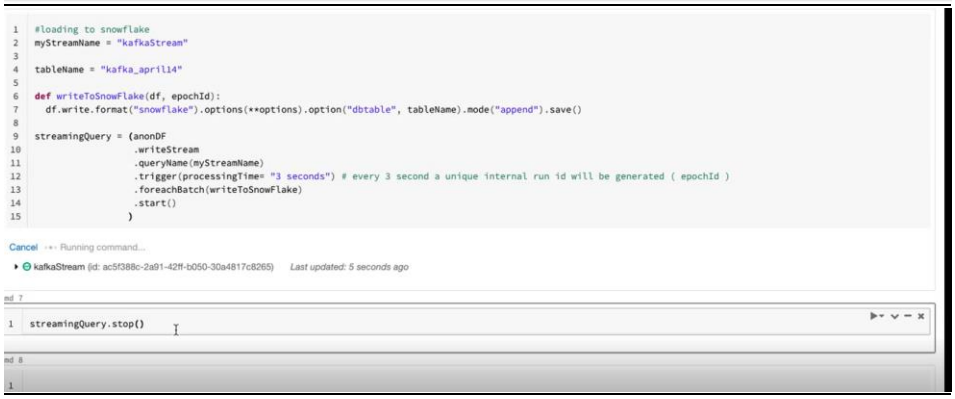
▸ myStream (id: 0a9c007b-bd0d-45cd-8b35-b142dfd766a2) *Last updated: 10 seconds ago*

	value
1	{"isRobot":false,"channel":"#en.wikipedia","timestamp":"2021-04-11T06:47:35.337Z","uri":"","isUnpatrolled":false,"page":"Special:Log/abusefilter","wikipedia":"en","wikipediaURL":"http://en.wikipedia.org","comment":"2409:4072:83:7C77:A3AD:3D64:81F9:BFE3 triggered [[Special:AbuseFilter/1113|filter 1113]], performing the action \"edit\" on [[\u000302List of Brahmins\u000310]]. Actions taken: none ([[Special:AbuseLog/29604609|details]])","userURL":"http://en.wikipedia.org/wiki/User:2409:4072:83:7C77:A3AD:3D64:81F9:BFE3","pageURL":"http://en.wikipedia.org/wiki/Special:Log/abusefilter","delta":null,"flag":"hit","isNewPage":false,"isAnonymous":true,"geocoding":{"countryCode2":null,"city":null,"latitude":null,"country":null,"longitude":null,"stateProvince":null,"countryCode3":null},"user":"2409:4072:83:7C77:A3AD:3D64:81F9:BFE3","namespace":"special"}
2	{"isRobot":false,"channel":"#en.wikipedia","timestamp":"2021-04-11T06:47:35.394Z","uri":"","isUnpatrolled":false,"page":"Special:Log/abusefilter","wikipedia":"en","wikipediaURL":"http://en.wikipedia.org","comment":"2409:4072:83:7C77:A3AD:3D64:81F9:BFE3 triggered [[Special:AbuseFilter/1116|filter 1116]], performing the action \"edit\" on [[\u000302List of Brahmins\u000310]]. Actions taken: none ([[Special:AbuseLog/29604610|details]])","userURL":"http://en.wikipedia.org/wiki/User:2409:4072:83:7C77:A3AD:3D64:81F9:BFE3","pageURL":"http://en.wikipedia.org/wiki/Special:Log/abusefilter","delta":null,"flag":"hit","isNewPage":false,"isAnonymous

Live streaming

```
myStreamName = "myStream"


mappedDF = (anonDF
  .groupBy("geocoding.countryCode3") # Aggregate by country (code)
  .count()                           # Produce a count of each aggregate
)

display(mappedDF, streamName = myStreamName)
```

Cancel



<u>Creating Delta Lake</u>

<u>SAVE THE DATA as delta file:</u>

<u>Create a deltla lake table</u>

<u>Rad for the table as as when needed.</u>

<u>Any data that come in to the file location needs to beappended so that thye awutomatically reflect in delta lake table</u>

Cmd 3

```
dbutils.fs.ls("/tmp/")
```

Cmd 4

```
dbutils.fs.rm("/tmp/", True)
```

Cmd 5

```
inputPath = "/mnt/training/online_retail/data-001/data.csv"

parquetDataPath  = "/tmp" + "/customer-data/"

deltaDataPath    = "/tmp" + "/customer-data-delta/"
```

Cmd 6

```
inputSchema = "InvoiceNo STRING, StockCode STRING, Description STRING, Quantity INT, InvoiceDate STRING, UnitPrice DOUBLE, CustomerID INT, Country STRING"

rawDF = (spark.read
  .option("header", "true")
  .schema(inputSchema)
  .csv(inputPath)
)
```

Cancel Running command...

```
# write using Databricks Delta format
(rawDF.write
  .mode("overwrite")
  .format("delta")
  .partitionBy("Country")
  .save(deltaDataPath) )
```

Cancel Running command...

▸ (1) Spark Jobs

```
(spark.read
  .option("header", "true")
  .schema(inputSchema)
  .csv(inputPath)
  .write
  .mode("overwrite")
  .format("delta")
  .partitionBy("Country")
  .save(deltaDataPath) )
```

▸ (4) Spark Jobs

Command took 14.76 seconds -- by deepak.rajak@exusia.com at 15/04/2021, 07:40:16 on My New Cluster

Cmd 28

```
%sql
drop table customer_delta_data
```

⊞Error in SQL statement: AnalysisException: Table or view not found: customer_delta_data;

Command took 7.33 seconds -- by deepak.rajak@exusia.com at 15/04/2021, 07:40:27 on My New Cluster

Cmd 29

```
spark.sql("""
    CREATE TABLE IF NOT EXISTS customer_delta_data
    USING delta
    OPTIONS (path = '{}')
  """.format(deltaDataPath))
```

SCD : Slowly Changing Dimensions

Cmd 7

Day1 - Load

```
(miniDataDF
  .write
  .mode("overwrite")
  .format("delta")
  .save(deltaMiniDataPath)
)

spark.sql("""
    CREATE TABLE IF NOT EXISTS customer_data_delta_mini
    USING DELTA
    LOCATION '{}'
  """.format(deltaMiniDataPath))
```

▸ (4) Spark Jobs

Out[3]: DataFrame[]

Command took 11.93 seconds -- by deepak.rajak@exusia.com at 18/03/2021, 07:56:30 on Exusia

```
sqlCmd = "SELECT * FROM customer_data_delta_mini WHERE CustomerID=20993"

display(spark.sql(sqlCmd))
```

▸ (1) Spark Jobs

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536371	32129	EverGlow Single	228	1/1/18 9:01	33.85	20993	Sierra Leone

Showing all 1 rows.

Cmd 11

```
%sql drop table customer_data_delta_to_upsert
```

⊞Error in SQL statement: AnalysisException: Table or view not found: customer_data_delta_to_upsert;

Command took 3.60 seconds -- by deepak.rajak@exusia.com at 15/04/2021, 07:54:15 on My New Cluster

Cmd 12

```
from pyspark.sql.functions import lit, col

customerSpecificDF = (miniDataDF
  .filter("CustomerID=20993")
  .withColumn("StockCode", lit(99999))
 )

spark.sql("DROP TABLE IF EXISTS customer_data_delta_to_upsert")

customerSpecificDF.write.saveAsTable("customer_data_delta_to_upsert")
```

Cmd 15

```
%sql MERGE INTO customer_data_delta_mini USING customer_data_delta_to_upsert ON customer_data_delta_mini.CustomerID = customer_data_delta_to_upsert.Customer
WHEN MATCHED THEN
UPDATE
SET
  customer_data_delta_mini.StockCode = customer_data_delta_to_upsert.StockCode
  WHEN NOT MATCHED THEN
INSERT
  (
    InvoiceNo,
    StockCode,
    Description,
    Quantity,
    InvoiceDate,
    UnitPrice,
    CustomerID,
    Country
  )
VALUES
  (
    customer_data_delta_to_upsert.InvoiceNo,
    customer_data_delta_to_upsert.StockCode,
    customer_data_delta_to_upsert.Description,
    customer_data_delta_to_upsert.Quantity,
    customer_data_delta_to_upsert.InvoiceDate,
    customer_data_delta_to_upsert.UnitPrice,
    customer_data_delta_to_upsert.CustomerID,
    customer_data_delta_to_upsert.Country
  )
```

```
# MERGE Operation
spark.sql(""" MERGE INTO del_delay as d
USING merge_table as m
on d.origin = m.origin
WHEN MATCHED THEN
  UPDATE SET *
WHEN NOT MATCHED
  THEN INSERT * """)
```

Cmd 18

Using Python code to merge:

Upsert (merge) in Delta Lake

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

# Create the 5 records ( We will be updating the column - name)
newIncrementalData = spark.range(5).withColumn("name", lit("Neha"))

# create DeltaTable instances using the path of the Delta table
deltaTable = DeltaTable.forPath(spark, "/tmp/ids")

# Execute Update + Insert ( Upsert = merge )
(deltaTable
 .alias("oldData")
 .merge(newIncrementalData.alias("newData"), "oldData.id = newData.id")
 .whenMatchedUpdate(set = { "name": col("newData.name") })
 .whenNotMatchedInsert(values = { "id": col("newData.id") , "name": col("newData.name")  })
 .execute()
)

# Display the records to check if the records are Merged
```

```
sqlCmd = "SELECT * FROM customer_data_delta_mini WHERE CustomerID=20993"
display(spark.sql(sqlCmd))
```

▸ (1) Spark Jobs

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536371	99999	EverGlow Single	228	1/1/18 9:01	33.85	20993	Sierra Leone

Showing all 1 rows.

Command took 3.15 seconds -- by deepak.rajak@exusia.com at 15/04/2021, 08:03:09 on My New Cluster

Merging scehama:

Day 1 with 5 columns

```
df.write.format("delta").save(delta_path)
```

▸ (5) Spark Jobs
▸ df: pyspark.sql.dataframe.DataFrame = [date: string, delay: string ... 3 more fields]

Command took 47.29 seconds -- by deepak.rajak@exusia.com at 16/04/2021, 07:12:13 on My New Cluster

Cmd 5

```
# Day1
delta_path = "/tmp/departure"

data_file = "/databricks-datasets/flights/departuredelays.csv"

spark.read.format("delta").load(delta_path).show(1) #We already have DELTA LAKE file

```

▸ (1) Spark Jobs

```
+--------+-----+--------+------+-----------+
|    date|delay|distance|origin|destination|
+--------+-----+--------+------+-----------+
|01011245|    6|     602|   ABE|        ATL|
+--------+-----+--------+------+-----------+
only showing top 1 row
```

Command took 6.34 seconds -- by deepak.rajak@exusia.com at 16/04/2021, 07:12:35 on My New Cluster

Cmd 6

```
%sql
select count(*) from delta.`/tmp/departure`
```

Cancel Command submitted to cluster for execution

Day2 if the data comes in with adiitional comumn :

```
# Day 2

from pyspark.sql.functions import lit

newDF = spark.read.format("delta").load(delta_path).limit(5).withColumn("country", lit("USA"))

display(newDF)
```

(1) Spark Jobs

newDF: pyspark.sql.dataframe.DataFrame

```
date: string
delay: string
distance: string
origin: string
destination: string
country: string
```

	date	delay	distance	origin	destination	country
1	01011245	6	602	ABE	ATL	USA
2	01020600	-8	369	ABE	DTW	USA
3	01021245	-2	602	ABE	ATL	USA
4	01020605	-4	602	ABE	ATL	USA
5	01031245	-4	602	ABE	ATL	USA

Showing all 5 rows.

<u>Error occurred as data mismatched:</u>

Cmd 8

```
# Append
newDF.write.format("delta").mode("append").save(delta_path)
```

AnalysisException: A schema mismatch detected when writing to the Delta table (Table ID: 6ad87511-49d9-4dac-9394-e7f5a5ad7d8f).

```
      1 # Append
----> 2 newDF.write.format("delta").mode("append").save(delta_path)

/databricks/spark/python/pyspark/sql/readwriter.py in save(self, path, format, mode, partitionBy, **options)
    828             self._jwrite.save()
    829         else:
--> 830             self._jwrite.save(path)
    831
    832     @since(1.4)

/databricks/spark/python/lib/py4j-0.10.9-src.zip/py4j/java_gateway.py in __call__(self, *args)
   1303         answer = self.gateway_client.send_command(command)
   1304         return_value = get_return_value(
-> 1305             answer, self.gateway_client, self.target_id, self.name)
   1306
   1307         for temp_arg in temp_args:

/databricks/spark/python/pyspark/sql/utils.py in deco(*a, **kw)
    131                 # Hide where the exception came from that shows a non-Pythonic
    132                 # JVM exception message.
--> 133                 raise_from(converted)
```

<u>To avoid this use option("mergeSchema","True")</u>

Cmd 9

```
newDF.write.format("delta").option("mergeSchema", True).mode("append").save(delta_path)
```

(4) Spark Jobs

Command took 7.11 seconds -- by deepak.rajak@exusia.com at 16/04/2021, 07:21:13 on My New Cluster

<u>History table-Every delta table has an history table associated.</u>

Command took 4.23 seconds -- by deepak.rajak@exusia.com at 16/04/2021, 07:24:04 on My New Cluster

Cmd 16

```
%sql
describe history delta.`/tmp/departure`
```

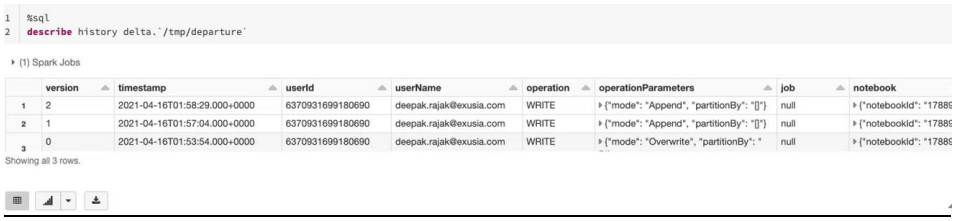
```
%sql
describe history delta.`/tmp/departure`
```

(1) Spark Jobs

	version	timestamp	userId	userName	operation	operationParameters	job	notebook
1	0	2021-04-16T01:53:54.000+0000	6370931699180690	deepak.rajak@exusia.com	WRITE	{"mode": "Overwrite", "partitionBy": "	null	{"notebookId": "17889

Showing all 1 rows.

<u>As the transaction increases , the version increases:</u>



```
%sql
describe history delta.`/tmp/departure`
```

(1) Spark Jobs

	version	timestamp	userId	userName	operation	operationParameters	job	notebook
1	2	2021-04-16T01:58:29.000+0000	6370931699180690	deepak.rajak@exusia.com	WRITE	{"mode": "Append", "partitionBy": "[]"}	null	{"notebookId": "17889
2	1	2021-04-16T01:57:04.000+0000	6370931699180690	deepak.rajak@exusia.com	WRITE	{"mode": "Append", "partitionBy": "[]"}	null	{"notebookId": "17889
3	0	2021-04-16T01:53:54.000+0000	6370931699180690	deepak.rajak@exusia.com	WRITE	{"mode": "Overwrite", "partitionBy": "	null	{"notebookId": "17889

Showing all 3 rows.

<u>Get the data of required version using option("versionAsof:<versionnumber>) this includes the data as of that version</u>

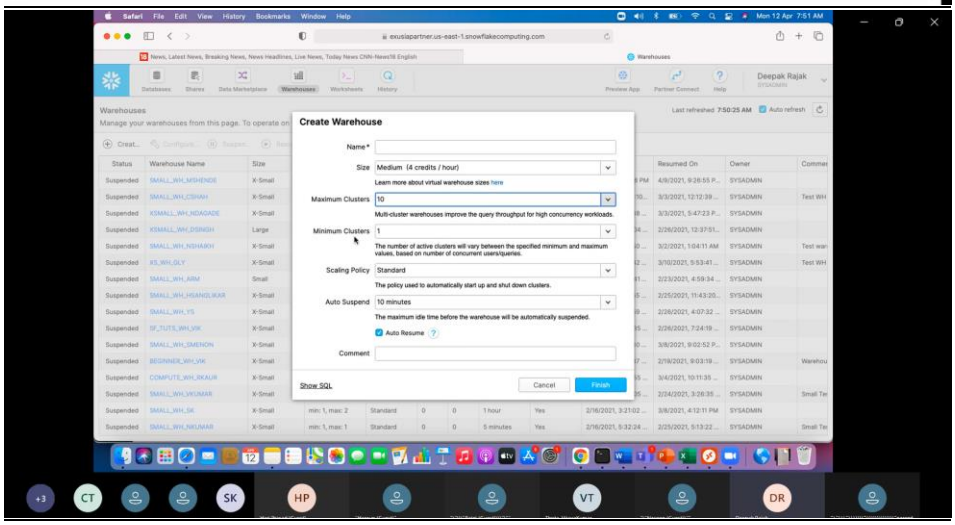
<u>If we are using parquet file instead of Delta lake ,</u>

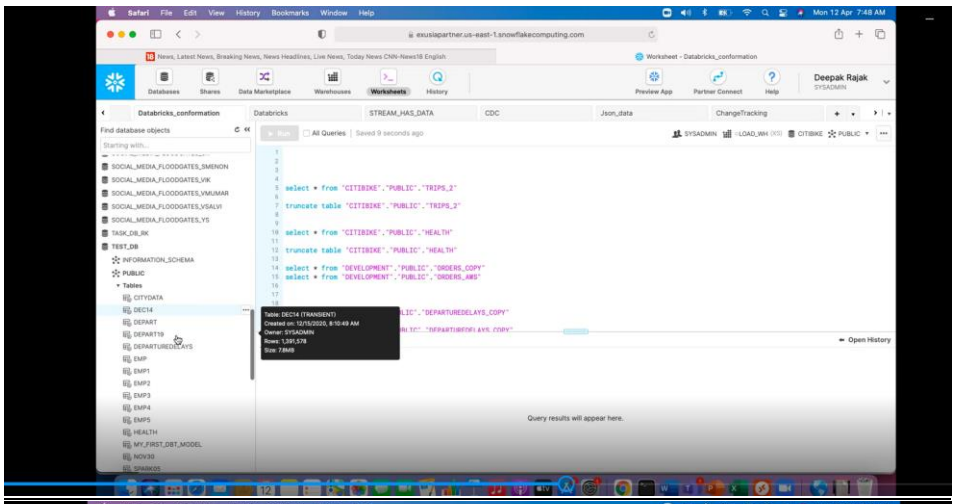
<u>Write data to Snow flake:</u>

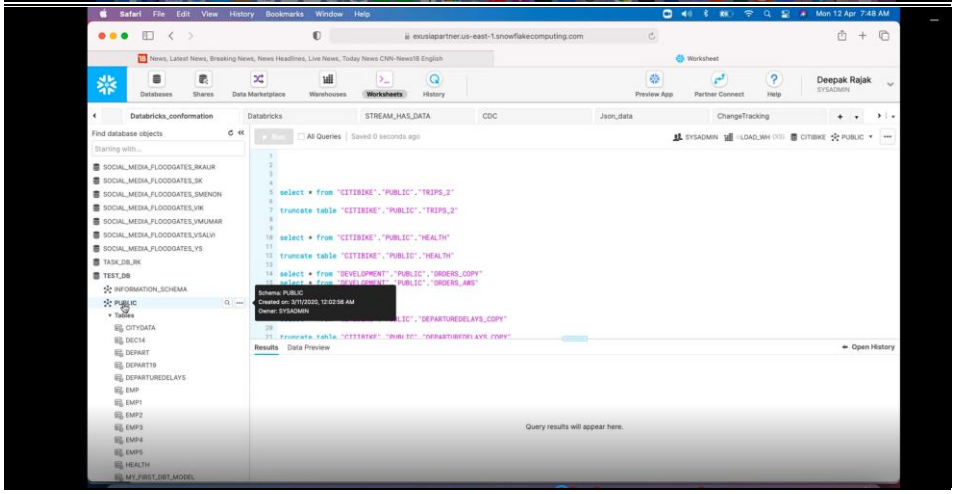
```
# snowflake connection options
options = dict(sfUrl="https://exusiapartner.us-east-1.snowflakecomputing.com",
               sfUser=user,
               sfPassword=password,
               sfDatabase="TEST_DB",
               sfSchema="PUBLIC",
               sfWarehouse="LOAD_WH")
```

Cmd 12

```
loadType = getArgument("LoadType")
# write the dataset to Snowflake table.
if loadType == "Snowflake":
            tableName = getArgument("TargetTable")
            df.write.format("snowflake").options(**options).option("dbtable", tableName).mode("overwrite").save()
            print(tableName, "table loaded successfully in Snowflake!!!")

elif loadType == "File":
                targetPath = getArgument("TargetPath") + "/out_" + getArgument("InputFile")
                 # write the dataset to DBFS .
                df.write.mode("overwrite").parquet(targetPath)
                print("file loaded successfully in DBFS!!!")
else:
  print("load not supported")
```







columns = train_data.columns

stages=[]

for i in train_data.dtypes:

** if(i[1]=='string'):**

**   columns.remove(i[0])**

**   stages.append(StringIndexer(inputCol=i[0], outputCol=i[0]+"_IX"))**

Auto load:

==========

```
checkpoint_path = '/tmp/delta/population_data/_checkpoints'
write_path = '/tmp/delta/population_data'

# Set up the stream to begin reading incoming files from the
# upload_path location.
df = spark.readStream.format('cloudFiles') \
  .option('cloudFiles.format', 'csv') \
  .option('header', 'true') \
  .schema('city string, year int, population long') \
  .load(upload_path)

# Start the stream.
# Use the checkpoint_path location to keep a record of all files that
# have already been uploaded to the upload_path location.
# For those that have been uploaded since the last check,
# write the newly-uploaded files' data to the write_path location.
df.writeStream.format('delta') \
  .option('checkpointLocation', checkpoint_path) \
  .start(write_path)
```

s

SnowFlake CONNE TION:

```
options=dict(sfurl="https://sg39612.us-east-2.aws.snowflakecomputing.com",
        sfUser="kvr",
        sfPassword="Sravanthi@1",
        sfDatabase="TEST",
        sfSchema="TEST" )
```

Writing to Table:

```
df.write.format("snowflake").options(**options).option("dbtable","sparktable").mode("overwrite").save()
```

Reading from Table

```
df1=spark.read.format("snowflake").options(**options).option("dbtable","sparktable").load()
```