Activity – 03

Ganesh Gundekarla
11700551

Task 01

1. Directed Acyclic graph.

DAG creates the logical plan in spark application on how the process has to be done .

So , in the topic of Big data analytics , big data processing frameworks exist like "pyspark" which is a directed acyclic graph , also termed as "dag", this commonly means the execution plan or (logical flow) of the transformations done on our data .

a. Directed means that it has a direction which is being associated with the edges present to the nodes in the graph. This represents the flow of transformations happened across one node to the other.
b. Acyclic: Here , this means that there is no cycles present in the graph , this is useful for ensuring that this is FINITE. It has no going from start node to the same starting node again. So this is made to ensure that this doesn't follow a infinite loop.
c. Graph: representation of the nodes which are being collected and connected by the edges or say "arrows".
d. Node = transformation , edge = flow of the data or arrow which is connections happening from one node to other.

2. Lazy evaluation :

Here ,the spark process does not evaluate any of the node or transformation with an immediate effect , instead , it make sures that all the transformations are being evaluated lazily . After this , the transformations are then added to the computation results. This is done to make the spark take some optimization decisions , and moreover these all transformations are then being visible to the engine of sprark before there is a need to perform any action specific . So, instead of immediately doing all the execution of the transformations , the lazy evaluations holds the execution of the operations until an actions is being triggered thus the name .

- Benefits : optimization can be achieved
- Overhead is reduced : which in turn boosts performance

- Fault tolerance : here, it can only check and rerun the failure part more easily ,computing the required parts only.

3. Fault tolerance :

The main aim of fault tolerance is to recover from the failures or the errors in the time of the execution of distributed computations . Here , the pyspark provides fault tolerance to make sure that the data processing jobs can be then continue to run smoothly even if there are any node error , failure and network issues .

Helps in achieving :

a. Reliability
b. Availability
c. Resilience in the big data work flows .
d. Job restartability

Key aspects in fault tolerance :

a. Resilient Distributed datasets are fault toleranct collections that can be made to operate in parallel to recompute the corrupt parts of the data in failure
b. Checkpointing in fault tolerance is being done periodically to save the rdd data to a more stable storage to minimize. The computation time to recover , which is helping us in limiting the depth of the rdd lineage , reducing the amount of data that is needed.
c. Task retry : done to handle transient failures during execution , failures happened due to network issues , or any errors which is then shifted to a different executor node.

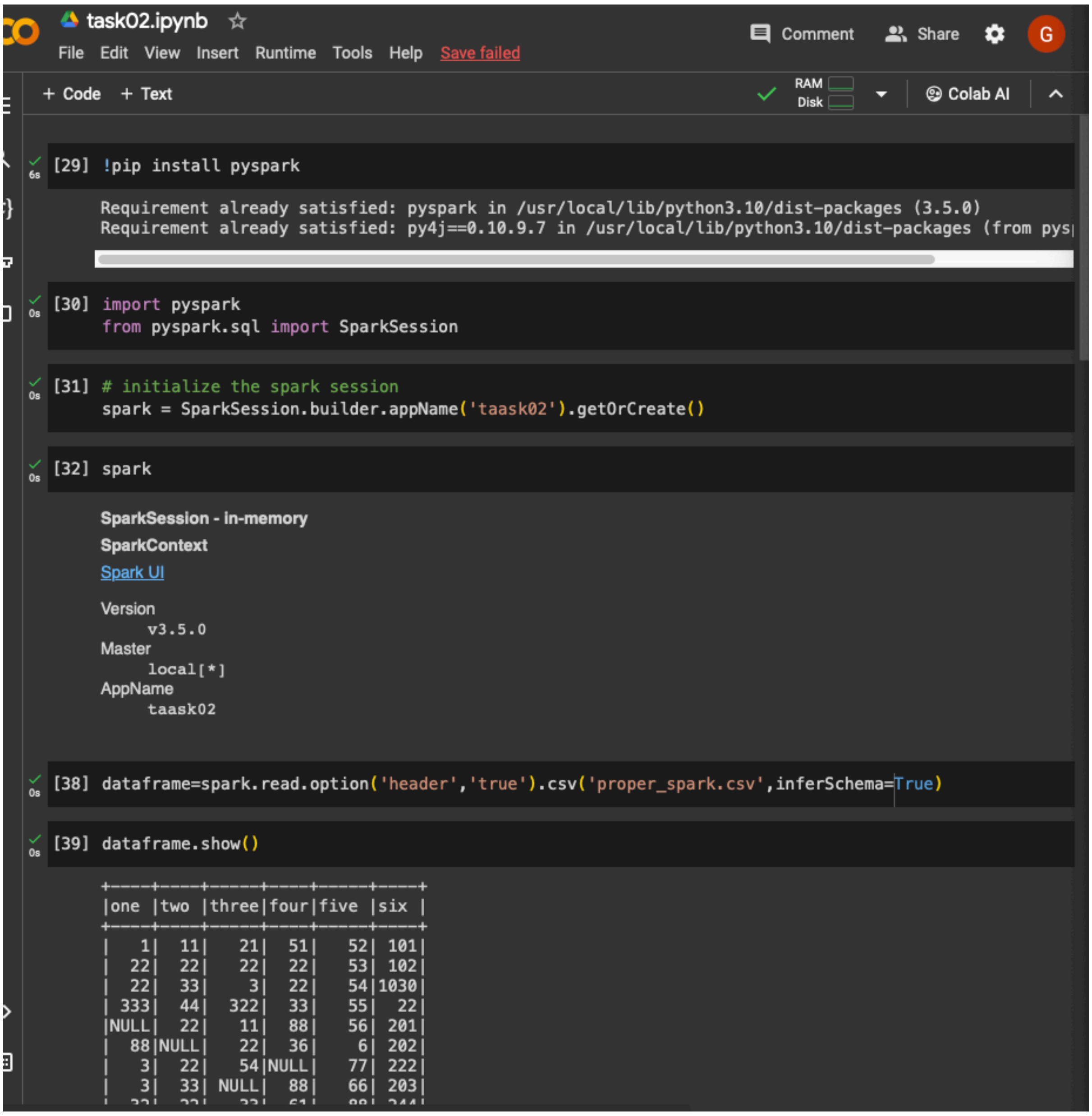
4. Resilient Distributed Datasets : this is data structure in the pyspark . These are mostly immutable and a series of distributed collections of elements that can be made to operate in parallel

Characteristics of rdd's:

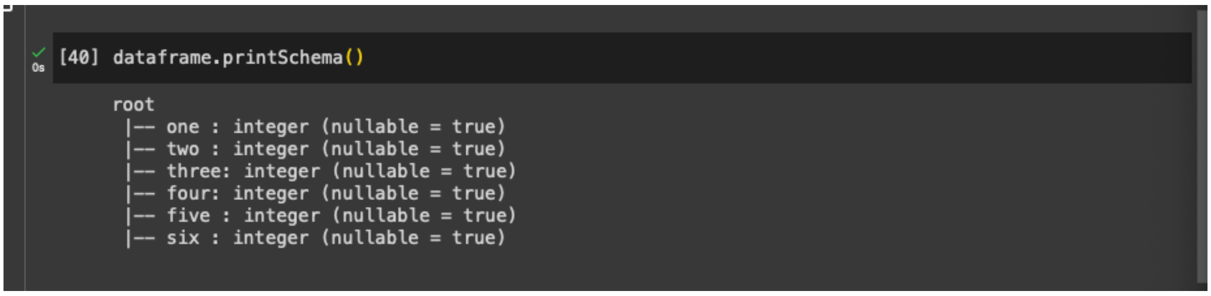
a. Resilient because they are fault tolerant
b. Distributed : because they are spanned in multiple nodes in a cluster.
c. Immutable : cant be modified
d. Lazy evaluation is being done .

Task 02:

- Installed the pyspark
- Imported required libs
- Initialized the session with a app name of “taask02”
- Took the schema to ensure the required



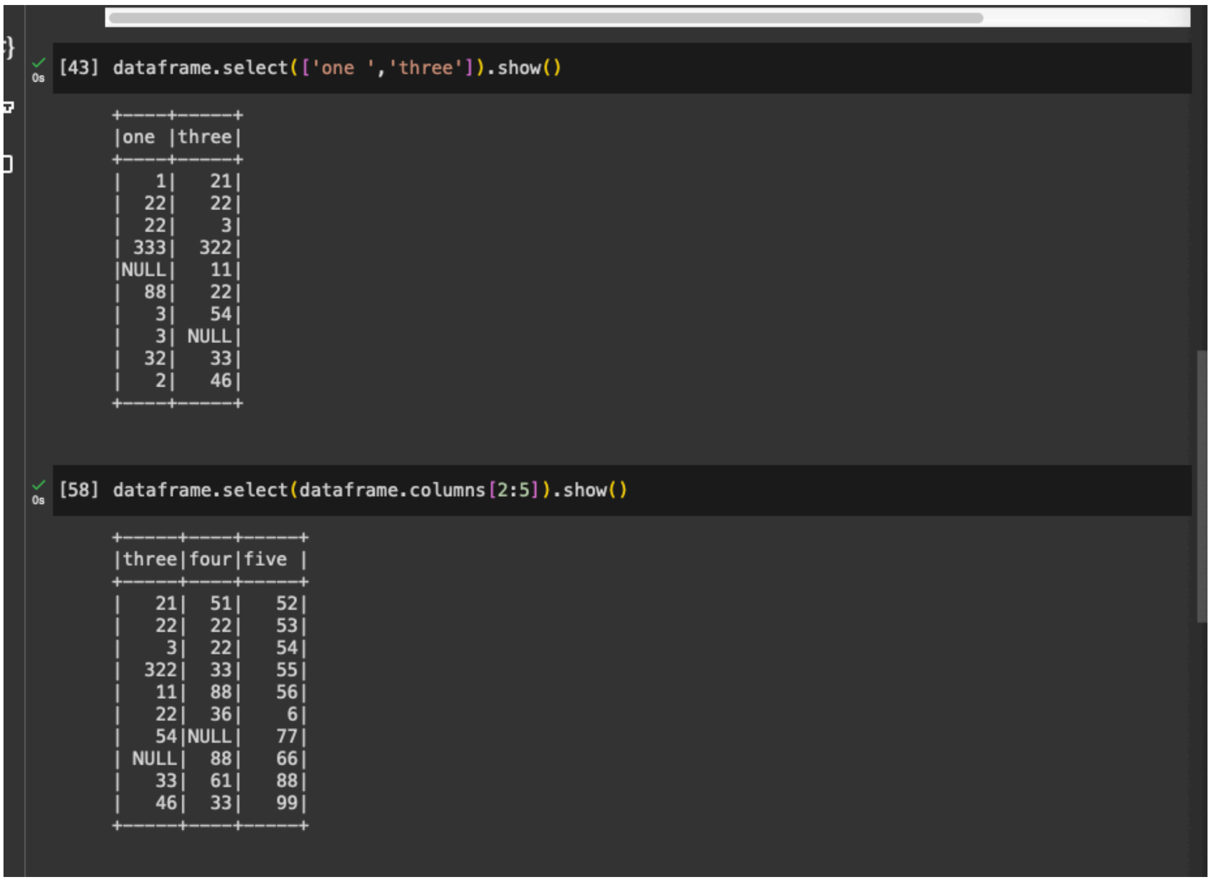
```
[29] !pip install pyspark

Requirement already satisfied: pyspark in /usr/local/lib/python3.10/dist-packages (3.5.0)
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.10/dist-packages (from pys

[30] import pyspark
     from pyspark.sql import SparkSession

[31] # initialize the spark session
     spark = SparkSession.builder.appName('taask02').getOrCreate()

[32] spark

SparkSession - in-memory
SparkContext
Spark UI

Version
    v3.5.0
Master
    local[*]
AppName
    taask02

[38] dataframe=spark.read.option('header','true').csv('proper_spark.csv',inferSchema=True)

[39] dataframe.show()

+----+----+-----+----+-----+----+
|one |two |three|four|five |six |
+----+----+-----+----+-----+----+
|   1|  11|   21|  51|   52| 101|
|  22|  22|   22|  22|   53| 102|
|  22|  33|    3|  22|   54|1030|
| 333|  44|  322|  33|   55|  22|
|NULL|  22|   11|  88|   56| 201|
|  88|NULL|   22|  36|    6| 202|
|   3|  22|   54|NULL|   77| 222|
|   3|  33| NULL|  88|   66| 203|
```



```
[40] dataframe.printSchema()

root
 |-- one : integer (nullable = true)
 |-- two : integer (nullable = true)
 |-- three: integer (nullable = true)
 |-- four: integer (nullable = true)
 |-- five : integer (nullable = true)
 |-- six : integer (nullable = true)
```

1. Selecting the columns which are ranged from the index 2-5

So , where in pyspark the select method is used to select the columns based on column name or using index of the columns .Show method is used to print the table .



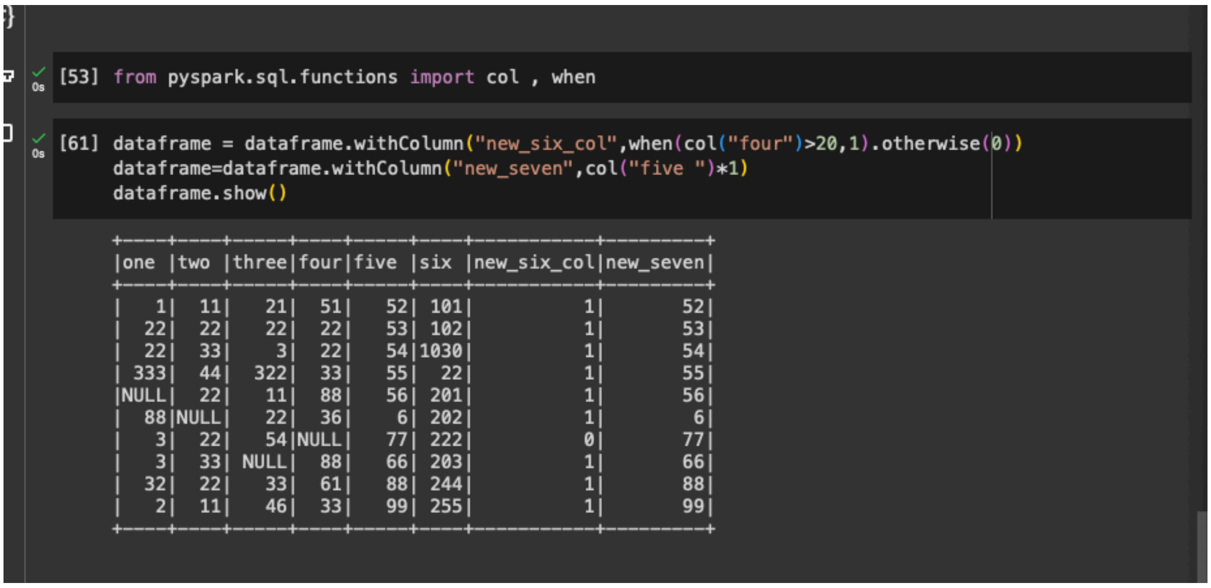
```
[43] dataframe.select(['one ','three']).show()

+----+-----+
|one |three|
+----+-----+
|   1|   21|
|  22|   22|
|  22|    3|
| 333|  322|
|NULL|   11|
|  88|   22|
|   3|   54|
|   3| NULL|
|  32|   33|
|   2|   46|
+----+-----+
```

```
[58] dataframe.select(dataframe.columns[2:5]).show()

+-----+----+-----+
|three|four|five |
+-----+----+-----+
|   21|  51|   52|
|   22|  22|   53|
|    3|  22|   54|
|  322|  33|   55|
|   11|  88|   56|
|   22|  36|    6|
|   54|NULL|   77|
| NULL|  88|   66|
|   33|  61|   88|
|   46|  33|   99|
+-----+----+-----+
```

2. Adding the two columns with necessary amount of data.

New libraries like col and when are needed from the pyspark library to select the columns. Here the withcolumn is used to insert a new column with the existing table with a new column with the condition in the bracket .

Dataframe = dataset.withcolumn(“column_name”,condtion)

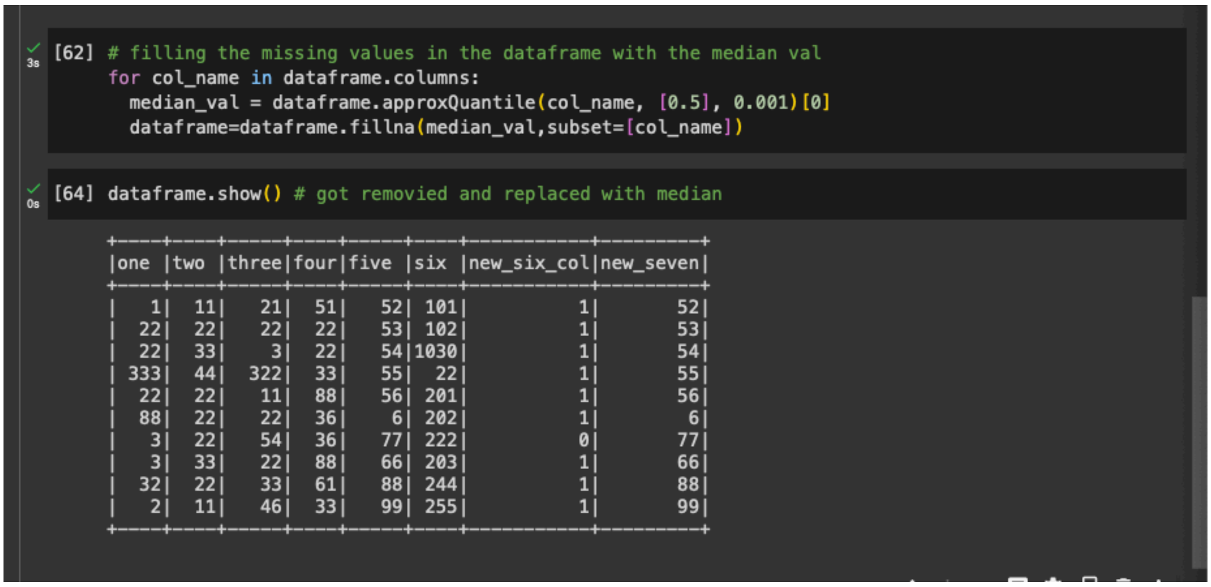


```
[53] from pyspark.sql.functions import col , when
```

```
[61] dataframe = dataframe.withColumn("new_six_col",when(col("four")>20,1).otherwise(0))
     dataframe=dataframe.withColumn("new_seven",col("five ")*1)
     dataframe.show()
```

```
+----+----+-----+----+-----+----+-----------+---------+
|one |two |three|four|five |six |new_six_col|new_seven|
+----+----+-----+----+-----+----+-----------+---------+
|   1|  11|   21|  51|   52| 101|          1|       52|
|  22|  22|   22|  22|   53| 102|          1|       53|
|  22|  33|    3|  22|   54|1030|          1|       54|
| 333|  44|  322|  33|   55|  22|          1|       55|
|NULL|  22|   11|  88|   56| 201|          1|       56|
|  88|NULL|   22|  36|    6| 202|          1|        6|
|   3|  22|   54|NULL|   77| 222|          0|       77|
|   3|  33| NULL|  88|   66| 203|          1|       66|
|  32|  22|   33|  61|   88| 244|          1|       88|
|   2|  11|   46|  33|   99| 255|          1|       99|
+----+----+-----+----+-----+----+-----------+---------+
```

3. Filling the missing values with the median values of the each columns . Here , some columns contains null vals , so the median value has to be inserted here . To do this , we run the for loop for each columns and find median of the respective columns and then we insert the median value in the null val.

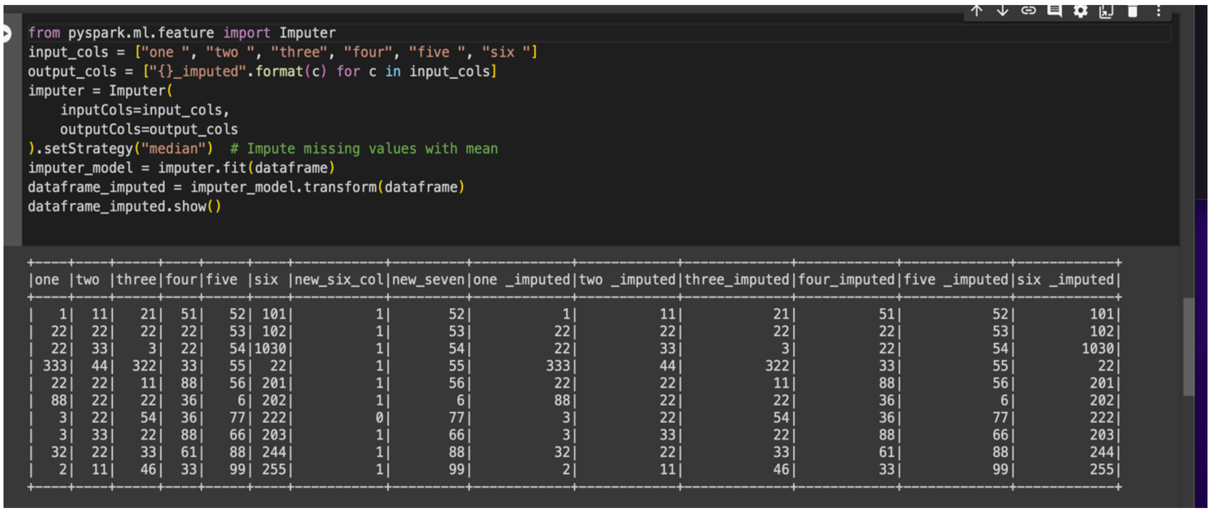


```
[62] # filling the missing values in the dataframe with the median val
     for col_name in dataframe.columns:
       median_val = dataframe.approxQuantile(col_name, [0.5], 0.001)[0]
       dataframe=dataframe.fillna(median_val,subset=[col_name])
```

```
[64] dataframe.show() # got removied and replaced with median
```

```
+----+----+-----+----+-----+----+-----------+---------+
|one |two |three|four|five |six |new_six_col|new_seven|
+----+----+-----+----+-----+----+-----------+---------+
|   1|  11|   21|  51|   52| 101|          1|       52|
|  22|  22|   22|  22|   53| 102|          1|       53|
|  22|  33|    3|  22|   54|1030|          1|       54|
| 333|  44|  322|  33|   55|  22|          1|       55|
|  22|  22|   11|  88|   56| 201|          1|       56|
|  88|  22|   22|  36|    6| 202|          1|        6|
|   3|  22|   54|  36|   77| 222|          0|       77|
|   3|  33|   22|  88|   66| 203|          1|       66|
|  32|  22|   33|  61|   88| 244|          1|       88|
|   2|  11|   46|  33|   99| 255|          1|       99|
+----+----+-----+----+-----+----+-----------+---------+
```

The same process is being done using transformer which is being imported from pyspark.ml.feature module in pyspark , here the imputer is created to handle the missing values in a specified columns of the table or dataframe by imputing them with a required strategy and in this case , the strategy used is “median” and it displays the imputed values of each columns .



```python
from pyspark.ml.feature import Imputer
input_cols = ["one ", "two ", "three", "four", "five ", "six "]
output_cols = ["{}_imputed".format(c) for c in input_cols]
imputer = Imputer(
    inputCols=input_cols,
    outputCols=output_cols
).setStrategy("median")  # Impute missing values with mean
imputer_model = imputer.fit(dataframe)
dataframe_imputed = imputer_model.transform(dataframe)
dataframe_imputed.show()
```

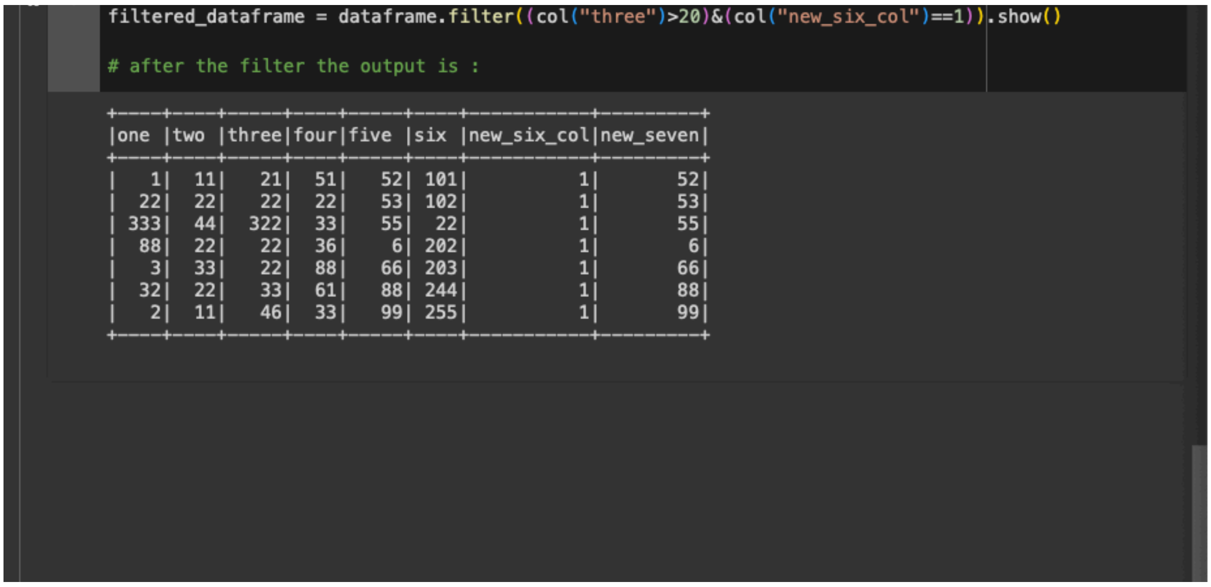
```
+----+----+-----+----+-----+----+-----------+---------+------------+------------+-------------+------------+-------------+------------+
|one |two |three|four|five |six |new_six_col|new_seven|one _imputed|two _imputed|three_imputed|four_imputed|five _imputed|six _imputed|
+----+----+-----+----+-----+----+-----------+---------+------------+------------+-------------+------------+-------------+------------+
|   1|  11|   21|  51|   52| 101|          1|       52|           1|          11|           21|          51|           52|         101|
|  22|  22|   22|  22|   53| 102|          1|       53|          22|          22|           22|          22|           53|         102|
|  22|  33|    3|  22|   54|1030|          1|       54|          22|          33|            3|          22|           54|        1030|
| 333|  44|  322|  33|   55|  22|          1|       55|         333|          44|          322|          33|           55|          22|
|  22|  22|   11|  88|   56| 201|          1|       56|          22|          22|           11|          88|           56|         201|
|  88|  22|   22|  36|    6| 202|          1|        6|          88|          22|           22|          36|            6|         202|
|   3|  22|   54|  36|   77| 222|          0|       77|           3|          22|           54|          36|           77|         222|
|   3|  33|   22|  88|   66| 203|          1|       66|           3|          33|           22|          88|           66|         203|
|  32|  22|   33|  61|   88| 244|          1|       88|          32|          22|           33|          61|           88|         244|
|   2|  11|   46|  33|   99| 255|          1|       99|           2|          11|           46|          33|           99|         255|
+----+----+-----+----+-----+----+-----------+---------+------------+------------+-------------+------------+-------------+------------+
```

4. Applying two filter conditions .

Here we applied two filter conditions in one statement using a and operator which checks that both the conditions are fulfilled.

We use the filter method and condition

Dataset = dataset.filter(condition & , || ,++ condition 2)



```python
filtered_dataframe = dataframe.filter((col("three")>20)&(col("new_six_col")==1)).show()

# after the filter the output is :
```

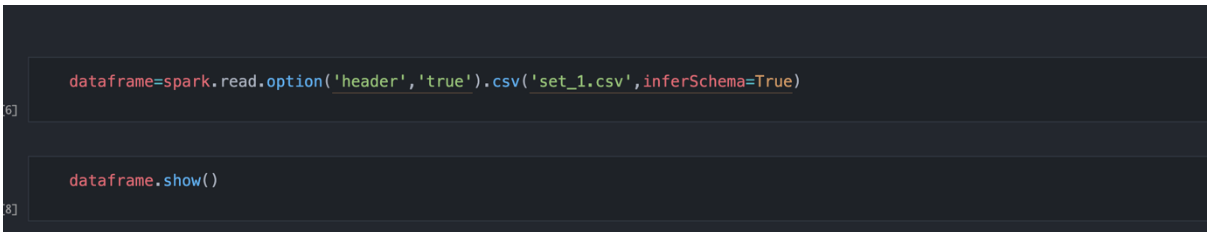
```
+----+----+-----+----+-----+----+-----------+---------+
|one |two |three|four|five |six |new_six_col|new_seven|
+----+----+-----+----+-----+----+-----------+---------+
|   1|  11|   21|  51|   52| 101|          1|       52|
|  22|  22|   22|  22|   53| 102|          1|       53|
| 333|  44|  322|  33|   55|  22|          1|       55|
|  88|  22|   22|  36|    6| 202|          1|        6|
|   3|  33|   22|  88|   66| 203|          1|       66|
|  32|  22|   33|  61|   88| 244|          1|       88|
|   2|  11|   46|  33|   99| 255|          1|       99|
+----+----+-----+----+-----+----+-----------+---------+
```

Task 03

Transformations performed and take on activity;

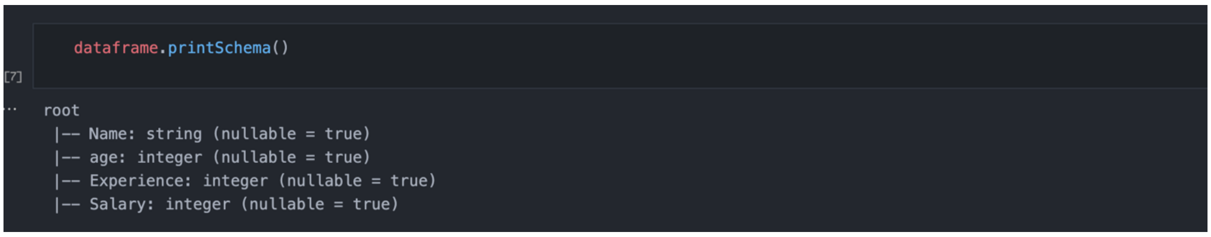
There are multiple transformations that are being performed ,

a. Reading the csv file: here the session is initiated by initiating the spark session after installing spark and then importing several libraries. Here the code starts and reads two csv files into two sets of spark data frames
   .



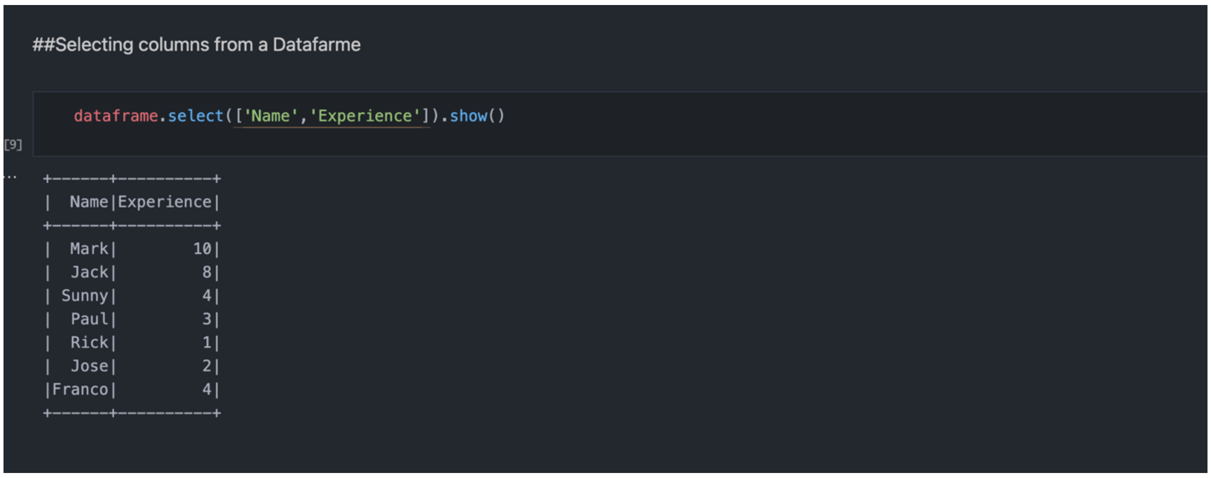
```
[6]
dataframe=spark.read.option('header','true').csv('set_1.csv',inferSchema=True)

[8]
dataframe.show()
```

b. Then the data is being displayed by using the show () method to have a visual representation of the data.
c. Schema inspection is being done by printSchema() a method that is being used to print the schema of the data. It gives the structure like whether the column is a string or an integer and is it nullable or not .



```
[7]
dataframe.printSchema()

root
 |-- Name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- Experience: integer (nullable = true)
 |-- Salary: integer (nullable = true)
```

d. The method 'select() ' method is then used to select a particular column to manipulate it with , on the data frame which is used to gain some valuable insights about a particular columns .This actively selects the cols which is beneficial .



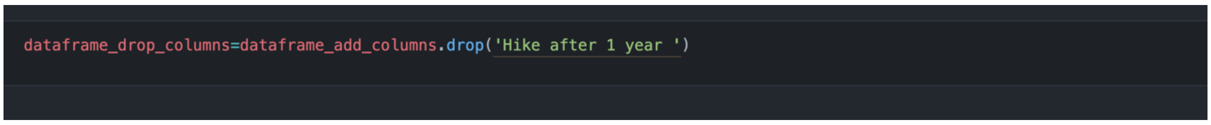
##Selecting columns from a Datafarme

```
[9]
dataframe.select(['Name','Experience']).show()

+------+----------+
|  Name|Experience|
+------+----------+
|  Mark|        10|
|  Jack|         8|
| Sunny|         4|
|  Paul|         3|
|  Rick|         1|
|  Jose|         2|
|Franco|         4|
+------+----------+
```

e. The descriptive stats (describe()) method is used to compute some stats about the existing data in the columns of the dataframe .This is very information providing one.

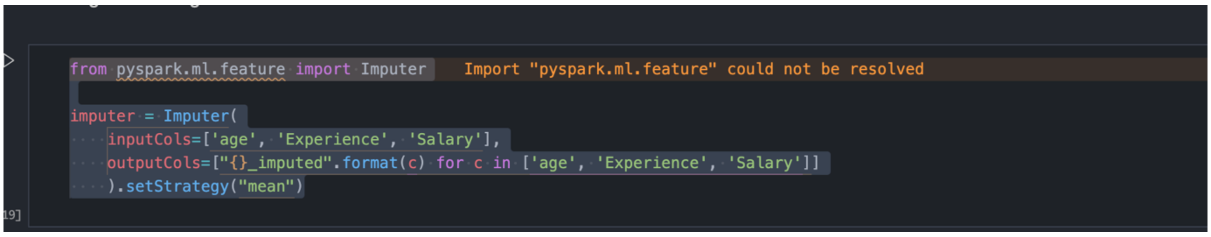
f. Adding new columns : Here , we are inserting a new columns is added to the dataframe done using the 'withcolumn()' method , which basically combines table with a new column , this is a good step as the withcolumn combinely defines the new col name and then the condition to insert the values .

g. Dropping the columns : here the method drop method is then used to remove the any desired columns and this resolves the steps in a easy way .



```
dataframe_drop_columns=dataframe_add_columns.drop('Hike after 1 year ')
```

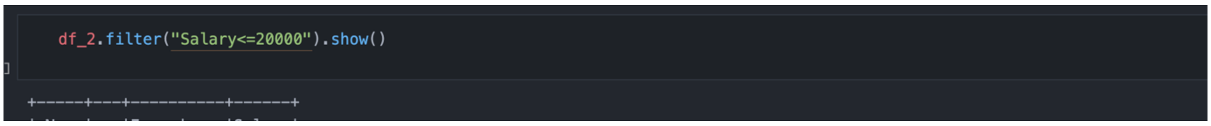
h. This is the most important step in big data
Handling Missing values : here , pysparks ml feature is used to remove the missing values in numeric columns and then imputes the selected strategy as mean().



```
from pyspark.ml.feature import Imputer    Import "pyspark.ml.feature" could not be resolved

imputer = Imputer(
    inputCols=['age', 'Experience', 'Salary'],
    outputCols=["{}_imputed".format(c) for c in ['age', 'Experience', 'Salary']]
    ).setStrategy("mean")
```

i. Handling missing values by inserting the selected vals is an important step .

j. Filtering data is a useful step in order to remove unwanted unnecessary data.



```
df_2.filter("Salary<=20000").show()

+------+---+----------+------+
```