databricks ShreyanthDE

(https://databricks.com)

TASK-1: RDD Usage

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("rdd task-1") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()


# File location and type
file_location = "/FileStore/tables/yahoo_stocks.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be
ignored.
yahoo_rdd = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location).rdd


#To check the type of the dataframe
type(yahoo_rdd)

Out[281]: pyspark.rdd.RDD


#return the length of the RDD and first 2 values of the RDD to the driver
print('RDD Count:', yahoo_rdd.count())

RDD Count: 1193


#Lists the contents of the RDD
print(yahoo_rdd.collect())
```

```
[Row(Date=datetime.datetime(2001, 1, 2, 0, 0), Open=30.3125, High=30.375, Lo
w=27.5, Close=28.1875, Volume=21939200, Adj Close=14.09375), Row(Date=dateti
me.datetime(2000, 12, 29, 0, 0), Open=30.3125, High=31.1875, Low=29.5625, Cl
ose=30.0625, Volume=20893400, Adj Close=15.03125), Row(Date=datetime.datetim
e(2000, 12, 28, 0, 0), Open=29.4375, High=31.75, Low=29.125, Close=31.0, Vol
ume=24374600, Adj Close=15.5), Row(Date=datetime.datetime(2000, 12, 27, 0,
0), Open=31.0, High=31.5, Low=29.125, Close=29.75, Volume=22045400, Adj Clos
e=14.875), Row(Date=datetime.datetime(2000, 12, 26, 0, 0), Open=32.0, High=3
4.0, Low=30.125, Close=31.1875, Volume=37536200, Adj Close=15.59375), Row(Da
te=datetime.datetime(2000, 12, 22, 0, 0), Open=26.4375, High=29.875, Low=26.
0625, Close=29.5625, Volume=28347400, Adj Close=14.78125), Row(Date=datetim
e.datetime(2000, 12, 21, 0, 0), Open=26.75, High=28.25, Low=25.0625, Close=2
5.625, Volume=27794400, Adj Close=12.8125), Row(Date=datetime.datetime(2000,
12, 20, 0, 0), Open=25.8125, High=28.375, Low=25.5, Close=27.9375, Volume=44
862800, Adj Close=13.96875), Row(Date=datetime.datetime(2000, 12, 19, 0, 0),
Open=30.5625, High=31.9687, Low=28.0, Close=28.0, Volume=36131600, Adj Close
=14.0), Row(Date=datetime.datetime(2000, 12, 18, 0, 0), Open=33.875, High=3
4.0, Low=30.25, Close=32.0, Volume=31697600, Adj Close=16.0), Row(Date=datet
ime.datetime(2000, 12, 15, 0, 0), Open=32.0, High=34.0, Low=31.0625, Close=3
3.0, Volume=40448000, Adj Close=16.5), Row(Date=datetime.datetime(2000, 12,
14, 0, 0), Open=35.3125, High=35.9062, Low=31.9375, Close=32.0, Volume=20899
```

```
#Building DataFrame from RDD
yd = sc.textFile("/FileStore/tables/yahoo_stocks.csv", use_unicode=True) \
                    .map(lambda x:x.replace('"', "")) \
                    .map(lambda x:x.split(","))
yd.take(2)
```

```
Out[284]: [['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close'],
 ['2001-01-02',
  '30.3125',
  '30.375',
  '27.50',
  '28.1875',
  '21939200',
  '14.09375']]
```

```
#RDD Transformation
newRDD = yahoo_rdd.map(lambda x: x-1)
filteredRDD = newRDD.filter(lambda x : x<10)
```

```
# display rdd with various methods available in pyspark - using parllelize()
function
rdd_method = spark.sparkContext.parallelize([(1,2,3,'a b c'),(4,5,6,'d e
f'),(7,8,9,'g h i')]).toDF(["col_1","col_2","col_3","col_4"]).rdd
```

```
rdd_method.collect()
```

```
Out[287]: [Row(col_1=1, col_2=2, col_3=3, col_4='a b c'),
 Row(col_1=4, col_2=5, col_3=6, col_4='d e f'),
 Row(col_1=7, col_2=8, col_3=9, col_4='g h i')]
```

```
print('RDD Parllelize Count:', rdd_method.count())
```

```
RDD Parllelize Count: 3
```

```
#Number of partitions occupied by RDD
print('RDD Num Partitions:', yahoo_rdd.getNumPartitions())
#Number of partitions occupied by RDD using parllelize method
print('RDD Num Partitions by parllelize:', rdd_method.getNumPartitions())
```

```
RDD Num Partitions: 1
RDD Num Partitions by parllelize: 4
```

```
# Another method for rdd creation - using createDataFrame( ) function
RDD_DF = spark.createDataFrame([
                        ('3224', 'Shreyanth',   '90000', 'Digital'),
                        ('3223', 'Srikanth', '80000', 'Digital'),
                        ('3125', 'Rahul',   '70000', 'QA'),
                        ('2762', 'Vijay',   '60000', 'Gaming')],
                        ['EMP_ID', 'Name', 'Salary','Division']
                       )
display(RDD_DF)
```

Table

	EMP_ID	Name	Salary	Division
1	3224	Shreyanth	90000	Digital
2	3223	Srikanth	80000	Digital
3	3125	Rahul	70000	QA
4	2762	Vijay	60000	Gaming

Showing all 4 rows.

```
#repartition() is used to increase or decrease the partitions
#Below example increases the partition from 1 to 4
yahoo_rdd_repart = yahoo_rdd.repartition(4)
print('Repartition size : ', yahoo_rdd_repart.getNumPartitions())
```

```
Repartition size :  4
```

```
#coalesce() is used only to reduce the number of partitions. It is the
improved version of repartition(). The data moevement across the partitions
is lower using coalesce. Hence for scaling up or down repartition is a good
choice
yahoo_rdd_coalesce = yahoo_rdd_repart.coalesce(2)
print('Coalesce size : ', yahoo_rdd_coalesce.getNumPartitions())
```

```
Coalesce size :  2
```

TASK-2: Dataframe Operations

1. DF USING LIST

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType

spark = SparkSession.builder.appName('DF_List').getOrCreate()

#Using List
dept = [("Digital",1),
        ("QA",2),
        ("Gaming",3),
        ("Sales",4)
      ]

deptColumns = ["dept_name","dept_id"]


deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.printSchema()
```

```
root
 |-- dept_name: string (nullable = true)
 |-- dept_id: long (nullable = true)
```

```
deptDF.show(truncate=False)
```

```
+---------+-------+
|dept_name|dept_id|
+---------+-------+
|Digital  |1      |
|QA       |2      |
|Gaming   |3      |
```

```
|Sales    |4      |
+---------+-------+
```

```
from pyspark.sql import Row
# List as rows
dept2 = [Row("Digital",1),
        Row("QA",2),
        Row("Gaming",3),
        Row("Sales",4)
      ]
```

```
deptDF2 = spark.createDataFrame(data=dept2, schema = deptColumns)
deptDF2.printSchema()
deptDF2.show(truncate=False)
```

```
root
 |-- dept_name: string (nullable = true)
 |-- dept_id: long (nullable = true)

+---------+-------+
|dept_name|dept_id|
+---------+-------+
|Digital  |1      |
|QA       |2      |
|Gaming   |3      |
|Sales    |4      |
+---------+-------+
```

```
# Convert list to RDD
rdd = spark.sparkContext.parallelize(dept)
rdd2 = spark.sparkContext.parallelize(dept2)
```

```
type(rdd)
type(rdd2)
```

```
Out[299]: pyspark.rdd.RDD
```

2. DF USING RANGE

```
spark = SparkSession.builder.appName('DF_Range').getOrCreate()
DF_range = spark.range(100,120)
DF_range.show()
```

```
+---+
| id|
+---+
|100|
|101|
|102|
|103|
|104|
|105|
|106|
|107|
|108|
|109|
|110|
|111|
|112|
|113|
|114|
|115|
|116|
|117|
```

3. DF USING DICTIONARY

```
spark = SparkSession.builder.appName('DF_Dictionary').getOrCreate()
DF_dict = [{'Department_ID':'1','Department_Name':'Digital','Employee
Numbers':'3000',
              'Location':'US, CHENNAI'},
             {'Department_ID':'2','Department_Name':'QA','Employee
Numbers':'2000',
              'Location':'Chennai, US, Singapore, Bangalore'},
            {'Department_ID':'2','Department_Name':'Gaming','Employee
Numbers':'1000',
              'Location':'US, CHENNAI, Hyderabad, Mumbai, London'},
            {'Department_ID':'2','Department_Name':'Sales','Employee
Numbers':'500',
              'Location':'All Locations'}]

DF_Dictionary = spark.createDataFrame(DF_dict)
DF_Dictionary.show()
```

```
+-------------+---------------+----------------+--------------------+
|Department_ID|Department_Name|Employee Numbers|            Location|
+-------------+---------------+----------------+--------------------+
|            1|        Digital|            3000|         US, CHENNAI|
|            2|             QA|            2000|Chennai, US, Sing...|
|            2|         Gaming|            1000|US, CHENNAI, Hyde...|
|            2|          Sales|             500|       All Locations|
```

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

4. DF USING RDD

```
spark = SparkSession.builder.appName('DF_RDD').getOrCreate()
DF_RDD = spark.createDataFrame(yahoo_rdd)
DF_RDD.show()
```

```
+-------------------+-------+-------+-------+-------+--------+---------+
|               Date|   Open|   High|    Low|  Close|  Volume|Adj Close|
+-------------------+-------+-------+-------+-------+--------+---------+
|2001-01-02 00:00:00|30.3125| 30.375|   27.5|28.1875|21939200| 14.09375|
|2000-12-29 00:00:00|30.3125|31.1875|29.5625|30.0625|20893400| 15.03125|
|2000-12-28 00:00:00|29.4375|  31.75| 29.125|   31.0|24374600|     15.5|
|2000-12-27 00:00:00|   31.0|   31.5| 29.125|  29.75|22045400|   14.875|
|2000-12-26 00:00:00|   32.0|   34.0| 30.125|31.1875|37536200| 15.59375|
|2000-12-22 00:00:00|26.4375| 29.875|26.0625|29.5625|28347400| 14.78125|
|2000-12-21 00:00:00|  26.75|  28.25|25.0625| 25.625|27794400|  12.8125|
|2000-12-20 00:00:00|25.8125| 28.375|   25.5|27.9375|44862800| 13.96875|
|2000-12-19 00:00:00|30.5625|31.9687|   28.0|   28.0|36131600|     14.0|
|2000-12-18 00:00:00| 33.875|   34.0|  30.25|   32.0|31697600|     16.0|
|2000-12-15 00:00:00|   32.0|   34.0|31.0625|   33.0|40448000|     16.5|
|2000-12-14 00:00:00|35.3125|35.9062|31.9375|   32.0|20899800|     16.0|
|2000-12-13 00:00:00|38.3125| 38.625|  34.25| 34.875|33640400|  17.4375|
|2000-12-12 00:00:00|  33.25|   39.5|32.9375|35.8125|79275800| 17.90625|
|2000-12-11 00:00:00| 33.625|37.0625| 30.625| 33.875|71038800|  16.9375|
|2000-12-08 00:00:00| 37.125| 37.125| 32.125|34.9375|49184000| 17.46875|
|2000-12-07 00:00:00|36.0625|36.2187|   31.5|34.9375|55136200| 17.46875|
|2000-12-06 00:00:00| 41.625|42.9375| 37.125|   37.5|32559800|    18.75|
```

Compare Pandas DF vs Spark DF | Spark SQL

```
from io import StringIO
import pandas as pd

data1 = """employee_id,name,salary,tax_paid,joining_date
I3223,Shreyanth S,85000,1453.124,2020-08-01
I3224,Srikanth Patange,85000,,2020-09-10
I3225,Vijay,70000,1375.22,2021-01-15
I3226,Rahul Jain,65000,1325.65,2022-04-09
I3227,Sunil,40000,,2022-07-29
"""

data2 = """employee_id2,name,salary2,tax_paid,joining_date
I3223,Shreyanth S,85000,,2020-08-01
I3224,Srikanth Patange,85000,1453.124,2020-09-10
I3225,Vijay,70000,1457.33,2021-01-15
I3226,Rahul Jain,65000,,2022-04-09
I3227,Sunil,40000,1296.54,2022-07-29
"""

pandas_df_1 = pd.read_csv(StringIO(data1))
pandas_df_2 = pd.read_csv(StringIO(data2))

display(pandas_df_1)
display(pandas_df_2)
```

Table

	employee_id	name	salary	tax_paid	joining_date
1	I3223	Shreyanth S	85000	1453.124	2020-08-01
2	I3224	Srikanth Patange	85000	null	2020-09-10
3	I3225	Vijay	70000	1375.22	2021-01-15
4	I3226	Rahul Jain	65000	1325.65	2022-04-09
5	I3227	Sunil	40000	null	2022-07-29

Showing all 5 rows.

Table

	employee_id2	name	salary2	tax_paid	joining_date
1	I3223	Shreyanth S	85000	null	2020-08-01
2	I3224	Srikanth Patange	85000	1453.124	2020-09-10
3	I3225	Vijay	70000	1457.33	2021-01-15
4	I3226	Rahul Jain	65000	null	2022-04-09
5	I3227	Sunil	40000	1296.54	2022-07-29

Showing all 5 rows.

```
print (pd.merge(pandas_df_1, pandas_df_2.rename(columns=
{'employee_id2':'employee_id'}), on='employee_id',  how='left'))
```

```
  employee_id            name_x  salary  ...  salary2 tax_paid_y joining_dat
e_y
0       I3223       Shreyanth S   85000  ...    85000        NaN     2020-08
-01
1       I3224  Srikanth Patange   85000  ...    85000   1453.124     2020-09
-10
2       I3225             Vijay   70000  ...    70000   1457.330     2021-01
-15
3       I3226        Rahul Jain   65000  ...    65000        NaN     2022-04
-09
4       I3227             Sunil   40000  ...    40000   1296.540     2022-07
-29

[5 rows x 9 columns]
```

```
import datetime
from pyspark.sql import Row

# This example assumes you have a SparkSession named "spark" in your
environment, as you
# do when running `pyspark` from the terminal or in a Databricks notebook
(Spark v2.0 and higher)

data1 = [
    Row(employee_id='I3223', Salary=85000, name='Shreyanth S',
tax_paid=14530.1555,
        joining_date=datetime.date(2020, 8, 1)),
    Row(employee_id='I3224', Salary=85000, name='Srikanth Patange',
tax_paid=1.0,
        joining_date=datetime.date(2020, 9, 10)),
    Row(employee_id='I3225', Salary=70000, name='Vijay', tax_paid=334.65,
        joining_date=datetime.date(2021, 1,15)),
    Row(employee_id='I3226', Salary=65000, name='Rahul Jain',
tax_paid=345.12,
        joining_date=datetime.date(2022, 4, 9)),
    Row(employee_id='I3227', Salary=40000, name='Sunil', tax_paid=None,
        joining_date=datetime.date(2022, 7, 29))
]

data2 = [
    Row(employee_id='I3223', Salary=85000, name='Shreyanth S',
tax_paid=1753.23,
        joining_date=datetime.date(2020, 8, 1)),
    Row(employee_id='I3224', Salary=85000, name='Srikanth Patange',
tax_paid=1487.55,
        joining_date=datetime.date(2020, 9, 10)),
    Row(employee_id='I3225', Salary=70000, name='Vijay', tax_paid=1365.43,
        joining_date=datetime.date(2021, 1,15)),
    Row(employee_id='I3226', Salary=65000, name='Rahul Jain', tax_paid=None,
        joining_date=datetime.date(2022, 4, 9)),
    Row(employee_id='I3227', Salary=40000, name='Sunil', tax_paid=1137.86,
        joining_date=datetime.date(2022, 7, 29))
]

spark_df_1 = spark.createDataFrame(data1)
spark_df_2 = spark.createDataFrame(data2)

display(spark_df_1)
display(spark_df_2)
```

Table

	employee_id	Salary	name	tax_paid	joining_date
1	I3223	85000	Shreyanth S	14530.1555	2020-08-01

2	I3224	85000	Srikanth Patange	1	2020-09-10
3	I3225	70000	Vijay	334.65	2021-01-15
4	I3226	65000	Rahul Jain	345.12	2022-04-09
5	I3227	40000	Sunil	null	2022-07-29

Showing all 5 rows.

Table

	employee_id	Salary	name	tax_paid	joining_date
1	I3223	85000	Shreyanth S	1753.23	2020-08-01
2	I3224	85000	Srikanth Patange	1487.55	2020-09-10
3	I3225	70000	Vijay	1365.43	2021-01-15
4	I3226	65000	Rahul Jain	null	2022-04-09
5	I3227	40000	Sunil	1137.86	2022-07-29

Showing all 5 rows.

```
spark_df_1.createOrReplaceTempView("table_df_1")
spark_df_2.registerTempTable("table_df_2")
```

```
/databricks/spark/python/pyspark/sql/dataframe.py:146: FutureWarning: Deprec
ated in 2.0, use createOrReplaceTempView instead.
  warnings.warn(
```

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
table_DF = sqlContext.sql("select t1.employee_id, t1.salary, t1.name,
(t1.tax_paid-t2.tax_paid)" +
  "from table_df_1 t1 inner join  table_df_2 t2 " +
  "on t1.employee_id = t2.employee_id").show()
```

```
/databricks/spark/python/pyspark/sql/context.py:82: FutureWarning: Deprecate
d in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
+-----------+------+---------------+--------------------+
|employee_id|salary|           name|(tax_paid - tax_paid)|
+-----------+------+---------------+--------------------+
|      I3223| 85000|    Shreyanth S|   12776.925500000001|
|      I3224| 85000|Srikanth Patange|             -1486.55|
|      I3225| 70000|          Vijay|  -1030.7800000000002|
|      I3226| 65000|     Rahul Jain|                null|
|      I3227| 40000|          Sunil|                null|
+-----------+------+---------------+--------------------+
```

```
%sql
CREATE TABLE customer_data USING CSV LOCATION
'/FileStore/tables/customer_contact.csv'
```

OK

```
%sql
select * from customer_data
```

Table

	_c0	_c1	_c2	_c3
1	Unnamed: 0	CustAddrID	CustomerInfoId	Address1
2	510000	546716	555812	house no 47 Paliya Masudpur TEH Naw
3	510001	545479	555813	Paliyamasudpur
4	510002	550138	555814	Paliyamasudpur
5	510003	550137	555815	Ajadnagar
6	510004	547060	555816	bisauli
7	510005	547424	555817	bisauli

Truncated results, showing first 1,000 rows.

```
%sql
DROP TABLE customer_data
```

OK

```
# File location and type
file_location = "/FileStore/tables/customer_contact.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be
ignored.
customer_contact = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)


customer_contact.createOrReplaceTempView("customer_contact_table")
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Spark_sql').getOrCreate()
spark.sql("select custaddrid, customerinfoid, address1, zipcode1 from
customer_contact_table order by createdby").show()
```

```
+----------+--------------+--------------------+--------+
|custaddrid|customerinfoid|            address1|zipcode1|
+----------+--------------+--------------------+--------+
|    575571|        559206|Berue Rakswara ma...|  212216|
|    531577|        554433|Krishna nagar 208...|  281005|
|   47:40.4|       36:02.3|                1506|    null|
|      null|          null|                null|    null|
|    456730|        559859|giriya khalasakau...|  212216|
|   24:33.5|       28:12.9|                 526|    null|
|   50:42.4|       02:33.7|                1506|    null|
|   52:22.0|       59:40.0|                2420|    null|
|    463359|        559870|Giriya khalasakau...|  212216|
|      null|          null|                null|    null|
|   09:11.9|          null|                1506|    null|
|    568568|        558483|BELADIH,BELAGANJG...|  804403|
|    482097|        546125|       jatasankkar56|  470661|
|    568569|        558484|Belaganj,Bhuntoli...|  804403|
|      null|          null|                null|    null|
|   52:22.8|          null|                2420|    null|
|    485026|        547764|   193 soraun amethi|  227806|
|   16:54.7|          null|                1544|    null|
```

TASK-3: UDF IMPLEMENTATION

```
from pyspark.sql import SparkSession
import os
from pyspark.sql.functions import udf , col
from pyspark.sql.types import StringType

spark = SparkSession \
    .builder \
    .appName("UDF task-3") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

```
# File location and type
file_location = "/FileStore/tables/Department_Dataset.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be
ignored.
dept_df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)
```

```
dept_df.show(truncate=False)
```

```
+---+---------+---------+---------------+--------------+
|ID |Dept_name|location |travel_required|Average_salary|
+---+---------+---------+---------------+--------------+
|1  |HR       |pune     |yes            |1000000       |
|2  |Finance  |bangalore|no             |1500000       |
|3  |Finance  |bangalore|no             |1500000       |
|4  |Finance  |pune     |no             |1300000       |
|5  |Tech     |mumbai   |no             |2000000       |
|6  |Tech     |pune     |no             |1700000       |
|7  |Tech     |bangalore|yes            |2200000       |
|8  |HR       |bangalore|no             |1400000       |
|9  |HR       |pune     |no             |1000000       |
|10 |HR       |pune     |no             |1000000       |
|11 |HR       |mumbai   |no             |1200000       |
|12 |HR       |mumbai   |yes            |1200000       |
|13 |Finance  |bangalore|yes            |1500000       |
|14 |Tech     |bangalore|yes            |2200000       |
|15 |Tech     |mumbai   |yes            |2000000       |
|16 |Tech     |pune     |yes            |1700000       |
|17 |Tech     |bangalore|no             |2200000       |
|18 |Finance  |mumbai   |no             |1300000       |
```

```
# Convert the lower case string to upper case and store in the table
def convertCase(str):
    resStr=""
    arr = str.split(" ")
    for x in arr:
       resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "
    return resStr

# Converting function to UDF
convertUDF = udf(lambda z: convertCase(z))

dept_df.select(col("ID"), \
    convertUDF(col("location")).alias("location") ) \
.show(truncate=False)


@udf(returnType=StringType())
def upperCase(str):
    return str.upper()

upperCaseUDF = udf(lambda z:upperCase(z),StringType())

dept_df.withColumn("Uppercased location", upperCase(col("location"))) \
.show(truncate=False)
```

```
+---+----------+
|ID |location  |
+---+----------+
|1  |Pune      |
|2  |Bangalore |
|3  |Bangalore |
|4  |Pune      |
|5  |Mumbai    |
|6  |Pune      |
|7  |Bangalore |
|8  |Bangalore |
|9  |Pune      |
|10 |Pune      |
|11 |Mumbai    |
|12 |Mumbai    |
|13 |Bangalore |
|14 |Bangalore |
|15 |Mumbai    |
|16 |Pune      |
|17 |Bangalore |
|18 |Mumbai    |
```

```
# Using UDF on SQL
spark.udf.register("convertUDF", convertCase,StringType())
dept_df.createOrReplaceTempView("DEPARTMENT_TABLE")
spark.sql("select ID, Dept_name, convertUDF(location) as location,
Travel_required, Average_salary from DEPARTMENT_TABLE") \
     .show(truncate=False)

spark.sql("select ID, Dept_name, convertUDF(location) as New_location,
Travel_required, Average_salary from DEPARTMENT_TABLE " + \
          "where location is not null and convertUDF(location) like
'Bang%'") \
     .show(truncate=False)
```

```
+---+---------+----------+---------------+--------------+
|ID |Dept_name|location  |Travel_required|Average_salary|
+---+---------+----------+---------------+--------------+
|1  |HR       |Pune      |yes            |1000000       |
|2  |Finance  |Bangalore |no             |1500000       |
|3  |Finance  |Bangalore |no             |1500000       |
|4  |Finance  |Pune      |no             |1300000       |
|5  |Tech     |Mumbai    |no             |2000000       |
|6  |Tech     |Pune      |no             |1700000       |
|7  |Tech     |Bangalore |yes            |2200000       |
|8  |HR       |Bangalore |no             |1400000       |
|9  |HR       |Pune      |no             |1000000       |
|10 |HR       |Pune      |no             |1000000       |
|11 |HR       |Mumbai    |no             |1200000       |
|12 |HR       |Mumbai    |yes            |1200000       |
|13 |Finance  |Bangalore |yes            |1500000       |
|14 |Tech     |Bangalore |yes            |2200000       |
|15 |Tech     |Mumbai    |yes            |2000000       |
|16 |Tech     |Pune      |yes            |1700000       |
|17 |Tech     |Bangalore |no             |2200000       |
|18 |Finance  |Mumbai    |no             |1300000       |
```

```
# using UDF for null check - This will replace the row that has null (if
any) with empty record

spark.udf.register("nullsafeUDF", lambda str: convertCase(str) if not str is
None else "" , StringType())

dept_df.createOrReplaceTempView("DEPARTMENT_TABLE_2")

spark.sql("select ID, Dept_name, location, nullsafeUDF(location) as
new_location, Travel_required, Average_salary from DEPARTMENT_TABLE_2") \
     .show(truncate=False)

spark.sql("select ID, Dept_name, nullsafeUDF(location) as new_location,
Travel_required, Average_salary from DEPARTMENT_TABLE_2 " + \
          " where Average_salary is not null and nullsafeUDF(location) like
'Mys%'") \
     .show(truncate=False)
```

```
+---+---------+---------+------------+---------------+--------------+
|ID |Dept_name|location |new_location|Travel_required|Average_salary|
+---+---------+---------+------------+---------------+--------------+
|1  |HR       |pune     |Pune        |yes            |1000000       |
|2  |Finance  |bangalore|Bangalore   |no             |1500000       |
|3  |Finance  |bangalore|Bangalore   |no             |1500000       |
|4  |Finance  |pune     |Pune        |no             |1300000       |
|5  |Tech     |mumbai   |Mumbai      |no             |2000000       |
|6  |Tech     |pune     |Pune        |no             |1700000       |
|7  |Tech     |bangalore|Bangalore   |yes            |2200000       |
|8  |HR       |bangalore|Bangalore   |no             |1400000       |
|9  |HR       |pune     |Pune        |no             |1000000       |
|10 |HR       |pune     |Pune        |no             |1000000       |
|11 |HR       |mumbai   |Mumbai      |no             |1200000       |
|12 |HR       |mumbai   |Mumbai      |yes            |1200000       |
|13 |Finance  |bangalore|Bangalore   |yes            |1500000       |
|14 |Tech     |bangalore|Bangalore   |yes            |2200000       |
|15 |Tech     |mumbai   |Mumbai      |yes            |2000000       |
|16 |Tech     |pune     |Pune        |yes            |1700000       |
|17 |Tech     |bangalore|Bangalore   |no             |2200000       |
|18 |Finance  |mumbai   |Mumbai      |no             |1300000       |
```

Task-4: OPTIMIZATION TECHNIQUES

Broadcast Variable

1. Broadcast variables are immutable shared variables which are cached on each worker nodes on a Spark cluster. Broadcast variables allow the programmer to

keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

2. Example for broadcast variable is if we have 5 nodes cluster with 50 partitions (10 partitions per node), this Array will be distributed at least 50 times (10 times to each node). But If we use broadcast it will be distributed once per node using efficient p2p protocol

```
import pyspark
import numpy as np
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
from pyspark.sql.types import StringType
from pyspark.sql.functions import udf
from pyspark.sql.types import *
import pyspark.sql.functions  as sf
from pyspark.sql.types import *



def main():
    arguments=sys.argv
    srcFile=arguments[1]
    lookup=arguments[2]
    trgtFilePath=arguments[3]
    trgtFileNm=arguments[4]

    finalfile=os.path.join(trgtFilePath,trgtFileNm)
    spark=SparkSession.builder.appName("Broadcast_udf").getOrCreate()
    sc=spark.sparkContext


    rdd=sc.textFile('/FileStore/tables/cities_stats.csv')
    header=rdd.first()
    rddforbroadcast=rdd.filter(lambda x: x!=header).map(lambda x:
((x.split(",")[0],x.split(",")[1]),(x.split(",")[2],x.split(",")[3])))
    broadcastVar=sc.broadcast(rddforbroadcast.collectAsMap())
    print(rdd.take(8))
    broadcast1=broadcastVar.value.get(('Chico','CA'))
    broadcast2=broadcastVar.value.get(('Bakersfield','CA'))
    broadcast3=broadcastVar.value.get(('Granite Bay','CA'))
    def updateAMT(city,state,amt):
        ct=city
        st=state
        print(repr(ct))
        print(repr(st))
        v=(ct,st)
        broadcastresult=broadcastVar.value.get(v)
        if broadcastresult is None:
            result=amt
        else:
            result=int(amt)+((int(amt)*int(broadcastresult[0]))/100)
        return result
    spark.udf.register("updateAMT",updateAMT)

    def updateDate(city,state,amtDate):
        ct=city
        st=state
```

```
        broadcastresult=broadcastVar.value.get((ct,st))
        if broadcastresult is None:
            result=amtDate
        else:
            result=broadcastresult[1]
        return result
    spark.udf.register("updateDate",updateDate)

    df=spark.read.csv('/FileStore/tables/broadcast_data.csv', header=True)
    df.createOrReplaceTempView("broadcast_table")
    newdf=spark.sql('''select OrgName,Cities,ST,updateAMT(Cities,ST,AnyAMT)
AnyAMT, updateDate(Cities,ST,AnyAMTDate) AnyDATE from broadcast_table order
by AnyDATE ''').show(10)

    #prints the broadcast call for cities within CA
    print('Adjust value for Chico is: ',broadcast1[0])
    print('Adjust value for Bakersfield is: ',broadcast2[0])
    print('Adjust value & Validate date for Granite Bay is:
',broadcast3[0],'&',broadcast3[1])


if __name__=='__main__':
    main()
else:
    print('Already Satisfied')
```

```
['Cities,ST,Adjust,Validate', 'Bakersfield,CA,2,1/1/16', 'Calabasas,CA,4,1/
1/16', 'Chico,CA,6,1/1/16', 'Culver City,CA,8,1/1/16', 'Granite Bay,CA,10,1/
1/16', 'Irvine,CA,2,1/1/16', 'La Jolla,CA,4,1/1/16']
+--------------------+--------------+---+--------+-------+
|             OrgName|        Cities| ST|  AnyAMT|AnyDATE|
+--------------------+--------------+---+--------+-------+
|         Jasper ing |        Jasper| GA| 17864.0| 1/1/16|
|    First Cherokee  |     Woodstock| GA|34019.44| 1/1/16|
|             First  |   Stockbridge| GA|20407.12| 1/1/16|
|        Gegia Trust |        Buford| GA|59003.94| 1/1/16|
|               Gegia|Peachtree City| GA|36669.36| 1/1/16|
|Montgomery  &amp;amp;...|         Ailey| GA| 21447.8| 1/1/16|
|          Eastside  |       Conyers| GA| 60450.0| 1/1/16|
|  Security Exchange |      Marietta| GA|36710.96| 1/1/16|
|           Douglas  |  Douglasville| GA|23380.92| 1/1/16|
|Palm Desert Natio...|   Palm Desert| CA|25522.56| 1/1/16|
+--------------------+--------------+---+--------+-------+
only showing top 10 rows

Adjust value for Chico is:  6
Adjust value for Bakersfield is:  2
```

Persist and Cache

1. Persist and Cache are optimization techniques to improve the performance of the jobs or applications.
2. Both caching and persisting are used to save the Spark RDD, Dataframe, and Dataset's. But, the difference is, RDD cache() method default saves it only to the memory whereas persist() method stores it to the user-defined storage level.
3. In persist, Each node stores its partitioned data in memory and reuses them in other forms on that dataset.
4. Similarly, the advantages of cache and persist are cost saving & Time saving.

How does the cache works in Spark for RDD?

when the first record of an RDD is initiated, it will test if this RDD should be cached. If yes, then the first record and all the followed records will be sent to blockManager's memoryStore. If memoryStore can not hold all the records, diskStore will be used instead.

Spark will test whether the RDD should be cached or not just before computing the first partition. If the RDD should be cached, the partition will be computed and cached into memory. After this it writes to the disk and this is called checkpoint.

After calling rdd.cache(), rdd becomes persistRDD whose storageLevel is MEMORY_ONLY. persistRDD will tell driver that it needs to be persisted. Then persist() starts the functioning

```
# File location and type
file_location = "/FileStore/tables/Department_Dataset.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be
ignored.
dept_df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location).cache()

# Size of dataframe memory utilised in cache
dept_df.cache().count()
```

```
Out[321]: 51
```

```
# Check if DF is cached
dept_df.is_cached
```

```
Out[322]: True
```

Using persist method, each persisted RDD/Dataset can be stored using a different storage level, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space), replicate it across nodes. These levels are set by passing a StorageLevel object (Scala, Java, Python) to persist() method.

```
# Stores the data in the memory level
dept_df.persist(pyspark.StorageLevel.MEMORY_ONLY)
```

```
Out[323]: DataFrame[ID: int, Dept_name: string, location: string, travel_req
uired: string, Average_salary: int]
```

```
dept_df.persist(pyspark.StorageLevel.MEMORY_ONLY).count()
```

```
Out[324]: 51
```

```
# Removes the data from the memory level
dept_df.unpersist()
```

```
Out[325]: DataFrame[ID: int, Dept_name: string, location: string, travel_req
uired: string, Average_salary: int]
```

```
# # Stores the data in the disk level
dept_df.persist(pyspark.StorageLevel.DISK_ONLY).count()
```

```
Out[326]: 51
```

```
Out[327]: DataFrame[ID: int, Dept_name: string, location: string, travel_req
uired: string, Average_salary: int]
```