Data processing for Big Data

Analysing Retail Data

Name : Syed Kabir

Part 1: Working with RDDs (30%)

1.1 Data Preparation and Loading (5%)

1.1.1: Write the code to create a SparkContext object using SparkSession, which tells Spark how to access a cluster. To create a SparkSession you first need to build a SparkConf object that contains information about your application, using Melbourne time as the session timezone. Give an appropriate name for your application and run Spark locally with as many working processors as logical cores on your machine.

Answer:

a) At first, SparkConf object is imported from the pyspark class

```
# Import SparkConf class into program
from pyspark import SparkConf

# local[*]: run Spark in local mode with as many working processors as logical cores on your machine
master = "local[*]"

# The `appName` field is a name to be shown on the Spark cluster UI page
app_name = "Retail Analysis App"

# Setup configuration parameters for Spark
spark_conf = SparkConf(). \
                setMaster(master).setAppName(app_name). \
                set("spark.sql.session.timeZone", "Australia/Melbourne") # setting the timezone for the session
```

b) SparkConf object is defined with the following parameters as shown in the above screenshot:
- Selecting all processing cores available in the matching using *
- Named the application as “Retail Analysis App”
- Setting session timezone as “Australia/Melbourne”

c) SparkSession object is imported from pyspark.sql class

d) A spark session is built from the builder object of SparkSession using config() and getOrCreate() methods and named as spark. ‘conf’ parameter is set as spark_conf (as defined in step b) within config() method. The details are shown in the following screenshot:

```
# Import SparkSession classes
from pyspark.sql import SparkSession # Spark SQL

# Method : Using SparkSession; Configuring time zone
spark = SparkSession.builder. \
                    config(conf=spark_conf). \
                    getOrCreate()
sc = spark.sparkContext
sc.setLogLevel('ERROR')
```

e) An instance of sparkcontext is created from the defined sparksession and named as sc as shown in the second last line of the above screenshot

f) Finally, ‘Error’ is set as setLogLevel to override any user-defined log settings.

1.1.2: Load the features, sales and stores csv file into features, sales, stores RDDs.

Ans:

textFile() method from sparkContext (as sc) is used to load features, sales and stores csv files from the local repository as shown in the following screenshot

```
features_rdd_raw = sc.textFile('features.csv')
sales_rdd_raw = sc.textFile('sales.csv')
stores_rdd_raw = sc.textFile('stores.csv')
```

1.1.3: For each features, sales and stores RDDs, remove the header rows and display the total count and first 10 records.

Answer: Steps:

a) At first, header is set using first() method

b) To remove header from the rdd, filter() method is used on imported rdd. The parameter is a lambda function which confirms the removal of header row from the rdds. As an example, a screenshot on features rdd is shown below:

```
1 header_features = features_rdd_raw.first()
2 features_rdd = features_rdd_raw.filter(lambda row: row != header_features)
3 print(f"Total number of rows in featutures_rdd: {features_rdd.count()}")
4 print("First 10 rows of featutures_rdd : ")
5 features_rdd.take(10)
```

```
Total number of rows in featutures_rdd: 8190
First 10 rows of featutures_rdd :

['1,05/02/2010,42.31,2.572,NA,NA,NA,NA,NA,211.0963582,8.106,FALSE',
 '1,12/02/2010,38.51,2.548,NA,NA,NA,NA,NA,211.2421698,8.106,TRUE',
 '1,19/02/2010,39.93,2.514,NA,NA,NA,NA,NA,211.2891429,8.106,FALSE',
 '1,26/02/2010,46.63,2.561,NA,NA,NA,NA,NA,211.3196429,8.106,FALSE',
 '1,05/03/2010,46.5,2.625,NA,NA,NA,NA,NA,211.3501429,8.106,FALSE',
 '1,12/03/2010,57.79,2.667,NA,NA,NA,NA,NA,211.3806429,8.106,FALSE',
 '1,19/03/2010,54.58,2.72,NA,NA,NA,NA,NA,211.215635,8.106,FALSE',
 '1,26/03/2010,51.45,2.732,NA,NA,NA,NA,NA,211.0180424,8.106,FALSE',
 '1,02/04/2010,62.27,2.719,NA,NA,NA,NA,NA,210.8204499,7.808,FALSE',
 '1,09/04/2010,65.86,2.77,NA,NA,NA,NA,NA,210.6228574,7.808,FALSE']
```

c) Count() is used inside print function to show the total number of rows in a rdd

d) Finally, take() method is used to show first 10 rows in rdds. So, parameter is just 10.

1.2 Data Partitioning in RDD (15%)

1.2.1: *How many partitions do the above RDDs have? How is the data in these RDDs partitioned by default, when we do not explicitly specify any partitioning strategy? Can you explain why it will be partitioned in this number? If I only have one single core CPU in my PC, what is the default partition's number?*

Answer: getNumPartitions() method is used individually to see how the features, sales and stores rdds are partitioned by default. There are 2 partitions for each rdd by default.
The source code from the following link shows that the parameters of textFile function:
https://spark.apache.org/docs/latest/api/python/_modules/pyspark/context.html#SparkContext.textFile

```
def textFile(self, name: str, minPartitions: Optional[int] = None, use_unicode: bool = True)
        minPartitions = minPartitions or min(self.defaultParallelism, 2)
        return RDD(self._jsc.textFile(name, minPartitions), self, UTF8Deserializer(use_unicode))
```

So, default partition number for textFile function is 2 unless minPartitions parameters is not set with any other number. Because the default setting is:

min(self.defaultParallelism, 2)

If we only have one single core CPU in our PC, then data will be partitioned to only one core.

1.2.2: Create a key value RDD for the store RDD, use the store type as the key and all of the columns as the value. Print out the first 5 records of the key-value RDD.

Answer: parseRecord() function is created following the tutorial-1 materal named “FIT5202 - Getting started with Apache Spark (1)” of ‘FIT5202 Data processing for big data - Summer B 2023’ placed in ‘Moodle’. The details of creating key value pair for store RDD is shown below:

a) As we see from the previous page screenshot for features_rdd.take(), each record has been placed within ‘’ and separated the records by comma(,). So, parseRecord function split all records using split() method with the parameter comma as shown in the following screenshot.
b) Then the function returns a tuple where key is set for array with index 1 because the index of Type in Store Rdd is 1. So, values are of index 0 and index 2.
c) Finally, the function is mapped on stores rdd using map() function placing parseRecord as the parameter of this function.
d) Take() method shows the store rdd has been arranged as per required format where key is Type and all other attributes are value.

```
# Implement function with logic to be applied to the RDDs
def parseRecord(line):
    # Split line separated by comma
    array_line = line.split(',')
    # Return a tuple with the car model as first element and the remaining as the second element
    return (array_line[1], [array_line[0],array_line[2]])

storetypekey_rdd = stores_rdd.map(parseRecord)
storetypekey_rdd.take(5)
```

```
[('A', ['1', '151315']),
 ('A', ['2', '202307']),
 ('B', ['3', '37392']),
 ('A', ['4', '205863']),
 ('B', ['5', '34875'])]
```

1.2.3: Write the code to separate the store key-value RDD based on the store type (the same type should be in the same partition). Print out the total partition's number and the number of records in each partition.

print_partitions () function is created following the tutorial-2 materal named “Activity: Parallel Search” of ‘FIT5202 Data processing for big data - Summer B 2023’ placed in ‘Moodle’. The details of creating key-value pair in the same partitions for each Store Type is shown below:

a) Creating print_partitions:
   - The function take the input as the rdd and return the partition numbers, number of records in each partition and data elements of each partition.

```
def print_partitions(data):
    numPartitions = data.getNumPartitions()
    partitions = data.glom().collect()
    print(f"####### NUMBER OF PARTITIONS: {numPartitions}")
    for index, partition in enumerate(partitions):
        # show partition if it is not empty
        if len(partition) > 0:
            print(f"Partition {index}: {len(partition)} records")
            print(partition)
```

   - So, getNumPartitions function is used to estimate the number of partitions.
   - data is collected using glom() and collect() and assigned to partitions variable

- next, a for loop is used to iterate over index of each partition and data points
- using print function, partition index and number of records in each partition and all the data elements in each partition are displayed as the output of the function

b) Creating hash_function():
   a) This function converts any key to hashed key using hash() method. This function will be used to convert categorical Type values to hashed Type values.

```
# Creating a hash function to convert catagorical value to hashed value
def hash_function(key):
    hashed_key = hash(key)
    return hashed_key
```

c) As we need to place same type of store in the same partitions, hash partitioning is the best to perform this requirement.
d) Hash Partitioning:
   b) Number of partitions are calculated from the distinct number of store Types.
   c) Hash partitioning is performed using partitionBy() functions implemented on storetypekey_rdd with two parameters. One for number of partitions and another one is hash function. The partitioned data is assigned to hash_partitioned_rdd variable
e) The partition number, partition index, records numbers and all records in each partition is displayed using print_partitions function as the parameter of hash_partitioned_rdd.

```
6  # hash partitioning
7  no_of_partitions= storetypekey_rdd.keys().distinct().count()
8  hash_partitioned_rdd = storetypekey_rdd.partitionBy(no_of_partitions, hash_function)
9
10 # Displaying partitioning
11 print_partitions(hash_partitioned_rdd)
```

```
######## NUMBER OF PARTITIONS: 3
Partition 0: 23 records
[('B', ['3', '37392']), ('B', ['5', '34875']), ('B', ['7', '70713']), ('B', ['9', '125833']), ('B', ['10', '126512']), ('B',
['12', '112238']), ('B', ['15', '123737']), ('B', ['16', '57197']), ('B', ['17', '93188']), ('B', ['18', '120653']), ('B',
['21', '140167']), ('B', ['22', '119557']), ('B', ['23', '114533']), ('B', ['25', '128107']), ('B', ['29', '93638']), ('C',
['30', '42988']), ('B', ['35', '103681']), ('C', ['37', '39910']), ('C', ['38', '39690']), ('C', ['42', '39690']), ('C', ['4
3', '41062']), ('C', ['44', '39910']), ('B', ['45', '118221'])]
Partition 1: 22 records
[('A', ['1', '151315']), ('A', ['2', '202307']), ('A', ['4', '205863']), ('A', ['6', '202505']), ('A', ['8', '155078']),
('A', ['11', '207499']), ('A', ['13', '219622']), ('A', ['14', '200898']), ('A', ['19', '203819']), ('A', ['20', '203742']),
('A', ['24', '203819']), ('A', ['26', '152513']), ('A', ['27', '204184']), ('A', ['28', '206302']), ('A', ['31', '203750']),
('A', ['32', '203007']), ('A', ['33', '39690']), ('A', ['34', '158114']), ('A', ['36', '39910']), ('A', ['39', '184109']),
('A', ['40', '155083']), ('A', ['41', '196321'])]
```

f) The result displayed in the above screenshot confirms that three partitions are being created. But 2 partitions are used. Partition 0 contains all data related to store type B & C which contains in total 23 records. Partition 1 contains only A type store which has 22 records.
g) As C type records are of very few numbers, spark placed all C type records along with B without placing c Type data in another partitions.
h) This will help to avoid costly IO operations and other overheads for very few number of records. So, overall processing will be faster.

1.3 Query/Analysis (10%)

Answer/ Explanation

1.3.1: Calculate the average weekly sales for each year.

Ans: Following step are followed for this query:

a) ParseRecord() function is restructured to grab the Date and Weekly_Sales values. The following screenshot shows that in detail.

```
def parseRecord(line):
    # Split line separated by comma
    array_line = line.split(',')
    # Return a tuple with 'Date' as first element and weekly sales as the second element
    return (array_line[2], array_line[3])

weeklySales_rdd = sales_rdd.map(parseRecord)
weeklySales_rdd.take(5)
```

```
[('05/02/2010', '24924.5'),
 ('12/02/2010', '46039.49'),
```

b) Next, parseRecord() is mapped to sales rdd which resulted tuple of date and weekly sales values.

c) Now we need to unpack year data from the date. To do so, weeklySales_rdd is mapped with a lambda function. The lambda function split 'Date' with the

```
weeklySales_year_rdd = weeklySales_rdd.map(lambda x: x[0].split('/')).map(lambda x: (x[2]))
weeklySales_year_rdd.distinct().collect()
```

```
['2010', '2011', '2012']
```

parameter '/' used in split command. Then 3rd value of the split list is selected to grab only year data.

d) This year data is zipped with the second element of weeklySales_rdd i.e. sales values. So, we get rdd that has only year and sales data. Then datatype is

```
weeklySales_rdd1 = weeklySales_year_rdd.zip(weeklySales_rdd. \
                                            map(lambda x : x[1])). \
                                            map(lambda x: [float(i) for i in x]) # Converting string values of sales t
```

changed using map function again from string to float data type for facilitating aggregation in next stage

e) Then, average value of the weekly sales is calculated by the following steps:

```
# Calculating average weekly sales for each year
weeklyAvgSales_rdd = weeklySales_rdd1.groupByKey(). \
                                      mapValues(lambda x: sum(x)/len(x))
weeklyAvgSales_rdd.collect()
```

```
[(2010.0, 16270.275737033313),
 (2012.0, 15694.948597357718),
 (2011.0, 15954.070675386392)]
```

- At first grouping the year data using groupByKey() commands on weeklySales_rdd1.
- Implementing a map function on the grouped rdd using a lambda function as

the parameter.

- The lambda function converts all sales value to average sales value for each group which is year.
- As a result of these steps, we obtained weekly average sales for each year.
- Collect() command shows the results.

1.3.2: Find the highest temperature record in 2011 in the 'type B' store. You should display the store ID, date, highest temperature, and type in the result.

Steps: Temperature and Type are available in features and stores rdd. So, perform this query, we have to join features and stores rdd on Store column using following steps

a) Selecting required elements from features rdd:
Creating purseRecordFeatures function to get a rdd that has tuple of 'Store' key element and 'Date' & 'Temperature' as the values elements.

b) Selecting required elements from stores rdd:
Creating purseRecordStores function to get a rdd that has 'Store' as the first element and 'Type' & 'Size' as the other elements.

c) Joining features and stores rdd:
Joining on Stores column will produce rdd that has Store as the key and Date, Temperature and Type as the values. However, Date and Temperature are within

```
# Joining feature and store rdds
feature_store_rdd = features_rdd1.join(stores_rdd1)
feature_store_rdd.take(5)
```

```
[('4', (('05/02/2010', '43.76'), 'A')),
 ('4', (('12/02/2010', '28.84'), 'A')),
 ('4', (('19/02/2010', '36.45'), 'A')),
```

a tuple. So, the elements are needed to be unpacked for further progress

d) Initial Unpacking of the elements:

- Using a map function on joined rdd where parameter is a lambda function.

```
feature_store_rdd1 = feature_store_rdd.map(lambda x : [x[0]] +list(x[1]))
feature_store_rdd1.take(3)
```

```
[['4', ('05/02/2010', '43.76'), 'A'],
 ['4', ('12/02/2010', '28.84'), 'A'],
 ['4', ('19/02/2010', '36.45'), 'A']]
```

- The lambda function converts each element of the joined rdd to a concatenated list.

e) Final Unpacking of the elements:

- Initial unpacking could not unpack the tuple elements within the rdd.

```
feature_store_rdd2 = feature_store_rdd1.map(lambda x: [x[0]] + list(x[1]) + [x[2]])
feature_store_rdd2.take(3)
```

```
[['4', '05/02/2010', '43.76', 'A'],
 ['4', '12/02/2010', '28.84', 'A'],
 ['4', '19/02/2010', '36.45', 'A']]
```

- So, same procedure of initial unpacking is repeated at this stage which unpacks all elements of the joined rdd. The output rdd is named as feature_store_rdd2.

f) Grabbing year data:

- Though final unpacking unpacks all elements, year data is not available yet as it is staying within Date.

```
# Unpacking year data from the date
feature_store_rdd3 = feature_store_rdd2.map(lambda x:  x[1].split('/')).map(lambda x: (x[2]))
feature_store_rdd3.take(3)
```

```
['2010', '2010', '2010']
```

- To do so, feature_store_rdd2 is mapped with a lambda function. The lambda function split 'Date' with the parameter '/' used in split command. Then

```
# Unpacking year data from the date
feature_store_rdd3 = feature_store_rdd2.map(lambda x:  x[1].split('/')).map(lambda x: (x[2]))
feature_store_rdd3.take(3)
```

```
['2010', '2010', '2010']
```

3rd value of the split list is selected to grab only year data. The output rdd is named as feature_store_rdd3

- This year data is zipped with other elements of feature_store_rdd2.

```
# Zipping year data with the joined unpacked RDD
feature_store_rdd4 = feature_store_rdd3.zip(feature_store_rdd2)
feature_store_rdd4.take(3)
```

```
[('2010', ['4', '05/02/2010', '43.76', 'A']),
 ('2010', ['4', '12/02/2010', '28.84', 'A']),
 ('2010', ['4', '19/02/2010', '36.45', 'A'])]
```

So, we get rdd that has year, date, size and temperature data and named as feature_store_rdd4

g) Rearranging and removing Date elements:

- To do further calculation, we need Store as the key. So, elements are needed to be rearranged.

```
# Rearranging and removing Date elements
feature_store_rdd5 = feature_store_rdd4.map(lambda x: [x[0]] + x[1]).map (lambda x : (x[1], x[0], x[4], x[3]))
feature_store_rdd5.take(3)
```

```
[('4', '2010', 'A', '43.76'),
 ('4', '2010', 'A', '28.84'),
 ('4', '2010', 'A', '36.45')]
```

- Date element is no longer needed as we already grab year data from the date
- So, map function is imposed on feature_store_rdd4 which use a lambda function as the parameter that unpacks the values.
- Another map is used on top of previous map function. This time, lambda function just selects the required elements that we want for queries.

h) Filtering for store type B and year = 2011:

- Filter method is used to filter out data for Type B and for the year 2011 in the following way:

```
feature_store_rdd_final = feature_store_rdd5.filter(lambda x : x[2] == 'B'). \
                                            filter(lambda x : x[1]=='2011')
```

i) *Finding final rdd with maximum value of Temperature Filtering for store type B and year = 2011*

- Using max function on Temperature using a lambda function as the parameter revealed the final result as shown in the following screenshot.

```
feature_store_rdd_max = feature_store_rdd_final.max(key=lambda x: float(x[3]))
print(feature_store_rdd_max)
```

```
('10', '2011', 'B', '95.36')
```

SEE NEXT PAGE FOR PART-2

Part 2. Working with DataFrames **(50%)**

2.1 Data Preparation and Loading (5%)

2.1.1: Load features, sales and stores data into three separate dataframes. When you create your dataframes, please refer to the metadata file and think about the appropriate data type for each column (Note: you could directly read the date column as the string type)

- At first, different libraries have been called for the data import and subsequent analysis.
- Mainly from pyspark; sql.types and sql.functions are used.
    - From sql.types following libraries are imported:

  StructType, StructField, StringType, IntegerType, DoubleType, DateType, BooleanType
    - StrucType() function is used to add different columns with appropriate data types followed by meta data documents for having three schemas (schema_features, schema_sales, schema_store) on three data sets. The selected data types are StringType for string, IntegerType, DoubleType for float and BooleanType for Boolean data. The schemas are shown below:

Schema of Features DataFrame	**Schema of Sales DataFrame**	**Schema of Stores DataFrame**

`schema_features = StructType() \`
`.add("Store",StringType(),True) \`
`.add("Date",StringType(),True) \`
`.add("Temperature",DoubleType(),True) \`
`.add("Fuel_Price",DoubleType(),True) \`
`.add("MarkDown1",DoubleType(),True) \`
`.add("MarkDown2",DoubleType(),True) \`
`.add("MarkDown3",DoubleType(),True) \`
`.add("MarkDown4",DoubleType(),True) \`
`.add("MarkDown5",DoubleType(),True) \`
`.add("CPI",DoubleType(),True) \`
`.add("Unemployment",DoubleType(),True) \`
`.add("IsHoliday",BooleanType(),True)`
	`schema_sales = StructType() \`
	`.add("Store",StringType(),True) \`
	`.add("Dept",StringType(),True) \`
	`.add("Date",StringType(),True) \`
	`.add("Weekly_Sales",DoubleType(),True) \`
	`.add("IsHoliday",BooleanType(),True)`
		`schema_stores = StructType() \`
		`.add("Store",StringType(),True) \`
		`.add("Type",StringType(),True) \`
		`.add("Size",IntegerType(),True)`

    - Store attribute is imported as integer instead of string datatypes for maintaining the orders on Store column in the next upcoming queries
    - Date - data types are not used for “Date” attribute for next transformation required for 2.2.1. Instead, string datatype is used to import Date attribute.
    - Finally, data is imported using spark.read() method by embedding data format and created schema.

2.1.2 : Display the schema of the features, sales and stores dataframes.

“printSchema()” function is used for each dataframe (feature, sales and store) to show the schema. It shows column name, data type of each column and the column is nullable or not. For simplicity, all columns are considered nullable. An example of the code (about features dataframe) is given bellow:

```
1 df_features.printSchema() # feature schema
```

```
root
 |-- Store: integer (nullable = true)
 |-- Date: string (nullable = true)
 |-- Temperature: double (nullable = true)
 |-- Fuel_Price: double (nullable = true)
 |-- MarkDown1: double (nullable = true)
 |-- MarkDown2: double (nullable = true)
 |-- MarkDown3: double (nullable = true)
 |-- MarkDown4: double (nullable = true)
 |-- MarkDown5: double (nullable = true)
 |-- CPI: double (nullable = true)
 |-- Unemployment: double (nullable = true)
 |-- IsHoliday: boolean (nullable = true)
```

2.2 Query/Analysis (45%)

2.2.1: Transform 'Date' column in both feature and sales dataframe to the date type, after that printout these two DFs schema to show the results.

- Using to_date() function from pyspark.sql. functions within withColumn() , 'Date' column is transformed in both feature and sales dataframe to the date type.
- Built-in format of Spark Date is “yyyy-mm-dd”
- After transforming datatypes, printSchema() command is used to confirm the transformation. A screenshot of the code is given below:

```
1 # Converting datatype of the Date column
2 df1_features = df_features.withColumn('Date', f.to_date('Date', 'dd/MM/yyy'))
3 # showing date datatype for the 'Date' column
4 df1_features.printSchema()
5 print(f"Number of records: {df1_features.count()}")
```

```
root
 |-- Store: integer (nullable = true)
 |-- Date: date (nullable = true)
 |-- Temperature: double (nullable = true)
 |-- Fuel_Price: double (nullable = true)
 |-- MarkDown1: double (nullable = true)
```

- The highlighted Date confirms the datatype transformation.
- here, f is for pyspark.sql.functions

2.2.2 : Calculate the average weekly sales for holiday week and non-holiday week separately, order your result based on the average weekly sales in descending order. Print out the IsHoliday and average sales columns

To make the query, following steps are being used:

- Using groupby() function on IsHoliday column of sales dataframe
- Performing mean aggregation on Weekly_Sales column with aliasing as ‘Average Sales’. For better viewing, round() function used for 3 decimals.
- Finally ordering ‘Average Sales’ in descending order using orderBy function with the parameter ascending = False
- Print out the result using show() command.

The screenshot of the code is given below:

```
Avg_sales = df_sales.groupby("IsHoliday"). \
                        agg(f.round(f.mean("Weekly_Sales"),3).alias('Average Sales')). \
                            orderBy("Average Sales", ascending = False).show(5)
```

```
+---------+-------------+
|IsHoliday|Average Sales|
+---------+-------------+
|     true|    17035.823|
|    false|    15901.445|
+---------+-------------+
```

2.2.3: Based on different years and months, calculate the average weekly sales.

This query is performed in two stages:

- Creating Month and Year column in Sales data frame where Date column is already been transformed to date datatype while working on 2.2.1. The extended Sales Data Frame is shown below:

```
+-----+----+----------+------------+---------+-----+----+
|Store|Dept|      Date|Weekly_Sales|IsHoliday|Month|Year|
+-----+----+----------+------------+---------+-----+----+
|    1|   1|2010-02-05|     24924.5|    false|    2|2010|
|    1|   1|2010-02-12|    46039.49|     true|    2|2010|
+-----+----+----------+------------+---------+-----+----+
only showing top 2 rows
```

- Then calculating average weekly sales using groupby() on Year and Month Column. This step is exactly similar to the previous query i.e. 2.2.2. So, the steps are not elaborated. The code with output is shown below:

```
avgSales_month = df2_sales.groupby("Year", "Month"). \
                 agg(f.round(f.mean("Weekly_Sales"),3).alias('Average Sales')). \
                 orderBy("Year","Month")
avgSales_month.show(15)
```

```
+----+-----+-------------+
|Year|Month|Average Sales|
+----+-----+-------------+
|2010|    2|    16076.779|
|2010|    3|    15432.627|
|2010|    4|    15745.551|
|2010|    5|    15996.482|
|2010|    6|    16486.251|
|2010|    7|    15972.813|
|2010|    8|    16171.689|
|2010|    9|    15120.087|
|2010|   10|    14806.151|
|2010|   11|    17320.131|
|2010|   12|    19570.351|
|2011|    1|    13997.774|
```

2.2.4 : Calculate the average MarkDown1 value in holiday week for all type C store.

The steps for this query are:

- Left joining Store and Features dataframes. Because ‘Type’ and ‘IsHoliday’ columns are not available in same dataframe.
- Filtering out joined dataframe for holiday week and for type c

- Performing mean aggregation on “MarDown1” column and printing out the result using show() command.
- The code with output is shown below:

```
# joining feature and store dataframes
featureStore_df = df_stores.join(df_features, \
                                 df_stores.Store == df_features.Store, \
                                 how='left')

# Filtering joined dataframe for holidays and type c
avgMk = featureStore_df.filter(featureStore_df.IsHoliday == "true"). \
                        filter(featureStore_df.Type == "C")

# Calculating average value of MarkDown1
avgMk.agg(f.mean("MarkDown1"). \
          alias("Average MarkDown1")).show() # Here null values are ignored
```

```
+------------------+
|Average MarkDown1|
+------------------+
|778.2502439024388|
+------------------+
```

- To query Average MarkDown1 for each related stores, following screenshot shows the details:

```
# joining feature and store dataframes
featureStore_df = df_stores.join(df_features, \
                                 df_stores.Store == df_features.Store, \
                                 how='left'). \
                                 select(df_stores["*"],df_features["MarkDown1"],df_features["IsHoliday"])
avgMk = featureStore_df.filter(featureStore_df.IsHoliday == "true"). \
                        filter(featureStore_df.Type == "C")
avgMk.groupby("Store","Type","IsHoliday").agg(f.mean("MarkDown1"). \
          alias("Average MarkDown1")).show()
```

```
+-----+----+---------+------------------+
|Store|Type|IsHoliday| Average MarkDown1|
+-----+----+---------+------------------+
|   42|   C|     true|445.26142857142855|
|   30|   C|     true| 806.7442857142858|
|   44|   C|     true|           1042.64|
|   37|   C|     true|            583.73|
|   43|   C|     true| 629.8657142857144|
|   38|   C|     true|           1199.03|
+-----+----+---------+------------------+
```

- The only difference is that we selected the required columns at first. Then we grouped data with Store using groupby function prior to aggregation.

2.2.5 : Show all stores total sales based on each different month and yearly total in 2011 for every different store, only keep two decimal places after the decimal point.

The steps for this query are:

a) Filtering out sales data for the year of 2011 using the result of first stage of 2.2.3 where Month and Year columns are being created.

```
# Filtering sales dataframe for only the year of 2011
df_sales_2011 = df2_sales.filter(col("Year")==2011)
```

b) Creating a new dataframe from filtered data which has no month column

```
# creating a dataframe from the filtered sales dataframe having no month column
df_sales_noMonth = df_sales_2011.drop("Month")
```

c) Creating two data frames:

- df_weeklySales is created from initial filtered dataset (df_sales_2011). Here total weekly sales are calculated by aggregating Weekly_Sales values for each month and for each store.

```
# Creating datframe for total weekly sales for each store and for each month
df_weeklySales = df_sales_2011.groupby("Store","Month"). \
                                        agg(f.round(f.sum("Weekly_Sales"),2). \
                                            alias("Sales")
                                           )

# Creating datframe for total weekly sales for each store only (That means for all months)
df_weeklysales_allMonths = df_sales_noMonth.withColumn("Month",f.lit(" ")).groupby("Store","Month"). \
                                                        agg(f.round(f.sum("Weekly_Sales"),2). \
                                                            alias ("Sales")
                                                           )
```

- df_weeklysales_allMonths is created from no months table generated in b) step. Here total weekly sales are calculated by aggregating Weekly_Sales values for each store only. So, sales figure here is for all months for a particular store.
- A new Month column with empty string value for all rows are created using lit() function within the “withColumn()” function.

d) Joining these two dataframe vertically using union() command. So, the joined dataset has Sales value for individual month as well as for all months.

```
# Joining the above two dataframes vertically using union
df_totalSales = df_weeklySales.union(df_weeklysales_allMonths)
# Let's check the schema of the dataframe
df_totalSales.printSchema()
```

```
root
 |-- Store: integer (nullable = true)
 |-- Month: string (nullable = true)
 |-- Sales: double (nullable = true)
```

e) The Month column is string type as shown above. So, it is needed to convert in integer type so that we can get each months ordered. This will convert empty space for total sales to null value.

```
# Converting month datatype to Integer type for ordering Month column
# This will convert all string value in Month column to null values
df_totalSales = df_totalSales.withColumn("Month", df_totalSales["Month"].cast(IntegerType()))
df_totalSales.printSchema() # This confirms that Month column is integer
```

```
root
 |-- Store: integer (nullable = true)
 |-- Month: integer (nullable = true)
 |-- Sales: double (nullable = true)
```

f) We at first sorted Total Sales based on Store and Month columns as shown in the following screenshot. Then Month column is converted into StringType again so that we can implement fillna() command to convert null value with the value "Total".

```
# At first ordering dataframe, then converting Month column to string
# finally null values are converted to Total value
df_totalSales = df_totalSales.orderBy("Store","Month"). \
                                        withColumn("Month", df_totalSales["Month"].cast(StringType())). \
                                        fillna("Total",["Month"]). \
                                        show(truncate=False)
```

g) Show() command confirms that the query result is in appropriate format.

```
+-----+-----+-------------+
|Store|Month|Sales        |
+-----+-----+-------------+
|1    |Total|8.092191883E7|
|1    |1    |5480050.97   |
|1    |2    |6399887.57   |
|1    |3    |6307375.48   |
|1    |4    |7689123.6    |
|1    |5    |6128431.8    |
|1    |6    |6194971.74   |
|1    |7    |7227654.31   |
|1    |8    |6144985.73   |
|1    |9    |7379542.34   |
|1    |10   |6072327.75   |
|1    |11   |6864972.83   |
|1    |12   |9032594.71   |
|2    |Total|9.860788142E7|
|2    |1    |6949000.95   |
|2    |2    |8011783.74   |
```

2.2.6 : Draw a scatter plot to show the relationship between weekly sales and unemployment rate, use the different colour for the holiday week point. After that, discuss your findings based on the scatter plot.

Steps:

a) At first, left joining sales and feature datasets based on two columns: 'Store' and 'Date' and named the joined dataframe as df_employment. Thus we obtained a dataframe which has dates only related to Sales (not for temperature). The code is shown below:

```
df_employment = df1_sales.join( \
                                df1_features, \
                                (df1_sales.Store == df1_features.Store) & (df1_sales.Date == df1_features.Date) , \
                                how ="left")
```

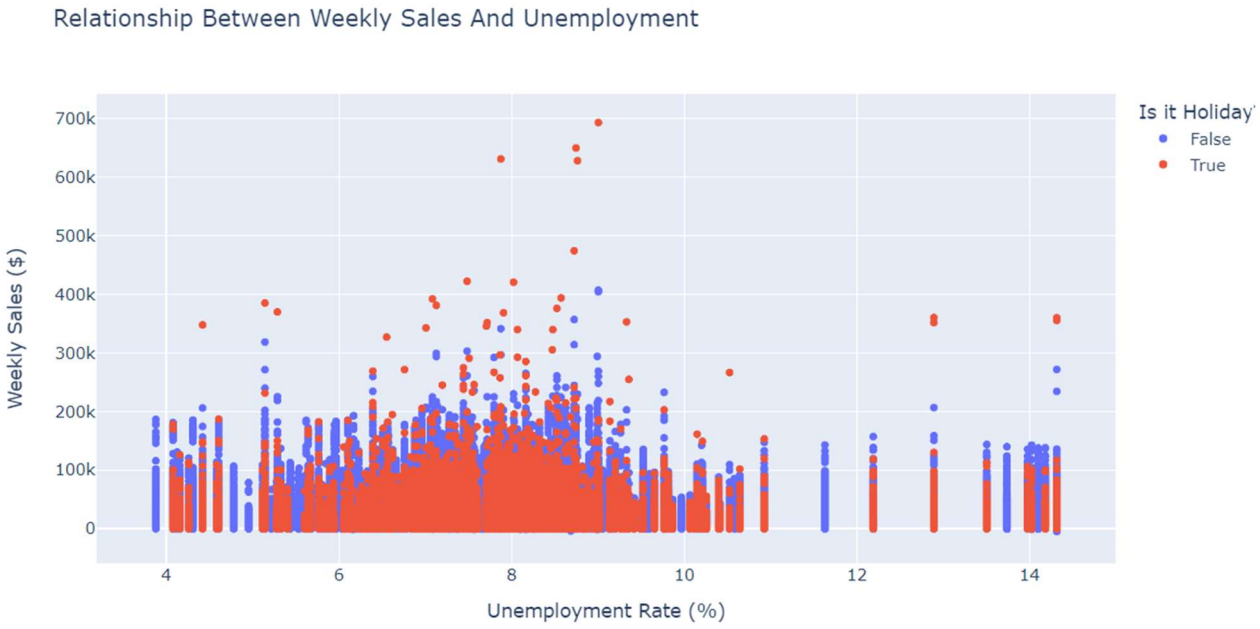
b) Converting df_employment to pandas dataframe using toPandas() method and named the pandas dataframe as df_employment_pd

c) Detailed explanation of visualisation code is shown below with the code screenshot:
   - Installation of plotly
   - Importing express library from plotly

```
#! pip install plotly
import plotly.express as px
df = df_employment_pd
c = df_employment_pd.iloc[:,4] # Choosing IsHoliday as the colour
fig = px.scatter(df, x="Unemployment", y="Weekly_Sales", color=c,
                 title="Relationship Between Weekly Sales And Unemployment",
                 labels={
                  "Unemployment": "Unemployment Rate (%)",
                  "Weekly_Sales": "Weekly Sales ($)",
                  "color": "Is it Holiday?"},)
fig.show()
```

   - Choosing IsHoliday attribute for the colour and assigned it to ‘c’ variable
   - Using scatter() function from plotly express to draw the plot using following parameters:
     - X-axis for Unemployment
     - Y-axis for Weekly_Sales
     - Color: c (which is “IsHoliday”)
     - labels: for labelling axis and legends
     - titles: for providing a title to the plot

A screenshot of the visualisation is show below:



The scatter plot confirms that:

- There is no linear relationship between unemployment and weekly sales.
- Majority of the weekly sales are below 200K for any unemployment rates.
- Non-holiday sales are more than holiday sales. (Probable reason is that there are more non-holidays than holidays in a calendar year)
- When unemployment rate surpasses 10%, weekly sales hardly go beyond 200K.
- Most of the sales over 300K are happened during holidays with few exceptions.
- All the very high sales (above 400K) happened when unemployment rate is in between 7% to 9.5 % and it happens during holidays.

Part3: RDDs vs DataFrame vs Spark SQL (20%)

Q: 3.1: Query: Calculate the average weekly fuel price for all stores' size larger than 150000.

Answer/ Explanation:

For RDD:

Following steps are being followed for the successful query using RDD concepts:

a) Two functions are being created to be implemented on Stores and Features RDDs consecutively. These functions are:
   - parseRecordStores () : which returns a tuple of 'Store' and 'Size' values.
   - parseRecordFeatures ()] : which returns a tuple of 'Store' and 'Fuel_Price' values.

```
# Implement function with logic to be applied to the RDDs

def parseRecordStores(line):
    # Split line separated by comma
    array_line = line.split(',')
    # Return a tuple with the store id as first element and store type as the second element
    return (array_line[0], array_line[2])

stores_rdd1 = stores_rdd.map(parseRecordStores) # implementing the function on stores rdd


def parseRecordFeatures(line):
    # Split line separated by comma
    array_line = line.split(',')
    # Return a tuple with the store id as first element and fuel_price as the second element
    return (array_line[0], array_line[3])

features_rdd1 = features_rdd.map(parseRecordFeatures) # implementing the function on featues rdd
```

So, first element is 'Store' for both the RDDs generated by implementing the functions on Store and Features RDD

b) Next 'Size' value of Store RDD is filtered to remove any value from the RDD which is less than 150,000.

c) Both the Store and Feature RDDs are being joined

```
%%time
# Filtering for the stores-size
store_rdd1 = stores_rdd1.filter (lambda x : int(x[1])>=150000)
# Joining feature and filtered store RDDs
feature_store_rdd = store_rdd1.join(features_rdd1)
# Unpacking elements of the joined rdd
feature_store_unpacked_rdd = feature_store_rdd.map(lambda x : [x[0]]+ list(x[1]))
# Calculating average fuel price
feature_store_rdd_avg = feature_store_unpacked_rdd.map(lambda x: float(x[2])).mean()
# Showing results
feature_store_rdd_avg
```

```
CPU times: user 3.54 ms, sys: 17.4 ms, total: 21 ms
Wall time: 247 ms

3.3882162087912087
```

d) Then map () function is used to unpack each element of the joined RDD

e) Finally average fuel price is calculated using map () and mean() function. To convert the string nature of Fuel_Price, float () function is used within lambda function embedded with map () function.
f) Result is Average fuel price :3.388

For DataFrame:

Following steps are being followed for the successful query using DataFrame:

g) Size' value of Store DataFrame is filtered to remove any value which is less than 150,000.

```
%%time

# Filter out store size less than 150000
df_stores = df_stores.filter(col("Size")>=150000)

# joining stores and features dataframe
featureStore_df = df_stores.join(df_features, \
                                 df_stores.Store == df_features.Store, \
                                 how='left')

# Calculating average fuel price
featureStore_avg_df = featureStore_df. \
                                 agg(f.mean("Fuel_Price"). \
                                       alias("Average Fuel Price")
                                    )

featureStore_avg_df.show()
```

```
+------------------+
|Average Fuel Price|
+------------------+
|3.3882162087912087|
+------------------+

CPU times: user 7.34 ms, sys: 5.47 ms, total: 12.8 ms
```

h) Both the Store and Feature Data Frames are being joined on 'Store' columns.
i) Finally average fuel price is calculated using mean() function on Fuel_Price column of the joined DataFrame.
j) Result is Average fuel price :3.388

For SparkSQL:

Following steps are being followed for the successful query using SparkSQL:

k) Two views are being created based on Stores and Features DataFrames consecutively. These views are:
   - sql_stores: built on Stores dataframe using createOrReplaceTempView () function.
   - sql_features: built on Features dataframe using createOrReplaceTempView () function.

```
%%time

df_stores.createOrReplaceTempView("sql_stores")
df_features.createOrReplaceTempView("sql_features")

# joining stores and features dataframe
# # Filter out store size less than 150000
# Calculating average fuel price

sql_featureStore = spark.sql('''
      SELECT AVG(Fuel_Price) as Avg_Fuel_Price
          FROM (
                  SELECT f.Fuel_Price
                  FROM sql_stores s LEFT JOIN sql_features f
                  ON s.Store = f.Store
                  WHERE s.Size >=150000)
                              ''')
sql_featureStore.collect()
```

```
CPU times: user 841 µs, sys: 19.9 ms, total: 20.8 ms
Wall time: 538 ms
```

l) Then spark.sql() function is used inside following steps are being followed:
m) An inner SQL query where two views are left joined on 'Stores' attribute, then filtered the inner query result using expressions : Store.Size>=150000
n) Next in outer SQL query, calculation of average fuel price is performed using AVG() function
o) Result is Average fuel price: 3.388

Q: 3.2: Also talk about this question in a detailed answer, "Why is RDD faster/slower than DF?"

Answer/ Explanation:

The time taken by RDD vs DataFrame vs SparkSQL structures for the given query is shown below:

Process Type	CPU Time		CPU Total Time (ms)	Wall Time (ms)
	User (ms)	Sys (ms)		
RDD	24.7	4.1	28.8	273
DataFrame	11.3	0.36	11.6	425
SparkSQL	10.4	0.707	11.1	607

CPU Time Analysis:

So, the above table conforms that DataFrame and SparkSQL are at least 200 %faster than RDD queries in terms of CPU time.
The reason is there is a performance limitation in RDD. We know in RDD, data is stored in memory to compute results. Being in-memory jvm objects, RDDs involve working with two

types of expensive overheads:

- overhead of ‘Garbage Collection’ and
- Java (or little better Kryo) Serialisation which are expensive when data grows.

On the other hand, according to Chandan Prash’s Blog “DataFrame offers huge performance improvement over RDDs because of 2 powerful features it has.

1. Custom Memory management (aka Project Tungsten)

Data is stored in off-heap memory in binary format. This saves a lot of memory space. There is also no ‘Garbage Collection’ overhead involved. By knowing the schema of data in advance and storing efficiently in binary format, expensive java Serialization is also avoided.

2. Optimized Execution Plans (aka Catalyst Optimizer)

Query plans are created for execution using Spark catalyst optimiser. After an optimised execution plan is prepared going through some steps, the final execution happens internally on RDDs only but that’s completely hidden from the users.”

Wall Time Analysis:

In terms of ‘wall time’, Rdd is faster than DataFrame and spark SQL.

Wall-clock time is the time that a clock on the wall (or a stopwatch in hand) would measure as having elapsed between the start of the process and 'now'.

Spark is best known for RDD, where a data can be stored in-memory and transformed based on the needs. Whereas dataframes are not stored as the data are being utilized in RDD. When data stored in the RDD (Similar to cache), spark can access fast than data stored as dataframe. RDD has specialized memory on which data can be stored and retrieved. Data are stored as partitions of chunks which enables parallelism of IO unlike DF which is not coupled with spark as a RDD does. Whenever we read a data from RDD due to partitions of data chunks and parallelism multiple threads will be hitting the data to perform IO operations which makes it faster than DF.

4. REFERENCE:

FIT5202 - Getting started with Apache SparkFile, Session 1 Moodle, FIT5202 Data processing for big data - Summer B 2023. Retrieved January 12,2023 from
https://lms.monash.edu/course/view.php?id=149349&section=4

FIT5202 - Parallel SearchFile, Session 2 Moodle, FIT5202 Data processing for big data - Summer B 2023. Retrieved January 12,2023 from
https://lms.monash.edu/course/view.php?id=149349&section=7

MyCloudera, Support Questions, *RDD vs DataFrame* Viewed on January 12, 2013 from
https://community.cloudera.com/t5/Support-Questions/RDD-vs-DATAFRAME/td-p/229114

Prakash, C. (2016). Apache Spark: RDD vs Dataframe vs Dataset. Viewed on January 12,2023 from
http://why-not-learn-something.blogspot.com.au/2016/07/apache-spark-rdd-vs-dataframe-vs-dataset.html