A1-TimothyJordan

August 25, 2023

1 FIT5202 Assignment 1 : Analysing eCommerce Data

1.1 Table of Contents

2 Part 1 : Working with RDDs

2.1 1.1 Working with RDD

In this section, you will need to create RDDs from the given datasets, perform partitioning in these RDDs and use various RDD operations to answer the queries for retail analysis.

2.1.1 1.1.1 Data Preparation and Loading

Write the code to create a SparkContext object using SparkSession. To create a SparkSession you first need to build a SparkConf object that contains information about your application, use 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.

[201]:
```
from datetime import datetime, date
from pyspark.sql.functions import col, udf, to_date, hour, col, udf,␣
 ↪collect_list, row_number
from pyspark.sql.functions import round as spark_round
from pyspark.sql.functions import sum as spark_sum
from pyspark.sql.functions import count as spark_count
from pyspark.sql.types import StringType, DateType, IntegerType, FloatType,␣
 ↪StringType, TimestampType
from pyspark.sql.window import Window
import matplotlib
import matplotlib.pyplot as plt
import math
```

[202]:
```
# 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
# If we want Spark to run locally with 'k' worker threads, we can specify as␣
 ↪"local[k]".
master = "local[*]"
# The `appName` field is a name to be shown on the Spark cluster UI page
app_name = "eCommerce Analysis"
# Setup configuration parameters for Spark
spark_conf = SparkConf().setMaster(master).setAppName(app_name)

#TODO : Initialize Spark Session and create a SparkContext Object
# Import SparkContext and SparkSession classes
from pyspark import SparkContext # Spark
from pyspark.sql import SparkSession # Spark SQL

spark = SparkSession.builder.config(conf=spark_conf).getOrCreate()
sc = spark.sparkContext
sc.setLogLevel('ERROR')

# Set Melbourne time as timezone
spark.conf.set("spark.sql.session.timeZone", "Australia/Melbourne")
```

1.1.2 Load CUPS csv files into four RDDs.

We first load the CUPS csv files

[203]:
```
category_rdd = sc.textFile('sales/category.csv')
product_rdd = sc.textFile('sales/product.csv')
sales_rdd = sc.textFile('sales/sales.csv')
users_rdd = sc.textFile('users/users.csv')
```

Then we perform data processing

[204]:
```
# separate the field values into multiple columns
category_rdd = category_rdd.map(lambda x: x.split(','))
product_rdd = product_rdd.map(lambda x: x.split(','))
sales_rdd = sales_rdd.map(lambda x: x.split(','))
users_rdd = users_rdd.map(lambda x: x.split(','))
```

1.1.3 For each RDD, remove the header rows and display the total count and first 10 records. (Hint: You can use csv.reader to parse rows into RDDs.)

We first remove the header rows

[205]:
```
# Retrieve header values
category_header = category_rdd.first()
```

```
product_header = product_rdd.first()
sales_header = sales_rdd.first()
users_header = users_rdd.first()
```

[206]:
```
# Filter out header values
category_rdd = category_rdd.filter(lambda x: x != category_header)
product_rdd = product_rdd.filter(lambda x: x != product_header)
sales_rdd = sales_rdd.filter(lambda x: x != sales_header)
users_rdd = users_rdd.filter(lambda x: x != users_header)
```

Print total count for each RDD

[36]:
```
print("Total count on RDDs")
print(f"category_rdd: {category_rdd.count()}")
print(f"product_rdd: {product_rdd.count()}")
print(f"sales_rdd: {sales_rdd.count()}")
print(f"users_rdd: {users_rdd.count()}")
```

```
Total count on RDDs
category_rdd: 1464
product_rdd: 208290
sales_rdd: 6848824
users_rdd: 15639803
```

[37]:
```
print(f"category_rdd first 10 records:\n{category_rdd.take(10)}")
print()
print(f"product_rdd first 10 records:\n{product_rdd.take(10)}")
```

```
category_rdd first 10 records:
[['2090971686529663114', 'appliances.environment.vacuum'],
['2232732116498514828', 'apparel.jeans'], ['2232732109628244704',
'apparel.shirt'], ['2232732103294845523', 'apparel.shoes.step_ins'],
['2232732086500851925', 'apparel.scarf'], ['2232732100660822557',
'country_yard.cultivator'], ['2053013558282682943', 'construction.tools.drill'],
['2053013562527318829', 'furniture.living_room.cabinet'],
['2110937143172923797', 'construction.tools.light'], ['2074462942123786261',
'kids.toys']]

product_rdd first 10 records:
[['100168127', '2053013554096767303', '', '30.8171196068388'], ['7101887',
'2232732103764607583', '', '33.6457534471799'], ['100058603',
'2053013555438944659', '', '32.1982006550096'], ['100007811',
'2232732112782361392', 'etro', '300.111251624047'], ['26400508',
'2053013553056579841', '', '139.458912588293'], ['100215722',
'2134905019189691101', '', '4.15035271832127'], ['32402016',
'2232732115777094520', '', '5.90104832419485'], ['13201248',
'2232732061804790604', 'brw', '400.468187201711'], ['2701879',
'2053013563911439225', 'beko', '249.969115457257'], ['15300266',
```

```
'2232732107698864813', '', '2.85509519214549']]
```

[38]:
```
print(f"sales_rdd first 10 records:\n{sales_rdd.take(10)}")
print()
print(f"users_rdd first 10 records:\n{users_rdd.take(10)}")
```

```
sales_rdd first 10 records:
[['338156802', '2020-03-27 02:03:48.000000 +00:00', '4804056', '171.56',
'513119357', '0ad58441-9db9-48bd-9ee7-accf603f06e0'], ['338156808', '2020-03-27
02:03:49.000000 +00:00', '1005223', '209.53', '522277649',
'2ecb2725-2c40-4acb-8ef1-05e39ea4c2cc'], ['338156913', '2020-03-27
02:04:11.000000 +00:00', '1005212', '174.25', '632847510',
'43609582-1818-4885-bd72-3791f3a4ec93'], ['338156921', '2020-03-27
02:04:13.000000 +00:00', '1005236', '231.15', '626579483',
'bbf972ef-8f8d-4a33-bcbd-ac09af67ea40'], ['338156953', '2020-03-27
02:04:18.000000 +00:00', '1005212', '174.25', '633245502',
'90166e0d-d7da-4997-b479-bc3a0fd724d6'], ['338156985', '2020-03-27
02:04:24.000000 +00:00', '100132156', '47.5', '541680528',
'4cd111cc-8716-4720-b7e3-bc3c8ddce98f'], ['338157055', '2020-03-27
02:04:37.000000 +00:00', '3500009', '290.84', '618162129',
'd0bda424-a759-4941-b9be-5091c28009db'], ['338157122', '2020-03-27
02:04:48.000000 +00:00', '2300214', '386.08', '626601722',
'ce913646-936a-4b2e-b6bb-9f6e0d781bc5'], ['338157141', '2020-03-27
02:04:52.000000 +00:00', '100070443', '35.01', '604369152', 'fd55e2cd-
dc42-4bf9-a0a1-f207f1c66e5d'], ['338157271', '2020-03-27 02:05:14.000000
+00:00', '1306659', '431.09', '616395888',
'a934f0d6-5fa6-475d-83df-528364fc8c31']]

users_rdd first 10 records:
[['514771925', 'MERVYN', 'Unknown', '1-5-1963', '30A LUCINDA AVENUE', '2260',
'NSW', 'male', 'WAMBERAL'], ['531947692', 'FALLON', 'Unknown', '16-10-2002', '82
CAMBERWARRA DRIVE', '6025', 'WA', 'female', 'CRAIGIE'], ['560231306', 'KALIN',
'Unknown', '19-8-2015', '74 SOLDIERS ROAD', '2281', 'NSW', 'male', 'PELICAN'],
['575248835', 'PIETTA', 'Unknown', '19-1-1962', '"UNIT 1', ' 1 LARK AVENUE"',
'5023', 'SA', 'female', 'SEATON'], ['575243330', 'SUMMAH', 'Unknown',
'19-4-1973', '44 CLARENDON ROAD', '2048', 'NSW', 'female', 'STANMORE'],
['516606243', 'KEIRA', 'Unknown', '9-6-2011', '1 VIOLA CLOSE', '4868', 'QLD',
'female', 'BAYVIEW HEIGHTS'], ['545807768', 'VERITY', 'Unknown', '1-2-1980',
'31-33 KING STREET', '3550', 'VIC', 'female', 'BENDIGO'], ['529118965',
'MAXWELL', 'Unknown', '16-5-2012', '15 HARWELL WAY', '6721', 'WA', 'male',
'WEDGEFIELD'], ['518625399', 'NATASHIA', 'Unknown', '5-6-1973', '"UNIT 3', ' 17
BOX STREET"', '6530', 'WA', 'female', 'WEBBERTON'], ['518273873', 'JUSTINE',
'Unknown', '27-8-2007', '"UNIT 909', ' 112 GODERICH STREET"', '6004', 'WA',
'male', 'EAST PERTH']]
```

1.1.4 Drop unnecessary columns from RDDs: firstname, lastname, user_session.

We first Determine which index's the store column values

[39]:
```
for index, field in enumerate(users_header):
    print(f"index: {index}, field: {field}")
```

```
index: 0, field: user_id
index: 1, field: firstname
index: 2, field: lastname
index: 3, field: dob
index: 4, field: address
index: 5, field: postcode
index: 6, field: state
index: 7, field: sex
index: 8, field: suburb
```

firstname and lastname are index 1 and 2 respectively

[40]:
```
for index, field in enumerate(sales_header):
    print(f"index: {index}, field: {field}")
```

```
index: 0, field: id
index: 1, field: sales_timestamp
index: 2, field: product_id
index: 3, field: price
index: 4, field: user_id
index: 5, field: user_session
```

user_session is index 5

Next, the requested columns are dropped in the `users_rdd` and `sales_rdd` RDDs

[207]:
```
users_rdd = users_rdd.map(lambda x: x[:1]+x[3:])
sales_rdd = sales_rdd.map(lambda x: x[:5])
```

We also remove the duplicate product IDs and Category IDs found in `product_rdd` and `category_rdd` respectively

[208]:
```
# Retrieve set of largest category IDs for each product id
unique_product_categories = set(product_rdd.map(lambda x: [x[0] , int(x[1])]).
 ↪groupByKey().mapValues(lambda x: max(x)).collect())
bd_unique_product_categories = sc.broadcast(unique_product_categories)

# Create function to filter product_rdd rows in this set
def filter_largest_category(row):
    val_tuple = (row[0], int(row[1]))
    return val_tuple in bd_unique_product_categories.value

# Apply filter
product_rdd = product_rdd.filter(filter_largest_category)

# Retrieve set of longest string of brand for each product id
```

```
unique_product_brands = product_rdd.map(lambda x: [(x[0], x[1]), x[2]])\
        .groupByKey().mapValues(max)\
        .map(lambda x: (x[0][0], x[0][1], x[1])).collect()
bd_unique_product_brands = sc.broadcast(unique_product_brands)

# Create function to filter product_rdd rows in this set
def filter_longest_brand(row):
    val_tuple = (row[0], row[1], row[2])
    return val_tuple in bd_unique_product_brands.value

# Apply filter
product_rdd = product_rdd.filter(filter_longest_brand)
```

[209]:
```
# Create UDF function to retrieve the longest category code string
def get_longest_string(ls):
    res = ""
    for val in ls:
        if len(val) > len(res):
            res = val
    return res
```

[210]:
```
unique_categories = set(category_rdd.groupByKey().mapValues(get_longest_string).
 ↪collect())
broadcast_unique_categories = sc.broadcast(unique_categories)

def filter_categories(row):
    val_tuple = (row[0], row[1])
    return val_tuple in broadcast_unique_categories.value

category_rdd = category_rdd.filter(filter_categories)
```

2.1.2 1.2 Data Partitioning in RDD

1.2.1 For each RDD, print out the total number of partitions and the number of records in each partition. Answer the following questions: 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? (Hint: search the Spark source code to try to answer this question.) Write code and your explanation in Markdown cells.

How many partitions do the above RDDs have?

First, we create a function to record number of records in each partition

[211]:
```
def get_partition_count(partition):
    count = 0
    for x in partition:
        count += 1
```

```
    yield count

def output_partition_res(res):
    for index, count in enumerate(partition_counts):
        print(f"partition {index}: {count} records")
```

[178]:
```
category_rdd.take(2)
```

[178]:
```
['category_id,category_code',
 '2090971686529663114,appliances.environment.vacuum']
```

[46]:
```
print("category_rdd partitions")
partition_counts = category_rdd.mapPartitions(get_partition_count).collect()
output_partition_res(partition_counts)
```

```
category_rdd partitions
partition 0: 647 records
partition 1: 611 records
```

[12]:
```
print("product_rdd partitions")
partition_counts = product_rdd.mapPartitions(get_partition_count).collect()
output_partition_res(partition_counts)
```

```
product_rdd partitions
partition 0: 81738 records
partition 1: 77586 records
```

[13]:
```
print("sales_rdd partitions")
partition_counts = sales_rdd.mapPartitions(get_partition_count).collect()
output_partition_res(partition_counts)
```

```
sales_rdd partitions
partition 0: 315832 records
partition 1: 315940 records
partition 2: 316009 records
partition 3: 319748 records
partition 4: 319808 records
partition 5: 319827 records
partition 6: 319861 records
partition 7: 319855 records
partition 8: 317174 records
partition 9: 316760 records
partition 10: 316656 records
partition 11: 316733 records
partition 12: 316682 records
partition 13: 316607 records
partition 14: 316439 records
partition 15: 316229 records
```

```
partition 16: 316253 records
partition 17: 316229 records
partition 18: 316133 records
partition 19: 316127 records
partition 20: 316023 records
partition 21: 187899 records
```

[14]:
```
print("users_rdd partitions")
partition_counts = users_rdd.mapPartitions(get_partition_count).collect()
output_partition_res(partition_counts)
```

```
users_rdd partitions
partition 0: 407684 records
partition 1: 407751 records
partition 2: 407663 records
partition 3: 407640 records
partition 4: 407740 records
partition 5: 407698 records
partition 6: 407764 records
partition 7: 407723 records
partition 8: 407732 records
partition 9: 407782 records
partition 10: 407851 records
partition 11: 407787 records
partition 12: 407659 records
partition 13: 407782 records
partition 14: 407827 records
partition 15: 407766 records
partition 16: 407888 records
partition 17: 407701 records
partition 18: 407788 records
partition 19: 407748 records
partition 20: 407745 records
partition 21: 407862 records
partition 22: 407724 records
partition 23: 407781 records
partition 24: 407785 records
partition 25: 407745 records
partition 26: 407680 records
partition 27: 407655 records
partition 28: 407691 records
partition 29: 407780 records
partition 30: 407737 records
partition 31: 407740 records
partition 32: 407693 records
partition 33: 407728 records
partition 34: 407713 records
partition 35: 407799 records
```

```
partition 36: 407740 records
partition 37: 407682 records
partition 38: 145549 records
```

Based on the above ouputs, `category_rdd` has 2 partitions, `product_rdd` has 2 partitions, `sales_rdd` has 22 partitions and `users_rdd` has 39 partitions

How is the data in these RDDs partitioned by default, when we do not explicitly specify any partitioning strategy?

The default partitioning strategy used is random-equal partitioning strategy. When using the `textFile` method to create an RDD, Spark creates on partition for each block of the file, which is 128MB by default.

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? (Hint: search the Spark source code to try to answer this question)

It is partitioned this number because each partition by default contains at most 1 block of the csv files whic is 128MB. Doing so allows the load to be evenly distributed amongst each partition and that each core of the CPU is being used effectively when performing parallel computations.

If I only have a single core CPU in my PC, then the default partition number will be 1. This is because PySpark is configured to partition the data based on the number of CPU cores in a PC. PySpark makes full use of the CPU processing cores to perform optimised parallel processing on any queries or operations we execute.

. . For a PC with only a single core CPU, then there will only be one partition.

1.2.2. Create a user defined function (UDF) to transform category_code to capitalized words. (e.g. apparel.shoes.ballet_shoes shall be converted to "Apparel Shots Ballet_shoes).

Do we use a user-defined function on an RDD? is it fine to just use the function?

Firstly, we create the user defined function

[212]:
```
def category_format(x):
    category_code = x
    words = category_code.split('.')
    for i in range(len(words)):
        words[i] = words[i].capitalize()
    return ' '.join(words)
category_format_udf = udf(category_format, StringType())
```

[213]:
```
category_rdd = category_rdd.map(lambda x: (x[0], category_format(x[1])))
```

1.2.3. Join Product and Category RDDs and Create a new key value RDD, using brand as the key and all of the categories of that brand as the value. Print out the first 5 records of the key-value RDD.

We retrieve relevant column values from `product_rdd` and `category_rdd`

[214]:
```
product_tuple_rdd = product_rdd.map(lambda x: tuple(x[1:3]))
category_tuple_rdd = category_rdd.map(lambda x: tuple(x))
```

We perform a join operation on the `category_id` and rearrange column RDD to retrieve brand to category_code pair values

[215]:
```
product_category_join = category_tuple_rdd.join(product_tuple_rdd)
product_category_join = product_category_join.map(lambda x: (x[1][1], x[1][0]))
```

Lastly, we perform a group by operation on `brand` and return a list of `category_code` for each brand

[216]:
```
brand_categories_rdd = product_category_join.groupByKey().mapValues(lambda x:␣
↪tuple(set(x)))
```

[113]:
```
brand_categories_rdd.take(5)
```

[113]:
```
[('domani-spa',
  ('Furniture Living_room Cabinet',
   'Electronics Clocks',
   'Furniture Bathroom Bath')),
 ('',
  ('Sport Tennis',
   'Apparel Shirt',
   'Appliances Kitchen Blender',
   'Appliances Kitchen Juicer',
   'Appliances Environment Water_heater',
   'Furniture Living_room Cabinet',
   'Apparel Shoes Keds',
   'Computers Peripherals Printer',
   'Appliances Kitchen Coffee_grinder',
   'Stationery Cartrige',
   'Kids Toys',
   'Electronics Telephone',
   'Apparel Underwear',
   'Apparel Shoes Slipons',
   'Computers Components Cdrw',
   'Electronics Audio Subwoofer',
   'Computers Components Sound_card',
   'Appliances Kitchen Washer',
   'Apparel Jacket',
   'Furniture Universal Light',
   'Construction Tools Welding',
   'Electronics Camera Photo',
   'Computers Components Videocards',
   'Apparel Trousers',
   'Kids Dolls',
   'Construction Tools Painting',
```

```
'Kids Carriage',
'Construction Tools Saw',
'Country_yard Watering',
'Furniture Living_room Sofa',
'Construction Tools Light',
'Sport Snowboard',
'Appliances Iron',
'Apparel Skirt',
'Appliances Kitchen Dishwasher',
'Construction Tools Axe',
'Furniture Bathroom Toilet',
'Furniture Kitchen Chair',
'Construction Components Faucet',
'Appliances Kitchen Hob',
'Appliances Environment Air_conditioner',
'Electronics Smartphone',
'Appliances Personal Hair_cutter',
'Appliances Kitchen Grill',
'Auto Accessories Light',
'Construction Tools Generator',
'Appliances Kitchen Microwave',
'Kids Swing',
'Construction Tools Pump',
'Sport Ski',
'Apparel Dress',
'Apparel Shoes Espadrilles',
'Apparel Glove',
'Auto Accessories Compressor',
'Medicine Tools Tonometer',
'Appliances Sewing_machine',
'Appliances Kitchen Mixer',
'Computers Components Motherboard',
'Appliances Kitchen Meat_grinder',
'Apparel Sock',
'Apparel Pajamas',
'Auto Accessories Player',
'Electronics Audio Microphone',
'Computers Peripherals Mouse',
'Construction Tools Screw',
'Apparel Tshirt',
'Computers Notebook',
'Appliances Kitchen Refrigerators',
'Apparel Belt',
'Electronics Camera Video',
'Electronics Audio Acoustic',
'Electronics Audio Music_tools Piano',
'Electronics Clocks',
```

```
'Apparel Jumper',
'Appliances Kitchen Steam_cooker',
'Apparel Shoes',
'Computers Components Cpu',
'Auto Accessories Parktronic',
'Apparel Scarf',
'Appliances Environment Fan',
'Electronics Audio Dictaphone',
'Appliances Kitchen Fryer',
'Sport Bicycle',
'Appliances Kitchen Toster',
'Furniture Living_room Shelving',
'Apparel Shorts',
'Appliances Ironing_board',
'Accessories Wallet',
'Appliances Personal Scales',
'Auto Accessories Alarm',
'Auto Accessories Winch',
'Computers Components Memory',
'Computers Ebooks',
'Appliances Kitchen Oven',
'Construction Tools Soldering',
'Appliances Environment Climate',
'Computers Components Hdd',
'Kids Skates',
'Apparel Costume',
'Auto Accessories Videoregister',
'Appliances Environment Air_heater',
'Computers Peripherals Camera',
'Electronics Video Tv',
'Computers Peripherals Keyboard',
'Furniture Bathroom Bath',
'Apparel Shoes Ballet_shoes',
'Apparel Jeans',
'Furniture Living_room Chair',
'Furniture Bedroom Pillow',
'Country_yard Lawn_mower',
'Others',
'Appliances Environment Vacuum',
'Computers Desktop',
'Apparel Shoes Moccasins',
'Computers Components Power_supply',
'Furniture Bedroom Blanket',
'Electronics Video Projector',
'Apparel Shoes Sandals',
'Accessories Umbrella',
'Computers Components Cooler',
```

```
  'Appliances Kitchen Kettle',
  'Apparel Shoes Step_ins',
  'Kids Fmcg Diapers',
  'Country_yard Cultivator',
  'Accessories Bag',
  'Country_yard Furniture Hammok',
  'Electronics Tablet',
  'Appliances Kitchen Hood',
  'Auto Accessories Anti_freeze',
  'Stationery Paper',
  'Electronics Audio Headphone',
  'Construction Tools Drill',
  'Auto Accessories Radar',
  'Sport Trainer',
  'Appliances Personal Massager',
  'Furniture Kitchen Table',
  'Furniture Bedroom Bed')),
('midea',
 ('Auto Accessories Videoregister',
  'Appliances Environment Air_heater',
  'Appliances Kitchen Hob',
  'Electronics Clocks',
  'Appliances Kitchen Blender',
  'Appliances Environment Air_conditioner',
  'Appliances Environment Water_heater',
  'Construction Tools Generator',
  'Appliances Kitchen Microwave',
  'Kids Swing',
  'Others',
  'Appliances Environment Vacuum',
  'Furniture Bedroom Blanket',
  'Appliances Kitchen Washer',
  'Computers Components Cooler',
  'Construction Tools Welding',
  'Electronics Camera Photo',
  'Appliances Kitchen Oven',
  'Appliances Kitchen Hood',
  'Apparel Tshirt',
  'Appliances Kitchen Refrigerators',
  'Appliances Personal Massager',
  'Apparel Belt',
  'Appliances Iron',
  'Appliances Kitchen Dishwasher',
  'Apparel Costume')),
('toro',
 ('Others',
  'Electronics Clocks',
```

```
    'Apparel Tshirt',
    'Apparel Shoes',
    'Apparel Shoes Keds',
    'Computers Peripherals Printer',
    'Construction Tools Generator',
    'Accessories Bag')),
 ('dyflon', ('Apparel Tshirt', 'Construction Tools Generator'))]
```

2.1.3 1.3 Query/Analysis

For this part, write relevant RDD operations to answer the following queries.

1.3.1 Calculate the average daily sales for each year, each month. Print the results as the following format(see assignment specification).

Firstly, we convert sales_timestamp column into datetime data type. Afterwards, we can extract the sales year, month and day

[217]:
```
timestamp_format = "%Y-%m-%d %H:%M:%S.%f %z"
daily_sales_rdd = sales_rdd.map(lambda x: [x[0]] + [datetime.strptime(x[1],␣
 ↪timestamp_format)] + x[2:])
daily_sales_rdd = daily_sales_rdd.map(lambda x: ((x[1].year, x[1].month, x[1].
 ↪day),1))
```

Then the total number of sales is calculated for each day using `groupByKey` function

[218]:
```
daily_sales_rdd = daily_sales_rdd.groupByKey().mapValues(len)
```

After calculating total daily sales, we can then perform another group-by function for each year, each month to retrieve the average total daily sales

[219]:
```
avg_sales_rdd = daily_sales_rdd.map(lambda x: ((x[0][0], x[0][1]), x[1]))
# group by month, year
avg_sales_rdd = avg_sales_rdd.groupByKey().mapValues(lambda x: round(sum(x)/
 ↪len(x),2))
# flatten the results
avg_sales_rdd = avg_sales_rdd.map(lambda x: [x[0][0], x[0][1], x[1]])
```

Our calculation of avg daily sales is then formally outputted using dataframe

[23]:
```
avg_sales_df = avg_sales_rdd.toDF()\
    .withColumnRenamed("_1","year").withColumnRenamed("_2","month").
 ↪withColumnRenamed("_3","avg sales")\
    .orderBy(col("year").asc(), col("month").asc())
avg_sales_df.show()
```

```
+----+-----+---------+
|year|month|avg sales|
+----+-----+---------+
```

```
|2019|   10| 23962.87|
|2019|   11| 31618.59|
|2019|   12| 37485.42|
|2020|    1| 27833.57|
|2020|    2| 41389.24|
|2020|    3| 33062.39|
|2020|    4|  32225.3|
+----+-----+---------+
```

1.3.2 Find 10 of the best selling brands. You should display the brand and total revenue in the result.

We first convert the price column from string to float data type

[220]:
```
sales_rdd = sales_rdd.map(lambda x: x[:3] + [float(x[3])] + x[4:])
```

[221]:
```
sales_mod_rdd = sales_rdd.map(lambda x: (x[2], x[3]))
product_mod_rdd = product_rdd.map(lambda x: (x[0], x[2]))
```

[222]:
```
join_rdd = product_mod_rdd.join(sales_mod_rdd)
join_rdd = join_rdd.map(lambda x: [x[1][0], x[1][1]])
```

[223]:
```
brand_revenue = join_rdd.groupByKey().mapValues(sum)
brand_revenue = brand_revenue.sortBy(lambda x: -x[1])
```

[121]:
```
brand_revenue.take(10)
```

[121]:
```
[('apple', 929385237.8800286),
 ('samsung', 425990400.8197582),
 ('xiaomi', 91863355.39001285),
 ('', 62416824.529973395),
 ('huawei', 42308406.85998937),
 ('lg', 38278431.37999987),
 ('sony', 28542732.010002367),
 ('lucente', 28392306.44000093),
 ('acer', 27610850.919998948),
 ('lenovo', 27212179.640004687)]
```

2.2 Part 2. Working with DataFrames

In this section, you will need to load the given datasets into PySpark DataFrames and use DataFrame functions to answer the queries. ### 2.1 Data Preparation and Loading

2.1.1. Load CUPS into four separate dataframes. When you create your dataframes, please refer to the metadata file and think about the appropriate data type for each columns (Note: Initially, you should read date/time related column as the string type).

[224]:
```
category_df = spark.read.csv('sales/category.csv', header=True)
product_df = spark.read.csv('sales/product.csv', header=True)
sales_df = spark.read.csv('sales/sales.csv', header=True)
users_df = spark.read.csv('users/users.csv', header=True)
```

2.1.2 Display the schema of the four dataframes.

We first transform the dataframes and convert the columns to the most appropriate data types

[225]:
```
# change product_df to have correct data types
product_df_clean = product_df.withColumn("product_id", col("product_id").
 ↪cast(IntegerType()))\
    .withColumn("avg_cost", col("avg_cost").cast(FloatType()))\

# change sales_df to have correct data types
sales_df_clean = sales_df.withColumn("id", col("id").cast(IntegerType()))\
    .withColumn("sales_timestamp", col("sales_timestamp").
 ↪cast(TimestampType()))\
    .withColumn("product_id", col("product_id").cast(IntegerType()))\
    .withColumn("price", col("price").cast(FloatType()))\
    .withColumn("user_id", col("user_id").cast(IntegerType()))

# change users_df to have correct data types
users_df_clean = users_df.withColumn("user_id", col("user_id").
 ↪cast(IntegerType()))\
    .withColumn("dob", to_date(col("dob"),"d-M-yyyy"))\
    .withColumn("postcode", col("postcode").cast(IntegerType()))
```

The schemas are then printed out

[282]:
```
print('category_df schema')
category_df.printSchema()
```

```
category_df schema
root
 |-- category_id: string (nullable = true)
 |-- category_code: string (nullable = true)
```

[283]:
```
print('product_df schema')
product_df_clean.printSchema()
```

```
product_df schema
root
 |-- product_id: integer (nullable = true)
 |-- category_id: string (nullable = true)
 |-- brand: string (nullable = true)
 |-- avg_cost: float (nullable = true)
```

[284]:
```
print('sales_df schema')
sales_df_clean.printSchema()
```

```
sales_df schema
root
 |-- id: integer (nullable = true)
 |-- sales_timestamp: timestamp (nullable = true)
 |-- product_id: integer (nullable = true)
 |-- price: float (nullable = true)
 |-- user_id: integer (nullable = true)
 |-- user_session: string (nullable = true)
 |-- sales_date: date (nullable = true)
 |-- sales_hour: integer (nullable = true)
```

[285]:
```
print('users_df schema')
users_df_clean.printSchema()
```

```
users_df schema
root
 |-- user_id: integer (nullable = true)
 |-- firstname: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- dob: date (nullable = true)
 |-- address: string (nullable = true)
 |-- postcode: integer (nullable = true)
 |-- state: string (nullable = true)
 |-- sex: string (nullable = true)
 |-- suburb: string (nullable = true)
 |-- age: integer (nullable = true)
```

2.2.1 2.2 QueryAnalysis

Implement the following queries using dataframes. You need to be able to perform operations like filtering, sorting, joining and group by using the functions provided by the DataFrame API.

2.2.1. Transform the 'sales_time' column in the sales dataframe to the date type; extract the hour in sales_date and create a new column "sales_hour"; after that, show the schema.

Firstly, convert values in sales_timestamp from String to Timestamp

[226]:
```
sales_df_mod = sales_df_clean.withColumn("sales_timestamp",sales_df_clean.
↪sales_timestamp.cast("timestamp"))
```

Secondly, create a new column `sales_date` based on data from `sales_timestamp`

[227]:
```
sales_df_mod = sales_df_mod.withColumn("sales_date", sales_df_mod.
  ↪sales_timestamp.cast("date"))
```

Thirdly, we create a new column `sales_hour` which retrieves the hour value from `sales_timestamp`

[228]:
```
sales_df_mod = sales_df_mod.withColumn("sales_hour", hour(sales_df_mod.
  ↪sales_timestamp))
```

We then print the schema

[229]:
```
sales_df_clean = sales_df_mod
sales_df_clean.printSchema()
```

```
root
 |-- id: integer (nullable = true)
 |-- sales_timestamp: timestamp (nullable = true)
 |-- product_id: integer (nullable = true)
 |-- price: float (nullable = true)
 |-- user_id: integer (nullable = true)
 |-- user_session: string (nullable = true)
 |-- sales_date: date (nullable = true)
 |-- sales_hour: integer (nullable = true)
```

2.2.2. Calculate total sales for each hour, sort your result based on each hour's sales in a descending order. Print out the sales_hour and total_sales columns.

[286]:
```
sales_by_hour_df = sales_df_clean.groupBy(sales_df_clean.sales_hour).count().
  ↪withColumnRenamed("count", "total_sales")
sales_by_hour_df = sales_by_hour_df.orderBy(sales_by_hour_df.total_sales.desc())
```

[287]:
```
sales_by_hour_df.show(24)
```

```
+----------+-----------+
|sales_hour|total_sales|
+----------+-----------+
|        20|     503621|
|        19|     503462|
|        18|     483906|
|        21|     482266|
|        17|     461138|
|        22|     443598|
|        16|     421224|
|        23|     409010|
|         0|     386173|
|         1|     372471|
|         2|     352587|
|        15|     351961|
|         3|     323915|
```

```
|          4|     287796|
|         14|     237628|
|          5|     225583|
|          6|     162968|
|         13|     116993|
|          7|     102286|
|          8|      64411|
|         12|      51705|
|          9|      42038|
|         11|      32010|
|         10|      30074|
+-----------+-----------+
```

2.2.3.Find 10 most profitable categories (profit can be simply defined as price - avg_cost). Print out the category name and total profit. Please print the category name in capitalized word format(hint: you can reuse the UDF defined in part 1.)

To find the 10 most profitable categories, we will need to perform a join operation with `category_df`, `product_df` and `sales_df`

We first remove the duplicate IDs in `category_df` by choosing the character code with the longest length.

[288]:
```
longest_string_udf = udf(get_longest_string, StringType())
# Used groupby function on df and aggregate using UDF function
deduped_category_df = category_df.groupBy("category_id").\
        agg(longest_string_udf(collect_list("category_code")).
 ↪alias("category_code"))
```

We then remove the duplicate product IDs in `product_df_clean` by selecting the instance with the largest category ID. We do this by using a combination of `row_number` function and `Window` to provide a ranking to each product id based on category id value

[289]:
```
# rank product_id by largest category_id and brand
product_dept = Window.partitionBy("product_id").orderBy([col("category_id").
 ↪desc(), col("brand").desc()])
# filter to get first rank
deduped_product_df = product_df_clean.withColumn("row", row_number().
 ↪over(product_dept))\
    .filter(col("row") == 1)\
    .drop("row")\
    .orderBy(col("product_id").asc())
```

We now join `dedupated_category_df`, `deduped_product_df` and `sales_df` to gather all the data we need to calculate sales profit

[290]:
```
join_df = sales_df_clean.join(deduped_product_df, sales_df_clean.product_id ==␣
 ↪deduped_product_df.product_id)\
```

```
        .drop(sales_df_clean.product_id)
join_df = join_df.join(deduped_category_df, join_df.category_id ==␣
 ↪deduped_category_df.category_id)\
        .drop(deduped_category_df.category_id)
```

We define a UDF function to calculate the profit

[291]:
```
profit_udf = udf(lambda price, avg_cost: price - avg_cost, FloatType())
```

We calculate the profit made for each sale, and then aggregate the table to get the total profit grouped by category name

[292]:
```
# Calculate profit
category_profits = join_df.select(col("category_code"), profit_udf(join_df.
 ↪price, join_df.avg_cost).alias("profit"))
# Sum profit grouped by category code
category_profits = category_profits.groupBy("category_code").sum("profit")
# format column and values
category_profits = category_profits.
 ↪withColumnRenamed("sum(profit)","total_profit")
```

Use `category_formatter` UDF function to transform `category_code`

[293]:
```
def category_format(category_code):
    words = category_code.split('.')
    for i in range(len(words)):
        words[i] = words[i].capitalize()
    return ' '.join(words)
category_format_udf = udf(category_format, StringType())

category_profits = category_profits.withColumn("category_code",␣
 ↪category_format_udf(col("category_code")))\
    .orderBy(category_profits.total_profit.desc())
```

Finally, we output the top 10 most profitable categories

[294]:
```
res = category_profits.limit(10)
```

[295]:
```
res.show(truncate=False)
```

```
+------------------------------+------------------+
|category_code                 |total_profit      |
+------------------------------+------------------+
|Construction Tools Light      |6.125602823422313E7|
|Appliances Personal Massager  |5196086.414781928 |
|Appliances Kitchen Refrigerators|4283076.883982658 |
|Others                        |4002670.567923337 |
|Computers Notebook            |2805912.1956562996|
```

```
|Electronics Clocks               |2113755.9995524883 |
|Appliances Kitchen Washer        |1432456.2404248714 |
|Sport Ski                        |1283055.6067435145 |
|Appliances Environment Vacuum    |1171350.9181756973 |
|Electronics Smartphone           |1021107.1154553294 |
+---------------------------------+-------------------+
```

2.2.4. Use DataFrame filters to find all transactions sold at loss (defined as price < avg_cost), calculate 10 worst loss margin in percentage. (margin is defined as (price - avg_cost)/avg_cost; if price - avg_cost > 0, it's call a profit margin; otherwise a loss margin)

[298]:
```
product_sales_df = sales_df_clean.join(deduped_product_df, \
                                       sales_df_clean.product_id ==␣
 ↪deduped_product_df.product_id,\
                                      how = 'left')\
        .drop(product_df.product_id)
```

[299]:
```
loss_margin_df = product_sales_df.withColumn("loss_margin",␣
 ↪spark_round(-((col("price") - col("avg_cost"))/col("avg_cost")*100),2))\
    .orderBy(col("loss_margin").desc())
loss_margin_df = loss_margin_df.select("id", "price", "avg_cost", "loss_margin")
```

[300]:
```
loss_margin_df.show(10)
```

```
+---------+-----+---------+-----------+
|       id|price| avg_cost|loss_margin|
+---------+-----+---------+-----------+
|204037762| 4.61|603.00494|      99.24|
|204017856| 4.61|603.00494|      99.24|
|396531551| 5.15| 512.3399|      98.99|
|394075934| 5.15| 512.3399|      98.99|
|394083902| 5.15| 512.3399|      98.99|
|394106370| 5.15| 512.3399|      98.99|
|394112198| 5.15| 512.3399|      98.99|
|394129754| 5.15| 512.3399|      98.99|
|396544717| 5.15| 512.3399|      98.99|
|396598353| 5.15| 512.3399|      98.99|
+---------+-----+---------+-----------+
only showing top 10 rows
```

2.2.5. Draw a barchart to show total sales from different states in each year.

We perform a left join between `sales_df` and `users_df` to receive state information for each sale transaction

[240]:
```
user_sales_df = sales_df_clean.join(users_df_clean, \
                                    sales_df_clean.user_id == users_df_clean.
↪user_id, how="left")\
       .drop(users_df_clean.user_id)
```

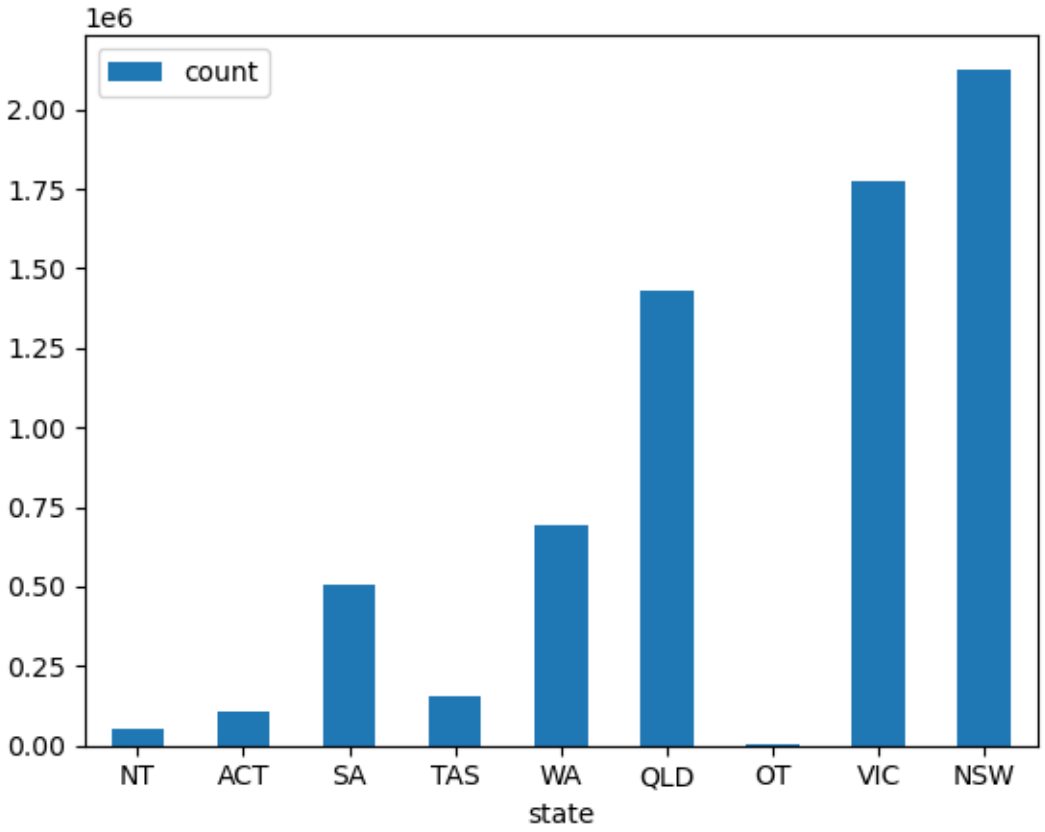
We then group the table by state and calculate the total count, which refers to the total number of sales made for each state

[241]:
```
state_total_sales_df = user_sales_df.groupBy(col("state")).count()
```

The Pyspark dataframe is then converted into a Pandas dataframe to plot it into a bar graph

[27]:
```
total_sales_pd = state_total_sales_df.toPandas()
total_sales_pd.plot.bar(x='state', y='count', rot=0)
```

[27]:
```
<Axes: xlabel='state'>
```



2.2.6. Draw a scatter plot of customer age and their total spending with MOTH. To limit the number of datapoints, you may show the top 1000 "most valuable" customers only. You may also use log scale for the XY axis.

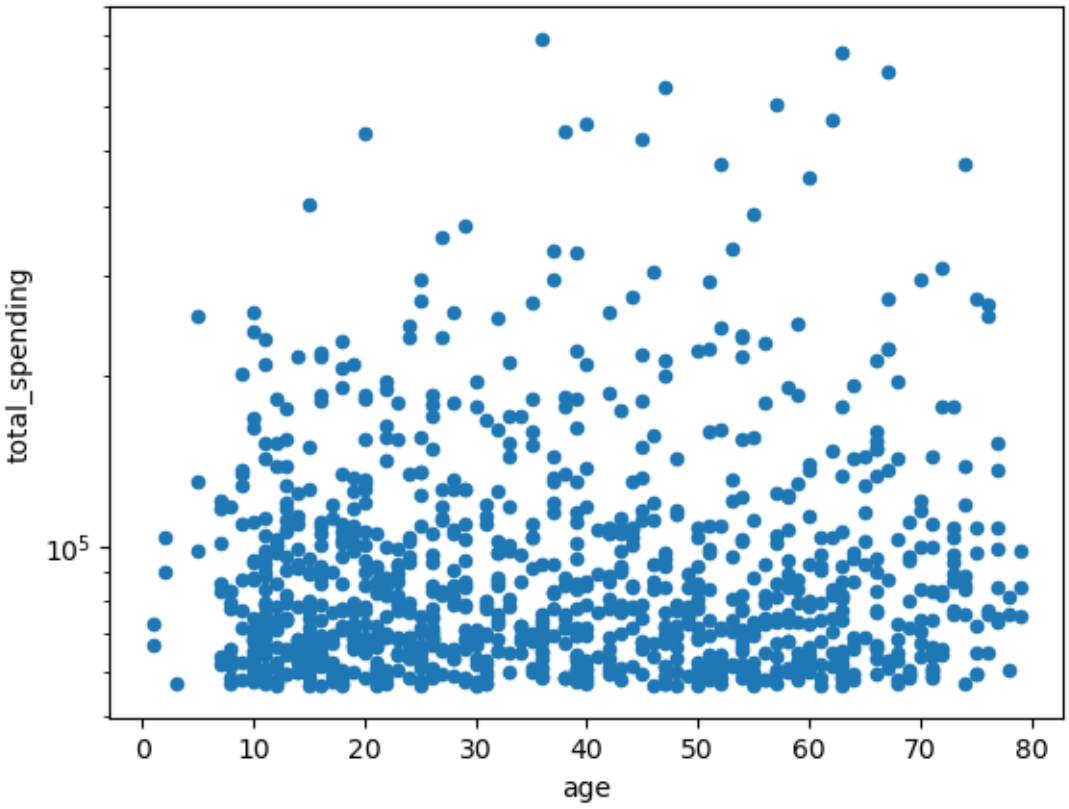
[242]:
```
def calc_age(dob):
    today = date.today()
    delta = today.year - dob.year - ((today.month, today.day) < (dob.month, dob.
 ↪day))
    return delta

calc_age_udf = udf(calc_age, IntegerType())
```

[243]:
```
age_spending_df = user_sales_df.select(col("user_id"), col("dob"),␣
 ↪col("price"), calc_age_udf(col("dob")).alias("age"))
age_spending_df = age_spending_df.groupBy(col("user_id"), col("age")).
 ↪sum("price")
age_spending_df = age_spending_df.withColumnRenamed("sum(price)",␣
 ↪"total_spending")
age_spending_df = age_spending_df.orderBy(age_spending_df.total_spending.desc())
age_spending_df = age_spending_df.limit(1000)
```

[30]:
```
age_spending_pd = age_spending_df.toPandas()
```

[31]:
```
age_spending_pd.plot.scatter(x='age', y='total_spending', logy=True)
```

[31]:
```
<Axes: xlabel='age', ylabel='total_spending'>
```



2.2.2 Part 3 RDDs vs DataFrame vs Spark SQL (15%)

Implement the following queries using RDDs, DataFrames in SparkSQL separately. Log the time taken for each query in each approach using the "%%time" built-in magic command in Jupyter Notebook and discuss the performance difference between these 3 approaches.

Query: Find top 100 most popular products (by total sales) among user age group 20-40, group by brand, and show total sales revenue of each brand.

3.1. RDD Implementation

[308]:
```
%%time
# pre-process user_rdd dataset to fix rows with additional columns showing␣
 ↪'Unknown' value
val1 = users_rdd.filter(lambda x: x[1] == 'Unknown').map(lambda x: x[:1] + x[2:
 ↪])
val2 = users_rdd.filter(lambda x: x[1] != 'Unknown')
users_mod_rdd = val1.union(val2)
# get user age
users_mod_rdd = users_mod_rdd.map(lambda x: x[:1] + [datetime.strptime(x[1],␣
 ↪"%d-%m-%Y")] + x[2:])
users_mod_rdd = users_mod_rdd.map(lambda x: x[:1] + [calc_age(x[1])])
# filter for users between the age 20-40 inclusive
users_mod_rdd = users_mod_rdd.filter(lambda x: x[-1] >= 20).filter(lambda x:␣
 ↪x[-1] <= 40)
sales_mod_rdd = sales_rdd.map(lambda x: [x[4], (x[2], x[3])])
# join user and sales dataset
join_rdd = sales_mod_rdd.join(users_mod_rdd)
product_sales_rdd = join_rdd.map(lambda x: x[1][0][:])
# group by product id and aggregate to get total number of sales and total␣
 ↪sales revenue
product_sales_rdd = product_sales_rdd.groupByKey().mapValues(lambda x:␣
 ↪(len(x),sum(x)) ).sortBy(lambda x: -x[1][0])
# Retrieve 100 most popular products based on total number of sales
product_sales_rdd = product_sales_rdd.zipWithIndex().filter(lambda x: x[1]<100).
 ↪map(lambda x: x[0][:])
# Get brand for each product_id
product_mod_rdd = product_rdd.map(lambda x: [x[0], x[2]])
brand_join_rdd = product_sales_rdd.join(product_mod_rdd).map(lambda x:␣
 ↪[x[1][1], x[1][0][1]])
# Group by brand and calculate sum of sales revenue
brand_revenue_rdd = brand_join_rdd.groupByKey().mapValues(sum).sortBy(lambda x:␣
 ↪-x[1])
# Output results
brand_revenue_rdd.collect()
```

```
CPU times: user 185 ms, sys: 86.6 ms, total: 272 ms
Wall time: 5min 8s
```

[308]:
```
[('apple', 225697189.42999905),
 ('samsung', 87188246.74999748),
 ('xiaomi', 14225832.960000172),
 ('huawei', 7733847.580000031),
 ('oppo', 5075319.259999837),
 ('sony', 1742158.7900000412),
 ('lenovo', 920414.8700000199),
 ('artel', 734490.7199999966)]
```

3.2. DataFrame Implementation Before performing the query, we need to first pre-process the data. Firstly is to calculate the age of each user

[302]:
```
%%time
# filter user age
users_df_clean = users_df_clean.withColumn("age", calc_age_udf(col("dob")))
filtered_users_df = users_df_clean.filter(col('age')>=20).filter(col('age')<=40)
# join sales_df and users_df
join_df = sales_df_clean.join(filtered_users_df, sales_df_clean.
 ↪user_id==filtered_users_df.user_id, how='inner')\
        .drop(filtered_users_df.user_id)
# group by product id and calculate total number of sales and sales revenue
product_sales_df = join_df.groupBy(col("product_id")).agg(
    spark_count("*").alias("total_sales"),
    spark_sum("price").alias("sales_revenue")
)
# get 100 most popular products by number of sales
product_sales_df = product_sales_df.orderBy(col("total_sales").desc()).
 ↪limit(100)
brand_join_df = product_sales_df.join(deduped_product_df,\
                    product_sales_df.product_id == deduped_product_df.
 ↪product_id,\
                    how='inner')\
        .drop(deduped_product_df.product_id)
# group by brand and aggregate the sum of sales revenue
brand_revenue_df = brand_join_df.groupBy(col("brand")).
 ↪agg(spark_sum("sales_revenue").alias("total_revenue"))\
                    .orderBy(col("total_revenue").desc())
# output results
brand_revenue_df.show()
```

```
+-------+--------------------+
|  brand|       total_revenue|
+-------+--------------------+
|  apple|2.2569718955223083E8|
```

```
|samsung| 8.718824659664154E7|
| xiaomi|1.4225832972858429E7|
| huawei|   7733847.568565369|
|   oppo|   5075319.240287781|
|   sony|   1742158.806427002|
| lenovo|   920414.8711090088|
|  artel|   734490.7239379883|
+-------+--------------------+

CPU times: user 11.4 ms, sys: 22.6 ms, total: 34 ms
Wall time: 29.9 s
```

3.3. Spark SQL Implementation We convert our pyspark dataframes into spark SQL views

[280]:
```
deduped_product_df.createOrReplaceTempView("product_sql")
sales_df_clean.createOrReplaceTempView("sales_sql")
users_df_clean.createOrReplaceTempView("users_sql")
```

We then run the pyspark SQL query

[301]:
```
%%time
brand_revenue_sql = spark.sql(
"""
with filtered_users as
(
    SELECT user_id
    FROM users_sql
    WHERE age >= 20 and age <= 40
)
, popular_products as
(
    SELECT product_id, count(*) as total_sales, sum(price) as total_revenue
    FROM sales_sql
    JOIN filtered_users
    ON sales_sql.user_id = filtered_users.user_id
    GROUP BY product_id
    ORDER BY total_sales desc
    LIMIT 100
)
SELECT brand, sum(total_revenue) as total_revenue
FROM popular_products
JOIN product_sql
ON popular_products.product_id = product_sql.product_id
GROUP BY brand
ORDER BY total_revenue desc
"""
)
brand_revenue_sql.show()
```

```
+-------+-------------------+
|  brand|       total_revenue|
+-------+-------------------+
|  apple|2.2569718955223083E8|
|samsung| 8.718824659664154E7|
| xiaomi|1.4225832972858429E7|
| huawei|   7733847.568565369|
|   oppo|   5075319.240287781|
|   sony|   1742158.806427002|
| lenovo|   920414.8711090088|
|  artel|   734490.7239379883|
+-------+-------------------+

CPU times: user 0 ns, sys: 15.1 ms, total: 15.1 ms
Wall time: 29.7 s
```

2.2.3 3.4 Observe the query execution time among RDD, DataFrame, SparkSQL, which is the fastest and why? (Maximum 500 words.)

Below is the summary of execution times for each query approach: * RDD - total CPU time: 272 ms, Wall time: 5min 8s * DataFrame - total CPU time: 34 ms, Wall time: 29.9 s * SparkSQL - total CPU time: 15.1 ms, Wall time: 29.7 s

From the results shown above, it is evident that SparkSQL has the shortest execution time. The next clost shortest execution time is Dataframe approach and then lastly RDD. These statements are true for both total CPU time and Wall time.

The reason for why SparkSQL and Dataframe approach had a significantly shorter execution time compared to RDD is due to its optimised strategies in join, sort and groupby operations. For example, SparkSQL and Dataframes both use a combination of Sort-Merge Join algorithm (has a complexity of O(nlogn) including sorting operations) for two large datasets and BroadcastHashJoin to join datasets where one is a size. For RDD, it is by default uses ShuffleHashJoin. Unlike RDDs, SparkSQL and Dataframe leverages various algorithms for join operations based on the dataset sizes in order to reduce execution time.

SparkSQL uses Catalyst Optimiser that is a powerful query optimisation framework that allows spark to define specific rules and transformations to the logical query plan to create an optimised physical execution plan.

SparkSQL is also very good at memory management compared to RDD. The method has been optimised to reduce garbage collection overhead and improve memory utilisation, allowing more memory to be used by processors and therefore leading to a faster query execution time

2.2.4 Some ideas on the comparison

Armbrust, M., Huai, Y., Liang, C., Xin, R., & Zaharia, M. (2015). Deep Dive into Spark SQL's Catalyst Optimizer. Retrieved September 30, 2017, from https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

Damji, J. (2016). A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets. Retrieved September 28, 2017, from https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-

spark-apis-rdds-dataframes-and-datasets.html

Data Flair (2017a). Apache Spark RDD vs DataFrame vs DataSet. Retrieved September 28, 2017, from http://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset

Prakash, C. (2016). Apache Spark: RDD vs Dataframe vs Dataset. Retrieved September 28, 2017, from http://why-not-learn-something.blogspot.com.au/2016/07/apache-spark-rdd-vs-dataframe-vs-dataset.html

Xin, R., & Rosen, J. (2015). Project Tungsten: Bringing Apache Spark Closer to Bare Metal. Retrieved September 30, 2017, from https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html