In [15]:
```
listEmployee = [(105,'Chloe',5),\
                (103, 'Paul', 3),\
                (101, 'John', 1),\
                (102, 'Lisa', 2),\
                (104, 'Evan', 4),\
                (106,  'Amy', 6)]
```

In [35]:
```
listDepartment = [(     100,  'Other'),\
                  (     3,'Engineering'),\
                  (     2,      'Sales'),\
                  (     1,  'Marketing')]
```

In [36]:
```
employeeDF = spark\
.createDataFrame(listEmployee,['id','name','deptno'])
```

In [37]:
```
departmentDF = spark\
.createDataFrame(listDepartment,['deptno','deptname'])
```

In [38]:
```
employeeDF.show()
```

```
+---+-----+------+
| id| name|deptno|
+---+-----+------+
|105|Chloe|     5|
|103| Paul|     3|
|101| John|     1|
|102| Lisa|     2|
|104| Evan|     4|
|106|  Amy|     6|
+---+-----+------+
```

In [39]:
```
departmentDF.show()
```

```
+------+-----------+
|deptno|   deptname|
+------+-----------+
|   100|      Other|
|     3|Engineering|
|     2|      Sales|
|     1|  Marketing|
+------+-----------+
```

In [50]:
```
# Inner Join/Natural join

# The inner join is the default join in Spark SQL.
# It selects rows that have matching values in both
# relations.

resDF = employeeDF\
.join(departmentDF,\
      employeeDF.deptno==departmentDF.deptno)

resDF.printSchema()
resDF.show()
```

```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptname: string (nullable = true)

+---+----+------+------+-----------+
| id|name|deptno|deptno|   deptname|
+---+----+------+------+-----------+
|101|John|     1|     1|  Marketing|
|103|Paul|     3|     3|Engineering|
|102|Lisa|     2|     2|      Sales|
+---+----+------+------+-----------+
```

In [52]:

```
# Inner Join

# The inner join is the default join in Spark SQL.
# It selects rows that have matching values in both
# relations.

resDF = employeeDF\
.join(departmentDF,\
      employeeDF.deptno==departmentDF.deptno,\
      'inner')

resDF.printSchema()
resDF.show()
```

```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptname: string (nullable = true)

+---+----+------+------+-----------+
| id|name|deptno|deptno|   deptname|
+---+----+------+------+-----------+
|101|John|     1|     1|  Marketing|
|103|Paul|     3|     3|Engineering|
|102|Lisa|     2|     2|      Sales|
+---+----+------+------+-----------+
```

In [64]:

```
# Left semi Join

# A left semi join returns values from the left
# side of the relation that has a match with the
# right. It is also referred to as a left join.

# 'leftsemi', 'left_semi',

resDF = employeeDF\
.join(departmentDF,\
      employeeDF.deptno==departmentDF.deptno,\
      'leftsemi')

resDF.printSchema()
resDF.show()
```

```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
```

```
 |-- deptno: long (nullable = true)

+---+----+------+
| id|name|deptno|
+---+----+------+
|101|John|     1|
|103|Paul|     3|
|102|Lisa|     2|
+---+----+------+
```

In [65]:

```
# Left outer Join

# A left outer join returns all values from the left
# relation and the matched values from the right
# relation, or appends NULL if there is no match.
# It is also referred to as a left join.

#'leftouter', 'left', 'left_outer',

resDF = employeeDF\
.join(departmentDF,\
      employeeDF.deptno==departmentDF.deptno, \
      'leftouter')

resDF.printSchema()
resDF.show()
```

```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptname: string (nullable = true)

+---+-----+------+------+-----------+
| id| name|deptno|deptno|   deptname|
+---+-----+------+------+-----------+
|106|  Amy|     6|  null|       null|
|105|Chloe|     5|  null|       null|
|101| John|     1|     1|  Marketing|
|103| Paul|     3|     3|Engineering|
|102| Lisa|     2|     2|      Sales|
|104| Evan|     4|  null|       null|
+---+-----+------+------+-----------+
```

In [66]:

```
# Right outer Join

# A right outer join returns all values from the right
# relation and the matched values from the left
# relation, or appends NULL if there is no match.
# It is also referred to as a right join.

#'rightouter', 'right', 'right_outer',

resDF = employeeDF\
.join(departmentDF,\
      employeeDF.deptno==departmentDF.deptno,\
      'rightouter')

resDF.printSchema()
resDF.show()
```

```
root
```

```
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptname: string (nullable = true)

+----+----+------+------+-----------+
|  id|name|deptno|deptno|   deptname|
+----+----+------+------+-----------+
| 101|John|     1|     1|  Marketing|
|null|null|  null|   100|      Other|
| 103|Paul|     3|     3|Engineering|
| 102|Lisa|     2|     2|      Sales|
+----+----+------+------+-----------+
```

In [67]:

```
# Full outer Join

# A full outer join returns all values from both
# relations, appending NULL values on the side that
# does not have a match. It is also referred to
# as a full join.

# 'outer', 'full', 'fullouter', 'full_outer',

resDF = employeeDF\
.join(departmentDF,\
      employeeDF.deptno==departmentDF.deptno,\
      'outer')

resDF.printSchema()
resDF.show()
```

```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptno: long (nullable = true)
 |-- deptname: string (nullable = true)

+----+-----+------+------+-----------+
|  id| name|deptno|deptno|   deptname|
+----+-----+------+------+-----------+
| 106|  Amy|     6|  null|       null|
| 105|Chloe|     5|  null|       null|
| 101| John|     1|     1|  Marketing|
|null| null|  null|   100|      Other|
| 103| Paul|     3|     3|Engineering|
| 102| Lisa|     2|     2|      Sales|
| 104| Evan|     4|  null|       null|
+----+-----+------+------+-----------+
```

In [68]:

```
# Left anti Join

# A left anti join returns values from the left
# relation that has no match with the right.
# It is also referred to as an anti join.

# 'leftanti', 'left_anti',

resDF = employeeDF\
.join(departmentDF,\
      employeeDF.deptno==departmentDF.deptno, \
      'leftanti')
```

```
resDF.printSchema()
resDF.show()
```

```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- deptno: long (nullable = true)

+---+-----+------+
| id| name|deptno|
+---+-----+------+
|106|  Amy|     6|
|105|Chloe|     5|
|104| Evan|     4|
+---+-----+------+
```

In [ ]:

In [1]:
```
import numpy as np
```

In [2]:
```
##integer
```

In [12]:
```
values = np.array([1,2,3,3])
```

In [13]:
```
integer = sc.parallelize(values)
```

In [14]:
```
integer.collect()
```

Out[14]: [1, 2, 3, 3]

In [15]:
```
greater = integer.filter(lambda i: i>2)
```

In [16]:
```
print(greater.collect())
```

```
[3, 3]
```

In [34]:
```
def square(i):
    return i*i
```

In [35]:
```
square = integer.map(square)
```

In [36]:
```
square.collect()
```

Out[36]: [1, 4, 9, 9]

In [34]:
```
distinct = integer.distinct()
```

In [35]:
```
print(distinct.collect())
```

```
[1, 2, 3]
```

In [81]:
```
def maxNum(x, y):
    if x>y:
        return x
    else:
        return y
sumInt = integer.reduce(lambda e1, e2: e1+e2)
maxInt = integer.reduce(maxNum)
print('sum:', sumInt, 'and max:', maxInt)
```

```
sum: 9 and max: 3
```

In [ ]:
```
##Length
```

In [9]:
```
lines = sc.textFile('myfile.txt')
```

In [10]:
```
def countLength(l):
    return len(l)
```

In [24]:
```
length = lines.map(countLength)
```

In [25]:
```
print(length.collect())
```

```
[27, 17, 23, 16]
```

In [26]:
```
##word list
```

In [31]:
```
wordListFlat = lines.flatMap(lambda l: l.split(sep=' '))
wordListMap = lines.map(lambda l: l.split(sep=' '))
```

In [32]:
```
print(wordListFlat.collect())
```

```
['this', 'is', 'about', 'word', 'counting', 'there', 'are', '4', 'lines', 'so', '4',
'partitions', 'fir', 'RDD', 'see', 'what', 'happens']
```

In [33]:
```
print(wordListMap.collect())
```

```
[['this', 'is', 'about', 'word', 'counting'], ['there', 'are', '4', 'lines'], ['so',
'4', 'partitions', 'fir', 'RDD'], ['see', 'what', 'happens']]
```

In [36]:
```
print(lines.distinct().collect())
```

```
['this is about word counting', 'there are 4 lines', 'so 4 partitions fir RDD', 'see
what happens']
```

In [37]:
```
## names
```

In [48]:
```
with open('names.txt') as f:
    l = f.readlines()
for i in l:
    print(i)
```

```
mori

ali

amir

khashi

amir
```

In [49]:
```
names = sc.textFile('names.txt')
```

In [67]:
```
print(names.countByValue())
```

```
defaultdict(<class 'int'>, {'mori': 1, 'ali': 1, 'amir': 2, 'khashi': 1})
```

In [78]:
```
print('first:', names.first(),\
      '..... take1:', names.take(1),\
      '..... takeOrder:', names.takeOrdered(2, lambda s: len(s)),\
      '..... top2:', names.top(2, lambda s: len(s)) )
```

```
first: mori ..... take1: ['mori'] ..... takeOrder: ['ali', 'mori'] ..... top2: ['kha
shi', 'mori']
```

In [50]:
```
sortNames = names.sortBy(lambda n:n)
```

In [51]:
```
print(sortNames.collect())
```

```
['ali', 'amir', 'amir', 'khashi', 'mori']
```

In [56]:
```
lenNames = names.sortBy(lambda n: (len(n), n), ascending=False)
```

In [57]:
```
print(lenNames.collect())
```

```
['khashi', 'mori', 'amir', 'amir', 'ali']
```

In [58]:
```
randomSample = lines.sample(False, 0.2)
```

In [59]:
```
print(randomSample.collect())
```

```
['there are 4 lines']
```

In [65]:
```
## two RDDs
```

In [60]:
```
u1 = sc.parallelize(np.array([1,2,2,3,3]))
u2 = sc.parallelize(np.array([3,4,5]))
```

In [61]:
```
union = u1.union(u2)
print(union.collect())
```

```
[1, 2, 2, 3, 3, 3, 4, 5]
```

In [62]:
```
intersection = u1.intersection(u2)
print(intersection.collect())
```

```
[3]
```

In [64]:
```
subtract = u1.subtract(u2)
print(subtract.collect())
```

```
[1, 2, 2]
```

In [66]:
```
cartesian = u1.cartesian(u2)
print(cartesian.collect())
```

```
[(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (2, 3), (2, 4), (2, 5), (3, 3), (3,
4), (3, 5), (3, 3), (3, 4), (3, 5)]
```

In [82]:
```
## fold action
```

In [84]:
```
strings = sc.parallelize(['this', 'is', 'a', 'test'])
```

In [95]:
```
def makeSentence(s1,s2):
    return s1+s2
#sentence = strings.fold('', lambda s1, s2: s1+s2)
sentence = strings.fold('', makeSentence)
print(sentence)
```

```
thisisatest
```

In [96]:
```
## aggregate
```

In [100…
```
zeroValue = (0,0)
sumCount = integer.aggregate(zeroValue,\
                             lambda acc, e:  (acc[0]+e, acc[1]+1),\
                             lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))
type(sumCount)
```

Out[100… tuple

In [101…
```
print(sumCount)
```

```
(9, 4)
```

In [ ]:
```

```

In [1]:
```
profilesList = [(19, "Justin"), (30, "Andy"),(None, "Michael")]
```

In [2]:
```
df = spark.createDataFrame(profilesList, ['age', 'name'])
```

In [3]:
```
df.printSchema()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

In [4]:
```
df.show()
```

```
+----+-------+
| age|   name|
+----+-------+
|  19| Justin|
|  30|   Andy|
|null|Michael|
+----+-------+
```

In [5]:
```
persons = spark.read.load('persons.csv',\
                    format='csv',\
                    header=True,\
                    inferSchema=True)
persons.show(), persons.count()
```

```
+-------+----+
|   name| age|
+-------+----+
|   Andy|  30|
|Michael|null|
| Justin|  19|
+-------+----+
```

Out[5]:
```
(None, 3)
```

In [6]:
```
rddRows = persons.rdd
rddRows.collect()
```

Out[6]:
```
[Row(name='Andy', age=30),
 Row(name='Michael', age=None),
 Row(name='Justin', age=19)]
```

In [10]:
```
rddNames = rddRows.map(lambda row: row.name)
rddNames.collect()
```

Out[10]:
```
['Andy', 'Michael', 'Justin']
```

In [12]:
```
personNames = persons.select('name')
personNames.collect()
```

Out[12]:
```
[Row(name='Andy'), Row(name='Michael'), Row(name='Justin')]
```

In [13]:
```
personNames.show()
```

```
+-------+
|   name|
+-------+
|   Andy|
|Michael|
| Justin|
+-------+
```

In [24]:
```
personAge = persons.selectExpr('age + 1 AS new_age')
personAge.show()
```

```
+-------+
|new_age|
+-------+
|     31|
|   null|
|     20|
+-------+
```

In [27]:
```
oldPerson = persons.filter('age > 20')
oldPerson.show()
```

```
+----+---+
|name|age|
+----+---+
|Andy| 30|
+----+---+
```

In [30]:
```
ageAve = persons.agg({'age': 'avg'})
ageAve.show()
```

```
+--------+
|avg(age)|
+--------+
|    24.5|
+--------+
```

In [14]:
```
profile = spark.read.load('profile.csv',\
                          format='csv',\
                          header=True,\
                          inferSchema=True)
profile.show()
```

```
+-------+---+
|   name|age|
+-------+---+
|   Andy| 30|
|Michael| 15|
| Justin| 19|
|   Andy| 40|
+-------+---+
```

In [34]:
```
avgTable = profile.groupBy('name').avg('age')
avgTable.show()
```

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

```
|   name|avg(age)|
+-------+--------+
|Michael|    15.0|
|   Andy|    35.0|
| Justin|    19.0|
+-------+--------+
```

In [36]:
```
numbers = profile.groupBy('name').agg({'age': 'avg','name': 'count'})
numbers.show()
```

```
+-------+-----------+--------+
|   name|count(name)|avg(age)|
+-------+-----------+--------+
|Michael|          1|    15.0|
|   Andy|          2|    35.0|
| Justin|          1|    19.0|
+-------+-----------+--------+
```

In [37]:
```
# SQL
```

In [38]:
```
pfg = spark.read.load('pfg.csv',\
                      format='csv',\
                      header=True,\
                      inferSchema=True)
```

In [39]:
```
pfg.show()
```

```
+-----+---+------+
| name|age|gender|
+-----+---+------+
| Paul| 40|  male|
| Paul| 38|  male|
|David| 15|  male|
|Susan| 40|female|
|Susan| 34|female|
+-----+---+------+
```

In [40]:
```
pfg.createOrReplaceTempView('people')
```

In [45]:
```
selectedPersons = spark.sql('\
                            SELECT *\
                            FROM people\
                            WHERE age>=30 and age<=41\
                            ')
```

In [46]:
```
selectedPersons.show()
```

```
+-----+---+------+
| name|age|gender|
+-----+---+------+
| Paul| 40|  male|
| Paul| 38|  male|
|Susan| 40|female|
|Susan| 34|female|
+-----+---+------+
```

In [47]:
```
selectedPersons.write.csv('selectedPersons.csv', header=True)
```

In [48]:
```
selectedPersons.explain()
```

```
== Physical Plan ==
*(1) Project [name#349, age#350, gender#351]
+- *(1) Filter ((isnotnull(age#350) && (age#350 >= 30)) && (age#350 <= 41))
   +- *(1) FileScan csv [name#349,age#350,gender#351] Batched: false, Format: CSV, L
ocation: InMemoryFileIndex[hdfs://BigDataHA/user/s296721/pfg.csv], PartitionFilters:
[], PushedFilters: [IsNotNull(age), GreaterThanOrEqual(age,30), LessThanOrEqual(age,
41)], ReadSchema: struct<name:string,age:int,gender:string>
```

In [ ]:
```

```

In [30]:
```
# Create a Spark Session object
#spark = SparkSession.builder.getOrCreate()
```

In [1]:
```
# Create a DataFrame from persons.csv
df = spark.read.load("persons.csv",\
                     format="csv",\
                     header=True,\
                     inferSchema=True)
```

In [2]:
```
df.printSchema()
```

```
root
 |-- Name: string (nullable = true)
 |-- Age: integer (nullable = true)
```

In [3]:
```
df.show()
```

```
+-------+----+
|   Name| Age|
+-------+----+
|   Andy|  30|
|Michael|null|
| Justin|  19|
+-------+----+
```

In [ ]:

In [4]:
```
# Create a DataFrame from persons.csv
#  A slightly different approach
df = spark.read.format("csv").option("header", True).option("inferSchema", True).loa
```

In [5]:
```
df.printSchema()
df.show()
```

```
root
 |-- Name: string (nullable = true)
 |-- Age: integer (nullable = true)

+-------+----+
|   Name| Age|
+-------+----+
|   Andy|  30|
|Michael|null|
| Justin|  19|
+-------+----+
```

In [ ]:

In [6]:
```
# Example with an input json file - inline format
df = spark.read.load( "persons.json",\
                     format="json")
```

In [7]:
```
df.printSchema()
df.show()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

In [9]:
```
# Example with an input folder with json files (one json file for each input object/
df = spark.read.load( "JSONFiles/",\
                      format="json",\
                      multiLine=True)
```

In [10]:
```
df.printSchema()
df.show()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

+----+-------+
| age|   name|
+----+-------+
|  19| Justin|
|  30|   Andy|
|null|Michael|
+----+-------+
```

In [ ]:
```
```

In [11]:
```
# DataFrame from local python list of tuples
profilesList = [(19, "Justin"), (30, "Andy"), (None, "Michael")]
```

In [12]:
```
df = spark.createDataFrame(profilesList,["age","name"])
```

In [13]:
```
df.printSchema()
df.show()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

+----+-------+
| age|   name|
+----+-------+
|  19| Justin|
|  30|   Andy|
|null|Michael|
+----+-------+
```

In [ ]:

In [14]:
```
#How to rename columns
df2 = df.withColumnRenamed("name", "myNameColumn")
```

In [15]:
```
df2.printSchema()
df2.show()
```

```
root
 |-- age: long (nullable = true)
 |-- myNameColumn: string (nullable = true)

+----+------------+
| age|myNameColumn|
+----+------------+
|  19|      Justin|
|  30|        Andy|
|null|     Michael|
+----+------------+
```

In [ ]:

In [17]:
```
# Create a DataFrame from persons_noheader.csv
# Input csv data without header
dfnoheader = spark.read.load("persons_noheader.csv",\
                     format="csv",\
                     header=False,\
                     inferSchema=True)
```

In [18]:
```
dfnoheader.printSchema()
dfnoheader.show()
```

```
root
 |-- _c0: string (nullable = true)
 |-- _c1: integer (nullable = true)

+-------+----+
|    _c0| _c1|
+-------+----+
|   Andy|  30|
|Michael|null|
| Justin|  19|
+-------+----+
```

In [22]:
```
# Create a DataFrame from persons_noheader.csv
# Rename its columns
dfnoheader = spark.read.load("persons_noheader.csv",\
                     format="csv",\
                     header=False,\
                     inferSchema=True)\
.withColumnRenamed("_c0", "name")\
.withColumnRenamed("_c1", "age")
```

In [23]:
```
dfnoheader.printSchema()
```

```
root
```

```
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
```

In [24]:
```
dfnoheader.show()
```

```
+-------+----+
|   name| age|
+-------+----+
|   Andy|  30|
|Michael|null|
| Justin|  19|
+-------+----+
```

In [25]:
```
# Create an RDD from a DataFrame
myRDD = dfnoheader.rdd
```

In [26]:
```
myRDD.collect()
```

Out[26]:
```
[Row(name='Andy', age=30),
 Row(name='Michael', age=None),
 Row(name='Justin', age=19)]
```

In [27]:
```
firstRecord = myRDD.first()
```

In [28]:
```
# Select the value of column name from the first record/row extracted from myRDD
firstRecord.name
```

Out[28]:
```
'Andy'
```

In [30]:
```
# Select the value of column name from the first record/row extracted from myRDD - a
firstRecord["name"]
```

Out[30]:
```
'Andy'
```

In [ ]:
```

```

In [32]:
```
# Read persons2.csv
df = spark.read.load( "persons2.csv", format="csv", header=True, inferSchema=True)
```

In [33]:
```
df.printSchema()
df.show()
```

```
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- gender: string (nullable = true)

+-----+---+------+
| name|age|gender|
+-----+---+------+
| Paul| 40|  male|
| John| 40|  male|
|David| 15|  male|
```

```
|Susan| 40|female|
|Karen| 34|female|
+-----+---+------+
```

In [34]:
```
# Select only columns name and age
dfNamesAges = df.select("name", "age")
```

In [35]:
```
dfNamesAges.printSchema()
dfNamesAges.show()
```

```
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)

+-----+---+
| name|age|
+-----+---+
| Paul| 40|
| John| 40|
|David| 15|
|Susan| 40|
|Karen| 34|
+-----+---+
```

In [37]:
```
# Select only columns name and age+1 (sum 1 to the original value of age).
dfnew = df.selectExpr("name", "age + 1 as newage")
```

In [38]:
```
dfnew.printSchema()
```

```
root
 |-- name: string (nullable = true)
 |-- newage: integer (nullable = true)
```

In [39]:
```
dfnew.show()
```

```
+-----+------+
| name|newage|
+-----+------+
| Paul|    41|
| John|    41|
|David|    16|
|Susan|    41|
|Karen|    35|
+-----+------+
```

In [ ]:
```

```

In [40]:
```
# Select only the records with age >= 30
df_filtered = df.filter("age>=30")
```

In [41]:
```
df_filtered.printSchema()
df_filtered.show()
```

```
root
 |-- name: string (nullable = true)
```

```
 |-- age: integer (nullable = true)
 |-- gender: string (nullable = true)

+-----+---+------+
| name|age|gender|
+-----+---+------+
| Paul| 40|  male|
| John| 40|  male|
|Susan| 40|female|
|Karen| 34|female|
+-----+---+------+
```

In [42]:
```
# Select only the records with age >= 30 - alternative syntax
df_filtered = df.filter(df.age>=30)
```

In [43]:
```
df_filtered.show()
```

```
+-----+---+------+
| name|age|gender|
+-----+---+------+
| Paul| 40|  male|
| John| 40|  male|
|Susan| 40|female|
|Karen| 34|female|
+-----+---+------+
```

In [ ]:

In [ ]:

In [44]:
```
# Compute the average value over column age
dfAverageAge = df.agg({"age": "avg"})
```

In [45]:
```
dfAverageAge.printSchema()
dfAverageAge.show()
```

```
root
 |-- avg(age): double (nullable = true)

+--------+
|avg(age)|
+--------+
|    33.8|
+--------+
```

In [46]:
```
# Compute the average value over column age and the maximum over name
dfAverageAge = df.agg({"age": "avg", "name": "max"})
```

In [47]:
```
dfAverageAge.printSchema()
dfAverageAge.show()
```

```
root
 |-- max(name): string (nullable = true)
 |-- avg(age): double (nullable = true)
```

```
+---------+--------+
|max(name)|avg(age)|
+---------+--------+
|    Susan|    33.8|
+---------+--------+
```

In [ ]:

In [2]:
```
# Solution Ex. 34
```

In [1]:
```
inputPath  = "/data/students/bigdata-01QYD/ex_data/Ex34/data/"
outputPath = "res_out_Ex34_Top/"
```

In [2]:
```
# Read the content of the input file
readingsRDD = sc.textFile(inputPath)
```

In [3]:
```
# Extract the PM10 values
# It can be implemented by using the map transformation
# Split each line and select the third field
pm10ValuesRDD = readingsRDD.map(lambda PM10Reading: float(PM10Reading.split(',')[2])
```

In [4]:
```
# Select/compute the maximum PM10 value by using takeOrdered
maxPM10Value = pm10ValuesRDD.takeOrdered(1, lambda num: -num)[0]
```

In [5]:
```
# Filter the content of readingsRDD
# Select only the line(s) associated with the maxPM10Value
selectedRecordsRDD = readingsRDD.filter(lambda PM10Reading: float(PM10Reading.split(
```

In [6]:
```
# Store the result in the output folder
selectedRecordsRDD.saveAsTextFile(outputPath)
```

In [ ]:
```
maplist = readingsRDD.map(lambda l: (l, float(l.split())))
```

Average value

In [1]:
```
# Solution Ex. 36
```

In [2]:
```
inputPath  = "/data/students/bigdata-01QYD/ex_data/Ex36/data/"
```

In [3]:
```
# Read the content of the input file
readingsRDD = sc.textFile(inputPath)
```

In [4]:
```
# Extract the PM10 values and return a tuple(PM10 value, 1)
# It can be implemented by using the map transformation
# PM10 is the third field of each input string
pm10ValuesRDD = readingsRDD.map(lambda PM10Reading: ( float(PM10Reading.split(',')[2
```

In [5]:
```
# Compute the sum of the PM10 values and the number of input lines (= sum of onses)
sumPM10ValuesCountLines = pm10ValuesRDD.reduce(lambda value1, value2: (value1[0]+val
```

In [ ]:
```
# Compute the average PM10 value
# sumPM10ValuesCountLines[0] is equal to the sum of the input PM10 values
# sumPM10ValuesCountLines[1] is equal to the number of input lines/input values
print("Average=", sumPM10ValuesCountLines[0]/sumPM10ValuesCountLines[1])
```

Critical dates analysis

In [1]:
```
# Solution Ex. 39 bis v1
```

In [1]:
```
inputPath  = "/data/students/bigdata-01QYD/ex_data/Ex39bis/data/sensors.txt" # argv[
outputPath = "res_out_Ex39bisv1" # argv[2]
```

In [2]:
```
# Read the content of the input file
readingsRDD = sc.textFile(inputPath)
```

In [3]:
```
# Apply a filter transformation to select only the lines with PM10>50
readingsHighValueRDD = readingsRDD.filter(lambda PM10Reading: float(PM10Reading.spli
```

In [8]:
```
# Create an RDD of key-value pairs
# Each pair contains a sensorId (key) and a date (value)
# It can be implemented by using the map transformation.
sensorsCriticalDatesRDD = readingsHighValueRDD.map(lambda PM10Reading: (PM10Reading.
```

In [9]:
```
# Create one pair for each sensor (key) with the list of dates associated with that
# by using the groupByKey transformation
finalSensorCriticalDates = sensorsCriticalDatesRDD.groupByKey()
```

In [10]:
```
# The map method is used to transform the content of the iterable over the values of
finalSensorCriticalDateStringFormat = finalSensorCriticalDates.mapValues(lambda date
```

In [30]:
```
#finalSensorCriticalDateStringFormat.collect()
```

In [31]:
```
# All sensors ID from the complete input file
allSensorsRDD = readingsRDD.map(lambda PM10Reading: PM10Reading.split(',')[0])
```

In [32]:
```
#allSensorsRDD.collect()
```

In [21]:
```
# Select the identifiers of the sensors that have never been associated with a PM10
sensorsNeverHighValueRDD = allSensorsRDD.subtract(finalSensorCriticalDates.keys())
```

In [33]:
```
#sensorsNeverHighValueRDD.collect()
```

In [25]:
```
# Map each sensor that has never been associated with a PM10 values greater than 50
# to a tuple/pair (sensorId, [])
sensorsNeverHighValueRDDEmptyList = sensorsNeverHighValueRDD.map(lambda sensorId: (s
```

In [34]:
```
#sensorsNeverHighValueRDDEmptyList.collect()
```

In [28]:
```
# Compute the final result
resultRDD = finalSensorCriticalDateStringFormat.union(sensorsNeverHighValueRDDEmptyL
```

In [35]:
```
#resultRDD.collect()
```

In [ ]:
```
# Store the result in the output folder
resultRDD.saveAsTextFile(outputPath)
```

In [1]:
```
# Solution Ex. 41 v2    Order sensors by number of critical days
```

In [ ]:
```
inputPath  = "/data/students/bigdata-01QYD/ex_data/Ex41/data/sensors.txt" # argv[1]
outputPath = "res_out_Ex41v2/" # argv[2]
k = 1 # sys.argv[3]
```

In [ ]:
```
# Read the content of the input file
readingsRDD = sc.textFile(inputPath)
```

In [3]:
```
# Apply a filter transformation to select only the lines with PM10>50
readingsHighValueRDD = readingsRDD.filter(lambda PM10Reading: float(PM10Reading.spli
```

In [4]:
```
# Create an RDD of key-value pairs
# Each pair contains a sensorId (key) and +1 (value)
# It can be implemented by using the map transformation.
# The function of the map transformation returns a tuple
sensorsPM10CriticalValuesRDD = readingsHighValueRDD.map(lambda PM10Reading: (PM10Rea
```

In [5]:
```
# Count the number of critical values for each sensor by using the reduceByKey trans
# The used function is the sum of the values (the sum of the ones)
sensorsCountsRDD = sensorsPM10CriticalValuesRDD.reduceByKey(lambda value1, value2: v
```

In [6]:
```
# Sort pairs by number of critical values - descending order
sortedNumCriticalValuesSensorRDD = sensorsCountsRDD.sortBy(lambda pair: pair[1], Fal
```

In [7]:
```
# Select the first k elements of sortedNumCriticalValuesSensorRDD.
# sortedNumCriticalValuesSensorRDD is sorted.
# Hence, the first k elements are the ones we are interested in
topKSensorsNumCriticalValues = sortedNumCriticalValuesSensorRDD.take(k)
```

In [8]:
```
# take is an action. Hence, topKCriticalSensors is a local Python variable of the Dr
# Create an RDD of pairs and store it in HDFS by means of the saveAsTextFile method
topKSensorsRDD = sc.parallelize(topKSensorsNumCriticalValues)
```

In [ ]:
```
topKSensorsRDD.saveAsTextFile(outputPath)
```

In [ ]:
```

```

In [19]:
```
# Solution Ex. 43
```
Critical bike sharing station analysis

In [20]:
```
#inputPathReadings = "/data/students/bigdata-01QYD/ex_data/Ex43/data/readings.txt"
#inputPathNeighbors = "/data/students/bigdata-01QYD/ex_data/Ex43/data/neighbors.txt"
#outputPath = "res_out_Ex43/"
#outputPath2 = "res_out_Ex43_2/"
#outputPath3 = "res_out_Ex43_3/"
#thresholdFreeSlots = 3
#thresholdCriticalPercentage = 0.8

inputPathReadings = "data/Ex43/data/readings.txt"
inputPathNeighbors = "data/Ex43/data/neighbors.txt"
outputPath = "res_out_Ex43/"
outputPath2 = "res_out_Ex43_2/"
outputPath3 = "res_out_Ex43_3/"
thresholdFreeSlots = 3
thresholdCriticalPercentage = 0.8
```

In [21]:
```
# Solution Ex. 43 - part I
# Selection of the stations with a percentage of critical situations
# greater than 80%
```

In [22]:
```
# Read the content of the readings file
readingsRDD = sc.textFile(inputPathReadings).cache()
```

In [23]:
```
def criticalSituation(line):
    fields = line.split(",")
    # fields[0] is the station id
    # fields[5] is the number of free slots
    stationId = fields[0]
    numFreeSlots = int(fields[5])

    if  numFreeSlots < thresholdFreeSlots:
        return (stationId, (1, 1))
    else:
        return (stationId, (1, 0))
```

In [24]:
```
# Count the number of total and critical readings for each station
# Create an RDD of pairs with
# key: stationId
# value: (numReadings, numCriticalReadings)
# ------- numReadings: 1 for each input line
# --------numCriticalReadings: 0 if the situation is not critical. 1 if it is critic
stationCountPairRDD = readingsRDD.map(criticalSituation)
```

In [27]:
```
stationCountPairRDD.collect()
```

Out[27]:
```
[('s1', (1, 0)),
 ('s2', (1, 0)),
 ('s3', (1, 0)),
 ('s4', (1, 1)),
 ('s5', (1, 0)),
 ('s1', (1, 0)),
 ('s2', (1, 0)),
```

```
('s3', (1, 1)),
('s4', (1, 1)),
('s5', (1, 0)),
('s1', (1, 0)),
('s3', (1, 0)),
('s4', (1, 1)),
('s5', (1, 1)),
('s1', (1, 0)),
('s2', (1, 0)),
('s3', (1, 0)),
('s5', (1, 0)),
('s5', (1, 0)),
('s1', (1, 1)),
('s2', (1, 1)),
('s3', (1, 1))]
```

In [26]:
```
# Compute the number of total and critical readings for each station
stationTotalCountPairRDD = stationCountPairRDD\
.reduceByKey(lambda c1, c2: (c1[0]+c2[0], c1[1]+c2[1]) )
```

In [28]:
```
stationTotalCountPairRDD.collect()
```

Out[28]:
```
[('s1', (5, 1)),
 ('s3', (5, 2)),
 ('s2', (4, 1)),
 ('s4', (3, 3)),
 ('s5', (5, 1))]
```

In [29]:
```
# Compute the percentage of critical situations for each station
stationPercentagePairRDD = stationTotalCountPairRDD\
.mapValues(lambda counters: counters[1]/counters[0])
```

In [30]:
```
stationPercentagePairRDD.collect()
```

Out[30]:
```
[('s1', 0.2), ('s3', 0.4), ('s2', 0.25), ('s4', 1.0), ('s5', 0.2)]
```

In [31]:
```
# Select stations with percentage > 80%
selectedStationsPairRDD = stationPercentagePairRDD\
.filter(lambda sensorPerc: sensorPerc[1]>thresholdCriticalPercentage)
```

In [32]:
```
selectedStationsPairRDD.collect()
```

Out[32]:
```
[('s4', 1.0)]
```

In [33]:
```
# Sort the stored stations by decreasing percentage of critical situations
selectedStationsSortedPairRDD = selectedStationsPairRDD\
.sortBy(lambda sensorPerc: sensorPerc[1], ascending=False)
```

In [34]:
```
selectedStationsSortedPairRDD.collect()
```

Out[34]:
```
[('s4', 1.0)]
```

In [35]:
```
selectedStationsSortedPairRDD.saveAsTextFile(outputPath)
```

In [36]:
```
# Solution Ex. 43 - part II
# Selection of the pairs (timeslot, station) with a percentage of
# critical situations greater than 80%
```

In [37]:
```
def criticalSituationTimeslots(line):

    fields = line.split(",")

    # fields[0] is the station id
    # fields[2] is the hour
    # fields[5] is the number of free slots

    stationId = fields[0]
    numFreeSlots = int(fields[5])

    minTimeslotHour = 4 * ( int(fields[2]) // int(4))
    maxTimeslotHour = minTimeslotHour + 3

    timestamp = "ts[" + str(minTimeslotHour) + "-" + str(maxTimeslotHour) + "]"

    key = (timestamp, stationId)

    if  numFreeSlots < thresholdFreeSlots:
        return (key, (1, 1))
    else:
        return (key, (1, 0))
```

In [38]:
```
# The input data are already in readingsRDD

# Count the number of total and critical readings for each (timeslot,stationId)
# Create an RDD of pairs with
# key: (timeslot,stationId)
# value: (numReadings, numCriticalReadings)
# ------- numReadings: 1 for each input line
# --------numCriticalReadings: 0 if the situation is not critical. 1 if it is critic

timestampStationCountPairRDD = readingsRDD.map(criticalSituationTimeslots)
```

In [39]:
```
timestampStationCountPairRDD.collect()
```

Out[39]:
```
[(('ts[0-3]', 's1'), (1, 0)),
 (('ts[0-3]', 's2'), (1, 0)),
 (('ts[0-3]', 's3'), (1, 0)),
 (('ts[0-3]', 's4'), (1, 1)),
 (('ts[0-3]', 's5'), (1, 0)),
 (('ts[0-3]', 's1'), (1, 0)),
 (('ts[0-3]', 's2'), (1, 0)),
 (('ts[0-3]', 's3'), (1, 1)),
 (('ts[0-3]', 's4'), (1, 1)),
 (('ts[0-3]', 's5'), (1, 0)),
 (('ts[12-15]', 's1'), (1, 0)),
 (('ts[12-15]', 's3'), (1, 0)),
 (('ts[12-15]', 's4'), (1, 1)),
 (('ts[12-15]', 's5'), (1, 1)),
 (('ts[12-15]', 's1'), (1, 0)),
 (('ts[12-15]', 's2'), (1, 0)),
 (('ts[12-15]', 's3'), (1, 0)),
 (('ts[12-15]', 's5'), (1, 0)),
 (('ts[0-3]', 's5'), (1, 0)),
 (('ts[0-3]', 's1'), (1, 1)),
```

```
(('ts[0-3]', 's2'), (1, 1)),
(('ts[0-3]', 's3'), (1, 1))]
```

In [22]:
```
# Compute the number of total and critical readings for each (timeslot,station)
timestampStationTotalCountPairRDD = timestampStationCountPairRDD \
.reduceByKey(lambda c1, c2: (c1[0]+c2[0], c1[1]+c2[1]) )
```

In [23]:
```
#timestampStationTotalCountPairRDD.collect()
```

In [24]:
```
# Compute the percentage of critical situations for each (timeslot,station)
timestampStationPercentagePairRDD = timestampStationTotalCountPairRDD\
.mapValues(lambda counters: counters[1]/counters[0])
```

In [25]:
```
#timestampStationPercentagePairRDD.collect()
```

In [26]:
```
# Select (timeslot,station) pairs with percentage > 80%
selectedTimestampStationsPairRDD = timestampStationPercentagePairRDD\
.filter(lambda sensorPerc: sensorPerc[1]>thresholdCriticalPercentage)
```

In [27]:
```
#selectedTimestampStationsPairRDD.collect()
```

In [28]:
```
# Sort the stored pairs by decreasing percentage of critical situations
percentageTimestampStationsSortedPairRDD = selectedTimestampStationsPairRDD\
.sortBy(lambda sensorPerc: sensorPerc[1], ascending=False)
```

In [30]:
```
#percentageTimestampStationsSortedPairRDD.collect()
```

In [31]:
```
percentageTimestampStationsSortedPairRDD.saveAsTextFile(outputPath2)
```

In [40]:
```
# Solution Ex. 43 - part III
# Select a reading (i.e., a line) of the first input file if and only if the followi
# - The line is associated with a full station situation
# - All the neighbor stations of the station Si are full in the time stamp associate
```

In [41]:
```
# Read the file containing the list of neighbors for each station
neighborsRDD = sc.textFile(inputPathNeighbors)
```

In [42]:
```
# Map each line of the input file to a pair stationid, list of neighbor stations
nPairRDD = neighborsRDD.map(lambda line: (line.split(",")[0], line.split(",")[1].spl
```

In [43]:
```
nPairRDD.collect()
```

Out[43]:
```
[('s1', ['s2', 's3']),
 ('s2', ['s1', 's5']),
 ('s3', ['s1']),
 ('s4', ['s5']),
 ('s5', ['s4', 's2'])]
```

In [44]:
```
# Create a local dictionary in the main memory of the driver that will be used to st
# stationid -> list of neighbors
# There are only 100 stations. Hence, you can suppose that data about neighbors can
neighbors=nPairRDD.collectAsMap()
```

In [46]:
```
neighbors
# The input data are already in readingsRDD
```

Out[46]:
```
{'s1': ['s2', 's3'],
 's2': ['s1', 's5'],
 's3': ['s1'],
 's4': ['s5'],
 's5': ['s4', 's2']}
```

In [47]:
```
# Select the lines/readings associated with a full status (number of free slots equa
fullStatusLines = readingsRDD.filter(lambda line: int(line.split(",")[5])==0)
```

In [48]:
```
def extractTimestamp(reading):
    fields = reading.split(",")
    timestamp = fields[1] + fields[2] + fields[3]

    return timestamp
```

In [49]:
```
# Create an RDD of pairs with key = timestamp and value=reading associated with that
# The concatenation of fields[1], fields[2], fields[3] is the timestamp of the readi
fullLinesPRDD = fullStatusLines.map(lambda reading: (extractTimestamp(reading), read
```

In [50]:
```
fullLinesPRDD.collect()
```

Out[50]:
```
[('2015-05-020000', 's1,2015-05-02,00,00,9,0'),
 ('2015-05-020000', 's2,2015-05-02,00,00,8,0'),
 ('2015-05-020000', 's3,2015-05-02,00,00,9,0')]
```

In [51]:
```
#  Collapse all the values with the same key in one single pair (timestamp, reading
fullReadingsPerTimestamp = fullLinesPRDD.groupByKey()
```

In [52]:
```
fullReadingsPerTimestamp.mapValues(lambda v: list(v)).collect()
```

Out[52]:
```
[('2015-05-020000',
  ['s1,2015-05-02,00,00,9,0',
   's2,2015-05-02,00,00,8,0',
   's3,2015-05-02,00,00,9,0'])]
```

In [53]:
```
def selectReadingssFunc(pairTimeStampListReadings):
    # Extract the list of stations that appear in the readings
    # associated with the current key
    # (i.e., the list of stations that are full in this timestamp)
    # The list of readings is in the value part of the inpput key-value pair
    stations = []
    for reading in pairTimeStampListReadings[1]:
        # Extract the stationid from each reading
        fields = reading.split(",")
        stationId = fields[0]
        stations.append(stationId)
```

```
    # Iterate again over the list of readings to select the readings satistying the
    # full status situation of all neighboors
    selectedReading = []

    for reading in pairTimeStampListReadings[1]:
        # This reading must be selected if all the neighbors of
        # the station of this reading are also in the value of
        # the current key-value pair (i.e., if they are in list stations)
        # Extract the stationid of this reading
        fields = reading.split(",")
        stationId = fields[0]

        # Select the list of neighbors of the current station
        nCurrentStation = neighbors[stationId]

        # Check if all the neighbors of the current station are in value
        # (i.e., the local list stations) of the current key-value pair
        allNeighborsFull = True

        for neighborStation in nCurrentStation:
            if neighborStation not in stations:
                # There is at least one neighbor of th current station
                # that is not in the full status in this timestamp
                allNeighborsFull = False

        if allNeighborsFull == True:
            selectedReading.append(reading)

    return selectedReading
```

In [54]:
```
# Each pair contains a timestamp and the list of readings (with number of free slots
# associated with that timestamp.
# Check, for each reading in the list, if all the neighbors of the station of that r
# also present in this list of readings
# Emit one "string" for each reading associated with a completely full status
selectedReadingsRDD = fullReadingsPerTimestamp.flatMap(selectReadingssFunc)
```

In [55]:
```
selectedReadingsRDD.collect()
```

Out[55]:
```
['s1,2015-05-02,00,00,9,0', 's3,2015-05-02,00,00,9,0']
```

In [130…
```
# Store the result in HDFS
selectedReadingsRDD.saveAsTextFile(outputPath3)
```

In [ ]:

In [1]:
```
# Solution Ex. 45     Profile update
```

In [2]:
```
#inputPathWatched = "/data/students/bigdata-01QYD/ex_data/Ex45/data/watchedmovies.tx
#inputPathPreferences = "/data/students/bigdata-01QYD/ex_data/Ex45/data/preferences.
#inputPathMovies = "/data/students/bigdata-01QYD/ex_data/Ex45/data/movies.txt"
#outputPath = "res_out_Ex45/"
#threshold = 0.5

inputPathWatched = "data/Ex45/data/watchedmovies.txt"
inputPathPreferences = "data/Ex45/data/preferences.txt"
inputPathMovies = "data/Ex45/data/movies.txt"
outputPath = "res_out_Ex45/"
threshold = 0.5
```

In [3]:
```
# Read the content of the watched movies file
watchedRDD = sc.textFile(inputPathWatched)
```

In [4]:
```
# Select only userid and movieid
# Define an RDD or pairs with movieid as key and userid as value
movieUserPairRDD = watchedRDD.map(lambda line:  (line.split(",")[1], line.split(",")
movieUserPairRDD.collect()
```

Out[4]:
```
[('movie1', 'user1'),
 ('movie3', 'user1'),
 ('movie4', 'user1'),
 ('movie5', 'user1'),
 ('movie6', 'user2'),
 ('movie3', 'user2'),
 ('movie4', 'user2'),
 ('movie7', 'user2'),
 ('movie3', 'user2'),
 ('movie4', 'user2')]
```

In [5]:
```
# Read the content of the movies file
moviesRDD = sc.textFile(inputPathMovies)
```

In [6]:
```
# Select only movieid and genre
# Define an RDD of pairs with movieid as key and genre as value
movieGenrePairRDD = moviesRDD.map(lambda line:  (line.split(",")[0], line.split(",")
movieGenrePairRDD.collect()
```

Out[6]:
```
[('movie1', 'Animation'),
 ('movie2', 'Adventure'),
 ('movie3', 'Comedy'),
 ('movie4', 'Comedy'),
 ('movie5', 'Comedy'),
 ('movie6', 'Action'),
 ('movie7', 'Comedy'),
 ('movie8', 'Adventure'),
 ('movie9', 'Action'),
 ('movie10', 'Action')]
```

In [7]:
```
# Join watched movies with movies
joinWatchedGenreRDD = movieUserPairRDD.join(movieGenrePairRDD)
joinWatchedGenreRDD.collect()
```

Out[7]:
```
[('movie5', ('user1', 'Comedy')),
 ('movie1', ('user1', 'Animation')),
 ('movie4', ('user1', 'Comedy')),
 ('movie4', ('user2', 'Comedy')),
 ('movie4', ('user2', 'Comedy')),
 ('movie3', ('user1', 'Comedy')),
 ('movie3', ('user2', 'Comedy')),
 ('movie3', ('user2', 'Comedy')),
 ('movie6', ('user2', 'Action')),
 ('movie7', ('user2', 'Comedy'))]
```

In [8]:
```
# Select only userid (as key) and genre (as value)
usersWatchedGenresRDD = joinWatchedGenreRDD.map(lambda pair: (pair[1][0], pair[1][1]
usersWatchedGenresRDD.collect()
```

Out[8]:
```
[('user1', 'Comedy'),
 ('user1', 'Animation'),
 ('user1', 'Comedy'),
 ('user2', 'Comedy'),
 ('user2', 'Comedy'),
 ('user1', 'Comedy'),
 ('user2', 'Comedy'),
 ('user2', 'Comedy'),
 ('user2', 'Action'),
 ('user2', 'Comedy')]
```

In [9]:
```
# Read the content of preferences.txt
preferencesRDD = sc.textFile(inputPathPreferences)
```

In [10]:
```
# Define an RDD of pairs with userid as key and genre as value
userLikedGenresRDD = preferencesRDD.map(lambda line: (line.split(",")[0], line.split
```

In [11]:
```
# Cogroup the lists of watched and liked genres for each user
# There is one pair for each userid
# the value contains the list of genres (with repetitions) of the
# watched movies and the list of liked genres
userWatchedLikedGenres = usersWatchedGenresRDD.cogroup(userLikedGenresRDD)
userWatchedLikedGenres.mapValues(lambda v: (list(v[0]), list(v[1]))).collect()
```

Out[11]:
```
[('user2',
  (['Comedy', 'Comedy', 'Comedy', 'Comedy', 'Action', 'Comedy'], ['Action'])),
 ('user1',
  (['Comedy', 'Animation', 'Comedy', 'Comedy'], ['Animation', 'Comedy']))]
```

In [12]:
```
# This function is used in the next transformation to select users with a misleading
def misleadingProfileFunc(userWatchedLikedGenresLists):
    # Store in a local list the "small" set of liked genres
    # associated with the current user
    likedGenres = list(userWatchedLikedGenresLists[1][1])

    # Iterate over the watched movies (the genres of the watched movies)and count
    # - The number of watched movies for this user
    # - How many of watched movies are associated with a not liked genre
    numWatchedMovies = 0
    notLiked = 0

    for watchedGenre in userWatchedLikedGenresLists[1][0]:
        numWatchedMovies = numWatchedMovies+1
        if watchedGenre not in likedGenres:
            notLiked = notLiked+1
```

```
    # Check if the number of watched movies associated with a non-liked genre
    # is greater that threshold%
    if float(notLiked) > threshold * float(numWatchedMovies):
        return True
    else:
        return False
```

In [13]:
```
# Filter the users with a misleading profile
misleadingUsersListsRDD = userWatchedLikedGenres.filter(misleadingProfileFunc)
misleadingUsersListsRDD.mapValues(lambda v: (list(v[0]), list(v[1]))).collect()
```

Out[13]:
```
[('user2',
  (['Comedy', 'Comedy', 'Comedy', 'Comedy', 'Action', 'Comedy'], ['Action']))]
```

In [14]:
```
# This function is used in the next transformation to select the pairs (userid,misle
def misleadingGenresFunc(userWatchedLikedGenresLists):
    # Store in a local list the "small" set of liked genres
    # associated with the current user

    userId = userWatchedLikedGenresLists[0]
    likedGenres = list(userWatchedLikedGenresLists[1][1])

    # In this solution I suppose that the number of distinct genres for each user
    # is small and can be stored in a local variable.
    # The local variable is a dictionary that stores for each non-liked genre
    # also its number of occurrences in the list of watched movies of the current us
    numGenres = {}


    # Iterate over the watched movies (the genres of the watched movies).
    # Select the watched genres that are not in the liked genres and
    # count their number of occurrences. Store them in the numGenres dictionary
    for watchedGenre in userWatchedLikedGenresLists[1][0]:
        # Check if the genre is not in the liked ones
        if watchedGenre not in likedGenres:
            # Update the number of times this genre appears
            # in the list of movies watched by the current user
            if watchedGenre in numGenres:
                numGenres[watchedGenre] = numGenres[watchedGenre] + 1
            else:
                numGenres[watchedGenre] = 1

    # Select the genres, which are not in the liked ones,
    # which occur at least 5 times
    selectedGenres = []

    for genre, occurrences in numGenres.items():
        if occurrences>=5:
            selectedGenres.append( (userId, genre) )

    return selectedGenres
```

In [15]:
```
# Select the pairs (userid,misleading genre)
misleadingUserGenrePairRDD = misleadingUsersListsRDD.flatMap(misleadingGenresFunc)
```

In [16]:
```
misleadingUserGenrePairRDD.collect()
```

Out[16]:
```
[('user2', 'Comedy')]
```

In [ ]:
```
misleadingUserGenrePairRDD.saveAsTextFile(outputPath)
```

In [1]:
```
inputData = "/data/students/bigdata-01QYD/ex_data/Ex48/data/"
outputPath = "resOut_ex48/"
```

In [2]:
```
# Create a DataFrame from persons.csv
dfPersons = spark.read.load(inputData,\
                    format="csv",\
                    header=True,\
                    inferSchema=True)
```

Select the names occurring at least two times and store in the output folder name and average(age) of the selected names

In [3]:
```
dfPersons.printSchema()
dfPersons.show()
```

```
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- gender: string (nullable = true)

+-----+---+------+
| name|age|gender|
+-----+---+------+
| Paul| 40|  male|
| Paul| 38|  male|
|David| 15|  male|
|Susan| 40|female|
|Susan| 34|female|
+-----+---+------+
```

In [4]:
```
# Define one group for each name and count the number of records per name
dfNameCountRecords = dfPersons.groupBy("name").agg({"name": "count", "age": "avg"})
```

In [5]:
```
dfNameCountRecords.printSchema()
dfNameCountRecords.show()
```

```
root
 |-- name: string (nullable = true)
 |-- count(name): long (nullable = false)
 |-- avg(age): double (nullable = true)

+-----+-----------+--------+
| name|count(name)|avg(age)|
+-----+-----------+--------+
|Susan|          2|    37.0|
|David|          1|    15.0|
| Paul|          2|    39.0|
+-----+-----------+--------+
```

In [6]:
```
# Select only the name occurring at least two times
dfSelectedNames = dfNameCountRecords.filter("count(name)>=2")
```

In [8]:
```
dfSelectedNames.printSchema()
dfSelectedNames.show()
```

```
root
 |-- name: string (nullable = true)
 |-- count(name): long (nullable = false)
 |-- avg(age): double (nullable = true)
```

```
+-----+-----------+--------+
| name|count(name)|avg(age)|
+-----+-----------+--------+
|Susan|          2|    37.0|
| Paul|          2|    39.0|
+-----+-----------+--------+
```

In [9]:
```
# Select only the name and average(age) of the selected names
dfSelectedNamesAvgAge = dfSelectedNames.select("name", "avg(age)")
```

In [10]:
```
dfSelectedNamesAvgAge.printSchema()
dfSelectedNamesAvgAge.show()
```

```
root
 |-- name: string (nullable = true)
 |-- avg(age): double (nullable = true)

+-----+--------+
| name|avg(age)|
+-----+--------+
|Susan|    37.0|
| Paul|    39.0|
+-----+--------+
```

In [ ]:
```
dfSelectedNamesAvgAge.write.csv(outputPath, header=False)
```

In [ ]:
```
#dfSelectedNamesAvgAge.explain()
```

In [ ]:
```

```

In [29]:
```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
#from pyspark.ml.feature import Tokenizer
#from pyspark.ml.feature import StopWordsRemover
#from pyspark.ml.feature import HashingTF
#from pyspark.ml.feature import IDF
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
from pyspark.sql.types import *
```

In [30]:
```
# input and output folders
trainingData = "data/Ex51/data//trainingData.csv"
unlabeledData = "data/Ex51/data//unlabeledData.csv"
outputPath = "res_ex51/"
```

In [31]:
```
# ************************
# Training step
# ************************

# Create a DataFrame from trainingData.csv
# Training data in raw format
trainingData = spark.read.load(trainingData,\
                    format="csv",\
                    header=True,\
                    inferSchema=True)
```

In [32]:
```
trainingData.printSchema()
trainingData.show()
```

```
root
 |-- label: integer (nullable = true)
 |-- text: string (nullable = true)

+-----+--------------------+
|label|                text|
+-----+--------------------+
|    1|The Spark system ...|
|    1|Spark is a new di...|
|    0|Turin is a beauti...|
|    0|Turin is in the n...|
+-----+--------------------+
```

In [33]:
```
# Define a Python function that returns the number of words occurring in the input s
def countWords(text):
    return len(text.split(" "))
```

In [34]:
```
# Register a UDF function associated with countWords
# We explicitly report also the data type of the returned value
spark.udf.register("countWords", countWords, IntegerType())
```

Out[34]:
```
<function __main__.countWords(text)>
```

In [35]:
```
# Define a Python function that checks if the input string contain the work "Spark"
def containsSpark(text):
```

```
    return text.find("Spark")>=0
```

In [36]:
```
# Register a UDF function associated with containsSpark
# We explicitly report also the data type of the returned value
spark.udf.register("containsSpark", containsSpark, BooleanType())
```

Out[36]:
```
<function __main__.containsSpark(text)>
```

In [37]:
```
# Select the attributes label and text and create  two new columns:
# numLines and SparkWord
newFeaturesDF = trainingData\
.selectExpr("label", "text", "countWords(text)", "containsSpark(text)")\
.withColumnRenamed("countWords(text)", "numLines")\
.withColumnRenamed("containsSpark(text)", "SparkWord")
```

In [38]:
```
newFeaturesDF.printSchema()
newFeaturesDF.show()
```

```
root
 |-- label: integer (nullable = true)
 |-- text: string (nullable = true)
 |-- numLines: integer (nullable = true)
 |-- SparkWord: boolean (nullable = true)

+-----+--------------------+--------+---------+
|label|                text|numLines|SparkWord|
+-----+--------------------+--------+---------+
|    1|The Spark system ...|       7|     true|
|    1|Spark is a new di...|       6|     true|
|    0|Turin is a beauti...|       5|    false|
|    0|Turin is in the n...|       8|    false|
+-----+--------------------+--------+---------+
```

In [39]:
```
# Use an assembler to combine "numLines" and "SparkWord" in a Vector
assembler = VectorAssembler(inputCols=["numLines", "SparkWord"],\
                            outputCol="features")
```

In [40]:
```
# Create a classification model based on the logistic regression algorithm
# We can set the values of the parameters of the
# Logistic Regression algorithm using the setter methods.
lr = LogisticRegression()\
.setMaxIter(10)\
.setRegParam(0.01)
```

In [41]:
```
# Define the pipeline that is used to create the logistic regression
# model on the training data.
# In this case the pipeline is composed of five steps
# - text tokenizer
# - stopword removal
# - TF-IDF computation (performed in two steps)
# - Logistic regression model generation
pipeline = Pipeline().setStages([assembler, lr])
```

In [42]:
```
# Execute the pipeline on the training data to build the
# classification model
```

```
classificationModel = pipeline.fit(newFeaturesDF)

# Now, the classification model can be used to predict the class label
# of new unlabeled data
```

In [43]:
```
# ************************
# Prediction  step
# ************************
# Read unlabeled data
# Create a DataFrame from unlabeledData.csv
# Unlabeled data in raw format
unlabeledData = spark.read.load(unlabeledData,\
                    format="csv", header=True, inferSchema=True)
```

In [44]:
```
#unlabeledData.printSchema()
#unlabeledData.show()
```

In [45]:
```
newFeaturesDFunlabeled = unlabeledData\
.selectExpr("label", "text", "countWords(text)", "containsSpark(text)")\
.withColumnRenamed("countWords(text)", "numLines")\
.withColumnRenamed("containsSpark(text)", "SparkWord")
```

In [46]:
```
newFeaturesDFunlabeled.printSchema()
newFeaturesDFunlabeled.show()
```

```
root
 |-- label: string (nullable = true)
 |-- text: string (nullable = true)
 |-- numLines: integer (nullable = true)
 |-- SparkWord: boolean (nullable = true)

+-----+--------------------+--------+---------+
|label|                text|numLines|SparkWord|
+-----+--------------------+--------+---------+
| null|Spark performs be...|       5|     true|
| null|Comparison betwee...|       5|     true|
| null|Turin is in Piedmont|       4|    false|
+-----+--------------------+--------+---------+
```

In [47]:
```
# Make predictions on unlabeled documents by using the
# Transformer.transform() method.
# The transform will only use the 'features' columns
predictionsDF = classificationModel.transform(newFeaturesDFunlabeled)
```

In [48]:
```
predictionsDF.printSchema()
predictionsDF.show()
```

```
root
 |-- label: string (nullable = true)
 |-- text: string (nullable = true)
 |-- numLines: integer (nullable = true)
 |-- SparkWord: boolean (nullable = true)
 |-- features: vector (nullable = true)
 |-- rawPrediction: vector (nullable = true)
 |-- probability: vector (nullable = true)
 |-- prediction: double (nullable = false)
```

```
+-----+--------------------+--------+---------+---------+--------------------+------
--------------+----------+
|label|                text|numLines|SparkWord| features|       rawPrediction|
probability|prediction|
+-----+--------------------+--------+---------+---------+--------------------+------
--------------+----------+
| null|Spark performs be...|       5|     true|[5.0,1.0]|[-3.1272480248757...|[0.041
99718899423...|       1.0|
| null|Comparison betwee...|       5|     true|[5.0,1.0]|[-3.1272480248757...|[0.041
99718899423...|       1.0|
| null|Turin is in Piedmont|       4|    false|[4.0,0.0]|[3.19966999960023...|[0.960
82185681571...|       0.0|
+-----+--------------------+--------+---------+---------+--------------------+------
--------------+----------+
```

In [49]:

```
# The returned DataFrame has the following schema (attributes)
# |-- label: string (nullable = true)
# |-- text: string (nullable = true)
# |-- words: array (nullable = true)
# |    |-- element: string (containsNull = true)
# |-- filteredWords: array (nullable = true)
# |    |-- element: string (containsNull = true)
# |-- rawFeatures: vector (nullable = true)
# |-- features: vector (nullable = true)
# |-- rawPrediction: vector (nullable = true)
# |-- probability: vector (nullable = true)
# |-- prediction: double (nullable = false)

# Select only the original features (i.e., the value of the original text attribute)
# the predicted class for each record
predictions = predictionsDF.select("text", "prediction")
```

In [50]:

```
predictions.printSchema()
predictions.show(truncate=False)
```

```
root
 |-- text: string (nullable = true)
 |-- prediction: double (nullable = false)

+---------------------------------+----------+
|text                             |prediction|
+---------------------------------+----------+
|Spark performs better than Hadoop  |1.0       |
|Comparison between Spark and Hadoop|1.0       |
|Turin is in Piedmont               |0.0       |
+---------------------------------+----------+
```

In [51]:

```
# Save the result in an HDFS output folder
predictions.write.csv(outputPath, header="true")
```

In [ ]:

In [1]:
```
from graphframes import GraphFrame
```

In [2]:
```
inputPathVertexes = "data/Ex56/data/vertexes.csv"
inputPathEdges = "data/Ex56/data/edges.csv"
outputPath = "resOut_ex56/"
```

In [3]:
```
# Read the content of vertexes.csv
vDF = spark.read.load(inputPathVertexes,\
                      format="csv",
                      header=True,\
                      inferSchema=True)
```

In [4]:
```
vDF.printSchema()
vDF.show()
```

```
root
 |-- id: string (nullable = true)
 |-- entityName: string (nullable = true)
 |-- name: string (nullable = true)

+---+----------+--------+
| id|entityName|    name|
+---+----------+--------+
| V1|      user|   Paolo|
| V2|     topic|     SQL|
| V3|      user|   David|
| V4|     topic|Big Data|
| V5|      user|    John|
+---+----------+--------+
```

In [7]:
```
# Read the content of edges.csv
eDF = spark.read.load(inputPathEdges,\
                      format="csv",
                      header=True,\
                      inferSchema=True)
```

In [8]:
```
eDF.printSchema()
eDF.show()
```

```
root
 |-- src: string (nullable = true)
 |-- dst: string (nullable = true)
 |-- linktype: string (nullable = true)

+---+---+----------+
|src|dst|  linktype|
+---+---+----------+
| V1| V2|      like|
| V1| V3|    follow|
| V1| V4|    follow|
| V3| V2|    follow|
| V3| V4|    follow|
| V5| V2|  expertOf|
| V2| V4|correlated|
| V4| V2|correlated|
+---+---+----------+
```

In [9]:
```
# Only the "follow"  and "correlated" edges are useful
# Filter the input edge dataframe before creating the graph
filteredEdfes = eDF.filter("linktype='follow' OR linktype='correlated' ")
```

In [10]:
```
# Create the input graph
g = GraphFrame(vDF, filteredEdfes)
```

In [13]:
```
pathsDF = g.find("(v1)-[e1]->(v2);(v2)-[e2]->(v3)")
```

In [14]:
```
pathsDF.printSchema()
pathsDF.show()
```

```
root
 |-- v1: struct (nullable = false)
 |    |-- id: string (nullable = true)
 |    |-- entityName: string (nullable = true)
 |    |-- name: string (nullable = true)
 |-- e1: struct (nullable = false)
 |    |-- src: string (nullable = true)
 |    |-- dst: string (nullable = true)
 |    |-- linktype: string (nullable = true)
 |-- v2: struct (nullable = false)
 |    |-- id: string (nullable = true)
 |    |-- entityName: string (nullable = true)
 |    |-- name: string (nullable = true)
 |-- e2: struct (nullable = false)
 |    |-- src: string (nullable = true)
 |    |-- dst: string (nullable = true)
 |    |-- linktype: string (nullable = true)
 |-- v3: struct (nullable = false)
 |    |-- id: string (nullable = true)
 |    |-- entityName: string (nullable = true)
 |    |-- name: string (nullable = true)

+--------------------+--------------------+--------------------+--------------------
+--------------------+
|                  v1|                  e1|                  v2|                  e2
|                  v3|
+--------------------+--------------------+--------------------+--------------------
+--------------------+
|   [V1, user, Paolo]|    [V1, V3, follow]|   [V3, user, David]|    [V3, V4, follow]
|[V4, topic, Big D...|
|   [V1, user, Paolo]|    [V1, V3, follow]|   [V3, user, David]|    [V3, V2, follow]
|    [V2, topic, SQL]|
|   [V1, user, Paolo]|    [V1, V4, follow]|[V4, topic, Big D...|[V4, V2, correlated]
|    [V2, topic, SQL]|
|   [V3, user, David]|    [V3, V2, follow]|    [V2, topic, SQL]|[V2, V4, correlated]
|[V4, topic, Big D...|
|   [V3, user, David]|    [V3, V4, follow]|[V4, topic, Big D...|[V4, V2, correlated]
|    [V2, topic, SQL]|
|    [V2, topic, SQL]|[V2, V4, correlated]|[V4, topic, Big D...|[V4, V2, correlated]
|    [V2, topic, SQL]|
|[V4, topic, Big D...|[V4, V2, correlated]|    [V2, topic, SQL]|[V2, V4, correlated]
|[V4, topic, Big D...|
+--------------------+--------------------+--------------------+--------------------
+--------------------+
```

In [18]:
```
# Select the triples user -> follow -> topic -> correlated -> topic="Big data"
selectedPathsDF = pathsDF.filter("""v1.entityName='user'
AND e1.linktype='follow'
AND v2.entityName='topic'
```

```
AND e2.linktype='correlated'
AND v3.entityName='topic' AND v3.name='Big Data' """)
```

In [19]:

```
selectedPathsDF.printSchema()
selectedPathsDF.show()
```

```
root
 |-- v1: struct (nullable = false)
 |    |-- id: string (nullable = true)
 |    |-- entityName: string (nullable = true)
 |    |-- name: string (nullable = true)
 |-- e1: struct (nullable = false)
 |    |-- src: string (nullable = true)
 |    |-- dst: string (nullable = true)
 |    |-- linktype: string (nullable = true)
 |-- v2: struct (nullable = false)
 |    |-- id: string (nullable = true)
 |    |-- entityName: string (nullable = true)
 |    |-- name: string (nullable = true)
 |-- e2: struct (nullable = false)
 |    |-- src: string (nullable = true)
 |    |-- dst: string (nullable = true)
 |    |-- linktype: string (nullable = true)
 |-- v3: struct (nullable = false)
 |    |-- id: string (nullable = true)
 |    |-- entityName: string (nullable = true)
 |    |-- name: string (nullable = true)

+-----------------+----------------+----------------+--------------------+----------
----------+
|               v1|              e1|              v2|                  e2|
v3|
+-----------------+----------------+----------------+--------------------+----------
----------+
|[V3, user, David]|[V3, V2, follow]|[V2, topic, SQL]|[V2, V4, correlated]|[V4, topi
c, Big D...|
+-----------------+----------------+----------------+--------------------+----------
----------+
```

In [21]:

```
# Select name of the selected users
usersDF = selectedPathsDF.selectExpr("v1.name as username")
```

In [22]:

```
usersDF.show()
```

```
+--------+
|username|
+--------+
|   David|
+--------+
```

In [23]:

```
# Save the result in the output folder
usersDF.write.csv(outputPath, header=True)
```

In [ ]:

In [10]:
```
from graphframes import GraphFrame
```

In [11]:
```
inputPathVertexes = "data/Ex57/data/vertexes.csv"
inputPathEdges = "data/Ex57/data/edges.csv"
outputPath = "resOut_ex57/"
```

In [12]:
```
# Read the content of vertexes.csv
vDF = spark.read.load(inputPathVertexes,\
                              format="csv",
                              header=True,\
                              inferSchema=True)
```

In [13]:
```
vDF.printSchema()
vDF.show()
```

```
root
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)

+---+-----+---+
| id| name|age|
+---+-----+---+
| u1|Alice| 34|
| u2|  Bob| 36|
| u3| John| 30|
| u4|David| 29|
| u5| Paul| 32|
| u6| Adel| 36|
| u7| Eddy| 60|
+---+-----+---+
```

In [14]:
```
# Read the content of edges.csv
eDF = spark.read.load(inputPathEdges,\
                              format="csv",
                              header=True,\
                              inferSchema=True)
```

In [15]:
```
eDF.printSchema()
eDF.show()
```

```
root
 |-- src: string (nullable = true)
 |-- dst: string (nullable = true)
 |-- linktype: string (nullable = true)

+---+---+--------+
|src|dst|linktype|
+---+---+--------+
| u1| u2|  friend|
| u1| u4|  friend|
| u1| u5|  friend|
| u2| u1|  friend|
| u2| u3|  follow|
| u3| u2|  follow|
| u4| u1|  friend|
| u4| u5|  friend|
| u5| u1|  friend|
```

```
| u5| u4|  friend|
| u5| u6|  follow|
| u6| u3|  follow|
+---+---+--------+
```

In [17]:
```
# Create the input graph
g = GraphFrame(vDF, eDF)
```

In [18]:
```
# Compute for each vertex the length of the shortest path to u1
shortPathsLengDF = g.shortestPaths(['u1'])
```

In [20]:
```
shortPathsLengDF.printSchema()
shortPathsLengDF.show()
```

```
root
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- distances: map (nullable = true)
 |    |-- key: string
 |    |-- value: integer (valueContainsNull = false)

+---+-----+---+---------+
| id| name|age|distances|
+---+-----+---+---------+
| u6| Adel| 36|[u1 -> 3]|
| u3| John| 30|[u1 -> 2]|
| u2|  Bob| 36|[u1 -> 1]|
| u4|David| 29|[u1 -> 1]|
| u5| Paul| 32|[u1 -> 1]|
| u1|Alice| 34|[u1 -> 0]|
| u7| Eddy| 60|       []|
+---+-----+---+---------+
```

In [23]:
```
# Select only the users who can reach u1 in less than 3 "hops"
selectedUsersDF=shortPathsLengDF.filter("distances['u1']<3 AND id<>'u1' ")
```

In [24]:
```
selectedUsersDF.printSchema()
selectedUsersDF.show()
```

```
root
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- distances: map (nullable = true)
 |    |-- key: string
 |    |-- value: integer (valueContainsNull = false)

+---+-----+---+---------+
| id| name|age|distances|
+---+-----+---+---------+
| u3| John| 30|[u1 -> 2]|
| u2|  Bob| 36|[u1 -> 1]|
| u4|David| 29|[u1 -> 1]|
| u5| Paul| 32|[u1 -> 1]|
+---+-----+---+---------+
```

In [25]:
```
# Create a DataFrame with Columns name and numHops
```

```
usersNameNumHopsDF=selectedUsersDF.selectExpr("name", "distances['u1'] AS numHops")
```

In [26]:

```
usersNameNumHopsDF.printSchema()
usersNameNumHopsDF.show()
```

```
root
 |-- name: string (nullable = true)
 |-- numHops: integer (nullable = true)

+-----+-------+
| name|numHops|
+-----+-------+
| John|      2|
|  Bob|      1|
|David|      1|
| Paul|      1|
+-----+-------+
```

In [28]:

```
# Save the result in the output folder
usersNameNumHopsDF.write.csv(outputPath, header=True)
```

In [ ]:

In [88]:
```
# Exam 2020/06/27 - Exercise #2 - version #1
```

In [1]:
```
inputPathCars = "exam_ex2_data/Cars.txt"
inputPathFailures = "exam_ex2_data/CarsFailures.txt"
outputPathPart1 = "outPart1"
outputPathPart2 = "outPart2"
```

In [2]:
```
# ****************************************
# Exercise 2 - Part 1
# ****************************************
```

In [3]:
```
# Read the content of CarsFailures.txt
failuresRDD = sc.textFile(inputPathFailures).cache()
```

In [4]:
```
# Select only the failures related to years 2017 and 2018 and failureType=Engine
# 2015/01/05,08:45,Car15,Engine
def years1718EngineFilter(line):
    fields = line.split(",")

    date = fields[0]
    failureType = fields[3]

    if ( (date.startswith("2018") or date.startswith("2017")) and failureType=="Engi
        return True
    else:
        return False

selectedFailuresRDD = failuresRDD.filter(years1718EngineFilter)
```

In [5]:
```
# Count for each car the number of failures in years 2017 and 2018

# Map each input line to a pair
# key = carID
# value = +1 if year is 2018. -1 if year is 2017

def pairCarIDCounters(line):
    fields = line.split(",")
    date = fields[0]
    carId= fields[2]

    if (date.startswith("2017")):
        return (carId, -1)
    else:
        return (carId, +1)

carsCountersRDD = selectedFailuresRDD.map(pairCarIDCounters)
```

In [7]:
```
# Sum the ones
carsTotFailures1718RDD = carsCountersRDD.reduceByKey(lambda v1,v2: v1+v2)
```

In [8]:
```
# Select the cars associated with a value > 0 , i.e, cars with #failures 2018># fail
selectedCarsRDD = carsTotFailures1718RDD.filter(lambda pair: pair[1]>0)
```

In [10]:
```
# Read the content of Cars.txt
carsRDD = sc.textFile(inputPathCars)
```

In [11]:
```
# Extract (carId,model)
# Car12,Panda,FCA,Paris
carModelRDD = carsRDD.map(lambda line: (line.split(",")[0], line.split(",")[1]))
```

In [12]:
```
# Join the content of selectedCarsRDD with carModelRDD
joinSelectedRDD = selectedCarsRDD.join(carModelRDD)
```

In [14]:
```
# Select (carID,model) and store it in output first output folder
# carId is already the key. The must only change the value part in order to select o
selectedCarsModels = joinSelectedRDD.mapValues(lambda value: value[1])
```

In [100…
```
# Store the result
selectedCarsModels.saveAsTextFile(outputPathPart1)
```

In [ ]:
```

```

In [101…
```
# ****************************************
# Exercise 2 - Part 2
# ****************************************
```

In [102…
```
# This function is already provided
from datetime import datetime, timedelta
def previousDate(mydate):
    currentDate=datetime.strptime(mydate,"%Y/%m/%d")
    prevDate=currentDate-timedelta(days=1)
    return prevDate.strftime("%Y/%m/%d")
```

In [103…
```
# Analyze all the failures and emit a pair (CarId, Date) from each failure
def carIdDate(line):
    # 2015/01/05,08:45,Car15,Engine
    fields = line.split(",")

    date = fields[0]
    carId = fields[2]

    return (carId, date)

carsDatesRDD = failuresRDD.map(carIdDate)
```

In [104…
```
# Remove duplicates. The same car can have more than one failure per date
carsDatesDistinctRDD = carsDatesRDD.distinct()
```

In [105…
```
# Emit two pairs from each input pair
# - ((carId,  date), +1)   - First element of the window that starts in this date
# - ((carId,date-1), +1)   - Second element of the window that starts in the previou
def flatMapFunc(pair):
```

```
    returnedPairs = []

    carId = pair[0]
    currDate = pair[1]
    prevDate= previousDate(currDate)

    returnedPairs.append( ((carId, currDate), +1) )
    returnedPairs.append( ((carId, prevDate), +1) )

    return returnedPairs


windowsElementsRDD = carsDatesDistinctRDD.flatMap(flatMapFunc)
```

In [106…

```
# Count the number of elements per windows
windowsNumElementsRDD = windowsElementsRDD.reduceByKey(lambda v1, v2: v1+v2)
```

In [107…

```
# Select the windows with 2 elements
selectedWindowsRDD = windowsNumElementsRDD.filter(lambda pair: pair[1]==2)
```

In [108…

```
# Store the key part of the selected pairs (carId, first date of the sequence)
selectedWindowsRDD.keys().saveAsTextFile(outputPathPart2)
```

In [ ]:

In [1]:
```
# Exam 2020/06/27 - Exercise #2 - version #2
```

In [10]:
```
inputPathCars = "exam_ex2_data/Cars.txt"
inputPathFailures = "exam_ex2_data/CarsFailures.txt"
outputPathPart1 = "outPart1"
outputPathPart2 = "outPart2"
```

In [11]:
```
# ****************************************
# Exercise 2 - Part 1
# ****************************************
```

In [12]:
```
# Read the content of CarsFailures.txt
failuresRDD = sc.textFile(inputPathFailures).cache()
```

In [13]:
```
# Select only the failures related to years 2017 and 2018 and failureType=Engine
# 2015/01/05,08:45,Car15,Engine
def years1718EngineFilter(line):
    fields = line.split(",")

    date = fields[0]
    failureType = fields[3]

    if ( (date.startswith("2018") or date.startswith("2017")) and failureType=="Engi
        return True
    else:
        return False

selectedFailuresRDD = failuresRDD.filter(years1718EngineFilter)
```

In [14]:
```
# Count for each car the number of failures in years 2017 and 2018

# Map each input line to a pair
# key = carID
# value = (2017=1/0, 2018=1/0)

def pairCarIDCounters(line):
    fields = line.split(",")
    date = fields[0]
    carId= fields[2]

    if (date.startswith("2017")):
        return (carId, (1, 0))
    else:
        return (carId, (0, 1))

carsCountersRDD = selectedFailuresRDD.map(pairCarIDCounters)
```

In [15]:
```
# Sum the ones
carsTotFailures1718RDD = carsCountersRDD.reduceByKey(lambda v1,v2: (v1[0]+v2[0], v1[
```

In [16]:
```
# Select the cars with #failures 2018># failurs 2017
selectedCarsRDD = carsTotFailures1718RDD.filter(lambda pair: pair[1][1]>pair[1][0])
```

In [17]:
```
# Read the content of Cars.txt
carsRDD = sc.textFile(inputPathCars)
```

In [18]:
```
# Extract (carId,model)
# Car12,Panda,FCA,Paris
carModelRDD = carsRDD.map(lambda line: (line.split(",")[0], line.split(",")[1]))
```

In [19]:
```
# Join the content of selectedCarsRDD with carModelRDD
joinSelectedRDD = selectedCarsRDD.join(carModelRDD)
```

In [20]:
```
# Select (carID,model) and store it in output first output folder
# carId is already the key. The must only change the value part in order to select o
selectedCarsModels = joinSelectedRDD.mapValues(lambda value: value[1])
```

In [100…
```
# Store the result
selectedCarsModels.saveAsTextFile(outputPathPart1)
```

In [ ]:
```

```

In [101…
```
# ****************************************
# Exercise 2 - Part 2
# ****************************************
```

In [102…
```
# This function is already provided
from datetime import datetime, timedelta
def previousDate(mydate):
    currentDate=datetime.strptime(mydate,"%Y/%m/%d")
    prevDate=currentDate-timedelta(days=1)
    return prevDate.strftime("%Y/%m/%d")
```

In [103…
```
# Analyze all the failures and emit a pair (CarId, Date) from each failure
def carIdDate(line):
    # 2015/01/05,08:45,Car15,Engine
    fields = line.split(",")

    date = fields[0]
    carId = fields[2]

    return (carId, date)

carsDatesRDD = failuresRDD.map(carIdDate)
```

In [104…
```
# Remove duplicates. The same car can have more than one failure per date
carsDatesDistinctRDD = carsDatesRDD.distinct()
```

In [105…
```
# Emit two pairs from each input pair
# - ((carId,  date), +1)   - First element of the window that starts in this date
# - ((carId,date-1), +1)   - Second element of the window that starts in the previou
def flatMapFunc(pair):
```

```
    returnedPairs = []

    carId = pair[0]
    currDate = pair[1]
    prevDate= previousDate(currDate)

    returnedPairs.append( ((carId, currDate), +1) )
    returnedPairs.append( ((carId, prevDate), +1) )

    return returnedPairs


windowsElementsRDD = carsDatesDistinctRDD.flatMap(flatMapFunc)
```

In [106…

```
# Count the number of elements per windows
windowsNumElementsRDD = windowsElementsRDD.reduceByKey(lambda v1, v2: v1+v2)
```

In [107…

```
# Select the windows with 2 elements
selectedWindowsRDD = windowsNumElementsRDD.filter(lambda pair: pair[1]==2)
```

In [108…

```
# Store the key part of the selected pairs (carId, first date of the sequence)
selectedWindowsRDD.keys().saveAsTextFile(outputPathPart2)
```

In [ ]:

In [30]:
```
# Exam 2020/06/27 - Exercise #2 - SQL-based version #1
```

In [3]:
```
inputPathCars = "exam_ex2_data/Cars.txt"
inputPathFailures = "exam_ex2_data/CarsFailures.txt"
outputPathPart1 = "outPart1"
outputPathPart2 = "outPart2"
```

In [4]:
```
# ****************************************
# Exercise 2 - Part 1
# ****************************************
```

In [7]:
```
# Read the content of CarsFailures.txt
failuresDF = spark.read.load(inputPathFailures,\
                             format="csv",\
                             header=False,\
                             inferSchema=True)\
.withColumnRenamed("_c0", "date")\
.withColumnRenamed("_c1", "time")\
.withColumnRenamed("_c2", "carId")\
.withColumnRenamed("_c3", "failureType")\
.cache()
```

In [15]:
```
# Read the content of Cars.txt
carsDF = spark.read.load(inputPathCars,\
                             format="csv",\
                             header=False,\
                             inferSchema=True)\
.withColumnRenamed("_c0", "carId")\
.withColumnRenamed("_c1", "model")\
.withColumnRenamed("_c2", "company")\
.withColumnRenamed("_c3", "city")
```

In [16]:
```
# Associate the two dataFrames to two temporary tables
carsDF.createOrReplaceTempView("cars")
failuresDF.createOrReplaceTempView("failures")
```

In [17]:
```
# Define a UDF that returns 1 if the year is 2018. 0 otherwise.
def year2018Func(date):
    if date.startswith("2018"):
        return 1
    else:
        return 0


spark.udf.register("year2018", year2018Func)
```

Out[17]:
```
<function __main__.year2018Func(date)>
```

In [18]:
```
# Define a UDF that returns 1 if the year is 2017. 0 otherwise.
def year2017Func(date):
    if date.startswith("2017"):
        return 1
    else:
```

```
        return 0


spark.udf.register("year2017", year2017Func)
```

Out[18]: <function __main__.year2017Func(date)>

In [24]:
```
# Select only the failures related to years 2017 and 2018 and failureType=Engine
# Count
# - the number of engine failures in year 2017
# - the number of engine failures in year 2018
# Select only the car with #failures 2018 > #failures 2017

selectedCarsModelsDF = spark.sql("""SELECT cars.carId, model
FROM cars, failures
WHERE cars.carId=failures.carId
AND failures.failureType='Engine'
GROUP BY cars.carId, model
HAVING SUM( year2018(date)) > SUM( year2017(date)) """)
```

In [ ]:
```
# Store the result
```

In [26]:
```
selectedCarsModelsDF.write.csv(outputPathPart1,header=False)
```

In [ ]:
```

```

In [ ]:
```
# ****************************************
# Exercise 2 - Part 2
# ****************************************
```

In [49]:
```
# This function is already provided
from datetime import datetime, timedelta
def previousDate(mydate):
    currentDate=datetime.strptime(mydate,"%Y/%m/%d")
    prevDate=currentDate-timedelta(days=1)
    return prevDate.strftime("%Y/%m/%d")
```

In [50]:
```
# Select only carId and Date from each failure
# and remove duplicates
carsDatesDF = failuresDF.select("carId", "Date").distinct()
```

In [58]:
```
# Transform carsDatesDF into an RDD.
# Map the returned Row object to pairs (only for simplicity/for reusing the same cod
# solution based only on  RDDs for the next part)
carsDatesDistinctRDD = carsDatesDF.rdd\
.map(lambda row: (row["carId"], row["Date"]))
```

In [53]:
```
# Emit two pairs from each input pair
# - ((carId,  date), +1)   - First element of the window that starts in this date
# - ((carId,date-1), +1)   - Second element of the window that starts in the previou
def flatMapFunc(pair):
    returnedPairs = []
```

```
    carId = pair[0]
    currDate = pair[1]
    prevDate= previousDate(currDate)

    returnedPairs.append( ((carId, currDate), +1) )
    returnedPairs.append( ((carId, prevDate), +1) )

    return returnedPairs


windowsElementsRDD = carsDatesDistinctRDD.flatMap(flatMapFunc)
```

In [55]:
```
# Count the number of elements per windows
windowsNumElementsRDD = windowsElementsRDD.reduceByKey(lambda v1, v2: v1+v2)
```

In [56]:
```
# Select the windows with 2 elements
selectedWindowsRDD = windowsNumElementsRDD.filter(lambda pair: pair[1]==2)
```

In [57]:
```
# Store the key part of the selected pairs (carId, first date of the sequence)
selectedWindowsRDD.keys().saveAsTextFile(outputPathPart2)
```

In [ ]:
```

```

In [13]:
```
# Exam 2020/06/27 - Exercise #2 - SQL-based version #2
# Alternative solution completely based on Spark SQL
```

In [14]:
```
inputPathCars = "exam_ex2_data/Cars.txt"
inputPathFailures = "exam_ex2_data/CarsFailures.txt"
outputPathPart1 = "outPart1"
outputPathPart2 = "outPart2"
```

In [15]:
```
# ***************************************
# Exercise 2 - Part 1
# ***************************************
```

In [16]:
```
# Read the content of CarsFailures.txt
failuresDF = spark.read.load(inputPathFailures,\
                             format="csv",\
                             header=False,\
                             inferSchema=True)\
.withColumnRenamed("_c0", "date")\
.withColumnRenamed("_c1", "time")\
.withColumnRenamed("_c2", "carId")\
.withColumnRenamed("_c3", "failureType")\
.cache()
```

In [17]:
```
# Read the content of Cars.txt
carsDF = spark.read.load(inputPathCars,\
                             format="csv",\
                             header=False,\
                             inferSchema=True)\
.withColumnRenamed("_c0", "carId")\
.withColumnRenamed("_c1", "model")\
.withColumnRenamed("_c2", "company")\
.withColumnRenamed("_c3", "city")
```

In [18]:
```
# Associate the two dataFrames to two temporary tables
carsDF.createOrReplaceTempView("cars")
failuresDF.createOrReplaceTempView("failures")
```

In [19]:
```
# Define a UDF that returns 1 if the year is 2018. 0 otherwise.
def year2018Func(date):
    if date.startswith("2018"):
        return 1
    else:
        return 0


spark.udf.register("year2018", year2018Func)
```

Out[19]:
```
<function __main__.year2018Func(date)>
```

In [20]:
```
# Define a UDF that returns 1 if the year is 2017. 0 otherwise.
def year2017Func(date):
    if date.startswith("2017"):
        return 1
```

```
    else:
        return 0


spark.udf.register("year2017", year2017Func)
```

Out[20]:
```
<function __main__.year2017Func(date)>
```

In [21]:
```
# Select only the failures related to years 2017 and 2018 and failureType=Engine
# Count
# - the number of engine failures in year 2017
# - the number of engine failures in year 2018
# Select only the car with #failures 2018 > #failures 2017

selectedCarsModelsDF = spark.sql("""SELECT cars.carId, model
FROM cars, failures
WHERE cars.carId=failures.carId
AND failures.failureType='Engine'
GROUP BY cars.carId, model
HAVING SUM( year2018(date)) > SUM( year2017(date)) """)
```

In [24]:
```
# Store the result
selectedCarsModelsDF.write.csv(outputPathPart1,header=False)
```

In [ ]:
```

```

In [27]:
```
# ****************************************
# Exercise 2 - Part 2
# ****************************************
```

In [28]:
```
# This function is already provided
from datetime import datetime, timedelta
def previousDate(mydate):
    currentDate=datetime.strptime(mydate,"%Y/%m/%d")
    prevDate=currentDate-timedelta(days=1)
    return prevDate.strftime("%Y/%m/%d")
```

In [30]:
```
# Define a UDF that returns the previous date
spark.udf.register("prevDate", lambda currDate: previousDate(currDate))
```

Out[30]:
```
<function __main__.<lambda>(currDate)>
```

In [31]:
```
# Select only carId and Date from each failure
# and remove duplicates
carsDatesDF = failuresDF.select("carId", "Date").distinct()
```

In [35]:
```
# Create a new dataframe that contains one record carId, Date-1 for each record of c
# This is used to specify that each record of carsDatesDF is the second element of a
# starts at Date-1
carsDatesYesterdayDF = carsDatesDF.selectExpr("carId", "prevDate(Date) as Date")
```

In [38]:
```
# Apply union on the two dataFrames
```

```
windowsElementsDF = carsDatesDF.union(carsDatesYesterdayDF)
```

In [48]:

```
# Associate windowsElementsDF to temporary table
windowsElementsDF.createOrReplaceTempView("windowElements")
```

In [49]:

```
# Select only the groups (carId, Date) associated with two records. These are associ
# windows of two consecutive dates with at least one failure
selectedWindowsDF = spark.sql("""SELECT carId,Date
FROM windowElements
GROUP BY carId,Date
HAVING COUNT(*)=2""")
```

In [56]:

```
# Store the result
selectedWindowsDF.write.csv(outputPathPart2,header=False)
```

In [ ]:

In [ ]:
```
# Exam 2020/06/27 - Exercise #2
```

In [ ]:
```
# Input data and output folders
inputPathPurchases = "exam_ex2_data/Purchases.txt"
outputPathPart1 = "outPart1/"
outputPathPart2 = "outPart2/"

# Write your code here
```

In [ ]:
```
# ****************************************
# Exercise 2 - Part 1
# ****************************************
```

In [ ]:
```
# Read the content of Purchases.txt
purchasesRDD = sc.textFile(inputPathPurchases).cache()
```

In [ ]:
```
# Select only the purchases related to years 2019
# MID,CustomerId,Date,Price
def year2019(line):
    fields = line.split(",")

    date = fields[2]

    if (date.startswith("2019")):
        return True
    else:
        return False

selectedPurchasesRDD = purchasesRDD.filter(year2019)
```

In [ ]:
```
# Compute the income for each MID
```

In [ ]:
```
# Extract pairs (BID,price)
# Map each input line to a pair
# key = MID
# value = price

def pairMIDPrice(line):
    # MID,CustomerId,Date,Price
    fields = line.split(",")
    mid = fields[0]
    price = float(fields[3])

    return (mid, price)

midsPricesRDD = selectedPurchasesRDD.map(pairMIDPrice)
```

In [ ]:
```
# Apply reduceByKey to compute the (annual) income for each MID
midsIncomesRDD = midsPricesRDD.reduceByKey(lambda p1, p2: p1+p2).cache()
```

In [ ]:
```
# Select the maximum income in year 2019
```

In [ ]:
```
# Select only the annual incomes
incomesRDD = midsIncomesRDD.values()
```

In [ ]:
```
# Compute the maximum annual income
maxAnnualIncome = incomesRDD.reduce(lambda income1, income2: max(income1, income2))
```

In [ ]:
```
# Select the MIDs associated with the highest annual income in 2019
selectedMidsRDD = midsIncomesRDD.filter(lambda pair: pair[1]==maxAnnualIncome)
```

In [ ]:
```
# Store the result
selectedMidsRDD.keys().saveAsTextFile(outputPathPart1)
```

In [ ]:
```
# ****************************************
# Exercise 2 - Part 2
# ****************************************
```

In [ ]:
```
# Select the purchases of the last ten years (i.e., from year 2010 to year 2019)
# MID,CustomerId,Date,Price
def year20102019(line):
    fields = line.split(",")

    date = fields[2]
    year = int(date.split("/")[0])

    if (year>=2010 and year<=2019):
        return True
    else:
        return False

selectedPurchases1019RDD = purchasesRDD.filter(year20102019)
```

In [ ]:
```
# Compute the number of purchases for each MID in each year
```

In [ ]:
```
# Extract pairs (MID+year,+1)
# Map each input line to a pair
# key = (MID,year)
# value = +1

def pairMidYearOne(line):
    # MID,CustomerId,Date,Price
    fields = line.split(",")
    mid = fields[0]
    date = fields[2]
    year = int(date.split("/")[0])

    return ( (mid, year), 1)

midYearOneRDD = selectedPurchases1019RDD.map(pairMidYearOne)
```

In [ ]:
```
# Apply reduceByKey to compute the annual number of purchases for each MID in each y
midYearNumPurchasesRDD = midYearOneRDD.reduceByKey(lambda p1, p2: p1+p2).cache()
```

In [ ]:
```
# Compute the maximum number of purchases (associated with one single model) per yea
```

In [ ]:
```
# Extract pairs (year,annualPurchasesSingleModel)
# Map each input pair a pair
# key = year
# value = annualPurchasesSingleModel
yearNumPurchasesRDD = midYearNumPurchasesRDD.map(lambda pair: (pair[0][1], pair[1]))
```

In [ ]:
```
# Apply reduceByKey to compute the maximum number of purchases (associated with one
yearMaxNumPurchasesRDD = yearNumPurchasesRDD\
.reduceByKey(lambda numPurchases1, numPurchases2: max (numPurchases1, numPurchases2)
```

In [ ]:
```
# Map each input pair ( (MID,year), numPurchases) to
# ( year, (MID, numPurchases) )
yearMidNumPurchasesRDD  = midYearNumPurchasesRDD.map(lambda pair: (pair[0][1], (pair
```

In [ ]:
```
# join yearMidNumPurchasesRDD with yearMaxNumPurchasesRDD
midPurchasesvsMaxPurchasesRDD = yearMidNumPurchasesRDD.join(yearMaxNumPurchasesRDD)
```

In [ ]:
```
# Select the pairs with
# number of annual purchases per MID == maximum number of purchases (per single MID)
midsMaxAnnualPurchasesRDD = midPurchasesvsMaxPurchasesRDD.filter(lambda pair: pair[1
```

In [ ]:
```
# Count for each MID in how many years it is the most purchased model
```

In [ ]:
```
# Map each input pair to a new pair
# key = MID
# value = +1
midOneRDD = midsMaxAnnualPurchasesRDD.map(lambda pair: (pair[1][0][0], 1))
```

In [ ]:
```
midNumYearsMostPurchasedRDD = midOneRDD.reduceByKey(lambda v1,v2: v1+v2)
```

In [ ]:
```
# Select the MIDs that have been the most purchased smartphone model in at least two
selectedMIDsRDD = midNumYearsMostPurchasedRDD.filter(lambda pair: pair[1]>=2)
```

In [ ]:
```
# Store the selected MIDs
selectedMIDsRDD.keys().saveAsTextFile(outputPathPart2)
```

In [ ]:
```

```

In [ ]:
```
# Exam 2020/06/27 - Exercise #2 - Version #2
```

In [ ]:
```
# Input data and output folders
inputPathPurchases = "exam_ex2_data/Purchases.txt"
outputPathPart1 = "outPart1v2/"
outputPathPart2 = "outPart2v2/"

# Write your code here
```

In [ ]:
```
# ****************************************
# Exercise 2 - Part 1
# ****************************************
```

In [ ]:
```
# Read the content of Purchases.txt
purchasesRDD = sc.textFile(inputPathPurchases).cache()
```

In [ ]:
```
# Select only the purchases related to years 2019
# MID,CustomerId,Date,Price
def year2019(line):
    fields = line.split(",")

    date = fields[2]

    if (date.startswith("2019")):
        return True
    else:
        return False

selectedPurchasesRDD = purchasesRDD.filter(year2019)
```

In [ ]:
```
# Compute the income for each MID
```

In [ ]:
```
# Extract pairs (BID,price)
# Map each input line to a pair
# key = MID
# value = price

def pairMIDPrice(line):
    # MID,CustomerId,Date,Price
    fields = line.split(",")
    mid = fields[0]
    price = float(fields[3])

    return (mid, price)

midsPricesRDD = selectedPurchasesRDD.map(pairMIDPrice)
```

In [ ]:
```
# Apply reduceByKey to compute the (annual) income for each MID
midsIncomesRDD = midsPricesRDD.reduceByKey(lambda p1, p2: p1+p2).cache()
```

In [ ]:
```
# Select the maximum income in year 2019
```

In [ ]:
```
# Select only the annual incomes
incomesRDD = midsIncomesRDD.values()
```

In [ ]:
```
# Compute the maximum annual income
maxAnnualIncome = incomesRDD.reduce(lambda income1, income2: max(income1, income2))
```

In [ ]:
```
# Select the MIDs associated with the highest annual income in 2019
selectedMidsRDD = midsIncomesRDD.filter(lambda pair: pair[1]==maxAnnualIncome)
```

In [ ]:
```
# Store the result
selectedMidsRDD.keys().saveAsTextFile(outputPathPart1)
```

In [ ]:
```
# ****************************************
# Exercise 2 - Part 2
# ****************************************
```

In [ ]:
```
# Select the purchases of the last ten years (i.e., from year 2010 to year 2019)
# MID,CustomerId,Date,Price
def year20102019(line):
    fields = line.split(",")

    date = fields[2]
    year = int(date.split("/")[0])

    if (year>=2010 and year<=2019):
        return True
    else:
        return False

selectedPurchases1019RDD = purchasesRDD.filter(year20102019)
```

In [ ]:
```
# Compute the number of purchases for each MID in each year
```

In [ ]:
```
# Extract pairs (MID+year,+1)
# Map each input line to a pair
# key = (MID,year)
# value = +1

def pairMidYearOne(line):
    # MID,CustomerId,Date,Price
    fields = line.split(",")
    mid = fields[0]
    date = fields[2]
    year = int(date.split("/")[0])

    return ( (mid, year), 1)

midYearOneRDD = selectedPurchases1019RDD.map(pairMidYearOne)
```

In [ ]:
```
# Apply reduceByKey to compute the annual number of purchases for each MID in each y
midYearNumPurchasesRDD = midYearOneRDD.reduceByKey(lambda p1, p2: p1+p2).cache()
```

In [ ]:
```
# Compute the maximum number of purchases (associated with one single model) per yea
```

In [ ]:
```
# Extract pairs (year,annualPurchasesSingleModel)
# Map each input pair a pair
# key = year
# value = annualPurchasesSingleModel
yearNumPurchasesRDD = midYearNumPurchasesRDD.map(lambda pair: (pair[0][1], pair[1]))
```

In [ ]:
```
# Apply reduceByKey to compute the maximum number of purchases (associated with one
yearMaxNumPurchasesRDD = yearNumPurchasesRDD\
.reduceByKey(lambda numPurchases1, numPurchases2: max (numPurchases1, numPurchases2)
```

In [ ]:
```
# We are managing only 10 years.
# Hence, the content of yearMaxNumPurchasesRDD can be stored in a local variable of
yearMaxNumPurchases = yearMaxNumPurchasesRDD.collectAsMap()
```

In [ ]:
```
# Select the pairs with
# number of annual purchases per MID == maximum number of purchases (per single MID)
def maxNumPurchases(pair):
    year = pair[0][1]
    numPurchases = pair[1]

    if numPurchases==yearMaxNumPurchases[year]:
        return True
    else:
        return False

midsMaxAnnualPurchasesRDD = midYearNumPurchasesRDD.filter(maxNumPurchases)
```

In [ ]:
```
# Count for each MID in how many years it is the most purchased model
```

In [ ]:
```
# Map each input pair to a new pair
# key = MID
# value = +1
midOneRDD = midsMaxAnnualPurchasesRDD.map(lambda pair: (pair[0][0], 1))
```

In [ ]:
```
midNumYearsMostPurchasedRDD = midOneRDD.reduceByKey(lambda v1,v2: v1+v2)
```

In [ ]:
```
# Select the MIDs that have been the most purchased smartphone model in at least two
selectedMIDsRDD = midNumYearsMostPurchasedRDD.filter(lambda pair: pair[1]>=2)
```

In [ ]:
```
# Store the selected MIDs
selectedMIDsRDD.keys().saveAsTextFile(outputPathPart2)
```

In [ ]:
```

```

In [ ]:
```
from pyspark.sql.types import LongType
```

In [ ]:
```
# Exam 2020/06/27 - Exercise #2 - Spark SQL
```

In [ ]:
```
# Input data and output folders
inputPathPurchases = "exam_ex2_data/Purchases.txt"
outputPathPart1 = "outPart1SQL/"
outputPathPart2 = "outPart2SQL/"
```

In [ ]:
```
# ****************************************
# Exercise 2 - Part 1
# ****************************************
```

In [ ]:
```
# Read the content of Purchases.txt
purchasesDF = spark.read.load(inputPathPurchases,\
                              format="csv",\
                              header=False,\
                              inferSchema=True)\
.withColumnRenamed("_c0", "MID")\
.withColumnRenamed("_c1", "CustomerId")\
.withColumnRenamed("_c2", "Date")\
.withColumnRenamed("_c3", "Price")\
.cache()
```

In [ ]:
```
# Associate purchasesDF to a temporary table
purchasesDF.createOrReplaceTempView("purchases")
```

In [ ]:
```
# Compute the income for each MID in year 2019
midsIncomesDF = spark.sql("""SELECT MID, SUM(Price) as Income
FROM purchases
WHERE Date>='2019/01/01' and Date<='2019/12/31'
GROUP BY MID""")\
.cache()
```

In [ ]:
```
# Associate midsIncomesDF to a temporary table
midsIncomesDF.createOrReplaceTempView("MidsIncomesTable")
```

In [ ]:
```
# Select the MIDs associated with the highest annual income in 2019
selectedMidsDF = spark.sql("""SELECT MID
FROM MidsIncomesTable, (SELECT MAX(Income) as MaxIncome
                        FROM MidsIncomesTable) MaxIncomeTable
WHERE Income=MaxIncome""")
```

In [ ]:
```
#selectedMidsDF.show()
```

In [ ]:
```
# Store the result
selectedMidsDF.write.csv(outputPathPart1,header=False)
```

In [ ]:
```
# ***************************************
# Exercise 2 - Part 2
# ***************************************
```

In [ ]:
```
# Define a UDF that extract the year part from Date
def yearFunc(date):
    return int(date.split("/")[0])

spark.udf.register("yearFunc", yearFunc, LongType())
```

In [ ]:
```
# Select the purchases of the last ten years (i.e., from year 2010 to year 2019)
# Compute the number of purchases for each MID in each year
```

In [ ]:
```
midYearNumPurchasesDF = spark.sql("""SELECT MID,
yearFunc(Date) as Year,
count(*) as NumPurchases
FROM purchases
WHERE Date>='2010/01/01' and Date<='2019/12/31'
GROUP BY MID, yearFunc(Date)""")\
.cache()
```

In [ ]:
```
# Associate midYearNumPurchasesDF to a temporary table
midYearNumPurchasesDF.createOrReplaceTempView("MidYearNumPurchs")
```

In [ ]:
```
# We can provide the hint about the usefulness of the brodcast join
midsMaxAnnualPurchasesDF = spark.sql("""SELECT
/*+ BROADCAST(MaxYearlyPurchases) */
MID,
MidYearNumPurchs.Year,
NumPurchases
FROM MidYearNumPurchs, (SELECT Year, Max(NumPurchases) as MaxNumPurchases
                        FROM MidYearNumPurchs
                        GROUP BY Year) MaxYearlyPurchases
WHERE MidYearNumPurchs.NumPurchases=MaxYearlyPurchases.MaxNumPurchases
  AND MidYearNumPurchs.Year=MaxYearlyPurchases.Year""")
```

In [ ]:
```
#midsMaxAnnualPurchasesDF.explain()
```

In [ ]:
```
# Count for each MID in how many years it is the most purchased model
```

In [ ]:
```
# Associate midsMaxAnnualPurchasesRDD to a temporary table
midsMaxAnnualPurchasesDF.createOrReplaceTempView("midsMaxAnnual")
```

In [ ]:
```
selectedMIDsDFPartB = spark.sql("""SELECT MID
FROM midsMaxAnnual
GROUP BY MID
HAVING COUNT(*)>=2""")
```

In [ ]:
```
#selectedMIDsDFPartB.show()
```

In [ ]:
```
# Store the result
selectedMIDsDFPartB.write.csv(outputPathPart2,header=False)
```

In [ ]:
```

```

In [ ]:
```
# Input data and output folders
inUsers = "Users.txt"
inSongs = "Songs.txt"
inListenToSongs = "ListenToSongs.txt"

outputPathPart1 = "outPart1/"
outputPathPart2 = "outPart2/"
```

In [ ]:
```
# Write your code here
```

In [ ]:
```
# Part 1
```

In [ ]:
```
usersRDD = sc.textFile(inUsers)
songsRDD = sc.textFile(inSongs)
listenRDD = sc.textFile(inListenToSongs).cache()
```

In [ ]:
```
# Select young users
youngUsersRDD = usersRDD.filter(lambda line: int(line.split(",")[4])>1999)
```

In [ ]:
```
# Select the songs listened to in the last two years
def lastTwoYears(line):
    startTimestamp = line.split(",")[2]
    if (startTimestamp>="2018/09/14" and startTimestamp<="2020/09/13"):
        return True
    else:
        return False


listenLast2YearsRDD = listenRDD.filter(lastTwoYears)
```

In [ ]:
```
# Select the songs listened to by young users in the last two years
```

In [ ]:
```
# Map listenLast2YearsRDD to pairs (UID,SID)
def userSong(line):
    fields = line.split(",")
    sid = fields[0]
    uid = fields[1]

    return (uid,sid)

userSongTwoYearsRDD = listenLast2YearsRDD.map(userSong)
```

In [ ]:
```
# Map youngUsersRDD to pairs (UID,None)
def userNone(line):
    fields = line.split(",")
    uid = fields[0]

    return (uid,None)

youngUserPairRDD = youngUsersRDD.map(userNone)
```

In [ ]:
```
# Join the content of userSongTwoYearsRDD and youngUserPairRDD to identify the songs
songsListenedYoungUsersRDD = userSongTwoYearsRDD.join(youngUserPairRDD)
```

In [ ]:
```
# Select the SIDs of the songs listened to by young users in the last two years
sidListenedYoungUsersRDD = songsListenedYoungUsersRDD.map(lambda pair: pair[1][0])
```

In [ ]:
```
# Extract all the possible SIDs from songsRDD
allSidsRDD = songsRDD.map(lambda line: line.split(",")[0])
```

In [ ]:
```
# Select the SIDs of the songs that do not occur in sidListenedYoungUsersRDD
selectedSIDsRDD = allSidsRDD.subtract(sidListenedYoungUsersRDD)
```

In [ ]:
```
selectedSIDsRDD.saveAsTextFile(outputPathPart1)
```

In [ ]:
```

```

In [ ]:
```
# Part 2
```

In [ ]:
```
# Compute for each song its popularity in each year, i.e., the number of distinct us
```

In [ ]:
```
# Map the content of listenRDD to tuples (SID,Year,UID)
def tupleSidYearUser(line):
    fields = line.split(",")
    sid = fields[0]
    uid = fields[1]
    year = fields[2].split("/")[0]

    return (sid,year,uid)

sidYearUserRDD = listenRDD.map(tupleSidYearUser)
```

In [ ]:
```
# Remove duplicates. We are interested in the number of distinct users for each (son
sidYearDistinctUserRDD =sidYearUserRDD.distinct()
```

In [ ]:
```
# Count the number of distinct user for each pair (sid,year)
```

In [ ]:
```
# Map sidYearDistinctUserRDD to pairs ( (sid,year), +1)
sidYearOneRDD = sidYearDistinctUserRDD.map(lambda record: ( (record[0],record[1]), +
```

In [ ]:
```
# ReduceByKey to count the number of distinct user for each pair (sid,year)
sidYearPopularityRDD = sidYearOneRDD.reduceByKey(lambda v1,v2: v1+v2)
```

In [ ]:
```
# Compute the highest popularity for each song and the first year associated with th
```

In [ ]:
```
# Map sidYearPopularityRDD to pairs (SID, (annual popularity, year))
```

```
sidAnnualPopularityYearRDD = sidYearPopularityRDD.map(lambda pair: (pair[0][0], (pai
```

In [ ]:

```
# Apply reduceByKey to compute the highest annual popularity for each SID.
# Given two input pair values (annualPopularityV1, year1) and (annualPopularityV2, y
# I select the pair associated with the highest annualPopularity value.
# If the annualPopularity is the same for both pair, I select the pair associated wi
def maxPopFirstYear(v1, v2):
    annualPopularityV1 = v1[0]
    year1 = v1[1]
    annualPopularityV2 = v2[0]
    year2 = v2[1]

    if (annualPopularityV1>annualPopularityV2):
        return v1
    else:
        if (annualPopularityV1<annualPopularityV2):
            return v2
        else:
            if (year1<year2):
                return v1
            else:
                return v2

sidFirstYearMaxPopRDD =  sidAnnualPopularityYearRDD.reduceByKey(maxPopFirstYear)
```

In [ ]:

```
# Extract only SID and year
sidYearRDD = sidFirstYearMaxPopRDD.map(lambda pair: (pair[0], pair[1][1]))
```

In [ ]:

```
sidYearRDD.saveAsTextFile(outputPathPart2)
```

In [ ]:

In [ ]:
```
# Input data and output folders
inUsers = "Users.txt"
inSongs = "Songs.txt"
inListenToSongs = "ListenToSongs.txt"

outputPathPart1 = "outPart1/"
outputPathPart2 = "outPart2/"
```

In [ ]:
```
# Write your code here
```

In [ ]:
```
# Part 1
```

In [ ]:
```
usersRDD = sc.textFile(inUsers)
songsRDD = sc.textFile(inSongs)
listenRDD = sc.textFile(inListenToSongs).cache()
```

In [ ]:
```
# Select young users
youngUsersRDD = usersRDD.filter(lambda line: int(line.split(",")[4])>1999)
```

In [ ]:
```
# Select the songs listened to in the last two years
def lastTwoYears(line):
    startTimestamp = line.split(",")[2]
    if (startTimestamp>="2018/09/14" and startTimestamp<="2020/09/13"):
        return True
    else:
        return False


listenLast2YearsRDD = listenRDD.filter(lastTwoYears)
```

In [ ]:
```
# Select the songs listened to by young users in the last two years
```

In [ ]:
```
# Map listenLast2YearsRDD to pairs (UID,SID)
def userSong(line):
    fields = line.split(",")
    sid = fields[0]
    uid = fields[1]

    return (uid,sid)

userSongTwoYearsRDD = listenLast2YearsRDD.map(userSong)
```

In [ ]:
```
# Map youngUsersRDD to pairs (UID,None)
def userNone(line):
    fields = line.split(",")
    uid = fields[0]

    return (uid,None)

youngUserPairRDD = youngUsersRDD.map(userNone)
```

In [ ]:
```
# Join the content of userSongTwoYearsRDD and youngUserPairRDD to identify the songs
songsListenedYoungUsersRDD = userSongTwoYearsRDD.join(youngUserPairRDD)
```

In [ ]:
```
# Select the SIDs of the songs listened to by young users in the last two years
sidListenedYoungUsersRDD = songsListenedYoungUsersRDD.map(lambda pair: pair[1][0])
```

In [ ]:
```
# Extract all the possible SIDs from songsRDD
allSidsRDD = songsRDD.map(lambda line: line.split(",")[0])
```

In [ ]:
```
# Select the SIDs of the songs that do not occur in sidListenedYoungUsersRDD
selectedSIDsRDD = allSidsRDD.subtract(sidListenedYoungUsersRDD)
```

In [ ]:
```
selectedSIDsRDD.saveAsTextFile(outputPathPart1)
```

In [ ]:
```

```

In [ ]:
```
# Part 2
```

In [ ]:
```
# Compute for each song its popularity in each year, i.e., the number of distinct us
```

In [ ]:
```
# Map the content of listenRDD to tuples (SID,Year,UID)
def tupleSidYearUser(line):
    fields = line.split(",")
    sid = fields[0]
    uid = fields[1]
    year = fields[2].split("/")[0]

    return (sid,year,uid)

sidYearUserRDD = listenRDD.map(tupleSidYearUser)
```

In [ ]:
```
# Remove duplicates. We are interested in the number of distinct users for each (son
sidYearDistinctUserRDD =sidYearUserRDD.distinct()
```

In [ ]:
```
# Count the number of distinct user for each pair (sid,year)
```

In [ ]:
```
# Map sidYearDistinctUserRDD to pairs ( (sid,year), +1)
sidYearOneRDD = sidYearDistinctUserRDD.map(lambda record: ( (record[0],record[1]), +
```

In [ ]:
```
# ReduceByKey to count the number of distinct user for each pair (sid,year)
sidYearPopularityRDD = sidYearOneRDD.reduceByKey(lambda v1,v2: v1+v2).cache()
```

In [ ]:
```
# Compute the highest popularity for each song
```

In [ ]:
```
# Map sidYearPopularityRDD to pairs (SID, annual popularity)
```

```
sidPopularityRDD = sidYearPopularityRDD.map(lambda pair: (pair[0][0], pair[1]))
```

In [ ]:
```
sidMaxPopularityRDD = sidPopularityRDD.reduceByKey(lambda v1,v2: max(v1,v2))
```

In [ ]:
```
# Select for each song the year associated with the highest popularity
```

In [ ]:
```
# Map sidYearPopularityRDD to pairs (SID, (year, annual popularity))
sidYearPopRDD = sidYearPopularityRDD.map(lambda pair: (pair[0][0], (pair[0][1],pair[
```

In [ ]:
```
# Join sidMaxPopularityRDD and sidYearPopRDD and select for each song the years asso
joinSidMaxAnnualandLocalAnnualRDD = sidMaxPopularityRDD.join(sidYearPopRDD)
```

In [ ]:
```
# select for each song the years associated with its highest annual popularity
def annualEqualMax(pair):
    maxAnnualPopularity = pair[1][0]
    annualPopThisYear = pair[1][1][1]

    if (annualPopThisYear==maxAnnualPopularity):
        return True
    else:
        return False


selectedSongYearsRDD = joinSidMaxAnnualandLocalAnnualRDD.filter(annualEqualMax)
```

In [ ]:
```
# Select for each song the first year associated with its highest popularity
```

In [ ]:
```
# Extract pairs (sid,year)
# We can do this by changing the value pary
sidYearRDD = selectedSongYearsRDD.mapValues(lambda pair: pair[1][0] )
```

In [ ]:
```
# Select the first year for each key
sidFirstYearMaxPopRDD = sidYearRDD.reduceByKey(lambda v1,v2: min(v1,v2))
```

In [ ]:
```
sidFirstYearMaxPopRDD.saveAsTextFile(outputPathPart2)
```

In [ ]:
```

```

In [ ]:
```
# Input data and output folders

inProdPlants = "ProductionPlants.txt"
inRobots = "Robots.txt"
inFailures = "Failures.txt"

outputPathPart1 = "outPart1/"
outputPathPart2 = "outPart2/"
```

In [ ]:
```
# Write your code here
```

In [ ]:
```
# Part 1
```

In [ ]:
```
prodPlantsRDD = sc.textFile(inProdPlants)
```

In [ ]:
```
robotsRDD = sc.textFile(inRobots)
```

In [ ]:
```
# Select production plants located in Italy
prodPlantsItalyRDD = prodPlantsRDD.filter(lambda line: line.split(",")[2] == 'Italy'
```

In [ ]:
```
# Map each line of prodPlantsRDD to a pair (PlantID, City)
def plantIdCity(line):
    fields = line.split(",")
    plantID = fields[0]
    city = fields[1]

    return (plantID,city)

planIDCityRDD = prodPlantsItalyRDD.map(plantIdCity)
```

In [ ]:
```
# Map each line of robotsRDD to a pair (PlantID, +1)
ridNoneRDD = robotsRDD.map(lambda line: (line.split(",")[1], +1))
```

In [ ]:
```
# Join planIDCityRDD with ridNoneRDD
joinPlantIDRIDOneRDD = planIDCityRDD.join(ridNoneRDD)
```

In [ ]:
```
# Each value part cointains a pair (City, +1)
cityOneRDD = joinPlantIDRIDOneRDD.values()
```

In [ ]:
```
# Count the number of robots per city
citiesTotRobotsRDD = cityOneRDD.reduceByKey(lambda v1,v2: v1+v2)
```

In [ ]:
```
# Compute the maximum number of robots
maxRobots = citiesTotRobotsRDD.values().reduce(lambda v1,v2: max(v1,v2))
```

In [ ]:
```
# Select the city/ies) associated with the maximum value
```

```
selectedCitiesRDD = citiesTotRobotsRDD.filter(lambda pair: pair[1]==maxRobots)
```

In [ ]:
```
selectedCitiesRDD.keys().saveAsTextFile(outputPathPart1)
```

In [ ]:
```

```

In [ ]:
```
# Part 2
```

In [ ]:
```
failuresRDD = sc.textFile(inFailures)
```

In [ ]:
```
# Map each failure to a pair (RID, +1)
planIDCityRDD = failuresRDD.map(lambda line: (line.split(",")[0], +1))
```

In [ ]:
```
# Compute the number of failures for each robot
```

In [ ]:
```
failuresRobotRDD = planIDCityRDD.reduceByKey(lambda v1, v2: v1+v2)
```

In [ ]:
```
# Map each robot to a pair (RID, PlantId)
def ridPlantID(line):
    fields = line.split(",")
    rid = fields[0]
    pid = fields[1]

    return (rid,pid)


ridPidRDD = robotsRDD.map(ridPlantID)
```

In [ ]:
```
# Join ridPidRDD with failuresRobotRDD
joinRidPidNumFailuresRDD = ridPidRDD.join(failuresRobotRDD)
```

In [ ]:
```
# Map to pairs (PID, numFailuresRobot) and compute the maximum value for each PID
# This RDD does not contains the production plants without failures
pidMaxFailuresPerRobotRDD = joinRidPidNumFailuresRDD.values().reduceByKey(lambda v1,
```

In [ ]:
```
# Map input productin plants to pair (plantId, 0)
pidZeroRDD = prodPlantsRDD.map(lambda line: (line.split(",")[0], 0))
```

In [ ]:
```
# Retrieve the production plants with a maximum number of failures per robots equal
pidNoFailuresRDD = pidZeroRDD.subtractByKey(pidMaxFailuresPerRobotRDD)
```

In [ ]:
```
# The final result is the union of pidMaxFailuresPerRobotRDD and pidNoFailuresRDD
resPart2RDD = pidMaxFailuresPerRobotRDD.union(pidNoFailuresRDD)
```

In [ ]:
```
resPart2RDD.saveAsTextFile(outputPathPart2)
```

In [ ]:

In [6]:
```
import pyspark

from pyspark import SparkContext
from pyspark import SparkConf
from pyspark.sql import SparkSession
from pyspark.sql import functions
```

In [7]:
```
spark = SparkSession.builder.getOrCreate()
sc = spark.sparkContext
```

In [8]:
```
carModelsPath = 'data/CarModels.txt'
salesEUPath = 'data/SalesEU.txt'
salesExtraEUPath = 'data/SalesExtraEU.txt'


output1 = 'out1'
output2 = 'out2'
```

In [15]:
```
modelsRDD = sc.textFile(carModelsPath)
euSalesRDD = sc.textFile(salesEUPath)

def modelSalesMap(x):
    return (x[0], 1)

def modelPriceMap(x):
    modelId = x[0]
    priceTuple = (x[1], 1)
    return (modelId, priceTuple)

def keepModelPrice(line):
    fields = line.split(',')
    modelId = fields[2]
    price = float(fields[5])
    return (modelId, price)

# Select vehicles manufactured by FIAT
# (modelId, 'FIAT')
fiatModelsRDD = modelsRDD.filter(lambda line: line.split(',')[2] == 'FIAT')\
                         .map(lambda line: (line.split(',')[0], line.split(',')[2

# Select only sales in Italy
# (modelId, price)
italySalesRDD = euSalesRDD.filter(lambda line: line.split(',')[4] == 'Italy')\
                          .map(keepModelPrice)

# Select sales of modelIDs manufactured by FIAT
# (modelId, price)
italyFiatSalesRDD = italySalesRDD.join(fiatModelsRDD)\
                          .map(lambda x: (x[0], x[1][0]))

# Count the sales in Italy for each modelId and select only the entries with sales >
# (modelId, n. of sales)
modelItalyHighSalesRDD = italyFiatSalesRDD.map(modelSalesMap)\
                                  .reduceByKey(lambda v1, v2: v1 + v2)\
                                  .filter(lambda x: x[1] > 960) # 1000000 instead

# Compute the average price in Italy for each modelId and select only the entries wi
# (modelId, (price, +1)) ----> (modelId, avg_price)
```

```
modelItalyHighAvgPriceRDD = italyFiatSalesRDD.map(modelPriceMap)\
                                             .reduceByKey(lambda v1, v2: (v1[0] + v2[0], v1[1
                                             .mapValues(lambda x: x[0] / x[1])\
                                             .filter(lambda x: x[1] > 50000)

# Join the 2 RDDs and retrieve the keys only
fiatModelItalyHighSalesAndAvgPriceRDD = modelItalyHighSalesRDD.join(modelItalyHighAv
                                                                   .keys()

fiatModelItalyHighSalesAndAvgPriceRDD.saveAsTextFile(output1)
```

In [ ]:

```
# ____________________________________________________
# Task 2 v1
# ____________________________________________________
```

In [30]:

```
extraEURDD = sc.textFile(salesExtraEUPath)

def modelIdSalesPerYear(line):
    fields = line.split(',')
    modelId = fields[2]
    date = fields[3]
    year = int(date.split('/')[0])
    return ((modelId, year), 1)

# Obtain for the 2 sales datasets and RDD with
# ((modelId, year), +1)
euSalesPerYearRDD = euSalesRDD.map(modelIdSalesPerYear)
extraEUSalesPerYearRDD = extraEURDD.map(modelIdSalesPerYear)

# Merge the two RDDs and count for each year and each modelId the number of sales
# ((modelId, year), n. of sales)
globalSalesPerYear = euSalesPerYearRDD.union(extraEUSalesPerYearRDD)\
                                      .reduceByKey(lambda v1, v2: v1 + v2)

def checkIncreasingSales(x):
    salesList = x[1]
    salesList.sort()

    lastYear = -1
    lastSales = -1
    for year, sales in salesList:
        # verification
        assert lastYear == -1 or (lastYear + 1) == year
        if sales <= lastSales:
            return False
        lastSales = sales
        lastYear = year
    return True

# Map each ((modelId, year), n. of sales) in (modelId, (year, n. of sales))
# Group by key to obtain a list containing 50 elements, one per year, with the numbe
# Analyze the list to check if the number of sales is always increasing
# We suppose that the number of year is small enough to be stored and analyzed in on
modelIdsWithIncreasingSales = globalSalesPerYear.map(lambda x: (x[0][0], (x[0][1], x
                                                .groupByKey()\
                                                .mapValues(list)\
                                                .filter(checkIncreasingSales)
modelIdsWithIncreasingSales.keys().saveAsTextFile(output2)
```

In [ ]:
```
#from pyspark import SparkContext
#from pyspark.sql import SparkSession
```

In [ ]:
```
#spark = SparkSession.builder.appName('Exam20210705').getOrCreate()
#sc = spark.sparkContext
```

In [ ]:
```
items_path = '../exampleData/items.txt'
ads_path = '../exampleData/ads_sales.txt'
users_path = '../exampleData/users.txt'

out1 = 'out1'
out2 = 'out2'
```

In [ ]:
```
items_rdd = sc.textFile(items_path)
ads_rdd = sc.textFile(ads_path)
```

In [ ]:
```
### PART 1
# filter only the items that were actually purchased
purchased_rdd = ads_rdd.filter(lambda line: line.split(',')[3] == 'true').cache()
```

In [ ]:
```
def itemid_price(line):
    fields = line.split(',')
    itemid = fields[2]
    price = float(fields[4])
    return (itemid, price)

# obtain the pair rdd
# (itemId, salePrice)
sales_price_rdd = purchased_rdd.map(itemid_price)
```

In [ ]:
```
def itemid_recomPrice_category(line):
    fields = line.split(',')
    itemid = fields[0]
    recomPrice = float(fields[2])
    category = fields[3]

    return (itemid, (recomPrice, category))

# obtain the pair rdd
# (itemId, (recommendedPrice, category))
items_category_recomPrice_rdd = items_rdd.map(itemid_recomPrice_category).cache()
```

In [ ]:
```
# join the two pair rdds to obtain
# (itemId, (salePrice, (recommendedPrice, category)))
item_category_prices_rdd = sales_price_rdd.join(items_category_recomPrice_rdd)
```

In [ ]:
```
def obtain_counters(item):
    itemId = item[0]
    sale_price = item[1][0]
    recommended_price, category = item[1][1]
```

```
    numerator = 1 if sale_price > recommended_price else 0

    return ((itemId, category), (numerator, 1))

# map each element into a tuple of counters and keep the category field (by moving i
# one counter for numerator (= number of times the item was sold at price > recommen
# one counter for denominator (total number of times the item was sold)
# ((itemId, category), (numerator, denominator))
item_counters_category_rdd = item_category_prices_rdd.map(obtain_counters)
```

In [ ]:
```
# sum all the counters and compute the percentage
# ((itemId, category), percentage)
percentage_rdd = item_counters_category_rdd.reduceByKey(lambda v1, v2: (v1[0] + v2[0
                                            .mapValues(lambda it: it[0] / it[1])
```

In [ ]:
```
# get the result by filtering only those elements with percentage > 90% and keeping
result = percentage_rdd.filter(lambda it: it[1] >= 0.9)\
.keys()
```

In [ ]:
```
result.saveAsTextFile(out1)
```

In [ ]:
```
```

In [ ]:
```
### PART 2
# use an rdd computed in part 1 containing items with itemId, recommendedPrice and c
# and keep only itemId, category
# (itemId, (recommendedPrice, category))
# ->
# (itemId, category)
category_per_item_rdd = items_category_recomPrice_rdd.map(lambda it: (it[0], it[1][1
```

In [ ]:
```
def itemid_profit(line):
    fields = line.split(',')
    itemid = fields[2]
    price = float(fields[4])
    return (itemid, price)

# map purchased_rdd into an rdd containing itemId and price
# Only the lines with purchased equal to true are considered (the others have Price
# and hence are useless. For this reason we can use purchased_rdd instaed of ads_rdd
# (itemId, profitPerSingleItem)
profits_per_item_rdd = purchased_rdd.map(itemid_profit)
```

In [ ]:
```
# compute the total profits for each item
# (itemId, profit)
total_profits_rdd = profits_per_item_rdd.reduceByKey(lambda v1, v2: v1 + v2)
```

In [ ]:
```
# Use left outer join to join the two rdds previously computed to obtain the followi
# (itemId, (category, profit))
# note that profit is not defined for all items: for unadvertised items, profit is N
unadv_profits_rdd = category_per_item_rdd.leftOuterJoin(total_profits_rdd)
```

In [ ]:
```
# keep only unadvertised and low-profit items
lowprofits_unadv_items = unadv_profits_rdd.filter(lambda it: it[1][1] is None or it[
```

In [ ]:
```
def determine_low_profit_and_unadv(item):
    category = item[1][0]
    profit = item[1][1]
    x = 1 if profit is None else 0
    y = 1 if (not profit is None) and profit <= 100 else 0

    return (category, (x, y))

# map each element into a pair (category, (x, y))
# where:
# x = 1 if the item is unadvertised, 0 otherwise
# y = 1 if the item is low-profit, 0 otherwise
# and count for each category the number of unadvertised and low-profits items
# (category, (# items unadvertised, # items with low-profits))
lowprofits_unadv_per_category = lowprofits_unadv_items.map(determine_low_profit_and_
                                                           .reduceByKey(lambda v1, v2: (v1[0] +
```

In [ ]:
```
# filter only those categories with at least 10 low-profit items
# and at least 10 unadvertised items
result2 = lowprofits_unadv_per_category.filter(lambda it: it[1][0] >= 10 and it[1][1
# for testing purposes, set thresholds to 2 instead of 10
#result2 = lowprofits_unadv_per_category.filter(lambda it: it[1][0] >= 2 and it[1][1
```

In [ ]:
```
result2.saveAsTextFile(out2)
```

In [ ]:
```
import pyspark
import datetime

from pyspark import SparkContext
from pyspark.sql import SparkSession
```

In [ ]:
```
items_catalog_path = '../items.txt'
customers_path = '../customers.txt'
purchases_path = '../purchases.txt'

out1 = 'res/out1'
out2 = 'res/out2'
```

In [ ]:
```
purchases_rdd = sc.textFile(purchases_path).cache()
items_catalog_rdd = sc.textFile(items_catalog_path)
```

In [ ]:
```
# Part 1
def get_itemid_date1(line):
    fields = line.split(',')
    date = fields[0]
    itemid = fields[2]
    return itemid, date

def get_year(date):
    return int(date.split('/')[0])

def filter_year_2010_2019(line):
    _, date = get_itemid_date1(line)
    year = get_year(date)

    return 2010 <= year <= 2019

def map_purchases(line):
    itemid, date = get_itemid_date1(line)
    year = get_year(date)
    return ((itemid, year), 1)

# filter only the purchases between 2010 and 2019
# map each line into ((itemId, year), +1)
# and then use reduceByKey to count for each itemId and year, the number of time the
purchase_per_item_year_rdd = purchases_rdd.filter(filter_year_2010_2019)\
.map(map_purchases)\
.reduceByKey(lambda i1, i2: i1 + i2)

# map each item ((itemId, year), count)
# into a (itemId, count)
purchase_count_rdd = purchase_per_item_year_rdd.map(lambda it: (it[0][0], it[1]))

# filter only those lines with count >= 1000
# and map them into (itemId, +1)
# and sum using a reduceByKey
res1 = purchase_count_rdd.filter(lambda it: it[1] >= 1000)\
.map(lambda it: (it[0], 1))\
.reduceByKey(lambda i1, i2: i1 + i2)\
.filter(lambda it: it[1] == 10)\
.keys()
```

In [ ]:
```
res1.saveAsTextFile(out1)
```

In [ ]:
```

```

In [ ]:
```
# Part 2
```

In [ ]:
```
def get_itemid_date_category(line):
    fields = line.split(',')
    itemid = fields[0]
    category = fields[2]
    date = fields[3]
    return itemid, category, date


def filter_by_first_time_in_catalog(line):
    fields = line.split(',')
    date = fields[3]
    return date < '2010/01/01-00:00:00'

def map_item_catalog(line):
    itemid, category, date = get_itemid_date_category(line)
    return (itemid, (date, category))

# filter only items with firstTimeInCatalog < 2010/01/01 and then
# map item_catalog data into a tuple
# (itemId, (firstTimeInCatalog, category))
# to perform a join operation next
items_catalog_pair_rdd = items_catalog_rdd\
.filter(filter_by_first_time_in_catalog)\
.map(map_item_catalog)
```

In [ ]:
```
#items_catalog_pair_rdd.collect()
```

In [ ]:
```
def map_purchase2(line):
    itemid, date = get_itemid_date1(line)
    return (itemid, date)

# map purchases into tuple
# (itemId, purchaseTimestamp)
purchase_pair_rdd = purchases_rdd.map(map_purchase2)
```

In [ ]:
```
#purchase_pair_rdd.collect()
```

In [ ]:
```
# join the two RDDs
# (itemId, (purchaseTimestamp, (firstTimeInCatalog, category)))
# a rightOuterJoin was used to keep track of products which were never sold
joined_rdd = purchase_pair_rdd.rightOuterJoin(items_catalog_pair_rdd)
```

In [ ]:
```
def get_data_from_timestamp(t, has_hours=True):
    if has_hours:
        first, second = t.split('-')
        hours, minutes, seconds = second.split(':')
        hours, minutes, seconds = int(hours), int(minutes), int(seconds)
```

```
    else:
        first = t
        second = None
    year, month, day = first.split('/')
    year, month, day = int(year), int(month), int(day)
    return (year, month, day) if not has_hours else (year, month, day, hours, minute


def compute_difference(t1, t2):
    t1_t = datetime.datetime(*get_data_from_timestamp(t1))
    t2_t = datetime.datetime(*get_data_from_timestamp(t2))
    delta = t1_t - t2_t

    return delta.days / 365


def map_joined_data(it):
    itemid = it[0]
    purchase_date = it[1][0] if not it[1][0] is None else -1
    time_catalog = it[1][1][0]
    category = it[1][1][1]

    key = (itemid, category)
    value = 0 if purchase_date == -1 or compute_difference(purchase_date, time_catal
    return (key, value)

# map purchase info into a tuple
# ((itemId, category), 0/1)
# where:
# 1 is given in case purchaseTimestamp <= (firstTimeInCatalog + 2 years)
# 0 otherwise
# and then a reduceByKey is used to count the number of times each item was sold wit
# the two years after firstTimeInCatalog
# the final pairRDD is
# ((itemId, catalog), count)
sales_within_two_years_rdd = joined_rdd.map(map_joined_data)\
.reduceByKey(lambda it1, it2: it1 + it2)
```

In [ ]:

```
# Select only those items in which count == 0
# These are the NotSoldFirst2Years items
unsold_within_two_years_rdd = sales_within_two_years_rdd.filter(lambda it: it[1] ==
```

In [ ]:

```
# map each line into
# (category, +1)
# to count the number of items per category which was unsold within two years
unsold_items_count_per_category = unsold_within_two_years_rdd.map(lambda it: (it[0][
.reduceByKey(lambda i1, i2: i1 + i2)
```

In [ ]:

```
# keep only the categories with count >= 50
res2 = unsold_items_count_per_category.filter(lambda it: it[1] >= 50)
```

In [ ]:

```
res2.saveAsTextFile(out2)
```

In [ ]:
```
usersPath = 'ExampleData/Users.txt'
appsPath = 'ExampleData/Apps.txt'
actionPath = 'ExampleData/Actions.txt'
output1 = 'outPart1/'
output2 = 'outPart2/'
```

In [ ]:
```
### PART 1
actions_rdd = sc.textFile(actionPath).cache()
```

In [ ]:
```
#Select install and the year 2022
def Install2022(line):
    fields = line.split(',')
    timestamp = fields[2]
    action = fields[3]

    return timestamp.startswith('2022')==True and action=='Install'


actions2022_rdd = actions_rdd.filter(Install2022)
```

In [ ]:
```
# Map to (appId, userId) and
# remove repeated pairs

def appUser(line):
    fields = line.split(',')
    userId = fields[0]
    appId = fields[1]

    return (appId, userId)


appUser_rdd = actions2022_rdd.map(appUser)\
.distinct()
```

In [ ]:
```
apps_rdd = sc.textFile(appsPath)
```

In [ ]:
```
# Map apps to to (appId, Price)
def appPrice(line):
    fields = line.split(',')
    appId = fields[0]
    price = float(fields[2])

    return (appId, price)


appPrice_rdd = apps_rdd.map(appPrice)
```

In [ ]:
```
# Join appPrice_rdd with appUser_rdd
joinActionsPrices = appPrice_rdd.join(appUser_rdd)
```

In [ ]:
```
# Map each pair to (userId, (free , non-free))
def freeNonfree(pair):
    price=pair[1][0]
```

```
    userId=pair[1][1]

    if (price>0):
        return (userId, (0, 1))
    else:
        return (userId, (1, 0))


userFreeNonfreeApps_rdd = joinActionsPrices.map(freeNonfree)
```

In [ ]:
```
# Count the number of free and non-free apps for each user.
# Select the users with more non-free apps than free apps

selectedUsersPart1_rdd = userFreeNonfreeApps_rdd.reduceByKey(lambda v1,v2: (v1[0]+v2
.filter(lambda pair: pair[1][1]>pair[1][0])
```

In [ ]:
```
# Keep only userId and number of non-free apps and store the result
result1 = selectedUsersPart1_rdd.mapValues(lambda v: v[1])
```

In [ ]:
```
#result1.collect()
```

In [ ]:
```
result1.saveAsTextFile(output1)
```

In [ ]:
```

```

In [ ]:
```
### PART 2
users_rdd = sc.textFile(usersPath)
```

In [ ]:
```
# Select Italian users and return pairs (UserId, None)
italianUsers_rdd = users_rdd.filter(lambda line: line.split(',')[3]=='Italian')\
.map(lambda line: (line.split(',')[0], None))
```

In [ ]:
```
# Map each action to ( userId (appId,timestamp, action))
def  extractPair(line):
    fields = line.split(',')
    userId = fields[0]
    appId = fields[1]
    timestamp = fields[2]
    action = fields[3]

    return ( (userId, (appId, timestamp, action)) )


usersActions_rdd = actions_rdd.map(extractPair)
```

In [ ]:
```
# Join usersActions_rdd with italianUsers_rdd
# and select only Italian users
italianUsersActions_rdd  = usersActions_rdd.join(italianUsers_rdd)
```

In [ ]:
```
#italianUsersActions_rdd.collect()
```

In [ ]:
```
# Compute the last action for each combination Italian user+app
# Map to ( (userId, appId) , (timestamp, action))
# and then compute max timestamp + action

def maxTimestamp(time1_action1, time2_action2):
    if (time1_action1[0]>time2_action2[0]):
        return time1_action1
    else:
        return time2_action2

usersLasAction_rdd = italianUsersActions_rdd.map(lambda p: ( (p[0], p[1][0][0]), (p[
.reduceByKey(maxTimestamp)
```

In [ ]:
```
#usersLasAction_rdd.collect()
```

In [ ]:
```
# Select the apps with last action equal to Install => currently installed app
users_installedapps = usersLasAction_rdd.filter(lambda p: p[1][1]=='Install')
```

In [ ]:
```
# Count the number of currently installed apps for each user
usersNumInstalledapps = users_installedapps.map(lambda p: (p[0][0], +1))\
.reduceByKey(lambda v1,v2: v1+v2).cache()
```

In [ ]:
```
# Compute the maximum value of number of installed apps
maxInstalledApps = usersNumInstalledapps.values().max()
```

In [ ]:
```
# Select the users associated with maxInstalledApps
result2 = usersNumInstalledapps.filter(lambda p: p[1]==maxInstalledApps)\
.keys()
```

In [ ]:
```
#result2.collect()
```

In [ ]:
```
result2.saveAsTextFile(output2)
```

In [ ]:
```

```

In [1]:
```
# Exam example #1
# Exercise #2
```

In [2]:
```
inputPathPOIs = "exam_ex2_data/POIs.txt"
outputPathPartA = "out_PartA"
outputPathPartB = "out_PartB"
```

In [3]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Read the content of POIs.txt
# POI_ID,latitude,longitude,city,country,category,subcategory
poisRDD = sc.textFile(inputPathPOIs)
```

In [4]:
```
# Select only the Italian cities
poisItalyRDD = poisRDD.filter(lambda line: line.split(",")[4]=="Italy").cache()
```

In [5]:
```
# Select only the POIs with subcategory="Taxi" or subcategory="Busstop"

def filterSubcategories(line):
    fields = line.split(",")
    subcategory = fields[6]

    if (subcategory=="taxi" or subcategory=="busstop"):
        return True
    else:
        return False


poisItalySelectedSubcategoriesRDD = poisItalyRDD.filter(filterSubcategories)
```

In [6]:
```
# Generate for each of the selected input lines a pair with
# key = city
# value = (taxi=0/1,busstop=0/1)

def extractPairsCitySubcategoriesCounters(line):
    fields = line.split(",")
    city = fields[3]
    subcategory = fields[6]

    if (subcategory=="taxi"):
        return (city, (1, 0))
    else:
        return (city, (0, 1))



cityTaxiBusstopRDD = poisItalySelectedSubcategoriesRDD.map(extractPairsCitySubcatego
```

In [8]:
```
# For each Italian city, compute
# - number of taxi POIs
```

```
# - number of Busstop POIs
cityNumTaxiNumBusstopRDD = cityTaxiBusstopRDD.reduceByKey(lambda v1,v2: (v1[0]+v2[0]
```

In [9]:
```
# Select only the element with
# - number of taxi POIs >=1
# - number of Busstop =0
selectedCityNumTaxiNumBusstopRDD = cityNumTaxiNumBusstopRDD\
.filter(lambda pair: pair[1][0]>=1 and pair[1][1]==0)
```

In [10]:
```
# Save the selected cities = the keys of selectedCityNumTaxiNumBusstop
selectedCityNumTaxiNumBusstopRDD.keys()\
.saveAsTextFile(outputPathPartA)
```

In [ ]:
```

```

In [11]:
```
# ****************************************
# Exercise 2 - Part B
# ****************************************
```

In [13]:
```
# Count the number of museum POIs for each Italian city

# Generate for each input line a pair with
# key = city
# value = 1 if subcategory="museum"
# value = 0 if subcategory!="museum"

def extractCityMuseum(line):
    fields = line.split(",")
    city = fields[3]
    subcategory = fields[6]

    if (subcategory=="museum"):
        return (city, 1)
    else:
        return (city, 0)


cityMuseumPOIsRDD = poisItalyRDD.map(extractCityMuseum)
cityMuseumPOIsRDD.collect()
```

Out[13]:
```
[('Turin', 1),
 ('Turin', 1),
 ('Turin', 0),
 ('Turin', 0),
 ('Rome', 1),
 ('Rome', 0),
 ('Rome', 0),
 ('Naples', 0),
 ('Turin', 0),
 ('Turin', 0),
 ('Turin', 0),
 ('Turin', 0),
 ('Milan', 0),
 ('Milan', 0),
 ('Carmagnola', 0),
 ('Turin', 0)]
```

In [14]:
```

```

```
# Count the number of museum POIs for each Italian city
cityNumMuseumPOIsRDD = cityMuseumPOIsRDD.reduceByKey(lambda v1,v2: v1+v2).cache()
cityNumMuseumPOIsRDD.collect()
```

Out[14]: [('Milan', 0), ('Carmagnola', 0), ('Turin', 2), ('Rome', 1), ('Naples', 0)]

In [15]:
```
# Each input element represents
# - an italian city
# - the number of museum POIs for that city
# Emit one tuple for each input pair. Each tuple represents
# - numOfCities (set to 1 for each input pair)
# - numMuseumPOIs (for that city) = value of the input pair
numCitiesNumMuseumPOIsRDD = cityNumMuseumPOIsRDD.map(lambda pair: (1, pair[1]))
numCitiesNumMuseumPOIsRDD.collect()
```

Out[15]: [(1, 0), (1, 0), (1, 2), (1, 1), (1, 0)]

In [16]:
```
numCitiesNumMuseumPOIsRDD.count() # but it is an action
```

Out[16]: 5

In [ ]:
```
# Compute total number of Italian cities and total number of "museum" POIs the
# Italy (i.e., in the Italian cities)
# Store it in a local variable of the driver
totalNumCitiesNumMuseumPOIs = numCitiesNumMuseumPOIsRDD\
.reduce(lambda tuple1,tuple2: (tuple1[0]+tuple2[0], tuple1[1]+tuple2[1]) )
```

In [ ]:
```
# Compute average number of "museum" POIs per city in Italy
# Store it in a local variable of the driver
average = totalNumCitiesNumMuseumPOIs[1]/totalNumCitiesNumMuseumPOIs[0]
```

In [ ]:
```
# Select only the Italian cities with a number of "museum" POIs
# (subcategory="museum") greater than the average number of "museum" POIs per
# city in Italy
selectedCityNumMuseumPOIsRDD = cityNumMuseumPOIsRDD.\
filter(lambda inputPair: inputPair[1]> average)
```

In [ ]:
```
# Store the selected cities
selectedCityNumMuseumPOIsRDD.keys()\
.saveAsTextFile(outputPathPartB)
```

In [2]:
```
# Exam example #1
# Exercise #2
```

In [3]:
```
inputPathPOIs = "exam_ex2_data/POIs.txt"
outputPathPartA = "out_PartASQL"
outputPathPartB = "out_PartBSQL"
```

In [4]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Read the content of POIs.txt
# POI_ID,latitude,longitude,city,country,category,subcategory
poisDF = spark.read.load(inputPathPOIs,\
                         format="csv",\
                         header=False,\
                         inferSchema=True)\
.withColumnRenamed("_c0", "poiId")\
.withColumnRenamed("_c3", "city")\
.withColumnRenamed("_c4", "state")\
.withColumnRenamed("_c5", "category")\
.withColumnRenamed("_c6", "subcategory")
```

In [5]:
```
poisDF.printSchema()
poisDF.show()
```

```
root
 |-- poiId: string (nullable = true)
 |-- _c1: double (nullable = true)
 |-- _c2: double (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- category: string (nullable = true)
 |-- subcategory: string (nullable = true)

+-----+----------+--------+----------+-------+---------+-----------+
|poiId|       _c1|     _c2|      city|  state| category|subcategory|
+-----+----------+--------+----------+-------+---------+-----------+
| P101|45.0621644|7.578633|     Turin|  Italy|  tourism|     museum|
| P102|45.0621644|7.578633|     Turin|  Italy|  tourism|     museum|
| P103|45.0621644|7.578633|     Turin|  Italy|  tourism| artgallery|
| P104|45.0621644|7.578633|     Turin|  Italy|     shop|      shoes|
| P105|45.0621644|7.578633|      Rome|  Italy|  tourism|     museum|
| P106|45.0621644|7.578633|      Rome|  Italy|  tourism| artgallery|
| P107|45.0621644|7.578633|      Rome|  Italy|  tourism| artgallery|
| P108|45.0621644|7.578633|    Naples|  Italy|  tourism|      shoes|
| P109|45.0621644|7.578633|     Turin|  Italy|     shop|      shoes|
| P110|45.0621644|7.578633|     Turin|  Italy|     shop|      shoes|
| P111|45.0621644|7.578633|    Munich|Germany|  tourism|     museum|
| P112|45.0621644|7.578633|    Munich|Germany|  tourism|     museum|
| P113|45.0621644|7.578633|    Munich|Germany|  tourism| artgallery|
| P114|45.0621644|7.578633|     Turin|  Italy|transport|       taxi|
| P115|45.0621644|7.578633|     Turin|  Italy|transport|    busstop|
| P116|45.0621644|7.578633|     Milan|  Italy|transport|       taxi|
| P117|45.0621644|7.578633|     Milan|  Italy|transport|    busstop|
| P118|45.0621644|7.578633|Carmagnola|  Italy|transport|       taxi|
| P119|45.0621644|7.578633|     Turin|  Italy|transport|       taxi|
+-----+----------+--------+----------+-------+---------+-----------+
```

In [6]:
```
# "register" poisDF
poisDF.createOrReplaceTempView("poisTable")
```

In [7]:
```
# Define some UDFs
def taxi(subcategory):
    if subcategory=='taxi':
        return 1
    else:
        return 0

def busstop(subcategory):
    if subcategory=='busstop':
        return 1
    else:
        return 0




spark\
.udf\
.register("taxi", lambda subcategory: taxi(subcategory))

spark\
.udf\
.register("busstop", lambda subcategory: busstop(subcategory))
```

Out[7]:
```
<function __main__.<lambda>(subcategory)>
```

In [8]:
```
resADF = spark.sql("""SELECT city
FROM poisTable
WHERE state='Italy' AND
(subcategory='taxi' OR subcategory='busstop')
GROUP BY city
HAVING SUM(taxi(subcategory))>=1 AND
SUM(busstop(subcategory))=0
""")
```

In [9]:
```
resADF.show()
```

```
+----------+
|      city|
+----------+
|Carmagnola|
+----------+
```

In [10]:
```
resADF.write.csv(outputPathPartA, header=False)
```

In [ ]:

In [ ]:

In [ ]:
```
# ****************************************
# Exercise 2 - Part B
# ****************************************
```

In [11]:
```
# Define a UDF
def museum(subcategory):
    if subcategory=='museum':
        return 1
    else:
        return 0



spark\
.udf\
.register("museum", lambda subcategory: museum(subcategory))
```

Out[11]:
```
<function __main__.<lambda>(subcategory)>
```

In [12]:
```
# Num. Museums for each Italian city
numMuseumsCityDF = \
spark.\
sql("""SELECT city,
SUM(museum(subcategory)) AS numMuseums
FROM poisTable
WHERE state='Italy'
GROUP BY city
""")
```

In [13]:
```
numMuseumsCityDF.show()
```

```
+----------+----------+
|      city|numMuseums|
+----------+----------+
|Carmagnola|       0.0|
|    Naples|       0.0|
|     Milan|       0.0|
|      Rome|       1.0|
|     Turin|       2.0|
+----------+----------+
```

In [14]:
```
numMuseumsCityDF\
.createOrReplaceTempView("numMuseumsCityTable")
```

In [15]:
```
# Compute the average number of museum per Italian city
avgMuseumsDF = \
spark\
.sql("""SELECT avg(numMuseums) AS Threshold
FROM numMuseumsCityTable""")
```

In [17]:
```
avgMuseumsDF.show()
```

```
+---------+
|Threshold|
+---------+
|      0.6|
+---------+
```

In [16]:

```
avgMuseumsDF\
.createOrReplaceTempView("avgMuseumsTable")
```

In [ ]:

In [ ]:
```
# Compute the average number of museum per Italian city
resBDF = spark.sql("""SELECT city
FROM numMuseumsCityTable, avgMuseumsTable
WHERE numMuseumsCityTable.numMuseums>avgMuseumsTable.Threshold""")
```

In [ ]:
```
# Solution with table function
resBTDF = spark.sql("""SELECT city
FROM numMuseumsCityTable,
(SELECT avg(numMuseums) AS Threshold FROM numMuseumsCityTable) avgMuseumsTable
WHERE numMuseumsCityTable.numMuseums>avgMuseumsTable.Threshold""")
```

In [ ]:
```
#resBTDF.show()
```

In [ ]:
```
resBDF.write.csv(outputPathPartB, header=False)
```

In [ ]:

In [1]:
```
# Exam example #2
# Exercise #2
```

In [2]:
```
inputPathTemperatures = "exam_ex2_data/Temperatures.txt"
outputPathPartA = "out_PartA"
outputPathPartB = "out_PartB"
```

In [3]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Read the content of Temperatures.txt
temperatures = sc.textFile(inputPathTemperatures)
```

In [5]:
```
# Select data of year summer 2015
tempSummer2015 = temperatures\
.filter(lambda line: line.split(",")[0]>="2015/06/01" and line.split(",")[0]<="2015/
tempSummer2015.collect()
```

Out[5]:
```
['2015/07/20,Turin,Italy,32.5,26.0',
 '2015/07/21,Turin,Italy,32.5,26.0',
 '2015/07/22,Turin,Italy,32.5,26.0',
 '2015/07/23,Turin,Italy,32.5,26.0',
 '2015/07/24,Turin,Italy,36.5,26.0',
 '2015/07/25,Turin,Italy,32.5,-26.0',
 '2015/07/20,Rome,Italy,22.5,-26.0',
 '2015/07/21,Rome,Italy,22.5,-26.0',
 '2015/07/22,Rome,Italy,22.5,-26.0',
 '2015/07/23,Rome,Italy,22.5,-26.0',
 '2015/07/24,Rome,Italy,22.5,-26.0',
 '2015/07/25,Rome,Italy,22.5,1.0']
```

In [6]:
```
# Extract ( (city, county), (maxTemp,1) )
def extractPairCityCountryMaxTempOne(line):
    fields = line.split(",")
    city = fields[1]
    country = fields[2]
    maxTemp = float(fields[3])

    return (  (city, country), (maxTemp, 1)  )


cityCountryMaxTempCount = tempSummer2015.map(extractPairCityCountryMaxTempOne)
cityCountryMaxTempCount.collect()
```

Out[6]:
```
[(('Turin', 'Italy'), (32.5, 1)),
 (('Turin', 'Italy'), (32.5, 1)),
 (('Turin', 'Italy'), (32.5, 1)),
 (('Turin', 'Italy'), (32.5, 1)),
 (('Turin', 'Italy'), (36.5, 1)),
 (('Turin', 'Italy'), (32.5, 1)),
 (('Rome', 'Italy'), (22.5, 1)),
 (('Rome', 'Italy'), (22.5, 1)),
 (('Rome', 'Italy'), (22.5, 1)),
 (('Rome', 'Italy'), (22.5, 1)),
 (('Rome', 'Italy'), (22.5, 1)),
 (('Rome', 'Italy'), (22.5, 1))]
```

In [7]:
```
# Sum temps and num. lines
cityCountrySumMaxTempCount = cityCountryMaxTempCount\
.reduceByKey( lambda v1, v2: (v1[0]+v2[0], v1[1]+v2[1]))\
.cache()
cityCountrySumMaxTempCount.collect()
```

Out[7]:
```
[(('Turin', 'Italy'), (199.0, 6)), (('Rome', 'Italy'), (135.0, 6))]
```

In [8]:
```
# Compute average(max_temp) for each city+country
cityCountryAvg = cityCountrySumMaxTempCount.mapValues(lambda value: value[0]/value[1
cityCountryAvg.collect()
```

Out[8]:
```
[(('Turin', 'Italy'), 33.166666666666664), (('Rome', 'Italy'), 22.5)]
```

In [29]:
```
# Store the results of the first part
cityCountryAvg.saveAsTextFile(outputPathPartA)
```

In [ ]:
```

```

In [30]:
```
# ****************************************
# Exercise 2 - Part B
# ****************************************
```

In [32]:
```
# Extract pairs (country, (sum maxTemp, num lines)
# from cityCountrySumMaxTempCount
countryMaxTempCount = cityCountrySumMaxTempCount.map(lambda pair: (pair[0][1], pair[
```

In [34]:
```
# Sum temps and num. lines for each country
countrySumMaxTempCount = countryMaxTempCount\
.reduceByKey(lambda v1, v2: (v1[0]+v2[0], v1[1]+v2[1])  )
```

In [46]:
```
# Compute average(maxTemp) for each country
countryAvgTemp =countrySumMaxTempCount.mapValues(lambda value: value[0]/value[1])
```

In [47]:
```
countryAvgTemp.collect()
```

Out[47]:
```
[('Italy', 27.833333333333332)]
```

In [44]:
```
# Map cityCountryAvg to (country, (city, avg(maxtemp) )
county_CityAvgMaxTemp = cityCountryAvg\
.map(lambda pair: ( pair[0][1], (pair[0][0], pair[1]) ) )
```

In [45]:
```
county_CityAvgMaxTemp.collect()
```

Out[45]:
```
[('Italy', ('Turin', 33.166666666666664)), ('Italy', ('Rome', 22.5))]
```

In [ ]:
```

```

In [50]:
```
# Join (join condition -> same country)
# Results ( country, ((city, avgMaTempCity), avgMaxTempCountry) )
resJoin = county_CityAvgMaxTemp.join(countryAvgTemp)
```

In [49]:
```
resJoin.collect()
```

Out[49]:
```
[('Italy', (('Turin', 33.166666666666664), 27.833333333333332)),
 ('Italy', (('Rome', 22.5), 27.833333333333332))]
```

In [55]:
```
# Filter average city > average country+5

filteredResJoin = resJoin.filter(lambda pair: pair[1][0][1]> (pair[1][1]+5) )
```

In [56]:
```
filteredResJoin.collect()
```

Out[56]:
```
[('Italy', (('Turin', 33.166666666666664), 27.833333333333332))]
```

In [57]:
```
# Extract selecte cities
hotCities = filteredResJoin.map(lambda pair: pair[1][0][0])
```

In [59]:
```
# Save hot cities in the second output folder
hotCities.saveAsTextFile(outputPathPartB)
```

In [ ]:
```

```

In [ ]:
```
#SQL
```

In [37]:
```
data = spark.read.load(inputPathTemperatures,\
                       format='csv', header=False, InferSchema=True)\
                      .withColumnRenamed('_c0','date')\
                      .withColumnRenamed('_c1','city')\
                      .withColumnRenamed('_c2','country')\
                      .withColumnRenamed('_c3','max')
data.show()
```

```
+----------+-----+-------+----+-----+
|      date| city|country| max|  _c4|
+----------+-----+-------+----+-----+
|2015/07/20|Turin|  Italy|32.5| 26.0|
|2015/07/21|Turin|  Italy|32.5| 26.0|
|2015/07/22|Turin|  Italy|32.5| 26.0|
|2015/07/23|Turin|  Italy|32.5| 26.0|
|2015/07/24|Turin|  Italy|36.5| 26.0|
|2015/07/25|Turin|  Italy|32.5|-26.0|
|2015/07/20| Rome|  Italy|22.5|-26.0|
|2015/07/21| Rome|  Italy|22.5|-26.0|
|2015/07/22| Rome|  Italy|22.5|-26.0|
|2015/07/23| Rome|  Italy|22.5|-26.0|
|2015/07/24| Rome|  Italy|22.5|-26.0|
|2015/07/25| Rome|  Italy|22.5|  1.0|
|2015/01/01| Rome|  Italy|22.5|  1.0|
|2018/01/01| Rome|  Italy|22.5|  1.0|
|2014/12/11| Rome|  Italy|22.5|  1.0|
+----------+-----+-------+----+-----+
```

In [48]:
```
df = data.filter('date>"2015/06/01" and date<"2015/08/31"').cache()
df.show()
```

```
+----------+-----+-------+----+-----+
|      date| city|country| max|  _c4|
+----------+-----+-------+----+-----+
|2015/07/20|Turin|  Italy|32.5| 26.0|
|2015/07/21|Turin|  Italy|32.5| 26.0|
|2015/07/22|Turin|  Italy|32.5| 26.0|
|2015/07/23|Turin|  Italy|32.5| 26.0|
|2015/07/24|Turin|  Italy|36.5| 26.0|
|2015/07/25|Turin|  Italy|32.5|-26.0|
|2015/07/20| Rome|  Italy|22.5|-26.0|
|2015/07/21| Rome|  Italy|22.5|-26.0|
|2015/07/22| Rome|  Italy|22.5|-26.0|
|2015/07/23| Rome|  Italy|22.5|-26.0|
|2015/07/24| Rome|  Italy|22.5|-26.0|
|2015/07/25| Rome|  Italy|22.5|  1.0|
+----------+-----+-------+----+-----+
```

In [49]:
```
def concat(city, country):
    return city+'-'+country

df.createOrReplaceTempView('table')
spark.udf.register('concat', lambda city, country: concat(city, country))
```

Out[49]:
```
<function __main__.<lambda>(city, country)>
```

In [55]:
```
target = spark.sql(""" SELECT concat(city,country) AS name, avg(max) AS average
                   FROM table
                   GROUP BY city, country
                   """)
target.show()
```

```
+-----------+------------------+
|       name|           average|
+-----------+------------------+
| Rome-Italy|              22.5|
|Turin-Italy|33.166666666666664|
+-----------+------------------+
```

In [51]:
```
RDD = target.rdd
RDD.collect()
```

Out[51]:
```
[Row(name='Rome-Italy', average=22.5),
 Row(name='Turin-Italy', average=33.166666666666664)]
```

In [52]:
```
mapRDD = RDD.map(lambda row: (row.name, row.average))
mapRDD.collect()
```

Out[52]:
```
[('Rome-Italy', 22.5), ('Turin-Italy', 33.166666666666664)]
```

In [53]:
```
hotAvg = df.groupBy('country').agg({'max':'avg'})
hotAvg.show()
```

```
+-------+-----------------+
|country|         avg(max)|
+-------+-----------------+
```

```
|  Italy|27.833333333333332|
+-------+------------------+
```

In [ ]:

```
target.createOrReplaceTempView('avgCity')

hotCities = spark.sql("""
SELECT city
FROM table, avgCity
WHERE avg
""")
```

In [1]:
```
# Exam example #3
# Exercise #2
```

In [2]:
```
CPUthr=10.0
RAMthr=1.5
inputPath = "exam_ex2_data/PerformanceLog.txt"
outputPathPartA = "out_PartA"
outputPathPartB = "out_PartB"
```

In [3]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Read the content of PerformanceLog.txt
statistics = sc.textFile(inputPath)
```

In [6]:
```
# Select data related to May 2018
# Example data: 2018/05/01,15:40,VS1,10.5,0.5
statisticsMay2018 = statistics.filter(lambda line: line.startswith("2018/05")).cache
statisticsMay2018.take(3)
```

Out[6]:
```
['2018/05/01,15:40,VS1,10.5,0.5',
 '2018/05/01,15:41,VS1,10.5,0.5',
 '2018/05/01,15:42,VS1,10.5,0.5']
```

In [7]:
```
# Map each input line to a pair
# key = (VSID, Hour)
# value = (1, CPUUtiliation, RAMUtiliation)

def extractPair(line):
    fields = line.split(",")
    time = fields[1]
    hour = time.split(":")[0]
    vsid = fields[2]
    cpuUtil = float(fields[3])
    ramUtil = float(fields[4])

    return ( (vsid, hour), (1, cpuUtil, ramUtil)  )


vsidHourCounter = statisticsMay2018.map(extractPair)
```

In [8]:
```
# Sum the three parts of the value part
# Sum num lines
# Sum CPUutilization
# Sum RAMutilization
vsidHourTotals = vsidHourCounter\
.reduceByKey(lambda v1,v2: (v1[0]+v2[0], v1[1]+v2[1], v1[2]+v2[2]) )
```

In [9]:
```
# Filter pairs by applying the two thresholds
# Avg(CPUUsage%)>CPUthr
# Avg(RAMUsage%)>RAMthr

selectedVsidHour = vsidHourTotals\
.filter(lambda pair: (pair[1][1]/pair[1][0])>CPUthr and (pair[1][2]/pair[1][0])>RAMt
```

In [8]:
```
# Select only the key part and store the result in the output folder
selectedVsidHour.keys()\
.saveAsTextFile(outputPathPartA)
```

```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)
<ipython-input-8-abe7834d3b04> in <module>
      1 # Select only the key part and store the result in the output folder
      2 selectedVsidHour.keys()\
----> 3 .saveAsTextFile(outputPathPartA)

/opt/cloudera/parcels/CDH/lib/spark/python/pyspark/rdd.py in saveAsTextFile(self, pa
th, compressionCodecClass)
   1568             keyed._jrdd.map(self.ctx._jvm.BytesToString()).saveAsTextFile(pa
th, compressionCodec)
   1569         else:
-> 1570             keyed._jrdd.map(self.ctx._jvm.BytesToString()).saveAsTextFile(pa
th)
   1571
   1572     # Pair functions

/opt/cloudera/parcels/CDH/lib/spark/python/lib/py4j-0.10.7-src.zip/py4j/java_gatewa
y.py in __call__(self, *args)
   1255         answer = self.gateway_client.send_command(command)
   1256         return_value = get_return_value(
-> 1257             answer, self.gateway_client, self.target_id, self.name)
   1258
   1259         for temp_arg in temp_args:

/opt/cloudera/parcels/CDH/lib/spark/python/pyspark/sql/utils.py in deco(*a, **kw)
     61     def deco(*a, **kw):
     62         try:
---> 63             return f(*a, **kw)
     64         except py4j.protocol.Py4JJavaError as e:
     65             s = e.java_exception.toString()

/opt/cloudera/parcels/CDH/lib/spark/python/lib/py4j-0.10.7-src.zip/py4j/protocol.py
 in get_return_value(answer, gateway_client, target_id, name)
    326                 raise Py4JJavaError(
    327                     "An error occurred while calling {0}{1}{2}.\n".
--> 328                     format(target_id, ".", name), value)
    329             else:
    330                 raise Py4JError(

Py4JJavaError: An error occurred while calling o94.saveAsTextFile.
: org.apache.hadoop.mapred.FileAlreadyExistsException: Output directory hdfs://BigDa
taHA/user/s296721/out_PartA already exists
        at org.apache.hadoop.mapred.FileOutputFormat.checkOutputSpecs(FileOutputForm
at.java:131)
        at org.apache.spark.internal.io.HadoopMapRedWriteConfigUtil.assertConf(Spark
HadoopWriter.scala:287)
        at org.apache.spark.internal.io.SparkHadoopWriter$.write(SparkHadoopWriter.s
cala:71)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$1.appl
y$mcV$sp(PairRDDFunctions.scala:1096)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$1.appl
y(PairRDDFunctions.scala:1094)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$1.appl
y(PairRDDFunctions.scala:1094)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scal
a:151)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scal
a:112)
        at org.apache.spark.rdd.RDD.withScope(RDD.scala:363)
        at org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopDataset(PairRDDFunction
s.scala:1094)
```

```
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$4.apply$m
cV$sp(PairRDDFunctions.scala:1067)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$4.apply(P
airRDDFunctions.scala:1032)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$4.apply(P
airRDDFunctions.scala:1032)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scal
a:151)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scal
a:112)
        at org.apache.spark.rdd.RDD.withScope(RDD.scala:363)
        at org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopFile(PairRDDFunctions.s
cala:1032)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$1.apply$m
cV$sp(PairRDDFunctions.scala:958)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$1.apply(P
airRDDFunctions.scala:958)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$1.apply(P
airRDDFunctions.scala:958)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scal
a:151)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scal
a:112)
        at org.apache.spark.rdd.RDD.withScope(RDD.scala:363)
        at org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopFile(PairRDDFunctions.s
cala:957)
        at org.apache.spark.rdd.RDD$$anonfun$saveAsTextFile$1.apply$mcV$sp(RDD.scal
a:1499)
        at org.apache.spark.rdd.RDD$$anonfun$saveAsTextFile$1.apply(RDD.scala:1478)
        at org.apache.spark.rdd.RDD$$anonfun$saveAsTextFile$1.apply(RDD.scala:1478)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scal
a:151)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scal
a:112)
        at org.apache.spark.rdd.RDD.withScope(RDD.scala:363)
        at org.apache.spark.rdd.RDD.saveAsTextFile(RDD.scala:1478)
        at org.apache.spark.api.java.JavaRDDLike$class.saveAsTextFile(JavaRDDLike.sc
ala:550)
        at org.apache.spark.api.java.AbstractJavaRDDLike.saveAsTextFile(JavaRDDLike.
scala:45)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.jav
a:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI
mpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
        at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:357)
        at py4j.Gateway.invoke(Gateway.java:282)
        at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
        at py4j.commands.CallCommand.execute(CallCommand.java:79)
        at py4j.GatewayConnection.run(GatewayConnection.java:238)
        at java.lang.Thread.run(Thread.java:748)
```

In [ ]:

In [ ]:

In [ ]:
```
# ****************************************
# Exercise 2 - Part B
# ****************************************
```

In [10]:
```
# Map each input line to a pair
```

```
# key = (VSID, date, hour)
# value = CPUUtilization
# Example of input line: 2018/05/01,15:40,VS1,10.5,0.5

def extractVSIDDateHourCPU(line):
    fields = line.split(",")

    date = fields[0]
    time = fields[1]
    hour = time.split(":")[0]
    vsid = fields[2]
    cpuUtil = float(fields[3])

    return ( (vsid, date, hour), cpuUtil)


vsidDateHourCPUutil = statisticsMay2018.map(extractVSIDDateHourCPU)
vsidDateHourCPUutil.take(3)
```

Out[10]:
```
[(('VS1', '2018/05/01', '15'), 10.5),
 (('VS1', '2018/05/01', '15'), 10.5),
 (('VS1', '2018/05/01', '15'), 10.5)]
```

In [11]:
```
# Compute max CPUUtilization for each key (VSID, date, hour)
vsidDateHourMaxCPUutil = vsidDateHourCPUutil\
.reduceByKey(lambda cpuUtil1, cpuUtil2: max(cpuUtil1, cpuUtil2) )
vsidDateHourMaxCPUutil.take(3)
```

Out[11]:
```
[(('VS1', '2018/05/01', '15'), 10.5),
 (('VS2', '2018/05/01', '05'), 30.5),
 (('VS3', '2018/05/01', '03'), 90.4)]
```

In [12]:
```
# Select only the pairs with maxCPUUtilization>90 or <10
vsidDateHourMaxCPUHighLow = vsidDateHourMaxCPUutil\
.filter(lambda pair: pair[1]>90.0 or pair[1]<10.0)
vsidDateHourMaxCPUHighLow.take(3)
```

Out[12]:
```
[(('VS3', '2018/05/01', '03'), 90.4),
 (('VS3', '2018/05/01', '05'), 90.4),
 (('VS3', '2018/05/01', '06'), 90.4)]
```

In [13]:
```
# Return one pair of each input element
# key = (VSID, date)
# value = (1 if >90, 1 if <10)

def extractPairVSIDDateCounters(pair):
    if pair[1]>90:
        greaterThan90 = 1
        lessThan10 = 0
    else:
        lessThan10 = 1
        greaterThan90 = 0

    return (  (pair[0][0], pair[0][1]), (greaterThan90, lessThan10) )


vsidDateHighLow = vsidDateHourMaxCPUHighLow.map(extractPairVSIDDateCounters)
vsidDateHighLow.take(3)
```

Out[13]:
```
[(('VS3', '2018/05/01'), (1, 0)),
 (('VS3', '2018/05/01'), (1, 0)),
```

```
(('VS3', '2018/05/01'), (1, 0))]
```

In [14]:
```
# Compute
# how many hours with maxCPUutilization>90 and
# how many hours with maxCPUutilization<10
# Sum the two counters of the value part
vsidDateSumHighLow = vsidDateHighLow\
.reduceByKey(lambda v1, v2: (v1[0]+v2[0], v1[1]+v2[1]) )
vsidDateSumHighLow.collect()
```

Out[14]:
```
[(('VS3', '2018/05/01'), (15, 8)), (('VS2', '2018/05/01'), (1, 0))]
```

In [15]:
```
# Select only the pairs (VSID, date) for which
# (Num. of hours with max CPU utilization greater than 90.0) >= 8 hours
# (Num. of hours with max CPU utilization less than 10.0) >= 8 hours
selectedVsidDateSumHighLow = vsidDateSumHighLow\
.filter(lambda pair: pair[1][0]>=8 and pair[1][1]>=8)
```

In [ ]:
```
# Save the selected pairs (VSID, Date)
selectedVsidDateSumHighLow.keys()\
.saveAsTextFile(outputPathPartB)
```

In [ ]:
```
# Exam example #3
# Exercise #2 - SparkSQL version
```

In [ ]:
```
CPUthr=10.0
RAMthr=1.5
inputPath = "exam_ex2_data/PerformanceLog.txt"
outputPathPartA = "out_PartASQL"
outputPathPartB = "out_PartBSQL"
```

In [ ]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Create a DataFrame from PerformanceLog.txt
statisticsDF = spark.read.load(inputPath,\
                               format="csv",\
                               header=False,\
                               inferSchema=True)\
.withColumnRenamed("_c0","date")\
.withColumnRenamed("_c1","hour_minute")\
.withColumnRenamed("_c2","VSID")\
.withColumnRenamed("_c3","CPUUsage")\
.withColumnRenamed("_c4","RAMUsage")
```

In [ ]:
```
#statisticsDF.show()
#statisticsDF.printSchema()
```

In [ ]:
```
statisticsDF.createOrReplaceTempView("statistics")
```

In [ ]:
```
def extractHour(s):
    return s.split(":")[0]

spark.udf.register("extractHour", extractHour)
```

In [ ]:
```
selectedVsidHourDF = spark.sql("""SELECT VSID,extractHour(hour_minute) as Hour
FROM statistics
WHERE date>='2018/05/01' and date<='2018/05/31'
GROUP BY VSID,extractHour(hour_minute)
HAVING avg(CPUUsage)>"""+str(CPUthr)+"""
AND avg(RAMUsage)>"""+str(RAMthr))
```

In [ ]:
```
#selectedVsidHourDF.show()
```

In [ ]:
```
selectedVsidHourDF.write\
.csv(outputPathPartA, header=False)
```

In [ ]:
```

```

In [ ]:
```
# ****************************************
```

```
# Exercise 2 - Part B
# ************************************
```

In [ ]:
```
# Compute max CPUUtilization for each group (VSID, date, hour)
# and select only the groups with max(CPUusage)>90 or < 10
maxPerVSIDDateHourDF = spark.sql("""SELECT VSID,Date,
extractHour(hour_minute) as Hour,
max(CPUUsage) as MaxCPUUsage
FROM statistics
WHERE date>='2018/05/01' and date<='2018/05/31'
GROUP BY VSID,Date, extractHour(hour_minute)
HAVING max(CPUUsage)>90
OR max(CPUUsage)<10""")
```

In [ ]:
```
maxPerVSIDDateHourDF.createOrReplaceTempView("maxValuesTable")
```

In [ ]:
```
#maxPerVSIDDateHourDF.show()
```

In [ ]:
```
def greaterThan90(maxcpuusage):
    if maxcpuusage>90:
        return 1
    else:
        return 0

spark.udf.register("greaterThan90", greaterThan90)
```

In [ ]:
```
def lessThan10(maxcpuusage):
    if maxcpuusage<10:
        return 1
    else:
        return 0

spark.udf.register("lessThan10", lessThan10)
```

In [ ]:
```
# Compute
# how many hours with maxCPUutilization>90 and
# how many hours with maxCPUutilization<10
# for each combination VSID,Date
# and select those groups with at least 8 hours of both types
selectedVSIDDateDF = spark.sql("""SELECT VSID,Date
FROM maxValuesTable
GROUP BY VSID,Date
HAVING SUM(greaterThan90(MaxCPUUsage))>=8
AND SUM(lessThan10(MaxCPUUsage))>=8""")
```

In [ ]:
```
#selectedVSIDDateDF.show()
```

In [ ]:
```
selectedVSIDDateDF.write\
.csv(outputPathPartB, header=False)
```

In [ ]:
```

```

In [20]:
```
# Exam example #4
# Exercise #2
```

In [21]:
```
inputPathPrices = "exam_ex2_data/Stocks_Prices.txt"
outputPathPartA = "out_PartAv2"
outputPathPartB = "out_PartBv2"
```

In [22]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Read the content of stocks_prices.txt
stockPrices = sc.textFile(inputPathPrices)
```

In [23]:
```
# Select only year 2017 data
stockPrices2017 = stockPrices\
.filter(lambda line: line.split(",")[1].startswith("2017"))
```

In [24]:
```
# Create for each input line (of year 2017) a pair with
# key = (stockId,Date)
# value = (price,price)

def extractStockIdDatePrice(line):
    fields = line.split(",")
    stockId = fields[0]
    date = fields[1]
    price = float(fields[3])

    return ( (stockId, date), (price, price) )

stockDate_Price = stockPrices2017.map(extractStockIdDatePrice)
```

In [25]:
```
# Compute max price and min price for each (stockId, date)
stockDate_MinandMaxPrice = stockDate_Price\
.reduceByKey(lambda v1, v2: (max(v1[0],v2[0]), min(v1[1],v2[1])) )
```

In [26]:
```
# Compute daily variation for each (stockId, date)
stockDate_DailyVariation = stockDate_MinandMaxPrice\
.mapValues(lambda value: value[0]-value[1])\
.cache()
```

In [27]:
```
stockDate_DailyVariation.collect()
```

Out[27]:
```
[(('TSLA', '2017/05/21'), 50.0),
 (('GOOG', '2017/05/21'), 50.0),
 (('GOOG', '2017/06/21'), 50.0),
 (('FCA', '2017/06/21'), 0.010000000000000009),
 (('GOOG', '2017/05/22'), 2.0),
 (('GOOG', '2017/05/23'), 2.039999999999999)]
```

In [28]:
```
# Select only the elements with a daily variation > 10
stockDate_DailyVariationGreater10 = stockDate_DailyVariation\
```

```
.filter(lambda pair: pair[1]>10)
```

In [29]:
```
# Count the number of dates with daily variation > 10 for each stockId

# Map each input element to a pair (stockId, +1)
stockIdOne = stockDate_DailyVariationGreater10\
.map(lambda pair: (pair[0][0], 1) )
```

In [30]:
```
# ReduceByKey to count the number of dates for each stockId
stockIdNumDates = stockIdOne.reduceByKey(lambda v1, v2: v1 + v2)
```

In [19]:
```
#stockIdNumDates.collect()
```

In [ ]:
```
# Save result
stockIdNumDates.saveAsTextFile(outputPathPartA)
```

In [ ]:
```
```

In [ ]:
```
# ****************************************
# Exercise 2 - Part B
# ****************************************
```

In [31]:
```
# This funciton is already provided
from datetime import datetime, timedelta

def previousDate(mydate):
    currentDate = datetime.strptime(mydate, "%Y/%m/%d")
    prevDate=currentDate - timedelta(days=1)

    return prevDate.strftime("%Y/%m/%d")
```

In [32]:
```
# Consider stockDate_DailyVariation
# It contains for each stockid+Date the daily variation.
# Each element of stockDate_DailyVariation is the first date of a
# sequence of two dates and also the second date of another sequence of two dates.
# Emit for each input element (stockid+Date,DailyVariation) two pairs:
# 1 - key=stockId+Date - value=(DailyVariation, +1) at date Date -> This is the firs
#     element of the sequence of two dates starting at date "Date" associated with s
# 2 - key=stockId+(Date-1) - value=(DailyVariation, +1) at date Date -> This is the
#     element of the sequence of two dates starting at date "Date-1" associated with

def extractStockIdDateDailyVar(pair):
    stockId = pair[0][0]
    date = pair[0][1]
    prevDate = previousDate(date)
    dailyVariation = pair[1]

    returnedPairs = []
    # Append the pair key=stockId+Date - value=Date+DailyVariation
    returnedPairs.append( ((stockId, date) , (dailyVariation,+1)) )

    # Append the pair key=stockId+(Date-1) - value=Date+DailyVariation
    returnedPairs.append( ((stockId, prevDate) , (dailyVariation,+1)) )
```

```
    return returnedPairs


stockIdSequenceFirstDate_DailyVariations = stockDate_DailyVariation\
.flatMap(extractStockIdDateDailyVar)
```

In [33]:
```
stockIdSequenceFirstDate_DailyVariations.collect()
```

Out[33]:
```
[(('TSLA', '2017/05/21'), (50.0, 1)),
 (('TSLA', '2017/05/20'), (50.0, 1)),
 (('GOOG', '2017/05/21'), (50.0, 1)),
 (('GOOG', '2017/05/20'), (50.0, 1)),
 (('GOOG', '2017/06/21'), (50.0, 1)),
 (('GOOG', '2017/06/20'), (50.0, 1)),
 (('FCA', '2017/06/21'), (0.010000000000000009, 1)),
 (('FCA', '2017/06/20'), (0.010000000000000009, 1)),
 (('GOOG', '2017/05/22'), (2.0, 1)),
 (('GOOG', '2017/05/21'), (2.0, 1)),
 (('GOOG', '2017/05/23'), (2.039999999999999, 1)),
 (('GOOG', '2017/05/22'), (2.039999999999999, 1))]
```

In [34]:
```
# Apply reduceByKey to compute absolute difference between (two) values
# and number of values for each window
stockIdTwoConsecutiveDatesDailyVariations = stockIdSequenceFirstDate_DailyVariations
.reduceByKey(lambda v1,v2: ( abs(v1[0]-v2[0]) , v1[1]+v2[1] ) )
```

In [35]:
```
stockIdTwoConsecutiveDatesDailyVariations.collect()
```

Out[35]:
```
[(('TSLA', '2017/05/21'), (50.0, 1)),
 (('TSLA', '2017/05/20'), (50.0, 1)),
 (('GOOG', '2017/05/21'), (48.0, 2)),
 (('GOOG', '2017/05/20'), (50.0, 1)),
 (('GOOG', '2017/06/21'), (50.0, 1)),
 (('FCA', '2017/06/21'), (0.010000000000000009, 1)),
 (('GOOG', '2017/06/20'), (50.0, 1)),
 (('FCA', '2017/06/20'), (0.010000000000000009, 1)),
 (('GOOG', '2017/05/22'), (0.03999999999999915, 2)),
 (('GOOG', '2017/05/23'), (2.039999999999999, 1))]
```

In [36]:
```
# Select only the "stable trends"
def stableTrend(pair):
    # Select the two daily variations and compute the absolute
    # difference between those two values
    listDailyVariations=list(pair[1])

    #Check if there are at two daily variation values
    if (len(listDailyVariations)==2):
        dailyVariation1 = listDailyVariations[0]
        dailyVariation2 = listDailyVariations[1]

        # Check if their absolute difference betwen the two variations is at most 0.
        if abs(dailyVariation1-dailyVariation2) <= 0.1:
            return True
        else:
            return False

    else:
        return False
```

```
stockIDStableTrends = stockIdTwoConsecutiveDatesDailyVariations\
.filter(lambda value: value[1][0]<=0.1 and value[1][1]==2)
```

In [ ]:

```
# Select the keys and store them
stockIDStableTrends.keys()\
.saveAsTextFile(outputPathPartB)
```

In [37]:

```
stockIDStableTrends.keys().collect()
```

Out[37]:

```
[('GOOG', '2017/05/22')]
```

In [ ]:

In [1]:
```
# Exam example #4
# Exercise #2 - SQL
```

In [2]:
```
inputPathPrices = "exam_ex2_data/Stocks_Prices.txt"
outputPathPartA = "out_PartAv2SQL"
outputPathPartB = "out_PartBv2SQL"
```

In [3]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Read the content of stocks_prices.txt
stockPricesDF = spark.read.load(inputPathPrices,\
                                format="csv",\
                                header=False,\
                                inferSchema=True)\
.withColumnRenamed("_c0", "StockId")\
.withColumnRenamed("_c1", "Date")\
.withColumnRenamed("_c2", "Time")\
.withColumnRenamed("_c3", "Price")
```

In [4]:
```
stockPricesDF.printSchema()
stockPricesDF.show()
```

```
root
 |-- StockId: string (nullable = true)
 |-- Date: string (nullable = true)
 |-- Time: string (nullable = true)
 |-- Price: double (nullable = true)

+-------+----------+-----+-----+
|StockId|      Date| Time|Price|
+-------+----------+-----+-----+
|   TSLA|2017/05/21|15:01|45.32|
|   TSLA|2017/05/21|15:02|15.32|
|   TSLA|2017/05/21|15:03|45.32|
|   TSLA|2017/05/21|15:04|15.32|
|   TSLA|2017/05/21|15:05|45.32|
|   TSLA|2017/05/21|15:06|15.32|
|   TSLA|2017/05/21|15:07|45.32|
|   TSLA|2017/05/21|15:08| 5.32|
|   TSLA|2017/05/21|15:09|55.32|
|   TSLA|2017/05/21|15:09| 5.32|
|   GOOG|2017/05/21|15:01|45.32|
|   GOOG|2017/05/21|15:02|15.32|
|   GOOG|2017/05/21|15:03|45.32|
|   GOOG|2017/05/21|15:04|15.32|
|   GOOG|2017/05/21|15:05|45.32|
|   GOOG|2017/05/21|15:06|15.32|
|   GOOG|2017/05/21|15:07|45.32|
|   GOOG|2017/05/21|15:08| 5.32|
|   GOOG|2017/05/21|15:09|55.32|
|   GOOG|2017/05/21|15:09| 5.32|
+-------+----------+-----+-----+
only showing top 20 rows
```

In [5]:
```
# Associate stockPricesDF to a temporary table
stockPricesDF.createOrReplaceTempView("Prices")
```

In [6]:
```
# Select only year 2017 data
# and compute max-min price for each stock in year 2017.

stockDate_DailyVariationDF = spark.sql("""SELECT StockId,Date,
max(price)-min(price) AS dailyVar
FROM Prices
WHERE Date>='2017/01/01' and Date<='2017/12/31'
GROUP BY StockId,Date""").cache()
```

In [7]:
```
stockDate_DailyVariationDF.show()
```

```
+-------+----------+--------------------+
|StockId|      Date|            dailyVar|
+-------+----------+--------------------+
|   GOOG|2017/05/21|                50.0|
|   GOOG|2017/06/21|                50.0|
|   GOOG|2017/05/23|   2.039999999999999|
|    FCA|2017/06/21|0.010000000000000009|
|   TSLA|2017/05/21|                50.0|
|   GOOG|2017/05/22|                 2.0|
+-------+----------+--------------------+
```

In [13]:
```
stockDate_DailyVariationDF.groupBy('StockId').agg({'dailyVar':'sum'}).show()
```

```
+-------+--------------------+
|StockId|       sum(dailyVar)|
+-------+--------------------+
|   TSLA|                50.0|
|   GOOG|  104.03999999999999|
|    FCA|0.010000000000000009|
+-------+--------------------+
```

In [8]:
```
# Associate stockDate_DailyVariationDF to a temporary table
stockDate_DailyVariationDF.createOrReplaceTempView("DailyVarTable")
```

In [9]:
```
# Count the number of dates with daily variation > 10 for each stockId
stockIdNumDatesDF = spark.sql("""SELECT StockId, COUNT(*) as NumDates
FROM DailyVarTable
WHERE dailyVar>10
GROUP BY StockId""")
```

In [10]:
```
stockIdNumDatesDF.printSchema()
stockIdNumDatesDF.show()
```

```
root
 |-- StockId: string (nullable = true)
 |-- NumDates: long (nullable = false)

+-------+--------+
|StockId|NumDates|
+-------+--------+
|   TSLA|       1|
|   GOOG|       2|
+-------+--------+
```

In [ ]:
```
# Save result
```

```
selectedCarsModelsDF.write.csv(outputPathPartA,header=False)
```

In [ ]:

In [ ]:
```
# ****************************************
# Exercise 2 - Part B
# ****************************************
```

In [ ]:
```
# This funciton is already provided
from datetime import datetime, timedelta

def previousDate(mydate):
    currentDate = datetime.strptime(mydate, "%Y/%m/%d")
    prevDate=currentDate - timedelta(days=1)

    return prevDate.strftime("%Y/%m/%d")
```

In [ ]:
```
#Extract an RDD from stockIdNumDatesDF
stockDate_DailyVariation =stockDate_DailyVariationDF.rdd
```

In [ ]:
```
#stockDate_DailyVariation.collect()
```

In [ ]:
```
# Consider stockDate_DailyVariation
# It contains for each stockid+Date the daily variation.
# Each element of stockDate_DailyVariation is the first date of a
# sequence of two dates and also the second date of another sequence of two dates.
# Emit for each input element (stockid+Date,DailyVariation) two pairs:
# 1 - key=stockId+Date - value=(DailyVariation, +1) at date Date -> This is the firs
#     element of the sequence of two dates starting at date "Date" associated with s
# 2 - key=stockId+(Date-1) - value=(DailyVariation, +1) at date Date -> This is the
#     element of the sequence of two dates starting at date "Date-1" associated with

def extractStockIdDateDailyVar(rowRecord):
    stockId = rowRecord["StockId"]
    date = rowRecord["Date"]
    prevDate = previousDate(date)
    dailyVariation = rowRecord["dailyVar"]

    returnedPairs = []
    # Append the pair key=stockId+Date - value=Date+DailyVariation
    returnedPairs.append( ((stockId, date) , (dailyVariation,+1)) )

    # Append the pair key=stockId+(Date-1) - value=Date+DailyVariation
    returnedPairs.append( ((stockId, prevDate) , (dailyVariation,+1)) )

    return returnedPairs


stockIdSequenceFirstDate_DailyVariations = stockDate_DailyVariation\
.flatMap(extractStockIdDateDailyVar)
```

In [ ]:
```
#stockIdSequenceFirstDate_DailyVariations.collect()
```

In [ ]:

```
# Apply reduceByKey to compute absolute difference between (two) values
# and number of values for each window
stockIdTwoConsecutiveDatesDailyVariations = stockIdSequenceFirstDate_DailyVariations
.reduceByKey(lambda v1,v2: ( abs(v1[0]-v2[0]) , v1[1]+v2[1] ) )
```

In [ ]:

```
#stockIdTwoConsecutiveDatesDailyVariations.collect()
```

In [ ]:

```
# Select only the "stable trends"
def stableTrend(pair):
    # Select the two daily variations and compute the absolute
    # difference between those two values
    listDailyVariations=list(pair[1])

    #Check if there are at two daily variation values
    if (len(listDailyVariations)==2):
        dailyVariation1 = listDailyVariations[0]
        dailyVariation2 = listDailyVariations[1]

        # Check if their absolute difference betwen the two variations is at most 0.
        if abs(dailyVariation1-dailyVariation2) <= 0.1:
            return True
        else:
            return False

    else:
        return False

stockIDStableTrends = stockIdTwoConsecutiveDatesDailyVariations\
.filter(lambda value: value[1][0]<=0.1 and value[1][1]==2)
```

In [ ]:

```
# Select the keys and store them
stockIDStableTrends.keys()\
.saveAsTextFile(outputPathPartB)
```

In [ ]:

```
#stockIDStableTrends.keys().collect()
```

In [ ]:

In [ ]:
```
# Exam example #5
# Exercise #2
```

In [ ]:
```
inputServers = "exam_ex2_data/Servers.txt"
inputAnomalies = "exam_ex2_data/Servers_TemperatureAnomalies.txt"
outputPathPartA = "out_PartA"
outputPathPartB = "out_PartB"
```

In [ ]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Read the content of Servers_TemperatureAnomalies.txt
anomaliesRDD = sc.textFile(inputAnomalies)
```

In [ ]:
```
# Select anomalies associated with years 2010-2018
# Example: SID13,2018/03/01_15:40,110
def select20102018(line):
    fields = line.split(",")
    date = fields[1].split("_")[0]

    if date>="2010/01/01" and date<="2018/12/31":
        return True
    else:
        return False


anomalies2010_2018RDD = anomaliesRDD.filter(select20102018).cache()
```

In [ ]:
```
# Select anomalies with temperature > 100°C
# I used two filter in order to cache the output of the first filter that is
# used by the second part of the code (Part B).
# Example: SID13,2018/03/01_15:40,110
anomalies2010_2018RDDMoreThan100 = anomalies2010_2018RDD\
.filter(lambda line: int(line.split(",")[2]) > 100)
```

In [ ]:
```
# Map each input line to a pair
# key = (SID, Year)
# value = 1
# Example input data: SID13,2018/03/01_15:40,110

def extractPairSIDYearOne(line):
    fields = line.split(",")
    SID = fields[0]
    year = fields[1].split("/")[0]
    return ( (SID, year), 1)


SIDYearAnomaly = anomalies2010_2018RDDMoreThan100.map(extractPairSIDYearOne)
```

In [ ]:
```
# Count the number of temperatures > 100°C for each
# combination (SID, Year) (i.e., for each sid in each year between 2010-2018)
SIDYearNumAnomalies = SIDYearAnomaly.reduceByKey(lambda v1, v2: v1+v2)
```

In [ ]:
```
# Select the pairs with number of anomalies > 50
SIDYearNumAnomaliesMoreThan50 = SIDYearNumAnomalies.filter(lambda pair: pair[1]> 50)
```

In [ ]:
```
# The SIDs of the selected pairs are the SIDs of the servers with at least one
# year in which those servers had more than 50 anomalies with temperature > 100°C
# in years 2010-2018
# Extract the SID part of the key
selectedSIDs = SIDYearNumAnomaliesMoreThan50.map(lambda pair: pair[0][0])
```

In [ ]:
```
# The same server might had more than 50 anomalies in two different years
# of years 2010-2018
# Hence, duplicated must be removed before storing the result in the first output fo
selectedSIDs.distinct()\
.saveAsTextFile(outputPathPartA)
```

In [ ]:
```

```

In [ ]:
```
# ****************************************
# Exercise 2 - Part B
# ****************************************
```

In [ ]:
```
# Compute the number of anomalies for each server in the period 2010-2018
```

In [ ]:
```
# Map each input line to a pair
# key = SID
# value = 1

def extractPairSIDOne(line):
    # Example input data: SID13,2018/03/01_15:40,110
    fields = line.split(",")
    SID = fields[0]
    return (SID, 1)

SIDAnomaly = anomalies2010_2018RDD.map(extractPairSIDOne)
```

In [ ]:
```
# Compute the number of anomalies for each server in the period 2010-2018
SIDNumAnomaliesRDD = SIDAnomaly.reduceByKey(lambda v1, v2: v1+v2)
```

In [ ]:
```
# Select the SIDs of the servers with more than 10 anomalies
SIDNumAnomaliesMoreThan10 = SIDNumAnomaliesRDD.filter(lambda pair: pair[1]>10)
#SIDNumAnomaliesMoreThan10 = SIDNumAnomaliesRDD.filter(lambda pair: pair[1]>2)
```

In [ ]:
```
# Retrieve the data centers associated with the selected SIDs
```

In [ ]:
```
# Read the content of servers.txt
serversRDD = sc.textFile(inputServers).cache()
```

In [ ]:
```
# Map each input line to a pair
# Key: SID
```

```
# Value: Data center ID
def extractPairSIDDCID(line):
    # Example input record: SID13,PentiumV,DC10,Barcelona,Spain
    fields = line.split(",")
    SID = fields[0]
    DataCenterID = fields[2]
    return (SID, DataCenterID)


SIDDataCenters = serversRDD.map(extractPairSIDDCID)
```

In [ ]:
```
# Join SIDNumAnomaliesMoreThan10 with SIDDataCenters
# The result is:
# key: SID
# value: numAnomalies, Data Center ID
SIDNumAnomaliesDataCenter = SIDNumAnomaliesMoreThan10.join(SIDDataCenters)
```

In [ ]:
```
# Select the data centers for which there is at least one server with more than
# 10 anomalies.
# Extract Data Center ID from SIDNumAnomaliesDataCenter.
# These are the data centers for which there is at least one server with more than
# 10 anomalies.
dataCentersToBeRemoved = SIDNumAnomaliesDataCenter.map(lambda pair: pair[1][1])
```

In [ ]:
```
# Select all the data centers managed by PoliData (without duplicates)
def extractDCID(line):
    #Example input data: SID13,PentiumV,DC10,Barcelona,Spain
    fields = line.split(",")
    dataCenter = fields[2]
    return dataCenter

allDatacenters = serversRDD.map(extractDCID).distinct()
```

In [ ]:
```
# Select the data centers with all the servers with at most 10 anomalies
# (i.e., those data centers that are never associated with a server with more than 1
selectedDataCenters = allDatacenters.subtract(dataCentersToBeRemoved).cache()
```

In [ ]:
```
# Save the selected data centers
selectedDataCenters.saveAsTextFile(outputPathPartB)
```

In [ ]:
```
# Print on the standard output the number of select data centers
print(selectedDataCenters.count())
```

In [ ]:
```

```

In [20]:
```
# Exam example #4
# Exercise #2
```

In [21]:
```
inputPathPrices = "exam_ex2_data/Stocks_Prices.txt"
outputPathPartA = "out_PartAv2"
outputPathPartB = "out_PartBv2"
```

In [22]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Read the content of stocks_prices.txt
stockPrices = sc.textFile(inputPathPrices)
```

In [23]:
```
# Select only year 2017 data
stockPrices2017 = stockPrices\
.filter(lambda line: line.split(",")[1].startswith("2017"))
```

In [24]:
```
# Create for each input line (of year 2017) a pair with
# key = (stockId,Date)
# value = (price,price)

def extractStockIdDatePrice(line):
    fields = line.split(",")
    stockId = fields[0]
    date = fields[1]
    price = float(fields[3])

    return ( (stockId, date), (price, price) )

stockDate_Price = stockPrices2017.map(extractStockIdDatePrice)
```

In [25]:
```
# Compute max price and min price for each (stockId, date)
stockDate_MinandMaxPrice = stockDate_Price\
.reduceByKey(lambda v1, v2: (max(v1[0],v2[0]), min(v1[1],v2[1])) )
```

In [26]:
```
# Compute daily variation for each (stockId, date)
stockDate_DailyVariation = stockDate_MinandMaxPrice\
.mapValues(lambda value: value[0]-value[1])\
.cache()
```

In [27]:
```
stockDate_DailyVariation.collect()
```

Out[27]:
```
[(('TSLA', '2017/05/21'), 50.0),
 (('GOOG', '2017/05/21'), 50.0),
 (('GOOG', '2017/06/21'), 50.0),
 (('FCA', '2017/06/21'), 0.010000000000000009),
 (('GOOG', '2017/05/22'), 2.0),
 (('GOOG', '2017/05/23'), 2.039999999999999)]
```

In [28]:
```
# Select only the elements with a daily variation > 10
stockDate_DailyVariationGreater10 = stockDate_DailyVariation\
```

```
.filter(lambda pair: pair[1]>10)
```

In [29]:
```
# Count the number of dates with daily variation > 10 for each stockId

# Map each input element to a pair (stockId, +1)
stockIdOne = stockDate_DailyVariationGreater10\
.map(lambda pair: (pair[0][0], 1) )
```

In [30]:
```
# ReduceByKey to count the number of dates for each stockId
stockIdNumDates = stockIdOne.reduceByKey(lambda v1, v2: v1 + v2)
```

In [19]:
```
#stockIdNumDates.collect()
```

In [ ]:
```
# Save result
stockIdNumDates.saveAsTextFile(outputPathPartA)
```

In [ ]:
```

```

In [ ]:
```
# ****************************************
# Exercise 2 - Part B
# ****************************************
```

In [31]:
```
# This funciton is already provided
from datetime import datetime, timedelta

def previousDate(mydate):
    currentDate = datetime.strptime(mydate, "%Y/%m/%d")
    prevDate=currentDate - timedelta(days=1)

    return prevDate.strftime("%Y/%m/%d")
```

In [32]:
```
# Consider stockDate_DailyVariation
# It contains for each stockid+Date the daily variation.
# Each element of stockDate_DailyVariation is the first date of a
# sequence of two dates and also the second date of another sequence of two dates.
# Emit for each input element (stockid+Date,DailyVariation) two pairs:
# 1 - key=stockId+Date - value=(DailyVariation, +1) at date Date -> This is the firs
#     element of the sequence of two dates starting at date "Date" associated with s
# 2 - key=stockId+(Date-1) - value=(DailyVariation, +1) at date Date -> This is the
#     element of the sequence of two dates starting at date "Date-1" associated with

def extractStockIdDateDailyVar(pair):
    stockId = pair[0][0]
    date = pair[0][1]
    prevDate = previousDate(date)
    dailyVariation = pair[1]

    returnedPairs = []
    # Append the pair key=stockId+Date - value=Date+DailyVariation
    returnedPairs.append( ((stockId, date) , (dailyVariation,+1)) )

    # Append the pair key=stockId+(Date-1) - value=Date+DailyVariation
    returnedPairs.append( ((stockId, prevDate) , (dailyVariation,+1)) )
```

```
    return returnedPairs


stockIdSequenceFirstDate_DailyVariations = stockDate_DailyVariation\
.flatMap(extractStockIdDateDailyVar)
```

In [33]:
```
stockIdSequenceFirstDate_DailyVariations.collect()
```

Out[33]:
```
[(('TSLA', '2017/05/21'), (50.0, 1)),
 (('TSLA', '2017/05/20'), (50.0, 1)),
 (('GOOG', '2017/05/21'), (50.0, 1)),
 (('GOOG', '2017/05/20'), (50.0, 1)),
 (('GOOG', '2017/06/21'), (50.0, 1)),
 (('GOOG', '2017/06/20'), (50.0, 1)),
 (('FCA', '2017/06/21'), (0.010000000000000009, 1)),
 (('FCA', '2017/06/20'), (0.010000000000000009, 1)),
 (('GOOG', '2017/05/22'), (2.0, 1)),
 (('GOOG', '2017/05/21'), (2.0, 1)),
 (('GOOG', '2017/05/23'), (2.039999999999999, 1)),
 (('GOOG', '2017/05/22'), (2.039999999999999, 1))]
```

In [34]:
```
# Apply reduceByKey to compute absolute difference between (two) values
# and number of values for each window
stockIdTwoConsecutiveDatesDailyVariations = stockIdSequenceFirstDate_DailyVariations
.reduceByKey(lambda v1,v2: ( abs(v1[0]-v2[0]) , v1[1]+v2[1] ) )
```

In [35]:
```
stockIdTwoConsecutiveDatesDailyVariations.collect()
```

Out[35]:
```
[(('TSLA', '2017/05/21'), (50.0, 1)),
 (('TSLA', '2017/05/20'), (50.0, 1)),
 (('GOOG', '2017/05/21'), (48.0, 2)),
 (('GOOG', '2017/05/20'), (50.0, 1)),
 (('GOOG', '2017/06/21'), (50.0, 1)),
 (('FCA', '2017/06/21'), (0.010000000000000009, 1)),
 (('GOOG', '2017/06/20'), (50.0, 1)),
 (('FCA', '2017/06/20'), (0.010000000000000009, 1)),
 (('GOOG', '2017/05/22'), (0.03999999999999915, 2)),
 (('GOOG', '2017/05/23'), (2.039999999999999, 1))]
```

In [36]:
```
# Select only the "stable trends"
def stableTrend(pair):
    # Select the two daily variations and compute the absolute
    # difference between those two values
    listDailyVariations=list(pair[1])

    #Check if there are at two daily variation values
    if (len(listDailyVariations)==2):
        dailyVariation1 = listDailyVariations[0]
        dailyVariation2 = listDailyVariations[1]

        # Check if their absolute difference betwen the two variations is at most 0.
        if abs(dailyVariation1-dailyVariation2) <= 0.1:
            return True
        else:
            return False

    else:
        return False
```

```
stockIDStableTrends = stockIdTwoConsecutiveDatesDailyVariations\
.filter(lambda value: value[1][0]<=0.1 and value[1][1]==2)
```

In [ ]:

```
# Select the keys and store them
stockIDStableTrends.keys()\
.saveAsTextFile(outputPathPartB)
```

In [37]:

```
stockIDStableTrends.keys().collect()
```

Out[37]:

```
[('GOOG', '2017/05/22')]
```

In [ ]:

In [ ]:
```
# Exam example #5
# Exercise #2 - SQL
from pyspark.sql.types import LongType
```

In [ ]:
```
inputServers = "exam_ex2_data/Servers.txt"
inputAnomalies = "exam_ex2_data/Servers_TemperatureAnomalies.txt"
outputPathPartA = "out_PartASQL"
outputPathPartB = "out_PartBSQL"
```

In [ ]:
```
# ****************************************
# Exercise 2 - Part A
# ****************************************

# Read the content of Servers_TemperatureAnomalies.txt
anomaliesDF = spark.read.load(inputAnomalies,\
                              format="csv",\
                              header=False,\
                              inferSchema=True)\
.withColumnRenamed("_c0", "SID")\
.withColumnRenamed("_c1", "Timestamp")\
.withColumnRenamed("_c2", "AnomalousTemperatureValue")
```

In [ ]:
```
#anomaliesDF.printSchema()
#anomaliesDF.show()
```

In [ ]:
```
# Define a UDF that returns year from timestamp
def yearFunc(timestamp):
    return(int(timestamp.split("/")[0]))

spark.udf.register("yearFunc", yearFunc, LongType())
```

In [ ]:
```
# Select anomalies associated with years 2010-2018
anomalies2010_2018DF = anomaliesDF\
.filter("yearFunc(Timestamp)>=2010 and yearFunc(Timestamp)<=2018").cache()
```

In [ ]:
```
# Associate anomalies2010_2018DF to a temporary table
anomalies2010_2018DF.createOrReplaceTempView("anomalies1018Table")
```

In [ ]:
```
# The SIDs of the selected pairs are the SIDs of the servers with at least one
# year in which those servers had more than 50 anomalies with temperature > 100°C
# in years 2010-2018
# Extract the SID part of the key
selectedSIDsDF = spark.sql("""SELECT DISTINCT SID
FROM anomalies1018Table
WHERE AnomalousTemperatureValue>100
GROUP BY SID,yearFunc(Timestamp)
HAVING COUNT(*)>50""")
```

In [ ]:
```
#selectedSIDsDF.show()
```

In [ ]:
```
# Save result
selectedSIDsDF.write.csv(outputPathPartA,header=False)
```

In [ ]:
```

```

In [ ]:
```
# ****************************************
# Exercise 2 - Part B
# ****************************************
```

In [ ]:
```
# Read the content of servers.txt
serversDF = spark.read.load(inputServers,\
                            format="csv",\
                            header=False,\
                            inferSchema=True)\
.withColumnRenamed("_c0", "SID")\
.withColumnRenamed("_c1", "CPUVersion")\
.withColumnRenamed("_c2", "DataCenterID")\
.withColumnRenamed("_c3", "City")\
.withColumnRenamed("_c4", "Country")\
.cache()
```

In [ ]:
```
# Associate anomalies2010_2018DF to a temporary table
serversDF.createOrReplaceTempView("serversTable")
```

In [ ]:
```
# Compute the number of anomalies for each server in the period 2010-2018
# Identify the servers with more than 10 anomalies .
# Select the data centers for which there is at least one server with more than
# 10 anomalies.
dataCentersToBeRemovedDF = spark.sql("""SELECT DataCenterID
FROM anomalies1018Table, serversTable
WHERE anomalies1018Table.SID=serversTable.SID
GROUP BY anomalies1018Table.SID, DataCenterID
HAVING COUNT(*)>10""")
```

In [ ]:
```
# Select the data centers with all the servers with at most 10 anomalies
# (i.e., those data centers that are never associated with a
# server with more than 10 anomalies)
selectedDataCentersDF = serversDF\
.join(dataCentersToBeRemovedDF,\
      serversDF.DataCenterID==dataCentersToBeRemovedDF.DataCenterID,
      'left_anti')\
.select("DataCenterID")\
.distinct().cache()
```

In [ ]:
```
##Alternative approach

## Associate dataCentersToBeRemovedDF to a temporary table
#dataCentersToBeRemovedDF.createOrReplaceTempView("toberemovedTable")
#
#selectedDataCentersDF = spark.sql("""SELECT DISTINCT DataCenterID
#FROM serversTable LEFT ANTI JOIN toberemovedTable
#ON serversTable.DataCenterID==toberemovedTable.DataCenterID""").cache()
```

In [ ]:
```

```

```
#selectedDataCentersDF.show()
```

In [ ]:

```
# Save the selected data centers
selectedDataCentersDF.write.csv(outputPathPartB,header=False)
```

In [ ]:

```
# Print on the standard output the number of select data centers
print(selectedDataCentersDF.count())
```

In [ ]:

```

```

Distributed architectures for big data processing and analytics

Answer to the following questions. There is only one right answer for each question.

1. (2 points) Consider the following Spark Streaming applications.

(Application A)

```
from pyspark.streaming import StreamingContext

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

# Create a (Receiver) DStream that will connect to localhost:9999
# Define windows
inputWindowDStream = ssc.socketTextStream("localhost", 9999)\
.window(20, 10)\
.map(lambda value: int(value))

# Sum values
sumWindowDStream = inputWindowDStream.reduce(lambda v1,v2: v1+v2)

#Apply a filter
resDStream = sumWindowDStream.filter(lambda value : value>5)

# Print the result
resDStream.pprint()

ssc.start()
ssc.awaitTerminationOrTimeout(360)
ssc.stop(stopSparkContext=False)
```

(Application B)

```
from pyspark.streaming import StreamingContext

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

# Create a (Receiver) DStream that will connect to localhost:9999
inputDStream = ssc.socketTextStream("localhost", 9999)\
.map(lambda value: int(value))

# Sum values
sumDStream = inputDStream.reduce(lambda v1,v2: v1+v2)

# Define windows
sumWindowDStream = sumDStream.window(20, 10)

#Apply a filter
resDStream = sumWindowDStream.filter(lambda value : value>5)

# Print the result
resDStream.pprint()
```

```
ssc.start()
ssc.awaitTerminationOrTimeout(360)
ssc.stop(stopSparkContext=False)
```

(Application C)

```
from pyspark.streaming import StreamingContext

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

# Create a (Receiver) DStream that will connect to localhost:9999
inputDStream = ssc.socketTextStream("localhost", 9999)\
.map(lambda value: int(value))

# Define windows
inputWindowDStream = inputDStream.window(20, 10)

# Sum values
sumWindowDStream = inputWindowDStream.reduce(lambda v1,v2: v1+v2)

#Apply a filter
resDStream = sumWindowDStream.filter(lambda value : value>5)

# Print the result
resDStream.pprint()

ssc.start()
ssc.awaitTerminationOrTimeout(360)
ssc.stop(stopSparkContext=False)
```

Which one of the following statements is true?

a) Applications A, B, And C are equivalent in terms of returned result, i.e., given the same input they return the same result.

b) Applications A and B are equivalent in terms of returned result, i.e., given the same input they return the same result, while C is not equivalent to the other two applications.

c) Applications A and C are equivalent in terms of returned result, i.e., given the same input they return the same result, while B is not equivalent to the other two applications.

d) Applications B and C are equivalent in terms of returned result, i.e., given the same input they return the same result, while A is not equivalent to the other two applications.

2. (2 points) Consider the following Spark Streaming applications.

(Application A)

```
from pyspark.streaming import StreamingContext

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

# Create a (Receiver) DStream that will connect to localhost:9999
# Define windows
inputWindowDStream = ssc.socketTextStream("localhost", 9999)\
.window(20, 10)\
.map(lambda value: int(value))

# Sum values
sumWindowDStream = inputWindowDStream.reduce(lambda v1,v2: v1+v2)

#Apply a filter
resDStream = sumWindowDStream.filter(lambda value : value>5)

# Print the result
resDStream.pprint()

ssc.start()
ssc.awaitTerminationOrTimeout(360)
ssc.stop(stopSparkContext=False)
```

(Application B)

```
from pyspark.streaming import StreamingContext

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

# Create a (Receiver) DStream that will connect to localhost:9999
inputDStream = ssc.socketTextStream("localhost", 9999)\
.map(lambda value: int(value))

# Sum values
sumDStream = inputDStream.reduce(lambda v1,v2: v1+v2)

# Define windows
sumWindowDStream = sumDStream.window(20, 10)

#Apply a filter
resDStream = sumWindowDStream.filter(lambda value : value>5)

# Print the result
resDStream.pprint()

ssc.start()
ssc.awaitTerminationOrTimeout(360)
ssc.stop(stopSparkContext=False)
```

(Application C)

```
from pyspark.streaming import StreamingContext

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

# Create a (Receiver) DStream that will connect to localhost:9999
inputDStream = ssc.socketTextStream("localhost", 9999)\
.map(lambda value: int(value))

# Sum values
sumDStream = inputDStream.reduce(lambda v1,v2: v1+v2)

#Apply a filter
sumFilterDStream = sumDStream.filter(lambda value : value>5)

# Define windows
resDStream = sumFilterDStream.window(20, 10)

# Print the result
resDStream.pprint()

ssc.start()
ssc.awaitTerminationOrTimeout(360)
ssc.stop(stopSparkContext=False)
```

Which one of the following statements is true?

a) Applications A, B, And C are equivalent in terms of returned result, i.e., given the same input they return the same result.

b) Applications A and B are equivalent in terms of returned result, i.e., given the same input they return the same result, while C is not equivalent to the other two applications.

c) Applications A and C are equivalent in terms of returned result, i.e., given the same input they return the same result, while B is not equivalent to the other two applications.

d) Applications B and C are equivalent in terms of returned result, i.e., given the same input they return the same result, while A is not equivalent to the other two applications.

3. Which one of the following sentences about SparkStreaming is True?

a) Micro-batch execution offers lower latency than continuous execution

b) Structured Streaming API works with DStream objects

c) Two StreamingContext can be run in parallel on the same application

d) SparkStreaming provide Checkpoints to recover from failures

4. (2 points) Consider the following piece of Spark application:

```
myrdd=sc.parallelize(['I','am','an','example'])
myacc=sc.accumulator(0)

def strFun(line):
    l=len(line)
    if l<3:
        myacc.add(1)
    return l

v=myrdd.map(lambda l:strFun(l)).reduce(lambda l1,l2:l1+l2)

print(myacc.value,v)
```

What will be the output of the application?

a) 3 4
b) 0 'example'
c) 0 'Iamanexample'
d) 3 12

5. (2 points) Which one of the following statement about the preprocessing of data in Mllib is true?

a) A "vectorAssembler" estimator always merges all the columns of a Dataframe into a single vector

b) A "vectorAssembler" estimator can be applied to columns of type double

c) "indexToString" is useful to apply in preprocessing before training the a prediction model

d) "standardScaler" can be applied to strings

6. (2 points) Consider the following piece of Spark application.

```
from graphframes import GraphFrame

v = spark.read.load(inputPath1, format='csv',inferSchema=True).toDF("id","attribute")

e = spark.read.load(inputPath2, format='csv',inferSchema=True).toDF("src","dst","attribute")
```

```
g = GraphFrame(v, e)

n=g.vertices.count()

e2 = e.filter("attribute = 'typeA'")

g2 = GraphFrame(g.vertices, e2).dropIsolatedVertices()
```

Which one of the following statements is **false**?

a) The variable “n” is an integer number

b) “g.vertices” is a dataframe

c) “g2” is a subgraph of “g”

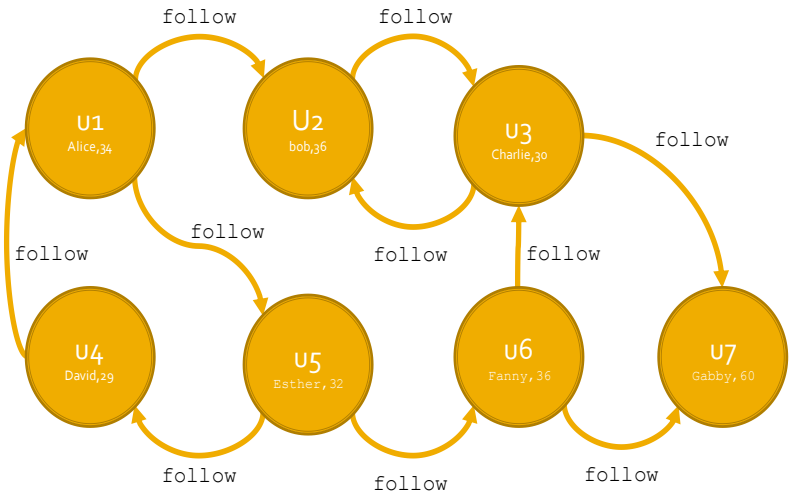
d) “e2” is a graphframe

Question 1: (c)
Question 2: (d)
Question 3: (d)
Question 4: (d)
Question 5: (b)
Question 6: (d)

Distributed architectures for big data processing and analytics

Answer to the following questions. There is only one right answer for each question.

1. (2 points) Consider the following graph and suppose g is its instantiation in GraphFrame.



Suppose the following command is executed on g.

motifs = g.find("(v1)-[]->(v2); (v2)-[]->(v3); !(v3)-[]->(v1)")

Which one of the following statements is **false**?

a) One of the rows stored into the Dataframe motifs is

v1	v2	v3
[u3, Charlie, 30]	[u2, Bob, 36]	[u3, Charlie, 30]

b) One of the rows stored into the Dataframe motifs is

v1	v2	v3
[u6, Fanny, 36]	[u3, Charlie, 30]	[u7, Gabby, 60]

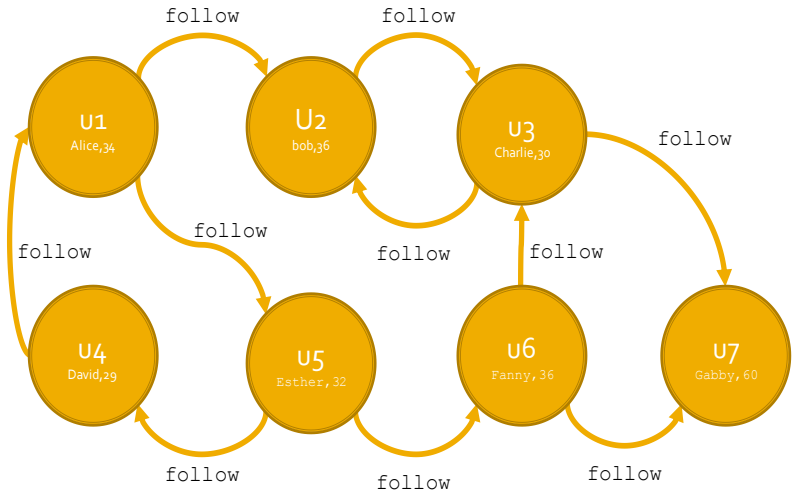
c) One of the rows stored into the Dataframe motifs is

v1	v2	v3
[u2, Bob, 36]	[u3, Charlie, 30]	[u2, Bob, 36]

d) One of the rows stored into the Dataframe motifs is

v1	v2	v3
[u5, Esther, 32]	[u4, David, 29]	[u1, Alice, 34]

2. (2 points) Consider the following graph and suppose g is its instantiation in GraphFrame.



Suppose the following command is executed on g.

motifs = g.find("(v1)-[]->(v2); (v2)-[]->(v3); !(v2)-[]->(v1)")

Which one of the following statements is **false**?

a) One of the rows stored into the Dataframe motifs is

v1	v2	v3
[u6, Fanny, 36]	[u3, Charlie, 30]	[u7, Gabby, 60]

b) One of the rows stored into the Dataframe motifs is

v1	v2	v3
[u4, David, 29]	[u1, Alice, 34]	[u5, Esther, 32]

c) One of the rows stored into the Dataframe motifs is

v1	v2	v3
[u2, Bob, 36]	[u3, Charlie, 30]	[u7, Gabby, 60]

d) One of the rows stored into the Dataframe motifs is

v1	v2	v3
[u5, Esther, 32]	[u4, David, 29]	[u1, Alice, 34]

Question 1: (d)
Question 2: (c)