# Analyzing the Bible and the Quran using Spark

Most of the data out there is unstructured, and Spark is an excellent tool for analyzing this type of data. Here, we will analyze the Bible and the Quran. We will see the distribution of words, the most common words in both scriptures and the average frequency. This could also be scaled to find the most common words and distribution of all words on the Internet. The books have been retrieved from Project Gutenberg. The Bible can be downloaded from here (http://www.gutenberg.org/files/10/10.txt) and the Quran from here (http://www.gutenberg.org/files/2800/2800.txt).

#### Import pyspark and initialize Spark

**pyspark** is the Spark Python API that exposes the Spark programming model to Python. **SparkContext** is main entry point for Spark functionality. In Spark, communication occurs between a driver and executors. The driver has Spark jobs that it needs to run and these jobs are split into tasks that are submitted to the executors for completion. The results from these tasks are delivered back to the driver. In order to use Spark and its API we will need to use a SparkContext. When running Spark, you start a new Spark application by creating a **SparkContext**. When the SparkContext is created, it asks the master for some cores to use to do work. The master sets these cores aside just for you; they won't be used for other applications. In the code below, we are also specifying some configuration parameters, including the fact that this Spark session is to use local machine since I am using my PC for this tutorial.

```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName("miniProject").setMaster("local[*]")
sc = SparkContext.getOrCreate(conf)
```

#### *Create Resilient Distributed Datasets (RDDs)*

A Spark context can be used to create Resilient Distributed Datasets (RDDs) on a cluster.

To convert a text file into an RDD, we use the SparkContext.textFile() method.

```
bibleRDD = sc.textFile("bible.txt")
quranRDD = sc.textFile("quran.txt")
```

#### Display sample data

**collect** return a list that contains all of the elements in the RDD.

```
bibleRDD.sample(withReplacement = False, fraction = 0.0002, seed = 80).collect()
```

```
['will not do it, if I find thirty there.',
 'work in the tabernacle of the congregation: 4:36 And those that were',
 'Israel, which they bring unto the priest, shall be his.',
 '',
 'under mine hand, but there is hallowed bread; if the young men have',
 '',
 '',
 'the king, All that thou didst send for to thy servant at the first I',
 '5:7 And it came to pass, when the king of Israel had read the letter,',
 'his clothes, and covered himself with sackcloth, and went into the',
 'flowers, and the lamps, and the tongs, made he of gold, and that',
 '',
 'which they commit here? for they have filled the land with violence,',
 'the house of Israel, to do it for them; I will increase them with men',
 'whole limit thereof round about shall be most holy. Behold, this is',
 '',
 'Father which is in heaven.',
 '21:31 Whether of them twain did the will of his father? They say unto',
 'which were early at the sepulchre; 24:23 And when they found not his',
 'peace.',
 'red dragon, having seven heads and ten horns, and seven crowns upon',
 'every man according to their works.']
```

```
quranRDD.sample(withReplacement = False, fraction = 0.0002, seed = 80).collect()
```

```
['Many of them suffered torture for their faith in him, and two of them died
as',
 'when ye halt; and from their wool and soft fur and hair, hath He supplied y
ou',
 'meet for their best deeds.',
 '',
 '']
```

Let's count the number of lines in each RDD.

```
print('The number of lines in the Bible text file is {}'.format(bibleRDD.count()))
```

```
The number of lines in the Bible text file is 100223
```

```
print('The number of lines in the Quran text file is {}'.format(quranRDD.count()))
```

```
The number of lines in the Quran text file is 27321
```

Words should be counted independent of their capitialization. So, we will change all words to lower case. We will also remove all punctuations. Further, any leading or trailing spaces on a line should be removed.

The function below removes all characters which are not alpha-numeric except space(s). It also changes them to lower letter and removes leading or trailing spaces. As you can see we are using the python module **re**.

```
import re

def wordclean(x):
    return re.sub("[^a-zA-Z0-9\s]+","", x).lower().strip()
```

Let's check the above function:

```
x = [" The Sun rises in the East and sets in the West!\n "
     "  He said, 'I am sure you know the answer!\n'  "]

for i in x:
  print(wordclean(i))
```

```
the sun rises in the east and sets in the west
   he said i am sure you know the answer
```

Now, can apply it to our Bible and Quran RDDS. We use the **map** (https://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=map#pyspark.RDD.map) RDD method.

```
bibleRDDList = bibleRDD.map(lambda x : wordclean(x))
quranRDDList = quranRDD.map(lambda x : wordclean(x))
```

Now, let's see how the RDDList files above look like. As shown below, all punctuation have been removed and all letters are lower-case.

```
bibleRDDList.take(60)[41: ]
```

```
['11 in the beginning god created the heavens and the earth',
 '',
 '12 and the earth was without form and void and darkness was upon',
 'the face of the deep and the spirit of god moved upon the face of the',
 'waters',
 '',
 '13 and god said let there be light and there was light',
 '',
 '14 and god saw the light that it was good and god divided the light',
 'from the darkness',
 '',
 '15 and god called the light day and the darkness he called night',
 'and the evening and the morning were the first day',
 '',
 '16 and god said let there be a firmament in the midst of the waters',
 'and let it divide the waters from the waters',
 '',
 '17 and god made the firmament and divided the waters which were',
 'under the firmament from the waters which were above the firmament']
```

```
quranRDDList.take(450)[414 : ]
```

```
['mohammed was born at mecca in ad 567 or 569  his flight hijra to medina',
 'which marks the beginning of the mohammedan era took place on 16th june 62
2',
 'he died on 7th june 632',
 '',
 '',
 '',
 'introduction',
 '',
 'the koran admittedly occupies an important position among the great religio
us',
 'books of the world though the youngest of the epochmaking works belonging',
 'to this class of literature it yields to hardly any in the wonderful effec
t',
 'which it has produced on large masses of men it has created an all but ne
w',
 'phase of human thought and a fresh type of character it first transformed
a',
 'number of heterogeneous desert tribes of the arabian peninsula into a natio
n',
 'of heroes and then proceeded to create the vast politicoreligious',
 'organisations of the muhammedan world which are one of the great forces wit
h',
 'which europe and the east have to reckon today',
 '',
 'the secret of the power exercised by the book of course lay in the mind',
 'which produced it it was in fact at first not a book but a strong living',
 'voice a kind of wild authoritative proclamation a series of admonitions',
 'promises threats and instructions addressed to turbulent and largely',
 'hostile assemblies of untutored arabs as a book it was published after th
e',
 'prophets death in muhammeds lifetime there were only disjointed notes',
 'speeches and the retentive memories of those who listened to them to spea
k',
 'of the koran is therefore practically the same as speaking of muhammed an
d',
 'in trying to appraise the religious value of the book one is at the same ti
me',
 'attempting to form an opinion of the prophet himself it would indeed be',
 'difficult to find another case in which there is such a complete identity',
 'between the literary work and the mind of the man who produced it',
 '',
 'that widely different estimates have been formed of muhammed is wellknown',
 'to moslems he is of course the prophet par excellence and the koran is',
 'regarded by the orthodox as nothing less than the eternal utterance of alla
h',
 'the eulogy pronounced by carlyle on muhammed in heroes and hero worship wil
l',
 'probably be endorsed by not a few at the present day the extreme contrary']
```

Apply a transformation that will split each element of the RDD by its spaces. For each element of the RDD, we are appling Python's string split() function. Note, we are using the [flatMap (https://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=flatmap#pyspark.RDD.flatMap)](https://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=flatmap#pyspark.RDD.flatMap) here.

```
bibleRDDwords = bibleRDDList.flatMap( lambda x: x.split(" "))
quranRDDwords = quranRDDList.flatMap( lambda x: x.split(" "))
```

Let's show sample words from each RDD.

```
bibleRDDwords.sample(withReplacement = False, fraction = 0.00001, seed = 90).collect()
```

```
['isaac', 'even', 'amon', 'a', 'there', 'his', 'by', 'but']
```

```
quranRDDwords.sample(withReplacement = False, fraction = 0.00005, seed = 90).collect()
```

```
['by', 'who', 'we', 'see', 'righteous', 'loveth', 'and', 'after', 'what', 'b
e']
```

Now, let's remove spaces. We use the **filter** (https://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=filter#pyspark.RDD.filter) method to achieve this.

```
bibleRDDwords = bibleRDDwords.filter(lambda x: len(x) != 0)
quranRDDwords = quranRDDwords.filter(lambda x: len(x) != 0)
```

Next, let's create word pairs. This helps us to count the frequency of each word and to select the most common words in each RDD.

```
bibleRDDwordPairs =  bibleRDDwords.map(lambda x: (x,1))
quranRDDwordPairs =  quranRDDwords.map(lambda x: (x, 1))
```

Let's show the first ten elements of each RDD.

```
bibleRDDwordPairs.take(10)
```

```
[('the', 1),
 ('project', 1),
 ('gutenberg', 1),
 ('ebook', 1),
 ('of', 1),
 ('the', 1),
 ('king', 1),
 ('james', 1),
 ('bible', 1),
 ('this', 1)]
```

```
quranRDDwordPairs.take(10)
```

```
[('the', 1),
 ('project', 1),
 ('gutenberg', 1),
 ('etext', 1),
 ('of', 1),
 ('the', 1),
 ('koran', 1),
 ('as', 1),
 ('translated', 1),
 ('by', 1)]
```

Now, we can find the frequency of each word.

The **reduceByKey()** transformation gathers together pairs that have the same key and applies a function to two associated values at a time. reduceByKey() operates by applying the function first within each partition on a per-key basis and then across the partitions.

```
bibleRDDwordCount =  bibleRDDwordPairs.reduceByKey(lambda a, b : a + b)
quranRDDwordCount =  quranRDDwordPairs.reduceByKey(lambda a, b : a + b)
```

```
bibleRDDwordCount.take(10)
```

```
[('admired', 1),
 ('11973', 1),
 ('shedeur', 5),
 ('stirs', 1),
 ('dispossessed', 2),
 ('tochen', 1),
 ('4833', 2),
 ('peor', 4),
 ('unblameable', 2),
 ('divers', 37)]
```

```
quranRDDwordCount.take(10)
```

```
[('carious', 1),
 ('heedful', 1),
 ('lxiii1the', 1),
 ('combats', 1),
 ('denunciations', 2),
 ('calamitous', 1),
 ('divers', 3),
 ('afford', 2),
 ('weeks', 1),
 ('tents', 3)]
```

The **takeOrdered()** action returns the first n elements of the RDD, using either their natural order or a custom comparator. The key advantage of using takeOrdered() instead of first() or take() is that takeOrdered() returns a deterministic result, while the other two actions may return differing results, depending on the number of partions or execution environment. takeOrdered() returns the list sorted in ascending order. Note below, we are using -x[1] to make it in descending order.

```
bibleRDDwordCount.takeOrdered(10, lambda x : -x[1])
```

```
[('shall', 9840),
 ('unto', 8997),
 ('lord', 7830),
 ('thou', 5474),
 ('thy', 4600),
 ('god', 4442),
 ('said', 3999),
 ('ye', 3983),
 ('thee', 3826),
 ('upon', 2750)]
```

```
quranRDDwordCount.takeOrdered(10, lambda x : -x[1])
```

```
[('god', 3180),
 ('shall', 2331),
 ('ye', 1798),
 ('hath', 951),
 ('lord', 921),
 ('said', 894),
 ('thou', 813),
 ('say', 758),
 ('thee', 640),
 ('day', 535)]
```

Next, let's remove stop words from our RDDs. Note that all old English stop words may not be included in the list of python stop words we are using here.

```
import nltk
nltk.download("stopwords")
```

```
[nltk_data] Downloading package stopwords to /home/fish/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
True
```

```
from nltk.corpus import stopwords
```

```
stopwords = stopwords.words('english')
```

```
len(stopwords)
```

```
153
```

```
stopwords[0:10]
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your']
```

```
bibleRDDwordCount = bibleRDDwordCount.filter(lambda x : x[0] not in stopwords)
quranRDDwordCount = quranRDDwordCount.filter(lambda x : x[0] not in stopwords)
```

Now, we see the most frequent words from each RDD after removing the stop words. As shown below, God is the most frequent word in the Quran but sixth most frequent word in the Bible. In the bible, the word lord, which usually means God, is third most frequent word. Lord is fifth most common word in the Quran.

```
bibleRDDwordCount.takeOrdered(10, lambda x : -x[1])
```

```
[('shall', 9840),
 ('unto', 8997),
 ('lord', 7830),
 ('thou', 5474),
 ('thy', 4600),
 ('god', 4442),
 ('said', 3999),
 ('ye', 3983),
 ('thee', 3826),
 ('upon', 2750)]
```

```
quranRDDwordCount.takeOrdered(15, lambda x : -x[1])
```

```
[('god', 3180),
 ('shall', 2331),
 ('ye', 1798),
 ('hath', 951),
 ('lord', 921),
 ('said', 894),
 ('thou', 813),
 ('say', 758),
 ('thee', 640),
 ('day', 535),
 ('one', 517),
 ('thy', 489),
 ('verily', 475),
 ('us', 468),
 ('sura', 458)]
```

But how many unique words do we have now in each RDD?

```
unique_words_bible = bibleRDDwordCount.count()
unique_words_quran = quranRDDwordCount.count()
```

```
print(" The total number of unique words in the bible is {} while the unique number of wor
ds in the Quran is {}".\
      format(unique_words_bible, unique_words_quran ))
```

```
 The total number of unique words in the bible is 16816 while the number of u
nique words in the Quran is 12551
```

To find the average occurence of a word, let's find the total number of words and divide that by the number of unique words.

```
total_words_bible = bibleRDDwordCount.map(lambda a: a[1]).reduce(lambda a, b : a + b)

print("Total number of words in the Bible: {}".format(total_words_bible))
```

```
Total number of words in the Bible: 407745
```

```
total_words_quran = quranRDDwordCount.map(lambda a: a[1]).reduce(lambda a, b : a + b)

print("Total number of words in the Quran: {}".format(total_words_quran))
```

```
Total number of words in the Quran: 100096
```

```
Average_word_count_bible = total_words_bible/unique_words_bible

Average_word_count_quran = total_words_quran/unique_words_quran
```

```
print('Average word frequency in the Bible is  {} while the average word frequency in the
 Quran is {}'.\
      format(round(Average_word_count_bible,1), round(Average_word_count_quran,1)))
```

```
Average word frequency in the Bible is  24.2 while the average word frequency
in the Quran is 8.0
```

we can now analyze the distribution of the words using standard python libraries such as numpy, pandas and matplotlib.

```
import numpy as np
```

Below, we are changing the word frequencies in the RDDs to numpy arrays and ploting them using matplotlib.

```
bibleRDDwordCount_numeric_values = bibleRDDwordCount.map(lambda x : x[1]).collect()
quranRDDwordCount_numeric_values = quranRDDwordCount.map(lambda x : x[1]).collect()
```

We can see the first ten elements of each list as below.

```
bibleRDDwordCount_numeric_values[:10]
```

```
[1, 1, 5, 1, 2, 1, 2, 4, 2, 37]
```

```
quranRDDwordCount_numeric_values[:10]
```

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

Below, we are converting the lists to numpy arrays.

```
bibleRDDwordCount_numeric_values_np = np.array(bibleRDDwordCount_numeric_values)
quranRDDwordCount_numeric_values_np = np.array(quranRDDwordCount_numeric_values)
```

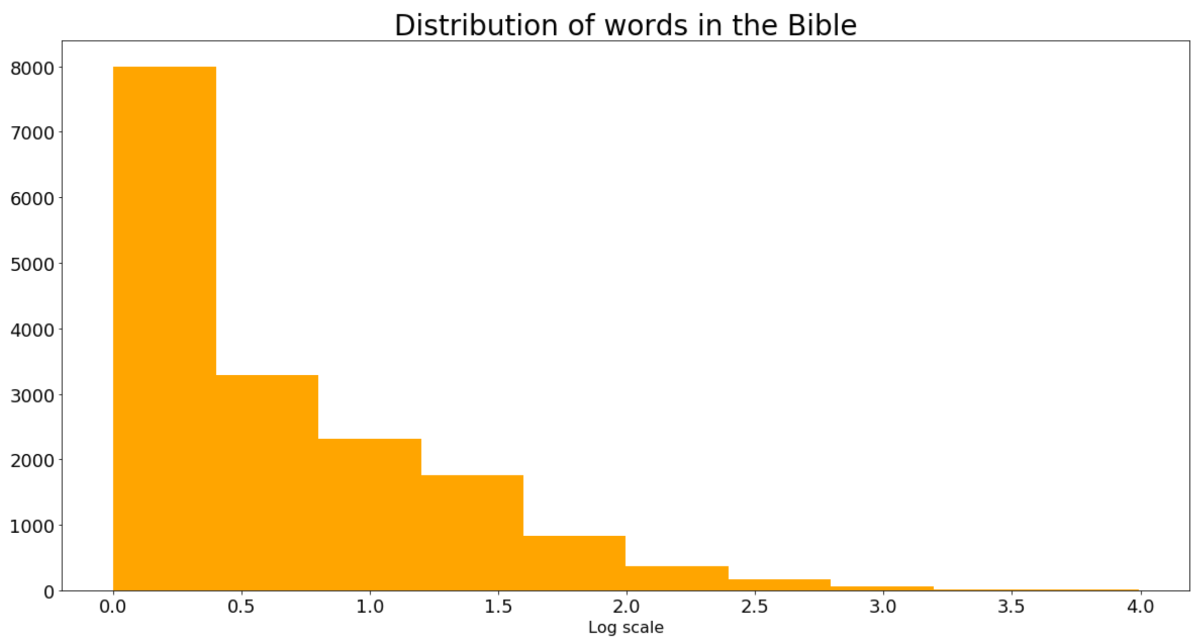
Check type of one of them:

```
type(bibleRDDwordCount_numeric_values_np)
```

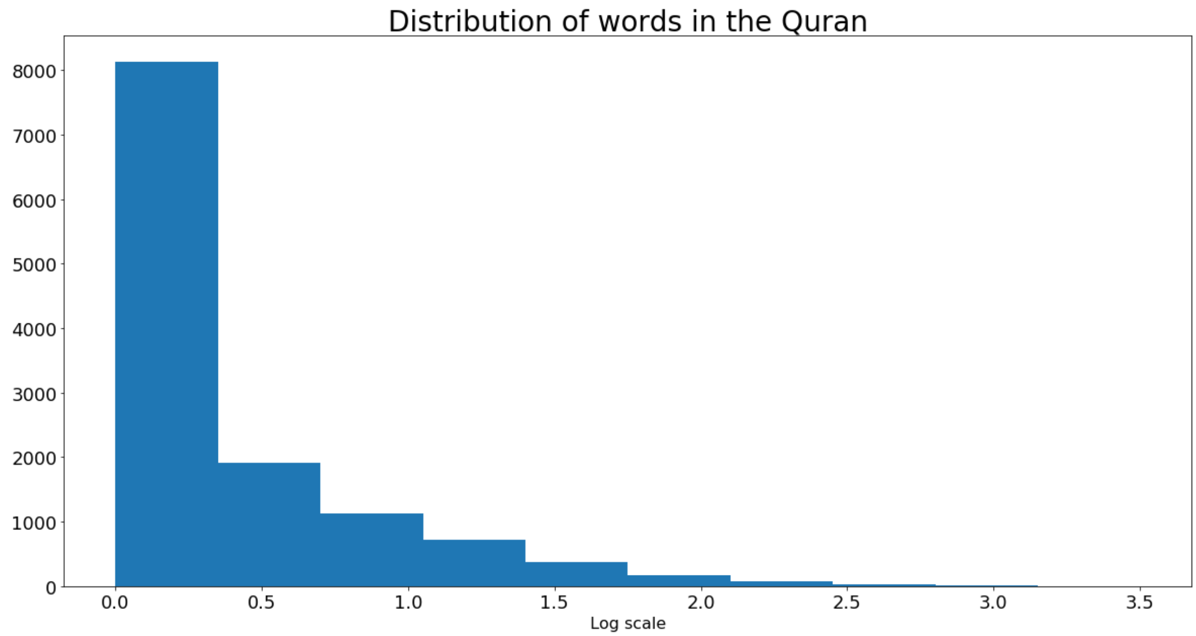
```
numpy.ndarray
```

```
import matplotlib.pyplot as plt
%matplotlib inline
```

```
plt.figure(figsize = (20, 10))
plt.hist(np.log10(bibleRDDwordCount_numeric_values_np), color = "orange")
plt.title("Distribution of words in the Bible", fontsize = 28)
plt.xlabel("Log scale", fontsize = 16)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.show()
```



```
plt.figure(figsize = (20, 10))
plt.hist(np.log10(quranRDDwordCount_numeric_values_np))
plt.title("Distribution of words in the Quran", fontsize = 28)
plt.xlabel("Log scale", fontsize = 16)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.show()
```




From the above histograms, we see that most words have frequencies less than 10.

Now, let's create a dataframe using the top 15 most common words.

```
import pandas as pd
```

```
bible_top15_words = bibleRDDwordCount.takeOrdered(15, lambda x : -x[1])
quran_top15_words = quranRDDwordCount.takeOrdered(15, lambda x : -x[1])
bible_words = [x[0] for x in bible_top15_words]
bible_count = [x[1] for x in bible_top15_words]
bible_dict = {"word": bible_words, "frequency": bible_count}

quran_words = [x[0] for x in quran_top15_words]
quran_count = [x[1] for x in quran_top15_words]
quran_dict = {"word": quran_words, "frequency": quran_count}

df_bible = pd.DataFrame(bible_dict)
df_quran = pd.DataFrame(quran_dict)
```

```
df_bible.head()
```

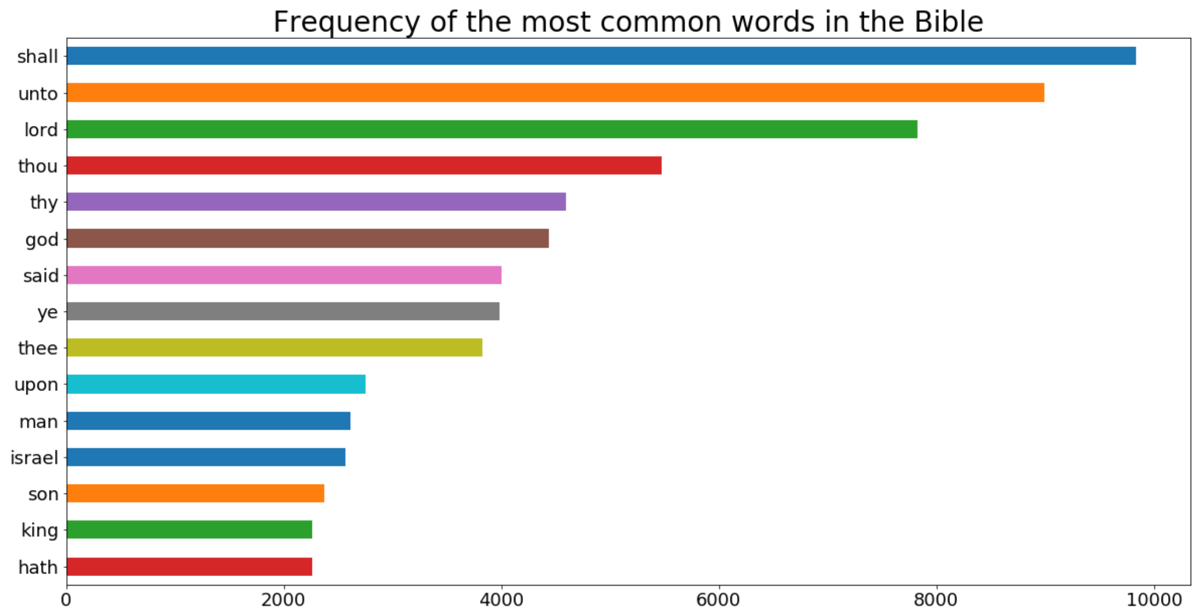
| | frequency | word |
|---|---|---|
| **0** | 9840 | shall |
| **1** | 8997 | unto |
| **2** | 7830 | lord |
| **3** | 5474 | thou |
| **4** | 4600 | thy |

```
df_quran.tail()
```

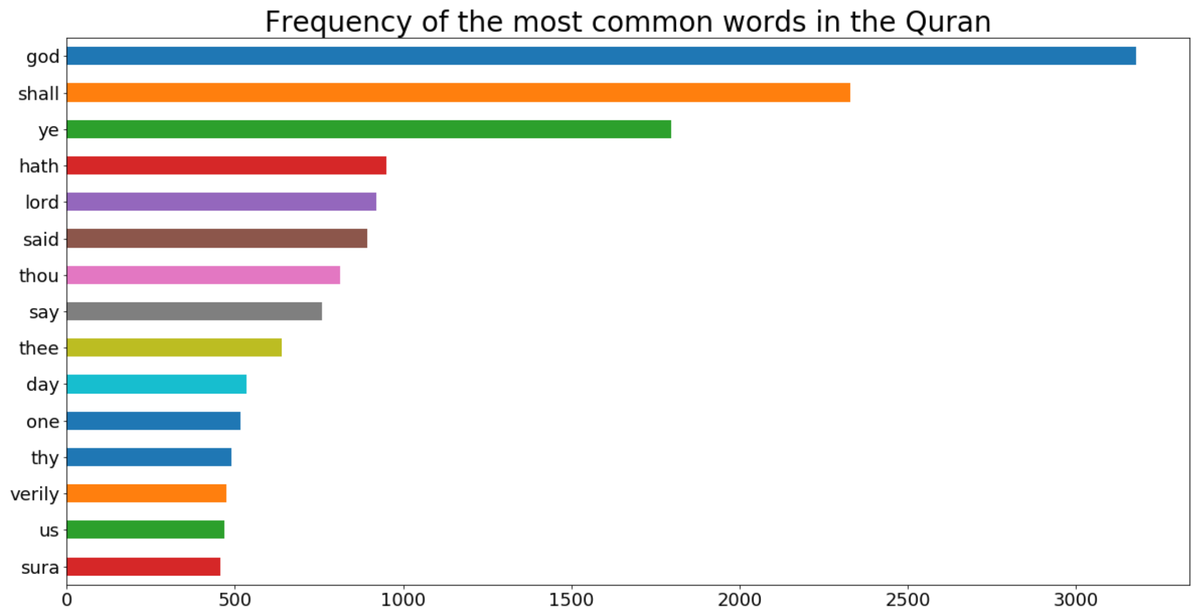
| | frequency | word |
|---|---|---|
| **10** | 517 | one |
| **11** | 489 | thy |
| **12** | 475 | verily |
| **13** | 468 | us |
| **14** | 458 | sura |

Finally, let's create a bar chart of the 15 most common words from each scripture.

```
my_plot = df_bible.plot(figsize = (20, 10),
              x = "word", y = "frequency", kind  = "barh", legend = False )

my_plot.invert_yaxis()

plt.title("Frequency of the most common words in the Bible", fontsize = 28)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.ylabel("")
plt.show()
```



```
my_plot = df_quran.plot(figsize = (20, 10),
              x = "word", y = "frequency", kind  = "barh", legend = False )
my_plot.invert_yaxis()
plt.title("Frequency of the most common words in the Quran", fontsize = 28)

plt.xticks(size = 18)
plt.yticks(size = 18)
plt.ylabel("")

plt.show()
```




## Summary

In this tutorial, we analyzed the Bible and the Quran using Spark, particularly the pyspark module. We calculated the average word frequency, the most common words and distribution of words in each scripture. God is the most frequent word in the Quran but sixth most frequent word in the Bible. In the bible, the word lord, which usually means God, is third most frequent word. Lord is fifth most common word in the Quran. We see that most words have frequencies less than 10. I plan to post various Spark tutorials and if you are interested in Spark, stay tuned.

# Spark DataFrames: Exploring Chicago Crimes

This is the second blog post on the Spark tutorial series to help big data enthusiasts prepare for Apache Spark Certification from companies such as Cloudera, Hortonworks, Databricks, etc. The first one is here (http://datascience-enthusiast.com/Python/analyzing_bible_quran_with_spark.html). If you want to learn/master Spark with Python or if you are preparing for a Spark Certification to show your skills in big data, these articles are for you.

In this tutorial, we will analyze crimes data from data.gov (https://data.cityofchicago.org/api/views/ijzp-q8t2/rows.csv?accessType=DOWNLOAD). The dataset reflects reported incidents of crime (with the exception of murders where data exists for each victim) that occurred in the City of Chicago since 2001.

A SparkSession can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files. It is the entry point to programming Spark with the DataFrame API. We can create a SparkSession, usfollowing builder pattern:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Chicago_crime_analysis").getOrCreate()
```

We can let Spark infer the schema of our csv data but proving pre-defined schema makes the reading process faster. Further,it helps us to make the colum names to have the format we want, for example, to avoid spaces in the names of the columns.

```
from pyspark.sql.types import  (StructType,
                                StructField,
                                DateType,
                                BooleanType,
                                DoubleType,
                                IntegerType,
                                StringType,
                               TimestampType)


crimes_schema = StructType([StructField("ID", StringType(), True),
                            StructField("CaseNumber", StringType(), True),
                            StructField("Date", StringType(), True ),
                            StructField("Block", StringType(), True),
                            StructField("IUCR", StringType(), True),
                            StructField("PrimaryType", StringType(), True  ),
                            StructField("Description", StringType(), True ),
                            StructField("LocationDescription", StringType(), True ),
                            StructField("Arrest", BooleanType(), True),
                            StructField("Domestic", BooleanType(), True),
                            StructField("Beat", StringType(), True),
                            StructField("District", StringType(), True),
                            StructField("Ward", StringType(), True),
                            StructField("CommunityArea", StringType(), True),
                            StructField("FBICode", StringType(), True ),
                            StructField("XCoordinate", DoubleType(), True),
                            StructField("YCoordinate", DoubleType(), True ),
                            StructField("Year", IntegerType(), True),
                            StructField("UpdatedOn", DateType(), True ),
                            StructField("Latitude", DoubleType(), True),
                            StructField("Longitude", DoubleType(), True),
                            StructField("Location", StringType(), True )
                            ])
```

**create crimes dataframe by providing the schema above.**

```
crimes = spark.read.csv("Chicago_crimes_2001_to_present.csv",
                       header = True,
                        schema = crimes_schema)
```

First, let'se see how many rows the crimes dataframe has:

```
print(" The crimes dataframe has {} records".format(crimes.count()))
```

```
 The crimes dataframe has 6481208 records
```

We can also see the columns, the data type of each column and the schema using the commands below.

```
crimes.columns
```

```
['ID',
 'CaseNumber',
 'Date',
 'Block',
 'IUCR',
 'PrimaryType',
 'Description',
 'LocationDescription',
 'Arrest',
 'Domestic',
 'Beat',
 'District',
 'Ward',
 'CommunityArea',
 'FBICode',
 'XCoordinate',
 'YCoordinate',
 'Year',
 'UpdatedOn',
 'Latitude',
 'Longitude',
 'Location']
```

```
crimes.dtypes
```

```
[('ID', 'string'),
 ('CaseNumber', 'string'),
 ('Date', 'string'),
 ('Block', 'string'),
 ('IUCR', 'string'),
 ('PrimaryType', 'string'),
 ('Description', 'string'),
 ('LocationDescription', 'string'),
 ('Arrest', 'boolean'),
 ('Domestic', 'boolean'),
 ('Beat', 'string'),
 ('District', 'string'),
 ('Ward', 'string'),
 ('CommunityArea', 'string'),
 ('FBICode', 'string'),
 ('XCoordinate', 'double'),
 ('YCoordinate', 'double'),
 ('Year', 'int'),
 ('UpdatedOn', 'date'),
 ('Latitude', 'double'),
 ('Longitude', 'double'),
 ('Location', 'string')]
```

```
crimes.printSchema()
```

```
root
 |-- ID: string (nullable = true)
 |-- CaseNumber: string (nullable = true)
 |-- Date: string (nullable = true)
 |-- Block: string (nullable = true)
 |-- IUCR: string (nullable = true)
 |-- PrimaryType: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- LocationDescription: string (nullable = true)
 |-- Arrest: boolean (nullable = true)
 |-- Domestic: boolean (nullable = true)
 |-- Beat: string (nullable = true)
 |-- District: string (nullable = true)
 |-- Ward: string (nullable = true)
 |-- CommunityArea: string (nullable = true)
 |-- FBICode: string (nullable = true)
 |-- XCoordinate: double (nullable = true)
 |-- YCoordinate: double (nullable = true)
 |-- Year: integer (nullable = true)
 |-- UpdatedOn: date (nullable = true)
 |-- Latitude: double (nullable = true)
 |-- Longitude: double (nullable = true)
 |-- Location: string (nullable = true)
```

We can also quickly see some rows as below. We select one or more columns using **select**. **show** helps us to print the first n rows.

```
crimes.select("Date").show(10, truncate = False)
```

```
+----------------------+
|Date                  |
+----------------------+
|01/31/2006 12:13:05 PM|
|03/21/2006 07:00:00 PM|
|02/09/2006 01:44:41 AM|
|03/21/2006 04:45:00 PM|
|03/21/2006 10:00:00 PM|
|03/20/2006 11:00:00 PM|
|02/01/2006 11:25:00 PM|
|03/21/2006 02:37:00 PM|
|02/09/2006 05:38:07 AM|
|11/29/2005 03:10:00 PM|
+----------------------+
only showing top 10 rows
```

### Change data type of a column

The **Date** column is in string format. Let's change it to timestamp format using the user defined functions (udf).

**withColumn** helps to create a new column and we remove one or more columns with **drop**.

```
from datetime import datetime
from pyspark.sql.functions import col,udf

myfunc =  udf(lambda x: datetime.strptime(x, '%m/%d/%Y %I:%M:%S %p'), TimestampType())
df = crimes.withColumn('Date_time', myfunc(col('Date'))).drop("Date")

df.select(df["Date_time"]).show(5)
```

```
+-------------------+
|          Date_time|
+-------------------+
|2006-01-31 12:13:05|
|2006-03-21 19:00:00|
|2006-02-09 01:44:41|
|2006-03-21 16:45:00|
|2006-03-21 22:00:00|
+-------------------+
only showing top 5 rows
```

## Calculate statistics of numeric and string columns

We can calculate the statistics of string and numeric columns using **describe**. When we select more than one columns, we have to pass the column names as a python list.

```
crimes.select(["Latitude","Longitude","Year","XCoordinate","YCoordinate"]).describe().show
()
```

```
+-------+-------------------+-------------------+------------------+---------
---------+------------------+
|summary|           Latitude|          Longitude|              Year|       XC
oordinate|       YCoordinate|
+-------+-------------------+-------------------+------------------+---------
---------+------------------+
|  count|            6393147|            6393147|           6479775|
  6393147|           6393147|
|   mean|  41.84186221474304| -87.67189839071902|2007.9269172154898|1164490.5
803256205|1885665.2150490205|
| stddev|0.09076954083441872|0.06277083346349299| 4.712584642906088|17364.095
200290543|32982.572778759975|
|    min|       36.619446395|      -91.686565684|              2001|
      0.0|               0.0|
|    max|       42.022910333|      -87.524529378|              2017|
1205119.0|         1951622.0|
+-------+-------------------+-------------------+------------------+---------
---------+------------------+
```

The above numbers are ugly. Let's round them using **format_number** from PySpark's the functions.

```
from pyspark.sql.functions import format_number
```

```
result = crimes.select(["Latitude","Longitude","Year","XCoordinate","YCoordinate"]).descri
be()
result.select(result['summary'],
              format_number(result['Latitude'].cast('float'),2).alias('Latitude'),
              format_number(result['Longitude'].cast('float'),2).alias('Longitude'),
              result['Year'].cast('int').alias('year'),
              format_number(result['XCoordinate'].cast('float'),2).alias('XCoordinate'),
              format_number(result['YCoordinate'].cast('float'),2).alias('YCoordinate')
             ).show()
```

```
+-------+------------+------------+-------+------------+------------+
|summary|    Latitude|   Longitude|   year| XCoordinate| YCoordinate|
+-------+------------+------------+-------+------------+------------+
|  count|6,394,450.00|6,394,450.00|6481208|6,394,450.00|6,394,450.00|
|   mean|       41.84|      -87.67|   2007|1,164,490.62|1,885,665.88|
| stddev|        0.09|        0.06|      4|   17,363.81|   32,982.29|
|    min|       36.62|      -91.69|   2001|        0.00|        0.00|
|    max|       42.02|      -87.52|   2017|1,205,119.00|1,951,622.00|
+-------+------------+------------+-------+------------+------------+
```

**How many primary crime types are there?**

**distinct** returns distinct elements.

```
crimes.select("PrimaryType").distinct().count()
```

```
35
```

We can also see a list of the primary crime types.

```
crimes.select("PrimaryType").distinct().show(n = 35)
```

```
+--------------------+
|         PrimaryType|
+--------------------+
|OFFENSE INVOLVING...|
|            STALKING|
|PUBLIC PEACE VIOL...|
|           OBSCENITY|
|NON-CRIMINAL (SUB...|
|               ARSON|
|   DOMESTIC VIOLENCE|
|            GAMBLING|
|   CRIMINAL TRESPASS|
|             ASSAULT|
|      NON - CRIMINAL|
|LIQUOR LAW VIOLATION|
| MOTOR VEHICLE THEFT|
|               THEFT|
|             BATTERY|
|             ROBBERY|
|            HOMICIDE|
|           RITUALISM|
|    PUBLIC INDECENCY|
| CRIM SEXUAL ASSAULT|
|   HUMAN TRAFFICKING|
|        INTIMIDATION|
|        PROSTITUTION|
|  DECEPTIVE PRACTICE|
|CONCEALED CARRY L...|
|         SEX OFFENSE|
|     CRIMINAL DAMAGE|
|           NARCOTICS|
|        NON-CRIMINAL|
|       OTHER OFFENSE|
|          KIDNAPPING|
|            BURGLARY|
|   WEAPONS VIOLATION|
|OTHER NARCOTIC VI...|
|INTERFERENCE WITH...|
+--------------------+
```

**How many homicides are there in the dataset?**

```
crimes.where(crimes["PrimaryType"] == "HOMICIDE").count()
```

```
8847
```

**how many domestic assualts there are?**

Make sure to add in the parenthesis separating the statements!

```
crimes.filter((crimes["PrimaryType"] == "ASSAULT") & (crimes["Domestic"] == "True")).count
()
```

```
86552
```

We can use **filter** or **where** to do filtering.

```
columns = ['PrimaryType', 'Description', 'Arrest', 'Domestic']

crimes.where((crimes["PrimaryType"] == "HOMICIDE") & (crimes["Arrest"] == "true"))\
                                                        .select(columns).show(10)
```

```
+-----------+-------------------+------+--------+
|PrimaryType|        Description|Arrest|Domestic|
+-----------+-------------------+------+--------+
|   HOMICIDE|FIRST DEGREE MURDER|  true|    true|
|   HOMICIDE|FIRST DEGREE MURDER|  true|   false|
|   HOMICIDE|FIRST DEGREE MURDER|  true|   false|
|   HOMICIDE|FIRST DEGREE MURDER|  true|   false|
|   HOMICIDE|FIRST DEGREE MURDER|  true|   false|
|   HOMICIDE|FIRST DEGREE MURDER|  true|   false|
|   HOMICIDE|FIRST DEGREE MURDER|  true|   false|
|   HOMICIDE|FIRST DEGREE MURDER|  true|   false|
|   HOMICIDE|FIRST DEGREE MURDER|  true|   false|
|   HOMICIDE|FIRST DEGREE MURDER|  true|    true|
+-----------+-------------------+------+--------+
only showing top 10 rows
```

We can use **limit** to limit the number of columns we want to retrieve from a dataframe.

```
crimes.select(columns).limit(10). show(truncate = True)
```

```
+-------------------+--------------------+------+--------+
|        PrimaryType|         Description|Arrest|Domestic|
+-------------------+--------------------+------+--------+
|          NARCOTICS|POSS: CANNABIS 30...|  true|   false|
|  CRIMINAL TRESPASS|             TO LAND|  true|   false|
|          NARCOTICS|POSS: CANNABIS 30...|  true|   false|
|              THEFT|           OVER $500| false|   false|
|              THEFT|      $500 AND UNDER| false|   false|
|MOTOR VEHICLE THEFT|          AUTOMOBILE| false|   false|
|          NARCOTICS|         POSS: CRACK|  true|   false|
|    CRIMINAL DAMAGE|         TO PROPERTY| false|   false|
|       PROSTITUTION|SOLICIT FOR PROST...|  true|   false|
|    CRIMINAL DAMAGE|   TO STATE SUP PROP| false|   false|
+-------------------+--------------------+------+--------+
```

**Create a new column with withColumn**

```
lat_max = crimes.agg({"Latitude" : "max"}).collect()[0][0]

print("The maximum latitude values is {}".format(lat_max))
```

```
The maximum latitude values is 42.022910333
```

Let's subtract each latitude value from the maximum latitude.

```
df = crimes.withColumn("difference_from_max_lat",lat_max - crimes["Latitude"])
```

```
df.select(["Latitude", "difference_from_max_lat"]).show(5)
```

```
+------------+----------------------+
|    Latitude|difference_from_max_lat|
+------------+----------------------+
|42.002478396|     0.0204319369999979|
|41.780595495|    0.24231483799999864|
|41.787955143|    0.23495519000000087|
|41.901774026|    0.12113630700000044|
|41.748674558|    0.27423577500000107|
+------------+----------------------+
only showing top 5 rows
```

#### Rename a column with withColumnRenamed

Let's rename Latitude to Lat.

```
df = crimes.withColumnRenamed("Latitude", "Lat")
df.columns
```

```
['ID',
 'CaseNumber',
 'Date',
 'Block',
 'IUCR',
 'PrimaryType',
 'Description',
 'LocationDescription',
 'Arrest',
 'Domestic',
 'Beat',
 'District',
 'Ward',
 'CommunityArea',
 'FBICode',
 'XCoordinate',
 'YCoordinate',
 'Year',
 'UpdatedOn',
 'Lat',
 'Longitude',
 'Location']
```

```
columns = ['PrimaryType', 'Description', 'Arrest', 'Domestic','Lat']

df.orderBy(df["Lat"].desc()).select(columns).show(10)
```

```
+-------------------+-------------------+------+--------+-----------+
|        PrimaryType|        Description|Arrest|Domestic|        Lat|
+-------------------+-------------------+------+--------+-----------+
|              THEFT|     $500 AND UNDER| false|   false|42.022910333|
|MOTOR VEHICLE THEFT|         AUTOMOBILE| false|   false|42.022878225|
|           BURGLARY|     UNLAWFUL ENTRY| false|   false|42.022709624|
|              THEFT|     POCKET-PICKING| false|   false|42.022671246|
|              THEFT|     $500 AND UNDER| false|   false|42.022671246|
|      OTHER OFFENSE|   PAROLE VIOLATION|  true|   false|42.022671246|
|    CRIMINAL DAMAGE|         TO VEHICLE| false|   false|42.022653914|
|            BATTERY|             SIMPLE| false|    true|42.022644813|
|          NARCOTICS|POSS: CANNABIS 30...|  true|   false|42.022644813|
|      OTHER OFFENSE|   TELEPHONE THREAT| false|    true|42.022644369|
+-------------------+-------------------+------+--------+-----------+
only showing top 10 rows
```

## Use PySpark's functions to calculate various statistics

Calculate average latitude value.

```
from pyspark.sql.functions import mean
df.select(mean("Lat")).alias("Mean Latitude").show()
```

```
+-----------------+
|         avg(Lat)|
+-----------------+
|41.84186221474304|
+-----------------+
```

We can also use the **agg** method to calculate the average.

```
df.agg({"Lat":"avg"}).show()
```

```
+------------------+
|          avg(Lat)|
+------------------+
|41.841863914298415|
+------------------+
```

**We can also calculate maximum and minimum values using functions from Pyspark.**

```
from pyspark.sql.functions import max,min
```

```
df.select(max("Xcoordinate"),min("Xcoordinate")).show()
```

```
+----------------+----------------+
|max(Xcoordinate)|min(Xcoordinate)|
+----------------+----------------+
|       1205119.0|             0.0|
+----------------+----------------+
```

#### What percentage of the crimes are domestic

```
df.filter(df["Domestic"]==True).count()/df.count() * 100
```

```
12.988412036768453
```

#### What is the Pearson correlation between Lat and Ycoordinate?

```
from pyspark.sql.functions import corr
df.select(corr("Lat","Ycoordinate")).show()
```

```
+----------------------+
|corr(Lat, Ycoordinate)|
+----------------------+
|    0.9999931390763287|
+----------------------+
```

#### Find the number of crimes per year

```
df.groupBy("Year").count().show()
```

```
+----+------+
|Year| count|
+----+------+
|2003|475921|
|2007|436966|
|2015|263496|
|2006|448066|
|2013|306846|
|2014|274839|
|2004|469362|
|2012|335798|
|2009|392601|
|2016|268160|
|2001|485735|
|2005|453687|
|2010|370230|
|2011|351654|
|2008|427000|
|2017|232670|
|2002|486744|
+----+------+
```

```
df.groupBy("Year").count().collect()
```

```
[Row(Year=2003, count=475921),
 Row(Year=2007, count=436966),
 Row(Year=2015, count=263496),
 Row(Year=2006, count=448066),
 Row(Year=2013, count=306846),
 Row(Year=2014, count=274839),
 Row(Year=2004, count=469362),
 Row(Year=2012, count=335798),
 Row(Year=2009, count=392601),
 Row(Year=2016, count=268160),
 Row(Year=2001, count=485735),
 Row(Year=2005, count=453687),
 Row(Year=2010, count=370230),
 Row(Year=2011, count=351654),
 Row(Year=2008, count=427000),
 Row(Year=2017, count=232670),
 Row(Year=2002, count=486744)]
```

#### We can also use matplotlib and Pandas to visualize the total number of crimes per year

```
count = [item[1] for item in df.groupBy("Year").count().collect()]
year = [item[0] for item in df.groupBy("Year").count().collect()]
```

```
number_of_crimes_per_year = {"count":count, "year" : year}
```

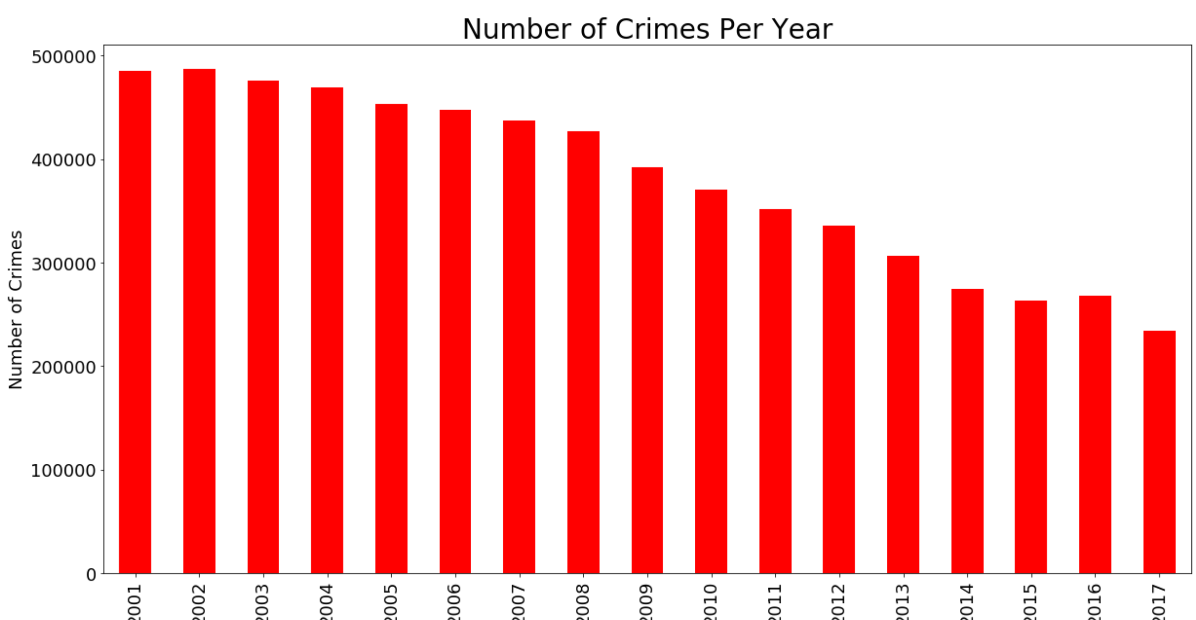
```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

```
number_of_crimes_per_year = pd.DataFrame(number_of_crimes_per_year)
```

```
number_of_crimes_per_year.head()
```

| | count | year |
|---|---|---|
| 0 | 475921 | 2003 |
| 1 | 436966 | 2007 |
| 2 | 263495 | 2015 |
| 3 | 448066 | 2006 |
| 4 | 306847 | 2013 |

```
number_of_crimes_per_year = number_of_crimes_per_year.sort_values(by = "year")

number_of_crimes_per_year.plot(figsize = (20,10), kind = "bar", color = "red",
                               x = "year", y = "count", legend = False)

plt.xlabel("", fontsize = 18)
plt.ylabel("Number of Crimes", fontsize = 18)
plt.title("Number of Crimes Per Year", fontsize = 28)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.show()
```



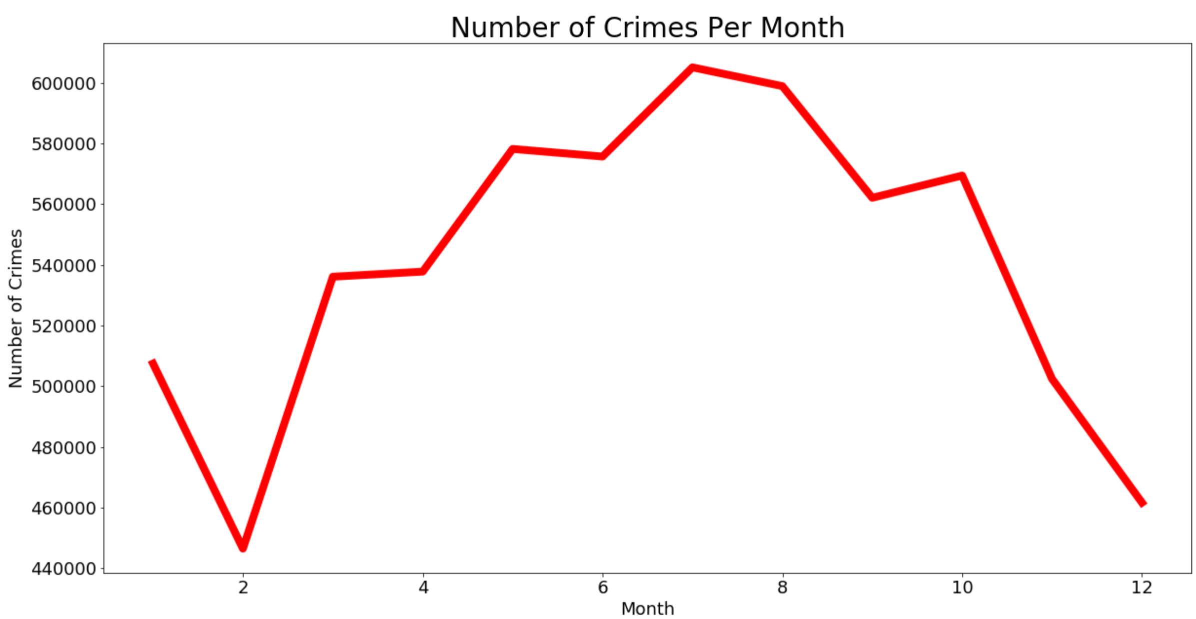

**Plot number of crimes by month**

we can use the month function from PySpark's functions to get the numeric month.

```
from pyspark.sql.functions import month
monthdf = df.withColumn("Month",month("Date_time"))
monthCounts = monthdf.select("Month").groupBy("Month").count()
monthCounts.show()
```

```
+-----+------+
|Month| count|
+-----+------+
|   12|461611|
|    1|507455|
|    6|575702|
|    3|536081|
|    5|578211|
|    9|562105|
|    4|537761|
|    8|598914|
|    7|605102|
|   10|569435|
|   11|502385|
|    2|446446|
+-----+------+
```

```
monthCounts = monthCounts.collect()

months = [item[0] for item in monthCounts]

count = [item[1] for item in monthCounts]

crimes_per_month = {"month":months, "crime_count": count}
crimes_per_month = pd.DataFrame(crimes_per_month)


crimes_per_month = crimes_per_month.sort_values(by = "month")

crimes_per_month.plot(figsize = (20,10), kind = "line", x = "month", y = "crime_count",
                      color = "red", linewidth = 8, legend = False)

plt.xlabel("Month", fontsize = 18)
plt.ylabel("Number of Crimes", fontsize = 18)
plt.title("Number of Crimes Per Month", fontsize = 28)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.show()
```




## Where do most crimes take pace?

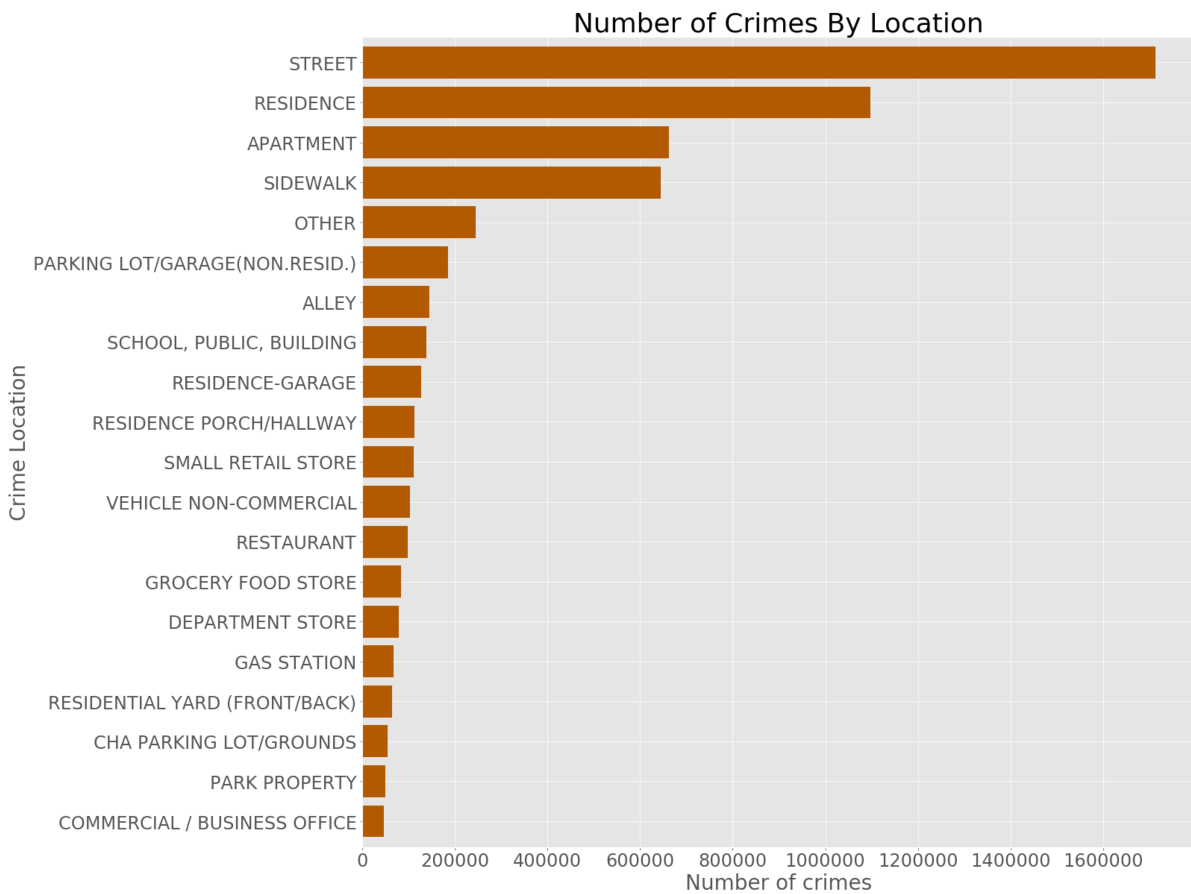
```
crimes.groupBy("LocationDescription").count().show()
```

```
+--------------------+------+
| LocationDescription| count|
+--------------------+------+
|   RAILROAD PROPERTY|    13|
|AIRPORT TERMINAL ...|  1417|
|EXPRESSWAY EMBANK...|     1|
|POLICE FACILITY/V...| 16518|
|               MOTEL|     5|
|            SIDEWALK|644570|
|AIRPORT TERMINAL ...|    67|
|PUBLIC GRAMMAR SC...|     1|
|CTA GARAGE / OTHE...|  9660|
|            CAR WASH|  2632|
|   TRUCKING TERMINAL|     1|
|    AIRPORT/AIRCRAFT| 16060|
|            HOSPITAL|     5|
|MEDICAL/DENTAL OF...|  6836|
|    FEDERAL BUILDING|   736|
|             TRAILER|     3|
|SCHOOL, PUBLIC, G...| 27969|
|         CTA STATION|  2760|
|SPORTS ARENA/STADIUM|  4733|
|               HOUSE|   497|
+--------------------+------+
only showing top 20 rows
```

```
crime_location  = crimes.groupBy("LocationDescription").count().collect()
location = [item[0] for item in crime_location]
count = [item[1] for item in crime_location]
crime_location = {"location" : location, "count": count}
crime_location = pd.DataFrame(crime_location)
crime_location = crime_location.sort_values(by = "count", ascending  = False)
crime_location.iloc[:5]
```

| | count | location |
|---|---|---|
| **58** | 1711956 | STREET |
| **95** | 1097012 | RESIDENCE |
| **125** | 662880 | APARTMENT |
| **5** | 644570 | SIDEWALK |
| **126** | 245385 | OTHER |

```
crime_location = crime_location.iloc[:20]

myplot = crime_location .plot(figsize = (20,20), kind = "barh", color = "#b35900", width =
 0.8,
                               x = "location", y = "count", legend = False)

myplot.invert_yaxis()

plt.xlabel("Number of crimes", fontsize = 28)
plt.ylabel("Crime Location", fontsize = 28)
plt.title("Number of Crimes By Location", fontsize = 36)
plt.xticks(size = 24)
plt.yticks(size = 24)
plt.show()
```




**We can also calculate the number of crimes per hour, day, and month.**

Let's add day of week and hour of day columns using the date_format.

```
from pyspark.sql.functions import date_format
df = df.withColumn("DayOfWeek", date_format("Date_time","E")).\
      withColumn("DayOfWeek_number", date_format("Date_time","u")).\
      withColumn("HourOfDay", date_format("Date_time","H"))
weekDaysCount  = df.groupBy(["DayOfWeek", "DayOfWeek_number"]).count()
weekDaysCount.show()
```

```
+---------+----------------+------+
|DayOfWeek|DayOfWeek_number| count|
+---------+----------------+------+
|      Fri|               5|976064|
|      Wed|               3|935274|
|      Sat|               6|925385|
|      Tue|               2|929622|
|      Mon|               1|914099|
|      Sun|               7|874663|
|      Thu|               4|926101|
+---------+----------------+------+
```

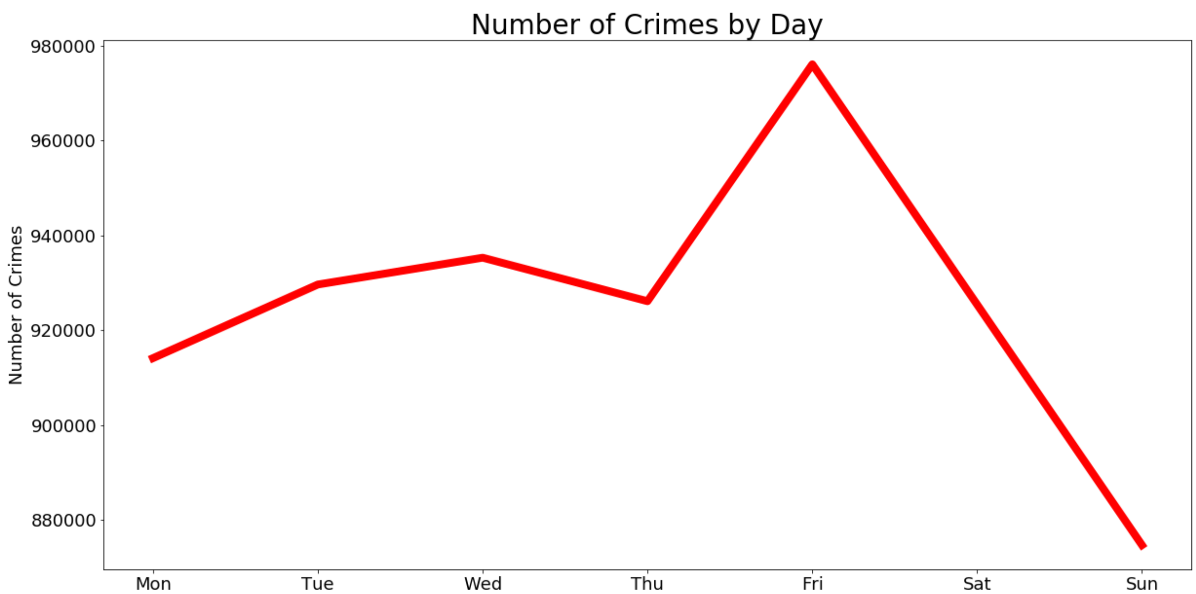
We can also print the schema to see the columns.

```
df.printSchema()
```

```
root
 |-- ID: string (nullable = true)
 |-- CaseNumber: string (nullable = true)
 |-- Block: string (nullable = true)
 |-- IUCR: string (nullable = true)
 |-- PrimaryType: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- LocationDescription: string (nullable = true)
 |-- Arrest: boolean (nullable = true)
 |-- Domestic: boolean (nullable = true)
 |-- Beat: string (nullable = true)
 |-- District: string (nullable = true)
 |-- Ward: string (nullable = true)
 |-- CommunityArea: string (nullable = true)
 |-- FBICode: string (nullable = true)
 |-- XCoordinate: double (nullable = true)
 |-- YCoordinate: double (nullable = true)
 |-- Year: integer (nullable = true)
 |-- UpdatedOn: date (nullable = true)
 |-- Latitude: double (nullable = true)
 |-- Longitude: double (nullable = true)
 |-- Location: string (nullable = true)
 |-- Date_time: timestamp (nullable = true)
 |-- DayOfWeek: string (nullable = true)
 |-- DayOfWeek_number: string (nullable = true)
 |-- HourOfDay: string (nullable = true)
```

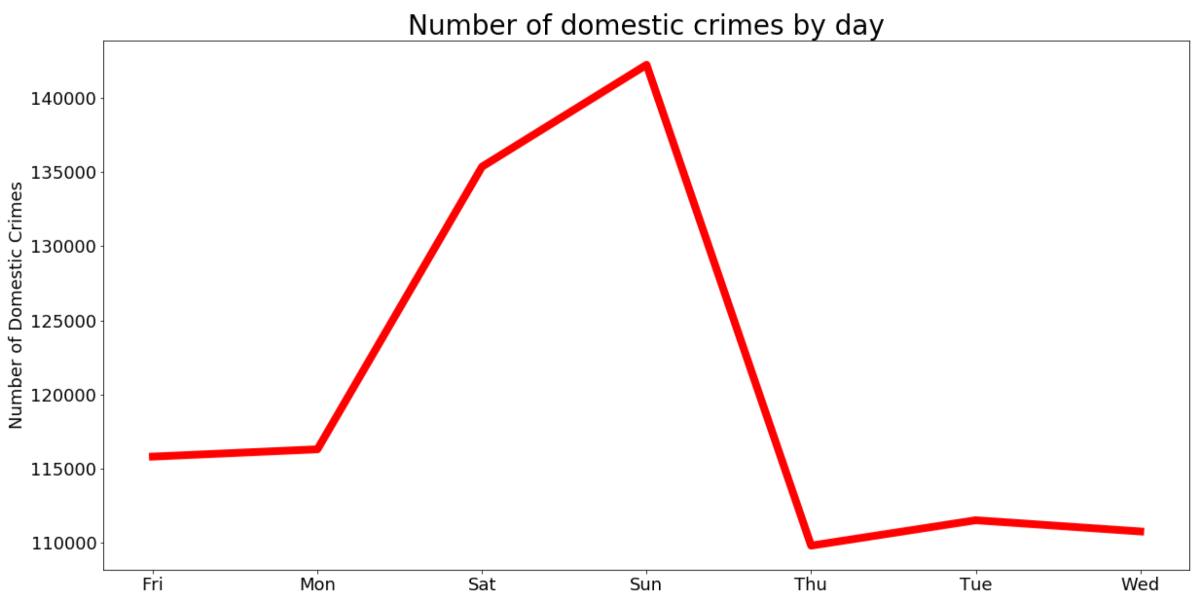
**Which days have the highest number of crimes?**

```
weekDaysCount  = df.groupBy(["DayOfWeek", "DayOfWeek_number"]).count().collect()
days = [item[0] for item in weekDaysCount]
count = [item[2] for item in weekDaysCount]
day_number = [item[1] for item in weekDaysCount]
crime_byDay = {"days" : days, "count": count, "day_number": day_number}
crime_byDay = pd.DataFrame(crime_byDay)
crime_byDay = crime_byDay.sort_values(by = "day_number", ascending  = True)


crime_byDay.plot(figsize = (20,10), kind = "line", x = "days", y = "count",
                      color = "red", linewidth = 8, legend = False)

plt.ylabel("Number of Crimes", fontsize = 18)
plt.xlabel("")
plt.title("Number of Crimes by Day", fontsize = 28)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.show()
```



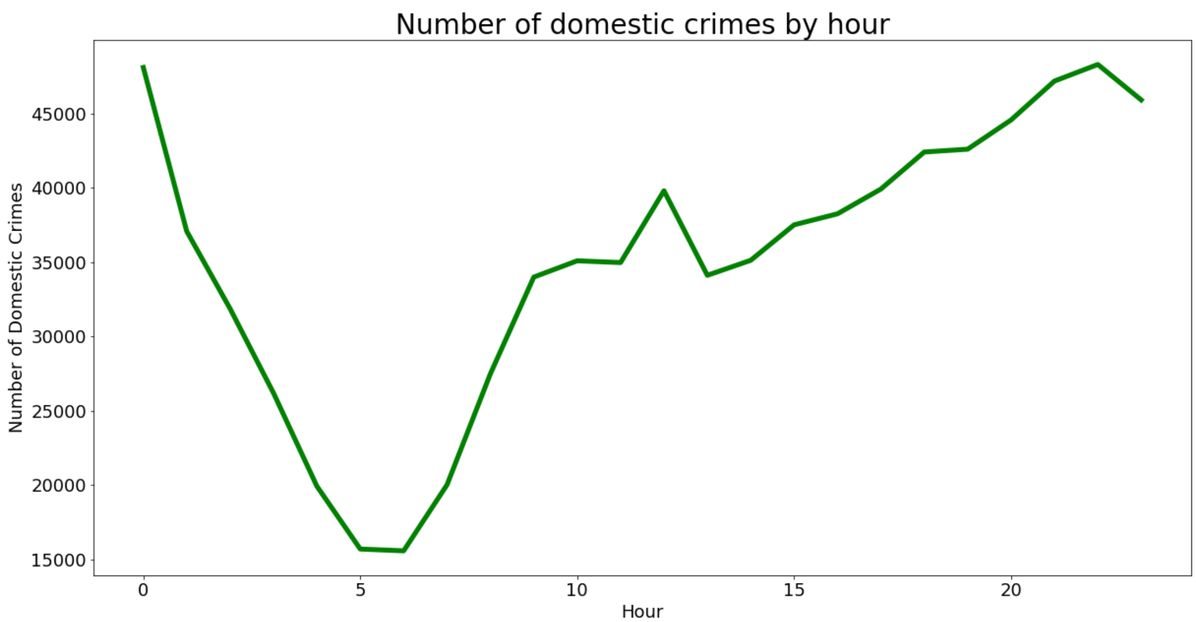

**we can also show only number of domestic crimes by day**

```
weekDaysCount  = df.filter(df["Domestic"] == "true").groupBy(["DayOfWeek", "DayOfWeek_numb
er"]).count().collect()
days = [item[0] for item in weekDaysCount]
count = [item[2] for item in weekDaysCount]
day_number = [item[1] for item in weekDaysCount]
crime_byDay = {"days" : days, "count": count, "day_number": day_number}
crime_byDay = pd.DataFrame(crime_byDay)
crime_byDay = crime_byDay.sort_values(by = "days", ascending  = True)


crime_byDay.plot(figsize = (20,10), kind = "line", x = "days", y = "count",
                      color = "red", linewidth = 8, legend = False)

plt.ylabel("Number of Domestic Crimes", fontsize = 18)
plt.xlabel("")
plt.title("Number of domestic crimes by day", fontsize = 28)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.show()
```




### Number of domestic crimes by hour

```
temp = df.filter(df["Domestic"] == "true")
temp = temp.select(df['HourOfDay'].cast('int').alias('HourOfDay'))
hourlyCount  = temp.groupBy(["HourOfDay"]).count().collect()

hours = [item[0] for item in hourlyCount]
count = [item[1] for item in hourlyCount]

crime_byHour = {"count": count, "hours": hours}
crime_byHour = pd.DataFrame(crime_byHour)
crime_byHour = crime_byHour.sort_values(by = "hours", ascending  = True)


crime_byHour.plot(figsize = (20,10), kind = "line", x = "hours", y = "count",
                      color = "green", linewidth = 5, legend = False)

plt.ylabel("Number of Domestic Crimes", fontsize = 18)
plt.xlabel("Hour", fontsize = 18)
plt.title("Number of domestic crimes by hour", fontsize = 28)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.show()
```

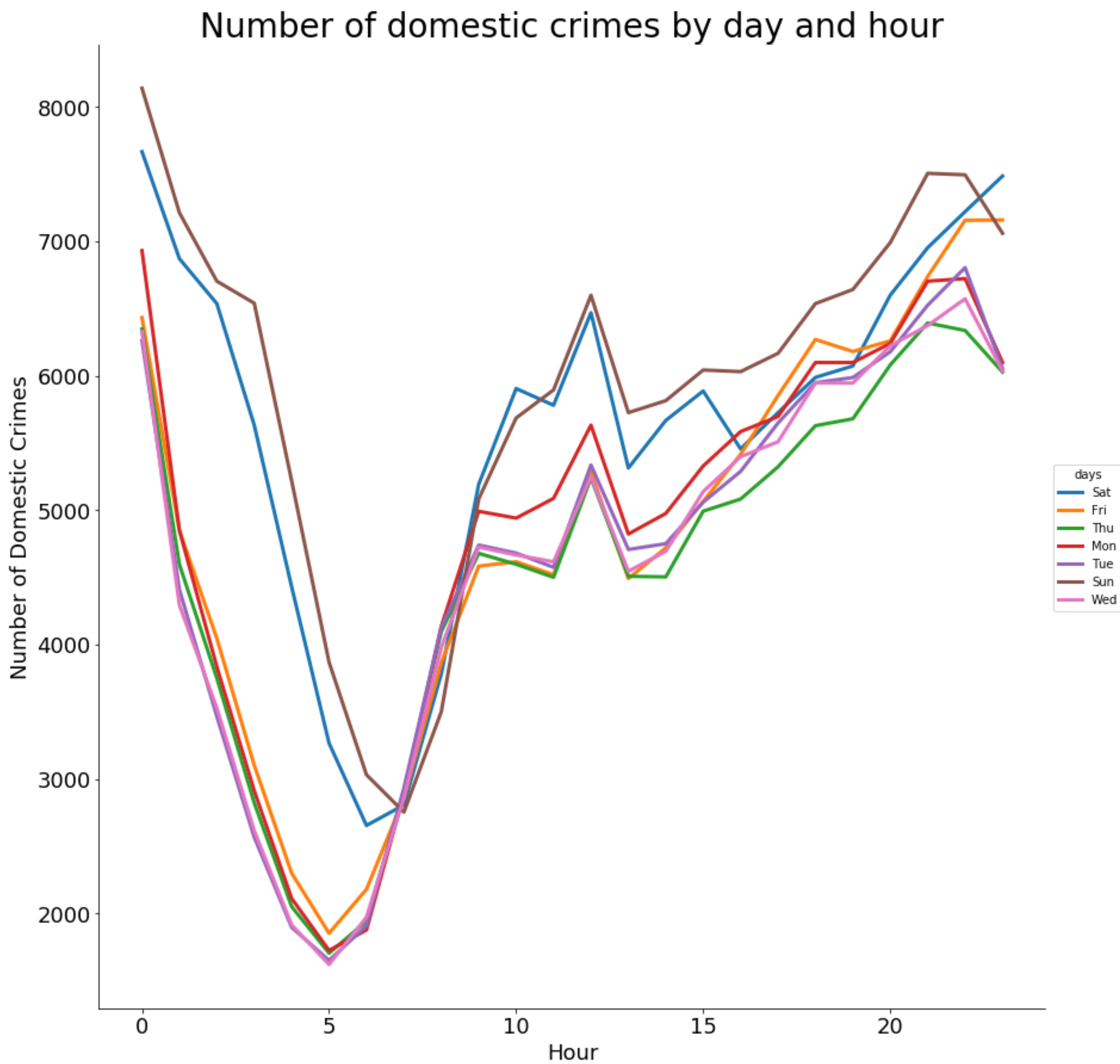



### We can also show number of domestic crimes by day and hour

```
import seaborn as sns
```

```
temp = df.filter(df["Domestic"] == "true")
temp = temp.select("DayOfWeek", df['HourOfDay'].cast('int').alias('HourOfDay'))
hourlyCount  = temp.groupBy(["DayOfWeek","HourOfDay"]).count().collect()

days = [item[0] for item in hourlyCount]
hours = [item[1] for item in hourlyCount]
count = [item[2] for item in hourlyCount]

crime_byHour = {"count": count, "hours": hours, "days": days}
crime_byHour = pd.DataFrame(crime_byHour)
crime_byHour = crime_byHour.sort_values(by = "hours", ascending  = True)
```

```
import seaborn as sns

g = sns.FacetGrid(crime_byHour, hue="days", size = 12)
g.map(plt.plot, "hours", "count", linewidth = 3)
g.add_legend()

plt.ylabel("Number of Domestic Crimes", fontsize = 18)
plt.xlabel("Hour", fontsize = 18)
plt.title("Number of domestic crimes by day and hour", fontsize = 28)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.show()
```




## Remark

This is the second blog post on the Spark tutorial series to help big data enthusiasts prepare for Apache Spark Certifications from companies such as Cloudera, Hortonworks, Databricks, etc. The first one is here (http://datascience-enthusiast.com/Python/analyzing_bible_quran_with_spark.html). If you want to learn/master Spark with Python or if you are preparing for a Spark Certification to show your skills in big data, these articles are for you.

# Spark RDDs vs DataFrames vs SparkSQL - part 1: Retrieving, Sorting and Filtering

Spark is a fast and general engine for large-scale data processing. It is a cluster computing framework which is used for scalable and efficient analysis of big data. With Spark, we can use many machines, which divide the tasks among themselves, and perform fault tolerant computations by distributing the data over a cluster.

Among the many capabilities of Spark, which made it famous, is its ability to be used with various programming languages through APIs. We can write Spark operations in Java, Scala, Python or R. Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

Spark components consist of Core Spark, Spark SQL, MLlib and ML for machine learning and GraphX for graph analytics. To help big data enthusiasts master Apache Spark, I have started writing tutorials. The first one is here (http://datascience-enthusiast.com/Python/analyzing_bible_quran_with_spark.html) and the second one is here (http://datascience-enthusiast.com/Python/SparkDataFrames-ExploringChicagoCrimes.html). For the next couple of weeks, I will write a blog post series on how to perform the same tasks using Spark Resilient Distributed Dataset (RDD), DataFrames and Spark SQL and this is the first one. I am using pyspark, which is the Spark Python API that exposes the Spark programming model to Python. The data can be downloaded from my GitHub repository (https://github.com/fissehab/Spark_certification/tree/master/data/AdventureWorksLT2012). The size of the data is not large, however, the same code works for large volume as well. Therefore, we can practice with this dataset to master the functionalities of Spark.

For this tutorial, we will work with the **SalesLTProduct.txt** data. Let's answer a couple of questions using RDD way, DataFrame way and Spark SQL.

SparkContext is main entry point for Spark functionality.

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SQLContext

conf = SparkConf().setAppName("miniProject").setMaster("local[*]")
sc = SparkContext.getOrCreate(conf)
```

**Create RDD from file**

```
products = sc.textFile("SalesLTProduct.txt")
```

**Retrieve the first row of the data**

```
products.first()
```

```
'ProductID\tName\tProductNumber\tColor\tStandardCost\tListPrice\tSize\tWeight
\tProductCategoryID\tProductModelID\tSellStartDate\tSellEndDate\tDiscontinued
Date\tThumbNailPhoto\tThumbnailPhotoFileName\trowguid\tModifiedDate'
```

We see that the first row is column names and the data is tab (\t) delimited. Let's remove the first row from the RDD and use it as column names.

We can see how many column the data has by spliting the first row as below

```
print("The data has {} columns".format(len(products.first().split("\t"))))
products.first().split("\t")
```

```
The data has 17 columns
```

```
['ProductID',
 'Name',
 'ProductNumber',
 'Color',
 'StandardCost',
 'ListPrice',
 'Size',
 'Weight',
 'ProductCategoryID',
 'ProductModelID',
 'SellStartDate',
 'SellEndDate',
 'DiscontinuedDate',
 'ThumbNailPhoto',
 'ThumbnailPhotoFileName',
 'rowguid',
 'ModifiedDate']
```

```
header = products.first()

content = products.filter(lambda line: line != header)
```

Now, we can see the first row in the data, after removing the column names.

```
content.first()
```

```
'680\tHL Road Frame - Black, 58\tFR-R92B-58\tBlack\t1059.31\t1431.50\t58\t101
6.04\t18\t6\t1998-06-01 00:00:00.000\tNULL\tNULL\t0x47494638396150003100F7000
00000008000000080008080000000808000800080808080C0C0C0FF000000FF00FFFF000000
FFFF00FF00FFFFFFFFFF0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000\t
no_image_available_small.gif\t43DD68D6-14A4-461F-9069-55309D90EA7E\t2004-03-1
1 10:01:36.827'
```

We have seen above using the header that the data has 17 columns. We can also check from the **content** RDD.

```
content.map(lambda line: len(line.split("\t"))).distinct().collect()
```

```
[17]
```

Now, let's solve questions using Spark RDDs and Spark DataFrames.

**1. Transportation costs are increasing and you need to identify the heaviest products. Retrieve the names of the top 15 products by weight.**

**RDD Way**

First, we will filter out NULL values because they will create problems to convert the wieght to numeric. Then, we will order our RDD using the weight column in descending order and then we will take the first 15 rows.

```
(content.filter(lambda line: line.split("\t")[7] != "NULL")
 .map(lambda line: (line.split("\t")[1], float(line.split("\t")[7])))
                     .takeOrdered(15, lambda x : -x[1])
                    )
```

```
[('Touring-3000 Blue, 62', 13607.7),
 ('Touring-3000 Yellow, 62', 13607.7),
 ('Touring-3000 Blue, 58', 13562.34),
 ('Touring-3000 Yellow, 58', 13512.45),
 ('Touring-3000 Blue, 54', 13462.55),
 ('Touring-3000 Yellow, 54', 13344.62),
 ('Touring-3000 Yellow, 50', 13213.08),
 ('Touring-3000 Blue, 50', 13213.08),
 ('Touring-3000 Yellow, 44', 13049.78),
 ('Touring-3000 Blue, 44', 13049.78),
 ('Mountain-500 Silver, 52', 13008.96),
 ('Mountain-500 Black, 52', 13008.96),
 ('Mountain-500 Silver, 48', 12891.03),
 ('Mountain-500 Black, 48', 12891.03),
 ('Mountain-500 Silver, 44', 12759.49)]
```

**DataFrame Way**

Hortonworks Spark Certification is with Spark 1.6 and that is why I am using SQLContext here. Otherwise, for recent Spark versions, SQLContext has been replaced by SparkSession as noted here (https://spark.apache.org/docs/2.0.0/sql-programming-guide.html#migration-guide)

```
from pyspark.sql import SQLContext

sqlContext = SQLContext(sc)
```

```
rdd1 = (content.filter(lambda line: line.split("\t")[7] != "NULL")
 .map(lambda line: (line.split("\t")[1], float(line.split("\t")[7])))
)
```

Now, we can create a DataFrame, order the DataFrame by weight in descending order and take the first 15 records.

```
df = sqlContext.createDataFrame(rdd1, schema = ["Name", "Weight"])
```

```
df.orderBy("weight", ascending = False).show(15, truncate = False)
```

```
+----------------------+--------+
|Name                  |Weight  |
+----------------------+--------+
|Touring-3000 Blue, 62  |13607.7 |
|Touring-3000 Yellow, 62|13607.7 |
|Touring-3000 Blue, 58  |13562.34|
|Touring-3000 Yellow, 58|13512.45|
|Touring-3000 Blue, 54  |13462.55|
|Touring-3000 Yellow, 54|13344.62|
|Touring-3000 Blue, 50  |13213.08|
|Touring-3000 Yellow, 50|13213.08|
|Touring-3000 Yellow, 44|13049.78|
|Touring-3000 Blue, 44  |13049.78|
|Mountain-500 Black, 52 |13008.96|
|Mountain-500 Silver, 52|13008.96|
|Mountain-500 Silver, 48|12891.03|
|Mountain-500 Black, 48 |12891.03|
|Mountain-500 Silver, 44|12759.49|
+----------------------+--------+
only showing top 15 rows
```

The **sql** function on a **SQLContext** enables applications to run SQL queries programmatically and returns the result as a DataFrame.

First, we have to register the DataFrame as a SQL temporary view.

#### Running SQL Queries Programmatically

```
df.createOrReplaceTempView("df_table")
```

```
sqlContext.sql(" SELECT * FROM df_table  ORDER BY Weight DESC limit 15").show()
```

```
+--------------------+--------+
|                Name|  Weight|
+--------------------+--------+
|Touring-3000 Yell...| 13607.7|
|Touring-3000 Blue...| 13607.7|
|Touring-3000 Blue...|13562.34|
|Touring-3000 Yell...|13512.45|
|Touring-3000 Blue...|13462.55|
|Touring-3000 Yell...|13344.62|
|Touring-3000 Yell...|13213.08|
|Touring-3000 Blue...|13213.08|
|Touring-3000 Blue...|13049.78|
|Touring-3000 Yell...|13049.78|
|Mountain-500 Blac...|13008.96|
|Mountain-500 Silv...|13008.96|
|Mountain-500 Blac...|12891.03|
|Mountain-500 Silv...|12891.03|
|Mountain-500 Silv...|12759.49|
+--------------------+--------+
```

**2. The heaviest ten products are transported by a specialist carrier, therefore you need to modify the previous query to list the heaviest 15 products not including the heaviest 10.**

First, let's remove the top 10 heaviest ones and take the top 15 records based on the weight column.

**RDD way**

```
top_10 = (content.filter(lambda line: line.split("\t")[7] != "NULL")
 .map(lambda line: (line.split("\t")[1], float(line.split("\t")[7])))
                    .takeOrdered(10, lambda x : -x[1])
                   )
```

```
name_weight_all_records = (content.filter(lambda line: line.split("\t")[7] != "NULL").
map(lambda line: (line.split("\t")[1], float(line.split("\t")[7]))))
```

```
name_weight_all_records.filter(lambda line: line not in top_10).takeOrdered(15, lambda x :
 -x[1])
```

```
[('Mountain-500 Silver, 52', 13008.96),
 ('Mountain-500 Black, 52', 13008.96),
 ('Mountain-500 Silver, 48', 12891.03),
 ('Mountain-500 Black, 48', 12891.03),
 ('Mountain-500 Silver, 44', 12759.49),
 ('Mountain-500 Black, 44', 12759.49),
 ('Touring-2000 Blue, 60', 12655.16),
 ('Mountain-500 Silver, 42', 12596.19),
 ('Mountain-500 Black, 42', 12596.19),
 ('Touring-2000 Blue, 54', 12555.37),
 ('Touring-2000 Blue, 50', 12437.44),
 ('Mountain-400-W Silver, 46', 12437.44),
 ('Mountain-500 Silver, 40', 12405.69),
 ('Mountain-500 Black, 40', 12405.69),
 ('Touring-2000 Blue, 46', 12305.9)]
```

#### DataFrame way

```
df = sqlContext.createDataFrame(name_weight_all_records, schema = ["Name", "Weight"])
```

```
top_10 = df.orderBy("Weight", ascending = False).take(10)
```

```
top_10_names = [x[0] for x in top_10]
top_10_weights = [x[1] for x in top_10]
```

```
from pyspark.sql.functions import col
```

```
(df.filter((~col("Name").isin(top_10_names)) & (~col("Weight").isin(top_10_names)))
.orderBy("Weight", ascending = False)
.show(15, truncate = False)
)
```

```
+-------------------------+--------+
|Name                     |Weight  |
+-------------------------+--------+
|Mountain-500 Black, 52   |13008.96|
|Mountain-500 Silver, 52  |13008.96|
|Mountain-500 Silver, 48  |12891.03|
|Mountain-500 Black, 48   |12891.03|
|Mountain-500 Silver, 44  |12759.49|
|Mountain-500 Black, 44   |12759.49|
|Touring-2000 Blue, 60    |12655.16|
|Mountain-500 Silver, 42  |12596.19|
|Mountain-500 Black, 42   |12596.19|
|Touring-2000 Blue, 54    |12555.37|
|Mountain-400-W Silver, 46|12437.44|
|Touring-2000 Blue, 50    |12437.44|
|Mountain-500 Silver, 40  |12405.69|
|Mountain-500 Black, 40   |12405.69|
|Touring-2000 Blue, 46    |12305.9 |
+-------------------------+--------+
only showing top 15 rows
```

As of now, I think Spark SQL does not support OFFSET.

**3. Retrieve product details for products where the product model ID is 1**

**RDD way**

Let's display the Name, Color, Size and product model

```
(content.filter(lambda line:line.split("\t")[9]=="1")
 .map(lambda line: (line.split("\t")[1],line.split("\t")[3], line.split("\t")[6], line.spl
it("\t")[9])).collect()
)
```

```
[('Classic Vest, S', 'Blue', 'S', '1'),
 ('Classic Vest, M', 'Blue', 'M', '1'),
 ('Classic Vest, L', 'Blue', 'L', '1')]
```

**DataFrame way**

```
rdd = content.map(lambda line: (line.split("\t")[1],line.split("\t")[3], line.split("\t")[
6], line.split("\t")[9])).collect()
```

```
df = sqlContext.createDataFrame(rdd, schema = ["Name", "Color", "Size","ProductModelID"])
```

```
df.filter(df["ProductModelID"]==1).show()
```

```
+--------------+-----+----+--------------+
|          Name|Color|Size|ProductModelID|
+--------------+-----+----+--------------+
|Classic Vest, S| Blue|   S|             1|
|Classic Vest, M| Blue|   M|             1|
|Classic Vest, L| Blue|   L|             1|
+--------------+-----+----+--------------+
```

### Running SQL Queries Programmatically

```
df.createOrReplaceTempView("df_table")
sqlContext.sql(" SELECT * FROM df_table  WHERE ProductModelID = 1").show()
```

```
+--------------+-----+----+--------------+
|          Name|Color|Size|ProductModelID|
+--------------+-----+----+--------------+
|Classic Vest, S| Blue|   S|             1|
|Classic Vest, M| Blue|   M|             1|
|Classic Vest, L| Blue|   L|             1|
+--------------+-----+----+--------------+
```

### 4. Retrieve the product number and name of the products that have a color of 'black', 'red', or 'white' and a size of 'S' or 'M'

### RDD way

```
colors = ["White","Black","Red"]
sizes = ["S","M"]

(content.filter(lambda line: line.split("\t")[6] in sizes)
.filter(lambda line: line.split("\t")[3] in colors)
.map(lambda line: (line.split("\t")[1],line.split("\t")[2], line.split("\t")[3],line.split
("\t")[6]))
 .collect()
)
```

```
[('Mountain Bike Socks, M', 'SO-B909-M', 'White', 'M'),
 ("Men's Sports Shorts, S", 'SH-M897-S', 'Black', 'S'),
 ("Men's Sports Shorts, M", 'SH-M897-M', 'Black', 'M'),
 ("Women's Tights, S", 'TG-W091-S', 'Black', 'S'),
 ("Women's Tights, M", 'TG-W091-M', 'Black', 'M'),
 ('Half-Finger Gloves, S', 'GL-H102-S', 'Black', 'S'),
 ('Half-Finger Gloves, M', 'GL-H102-M', 'Black', 'M'),
 ('Full-Finger Gloves, S', 'GL-F110-S', 'Black', 'S'),
 ('Full-Finger Gloves, M', 'GL-F110-M', 'Black', 'M'),
 ("Women's Mountain Shorts, S", 'SH-W890-S', 'Black', 'S'),
 ("Women's Mountain Shorts, M", 'SH-W890-M', 'Black', 'M'),
 ('Racing Socks, M', 'SO-R809-M', 'White', 'M')]
```

### DataFrame way

```
rdd = content.map(lambda line: (line.split("\t")[1],line.split("\t")[2], line.split("\t")[
3],line.split("\t")[6])).collect()
df = sqlContext.createDataFrame(rdd, schema = ["Name","ProductNumber","Color", "Size"])
```

```
colors = ["White","Black","Red"]
sizes = ["S","M"]
df.filter(col("Color").isin(colors) & col("Size").isin(sizes)).show()
```

```
+--------------------+-------------+-----+----+
|                Name|ProductNumber|Color|Size|
+--------------------+-------------+-----+----+
|Mountain Bike Soc...|    SO-B909-M|White|   M|
|Men's Sports Shor...|    SH-M897-S|Black|   S|
|Men's Sports Shor...|    SH-M897-M|Black|   M|
|   Women's Tights, S|    TG-W091-S|Black|   S|
|   Women's Tights, M|    TG-W091-M|Black|   M|
|Half-Finger Glove...|    GL-H102-S|Black|   S|
|Half-Finger Glove...|    GL-H102-M|Black|   M|
|Full-Finger Glove...|    GL-F110-S|Black|   S|
|Full-Finger Glove...|    GL-F110-M|Black|   M|
|Women's Mountain ...|    SH-W890-S|Black|   S|
|Women's Mountain ...|    SH-W890-M|Black|   M|
|     Racing Socks, M|    SO-R809-M|White|   M|
+--------------------+-------------+-----+----+
```

### Running SQL Queries Programmatically

```
df.createOrReplaceTempView("df_table")
sqlContext.sql(" SELECT * FROM df_table  WHERE Color IN ('White','Black','Red') AND Size I
N ('S','M')").show(truncate = False)
```

```
+-------------------------+-------------+-----+----+
|Name                     |ProductNumber|Color|Size|
+-------------------------+-------------+-----+----+
|Mountain Bike Socks, M   |SO-B909-M    |White|M   |
|Men's Sports Shorts, S   |SH-M897-S    |Black|S   |
|Men's Sports Shorts, M   |SH-M897-M    |Black|M   |
|Women's Tights, S        |TG-W091-S    |Black|S   |
|Women's Tights, M        |TG-W091-M    |Black|M   |
|Half-Finger Gloves, S    |GL-H102-S    |Black|S   |
|Half-Finger Gloves, M    |GL-H102-M    |Black|M   |
|Full-Finger Gloves, S    |GL-F110-S    |Black|S   |
|Full-Finger Gloves, M    |GL-F110-M    |Black|M   |
|Women's Mountain Shorts, S|SH-W890-S    |Black|S   |
|Women's Mountain Shorts, M|SH-W890-M    |Black|M   |
|Racing Socks, M          |SO-R809-M    |White|M   |
+-------------------------+-------------+-----+----+
```

### 5. Retrieve the product number, name, and list price of products whose product number begins with 'BK-'

### RDD way

```
(content.filter(lambda line: "BK" in line.split("\t")[2])
 .map(lambda line: (line.split("\t")[1],line.split("\t")[2], line.split("\t")[3],float(lin
e.split("\t")[5])))
.takeOrdered(10, lambda x: -x[3]))   # Displaying the heaviest 10
```

```
[('Road-150 Red, 62', 'BK-R93R-62', 'Red', 3578.27),
 ('Road-150 Red, 44', 'BK-R93R-44', 'Red', 3578.27),
 ('Road-150 Red, 48', 'BK-R93R-48', 'Red', 3578.27),
 ('Road-150 Red, 52', 'BK-R93R-52', 'Red', 3578.27),
 ('Road-150 Red, 56', 'BK-R93R-56', 'Red', 3578.27),
 ('Mountain-100 Silver, 38', 'BK-M82S-38', 'Silver', 3399.99),
 ('Mountain-100 Silver, 42', 'BK-M82S-42', 'Silver', 3399.99),
 ('Mountain-100 Silver, 44', 'BK-M82S-44', 'Silver', 3399.99),
 ('Mountain-100 Silver, 48', 'BK-M82S-48', 'Silver', 3399.99),
 ('Mountain-100 Black, 38', 'BK-M82B-38', 'Black', 3374.99)]
```

### DataFrame way

```
rdd = content.map(lambda line: (line.split("\t")[1],line.split("\t")[2], line.split("\t")[
3],float(line.split("\t")[5])))

df = sqlContext.createDataFrame(rdd, schema = ["Name","ProductNumber","Color", "ListPrice"
])
```

Here, we can use the **re** python module with the PySpark's User Defined Functions (udf).

```
from pyspark.sql.functions import udf
from pyspark.sql.types import BooleanType

import re

def is_match(line):
    pattern = re.compile("^(BK-)")
    return(bool(pattern.match(line)))

filter_udf = udf(is_match, BooleanType())

df.filter(filter_udf(df.ProductNumber)).orderBy("ListPrice", ascending = False).show(10, t
runcate = False)
```

```
+----------------------+-------------+------+---------+
|Name                  |ProductNumber|Color |ListPrice|
+----------------------+-------------+------+---------+
|Road-150 Red, 44      |BK-R93R-44   |Red   |3578.27  |
|Road-150 Red, 62      |BK-R93R-62   |Red   |3578.27  |
|Road-150 Red, 52      |BK-R93R-52   |Red   |3578.27  |
|Road-150 Red, 56      |BK-R93R-56   |Red   |3578.27  |
|Road-150 Red, 48      |BK-R93R-48   |Red   |3578.27  |
|Mountain-100 Silver, 48|BK-M82S-48   |Silver|3399.99  |
|Mountain-100 Silver, 44|BK-M82S-44   |Silver|3399.99  |
|Mountain-100 Silver, 42|BK-M82S-42   |Silver|3399.99  |
|Mountain-100 Silver, 38|BK-M82S-38   |Silver|3399.99  |
|Mountain-100 Black, 44 |BK-M82B-44   |Black |3374.99  |
+----------------------+-------------+------+---------+
only showing top 10 rows
```

### Running SQL Queries Programmatically

```
df.createOrReplaceTempView("df_table")
sqlContext.sql(" SELECT * FROM df_table  WHERE ProductNumber LIKE 'BK-%' ORDER BY ListPric
e DESC ").show(n = 10)
```

```
+-------------------+-------------+------+---------+
|               Name|ProductNumber| Color|ListPrice|
+-------------------+-------------+------+---------+
|    Road-150 Red, 44|   BK-R93R-44|   Red|  3578.27|
|    Road-150 Red, 62|   BK-R93R-62|   Red|  3578.27|
|    Road-150 Red, 52|   BK-R93R-52|   Red|  3578.27|
|    Road-150 Red, 56|   BK-R93R-56|   Red|  3578.27|
|    Road-150 Red, 48|   BK-R93R-48|   Red|  3578.27|
|Mountain-100 Silv...|   BK-M82S-48|Silver|  3399.99|
|Mountain-100 Silv...|   BK-M82S-44|Silver|  3399.99|
|Mountain-100 Silv...|   BK-M82S-42|Silver|  3399.99|
|Mountain-100 Silv...|   BK-M82S-38|Silver|  3399.99|
|Mountain-100 Blac...|   BK-M82B-44| Black|  3374.99|
+-------------------+-------------+------+---------+
only showing top 10 rows
```

**6. Modify your previous query to retrieve the product number, name, and list price of products whose product number begins 'BK-' followed by any character other than 'R', and ends with a '-' followed by any two numerals.**

```
def is_match(line):
    pattern = re.compile("^(BK-)[^R]+(-\d{2})$")
    return(bool(pattern.match(line)))
```

Let's check our function.

```
is_match("BK-M82S-38")
```

```
True
```

#### RDD way

```
(content.filter(lambda line: is_match(line.split("\t")[2]))
.map(lambda line: (line.split("\t")[1],line.split("\t")[2], line.split("\t")[3],float(line
.split("\t")[5])))
 .takeOrdered(10, lambda x: -x[3]))   # Displaying the heaviest 10
```

```
[('Mountain-100 Silver, 38', 'BK-M82S-38', 'Silver', 3399.99),
 ('Mountain-100 Silver, 42', 'BK-M82S-42', 'Silver', 3399.99),
 ('Mountain-100 Silver, 44', 'BK-M82S-44', 'Silver', 3399.99),
 ('Mountain-100 Silver, 48', 'BK-M82S-48', 'Silver', 3399.99),
 ('Mountain-100 Black, 38', 'BK-M82B-38', 'Black', 3374.99),
 ('Mountain-100 Black, 42', 'BK-M82B-42', 'Black', 3374.99),
 ('Mountain-100 Black, 44', 'BK-M82B-44', 'Black', 3374.99),
 ('Mountain-100 Black, 48', 'BK-M82B-48', 'Black', 3374.99),
 ('Touring-1000 Yellow, 46', 'BK-T79Y-46', 'Yellow', 2384.07),
 ('Touring-1000 Yellow, 50', 'BK-T79Y-50', 'Yellow', 2384.07)]
```

#### DataFrame way

```
filter_udf = udf(is_match, BooleanType())

df.filter(filter_udf(df.ProductNumber)).orderBy("ListPrice", ascending = False).show(10, t
runcate = False)
```

```
+----------------------+-------------+------+---------+
|Name                  |ProductNumber|Color |ListPrice|
+----------------------+-------------+------+---------+
|Mountain-100 Silver, 44|BK-M82S-44   |Silver|3399.99  |
|Mountain-100 Silver, 48|BK-M82S-48   |Silver|3399.99  |
|Mountain-100 Silver, 38|BK-M82S-38   |Silver|3399.99  |
|Mountain-100 Silver, 42|BK-M82S-42   |Silver|3399.99  |
|Mountain-100 Black, 42 |BK-M82B-42   |Black |3374.99  |
|Mountain-100 Black, 48 |BK-M82B-48   |Black |3374.99  |
|Mountain-100 Black, 44 |BK-M82B-44   |Black |3374.99  |
|Mountain-100 Black, 38 |BK-M82B-38   |Black |3374.99  |
|Touring-1000 Blue, 54  |BK-T79U-54   |Blue  |2384.07  |
|Touring-1000 Blue, 50  |BK-T79U-50   |Blue  |2384.07  |
+----------------------+-------------+------+---------+
only showing top 10 rows
```

**This is enough for today. See you in the next part of the DataFrames Vs RDDs in Spark tutorial series.**

# Spark RDDs Vs DataFrames vs SparkSQL - Part 2 : Working With Multiple Tables

This is the second tutorial on the Spark RDDs Vs DataFrames vs SparkSQL blog post series. The first one is available here (http://datascience-enthusiast.com/Python/DataFramesVsRDDsVsSQLSpark-Part1.html). In the first part, I showed how to retrieve, sort and filter data using Spark RDDs, DataFrames and SparkSQL. In this tutorial, we will see how to work with multiple tables in Spark the RDD way, the DataFrame way and with SparkSQL.

If you like this tutorial series, check also my other recent blos posts on Spark on Analyzing the Bible and the Quran using Spark (http://datascience-enthusiast.com/Python/analyzing_bible_quran_with_spark.html) and Spark DataFrames: Exploring Chicago Crimes (http://datascience-enthusiast.com/Python/SparkDataFrames-ExploringChicagoCrimes.html). The data and the notebooks can be downloaded from my GitHub repository (https://github.com/fissehab/Spark_certification). The size of the data is not large, however, the same code works for large volume as well. Therefore, we can practice with this dataset to master the functinalities of Spark.

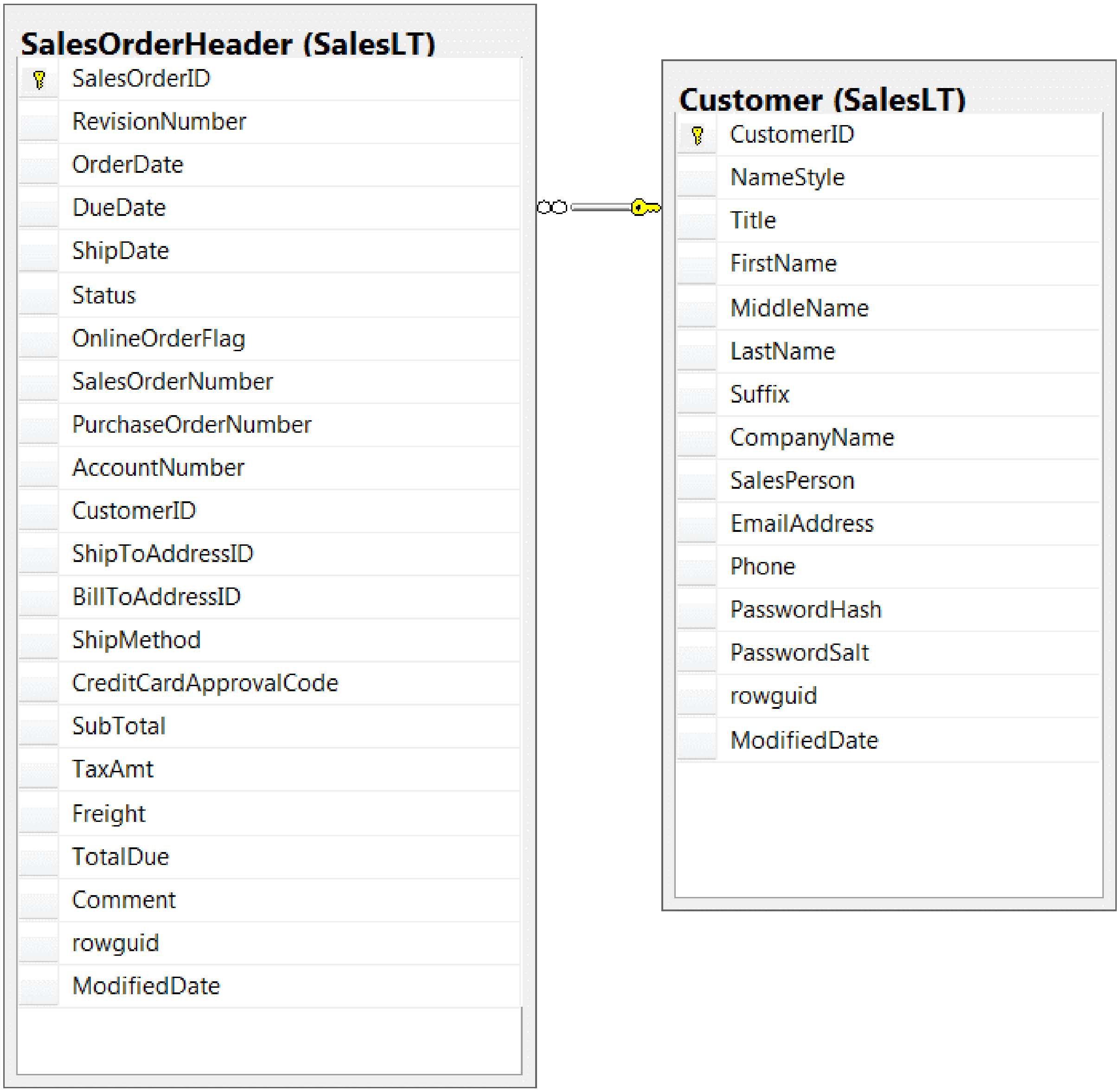
For this tutorial, we will work with the **SalesLTProduct.txt**,**SalesLTSalesOrderHeader.txt**, **SalesLTCustomer.txt**,**SalesLTAddress.txt** and **SalesLTCustomerAddress.txt** datasets. Let's answer a couple of questions using Spark Resilient Distiributed (RDD) way, DataFrame way and SparkSQL.

SparkContext is main entry point for Spark functionality.

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SQLContext


conf = SparkConf().setAppName("miniProject").setMaster("local[*]")
sc = SparkContext.getOrCreate(conf)

sqlcontext = SQLContext(sc)
```

## 1. Retrieve customer orders

As an initial step towards generating invoice report, write a query that returns the company name from the SalesLTCustomer.txt, and the sales order ID and total due from the SalesLTSalesOrderHeader.txt.




### RDD way

```
orderHeader = sc.textFile("SalesLTSalesOrderHeader.txt")
customer = sc.textFile("SalesLTCustomer.txt")
```

From the commnads below, we see that the first rows are column names and the datasets are tab delimited.

```
orderHeader.first()
```

```
'SalesOrderID\tRevisionNumber\tOrderDate\tDueDate\tShipDate\tStatus\tOnlineOr
derFlag\tSalesOrderNumber\tPurchaseOrderNumber\tAccountNumber\tCustomerID\tSh
ipToAddressID\tBillToAddressID\tShipMethod\tCreditCardApprovalCode\tSubTotal
\tTaxAmt\tFreight\tTotalDue\tComment\trowguid\tModifiedDate'
```

```
customer.first()
```

```
'CustomerID\tNameStyle\tTitle\tFirstName\tMiddleName\tLastName\tSuffix\tCompa
nyName\tSalesPerson\tEmailAddress\tPhone\tPasswordHash\tPasswordSalt\trowguid
\tModifiedDate'
```

Now, let's have the column names and the contents separated.

```
customer_header = customer.first()
customer_rdd = customer.filter(lambda line: line != customer_header)

orderHeader_header = orderHeader.first()
orderHeader_rdd = orderHeader.filter(lambda line: line != orderHeader_header)
```

customer_rdd and orderHeader_rdd are tab delimited as we can see it below.

```
customer_rdd.first()
```

```
'1\t0\tMr.\tOrlando\tN.\tGee\tNULL\tA Bike Store\tadventure-works\\pamela0\to
rlando0@adventure-works.com\t245-555-0173\tL/Rlwxzp4w7RWmEgXX+/A7cXaePEPcp+Kw
Qhl2fJL7w=\t1KjXYs4=\t3F5AE95E-B87D-4AED-95B4-C3797AFCB74F\t2001-08-01 00:00:
00.000'
```

We need only CustomerID and ComapanyName from the customers RDD. From the orderHeader RDD we need CustomerID,SalesOrderID and TotalDue then we are joining the two RDD using inner join.Finally, we are displaying 10 companies with the highest amout due.

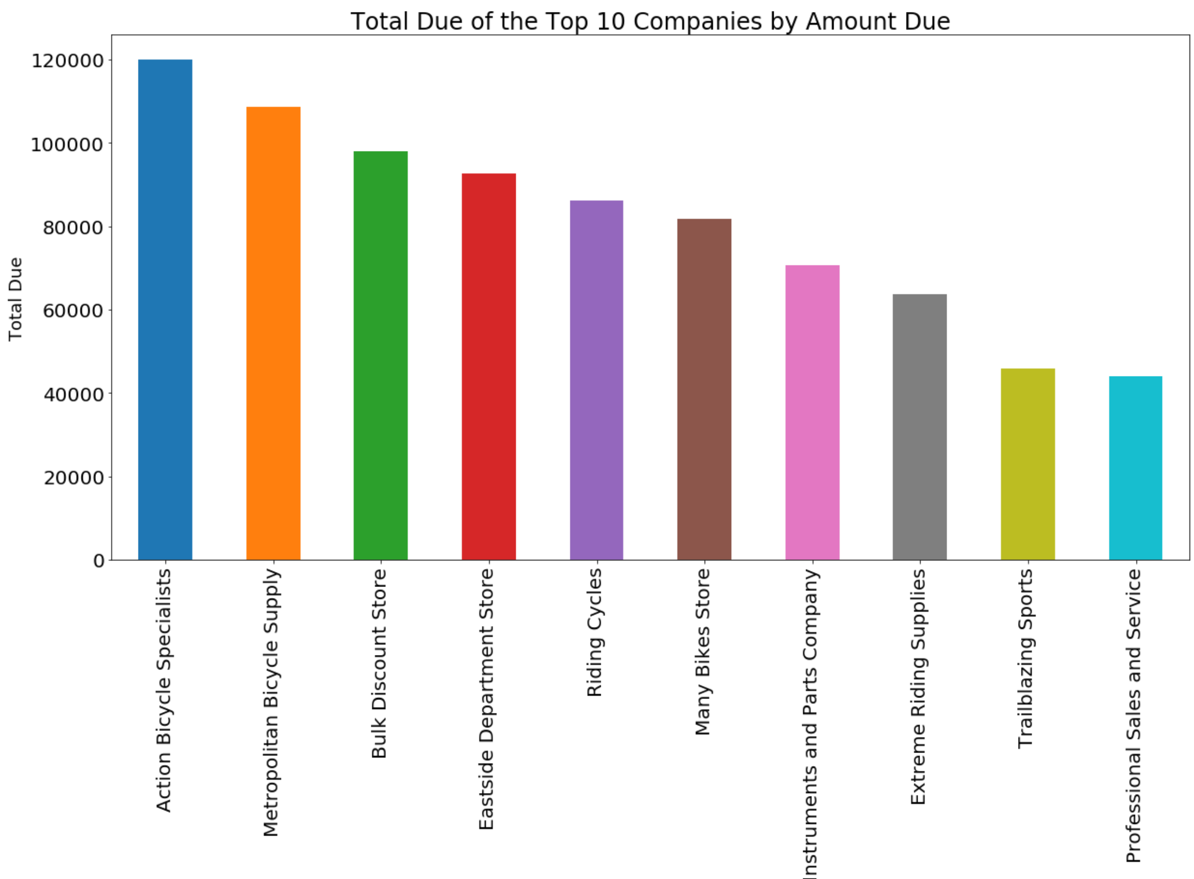
```
customer_rdd1 = customer_rdd.map(lambda line: (line.split("\t")[0], #CustomerID
                                                line.split("\t")[7]  #CompanyName
                                              ))

orderHeader_rdd1 = orderHeader_rdd.map(lambda line: (line.split("\t")[10], #CustomerID
                                                      (line.split("\t")[0], #SalesOrderID
                                                       float(line.split("\t")[-4]) # TotalD
ue
                                                      )))

invoice1 = customer_rdd1.join(orderHeader_rdd1)
invoice1.takeOrdered(10, lambda x: -x[1][1][1])
```

```
[('29736', ('Action Bicycle Specialists', ('71784', 119960.824))),
 ('30050', ('Metropolitan Bicycle Supply', ('71936', 108597.9536))),
 ('29546', ('Bulk Discount Store', ('71938', 98138.2131))),
 ('29957', ('Eastside Department Store', ('71783', 92663.5609))),
 ('29796', ('Riding Cycles', ('71797', 86222.8072))),
 ('29929', ('Many Bikes Store', ('71902', 81834.9826))),
 ('29932', ('Instruments and Parts Company', ('71898', 70698.9922))),
 ('29660', ('Extreme Riding Supplies', ('71796', 63686.2708))),
 ('29938', ('Trailblazing Sports', ('71845', 45992.3665))),
 ('29485', ('Professional Sales and Service', ('71782', 43962.7901)))]
```

If we want, once we collect the RDD resulting from our transformations and actions, we can use other Python packages to visualize our data.

```
import pandas as pd
top10 = invoice1.takeOrdered(10, lambda x: -x[1][1][1])
companies = [x[1][0] for x in top10]
total_due = [x[1][1][1] for x in top10]
top10_dict = {"companies": companies, "total_due":total_due}
top10_pd = pd.DataFrame(top10_dict)
```

```
import matplotlib.pyplot as plt
%matplotlib inline

top10_pd.plot(figsize = (20, 10),kind = "bar", legend  = False, x = "companies", y = "tota
l_due")
plt.xlabel("")
plt.ylabel("Total Due", fontsize = 18)
plt.title("Total Due of the Top 10 Companies by Amount Due", fontsize  = 24)
plt.xticks(size = 20)
plt.yticks(size = 20)
plt.show()
```




### DataFrame way

First, we create DataFrames from the RDDs by using the first row as schema.

```
customer_df = sqlcontext.createDataFrame(customer_rdd.map(lambda line: line.split("\t")),
                                        schema = customer.first().split("\t"))

orderHeader_df = sqlcontext.createDataFrame(orderHeader_rdd.map(lambda line: line.split("
\t")),
                                        schema = orderHeader.first().split("\t"))
```

We can select some columns and display some rows.

```
customer_df.select(["CustomerID",'CompanyName',"FirstName","MiddleName", "LastName"]).show
(10, truncate = False)
```

```
+----------+--------------------------+---------+----------+----------+
|CustomerID|CompanyName               |FirstName|MiddleName|LastName  |
+----------+--------------------------+---------+----------+----------+
|1         |A Bike Store              |Orlando  |N.        |Gee       |
|2         |Progressive Sports        |Keith    |NULL      |Harris    |
|3         |Advanced Bike Components  |Donna    |F.        |Carreras  |
|4         |Modular Cycle Systems     |Janet    |M.        |Gates     |
|5         |Metropolitan Sports Supply|Lucy     |NULL      |Harrington|
|6         |Aerobic Exercise Company  |Rosmarie |J.        |Carroll   |
|7         |Associated Bikes          |Dominic  |P.        |Gash      |
|10        |Rural Cycle Emporium      |Kathleen |M.        |Garza     |
|11        |Sharp Bikes               |Katherine|NULL      |Harding   |
|12        |Bikes and Motorbikes      |Johnny   |A.        |Caprio    |
+----------+--------------------------+---------+----------+----------+
only showing top 10 rows
```

Now, let's join the two DataFrames using the CustomerID column. We need to use inner join here. We are ordering the rows by TotalDue column in descending order but our result does not look normal. As we can see from the schema of the joined DataFrame, the TotalDue column is string. Therefore, we have to change that column to numeric field.

```
joined = customer_df.join(orderHeader_df, 'CustomerID', how = "inner")
joined.select(["CustomerID", 'CompanyName','SalesOrderID','TotalDue']).orderBy("TotalDue",
 ascending = False).show(10, truncate = False)
```

```
+----------+-----------------------------+------------+----------+
|CustomerID|CompanyName                  |SalesOrderID|TotalDue  |
+----------+-----------------------------+------------+----------+
|29546     |Bulk Discount Store          |71938       |98138.2131|
|29847     |Good Toys                    |71774       |972.785   |
|29957     |Eastside Department Store    |71783       |92663.5609|
|30072     |West Side Mart               |71776       |87.0851   |
|29796     |Riding Cycles                |71797       |86222.8072|
|29929     |Many Bikes Store             |71902       |81834.9826|
|29531     |Remarkable Bike Store        |71935       |7330.8972 |
|29932     |Instruments and Parts Company|71898       |70698.9922|
|30033     |Transport Bikes              |71856       |665.4251  |
|29660     |Extreme Riding Supplies      |71796       |63686.2708|
+----------+-----------------------------+------------+----------+
only showing top 10 rows
```

```
joined.printSchema()
```

```
root
 |-- CustomerID: string (nullable = true)
 |-- NameStyle: string (nullable = true)
 |-- Title: string (nullable = true)
 |-- FirstName: string (nullable = true)
 |-- MiddleName: string (nullable = true)
 |-- LastName: string (nullable = true)
 |-- Suffix: string (nullable = true)
 |-- CompanyName: string (nullable = true)
 |-- SalesPerson: string (nullable = true)
 |-- EmailAddress: string (nullable = true)
 |-- Phone: string (nullable = true)
 |-- PasswordHash: string (nullable = true)
 |-- PasswordSalt: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: string (nullable = true)
 |-- SalesOrderID: string (nullable = true)
 |-- RevisionNumber: string (nullable = true)
 |-- OrderDate: string (nullable = true)
 |-- DueDate: string (nullable = true)
 |-- ShipDate: string (nullable = true)
 |-- Status: string (nullable = true)
 |-- OnlineOrderFlag: string (nullable = true)
 |-- SalesOrderNumber: string (nullable = true)
 |-- PurchaseOrderNumber: string (nullable = true)
 |-- AccountNumber: string (nullable = true)
 |-- ShipToAddressID: string (nullable = true)
 |-- BillToAddressID: string (nullable = true)
 |-- ShipMethod: string (nullable = true)
 |-- CreditCardApprovalCode: string (nullable = true)
 |-- SubTotal: string (nullable = true)
 |-- TaxAmt: string (nullable = true)
 |-- Freight: string (nullable = true)
 |-- TotalDue: string (nullable = true)
 |-- Comment: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: string (nullable = true)
```

```
from pyspark.sql.functions import col, udf
from pyspark.sql.types import DoubleType
convert = udf(lambda x: float(x), DoubleType())
```

Now, let's change the TotalDue column to numeric.

```
joined2 = joined.withColumn('Total_Due',convert(col("TotalDue"))).drop("TotalDue")
joined2.dtypes[-1]  # we have created a new column with double type
```

```
('Total_Due', 'double')
```

```
joined2.select(["CustomerID", 'CompanyName','SalesOrderID','Total_Due'])\
.orderBy("Total_Due", ascending = False).show(10, truncate = False)
```

```
+----------+------------------------------+------------+-----------+
|CustomerID|CompanyName                   |SalesOrderID|Total_Due  |
+----------+------------------------------+------------+-----------+
|29736     |Action Bicycle Specialists    |71784       |119960.824 |
|30050     |Metropolitan Bicycle Supply   |71936       |108597.9536|
|29546     |Bulk Discount Store           |71938       |98138.2131 |
|29957     |Eastside Department Store     |71783       |92663.5609 |
|29796     |Riding Cycles                 |71797       |86222.8072 |
|29929     |Many Bikes Store              |71902       |81834.9826 |
|29932     |Instruments and Parts Company |71898       |70698.9922 |
|29660     |Extreme Riding Supplies       |71796       |63686.2708 |
|29938     |Trailblazing Sports           |71845       |45992.3665 |
|29485     |Professional Sales and Service|71782       |43962.7901 |
+----------+------------------------------+------------+-----------+
only showing top 10 rows
```

The result above is the same as the result we got using the RDD way above.

### Running SQL Queries Programmatically

First, let's create a local temporary view of the DataFrames and the we can use normal SQL commands to get the 10 companies with the highest amount due.

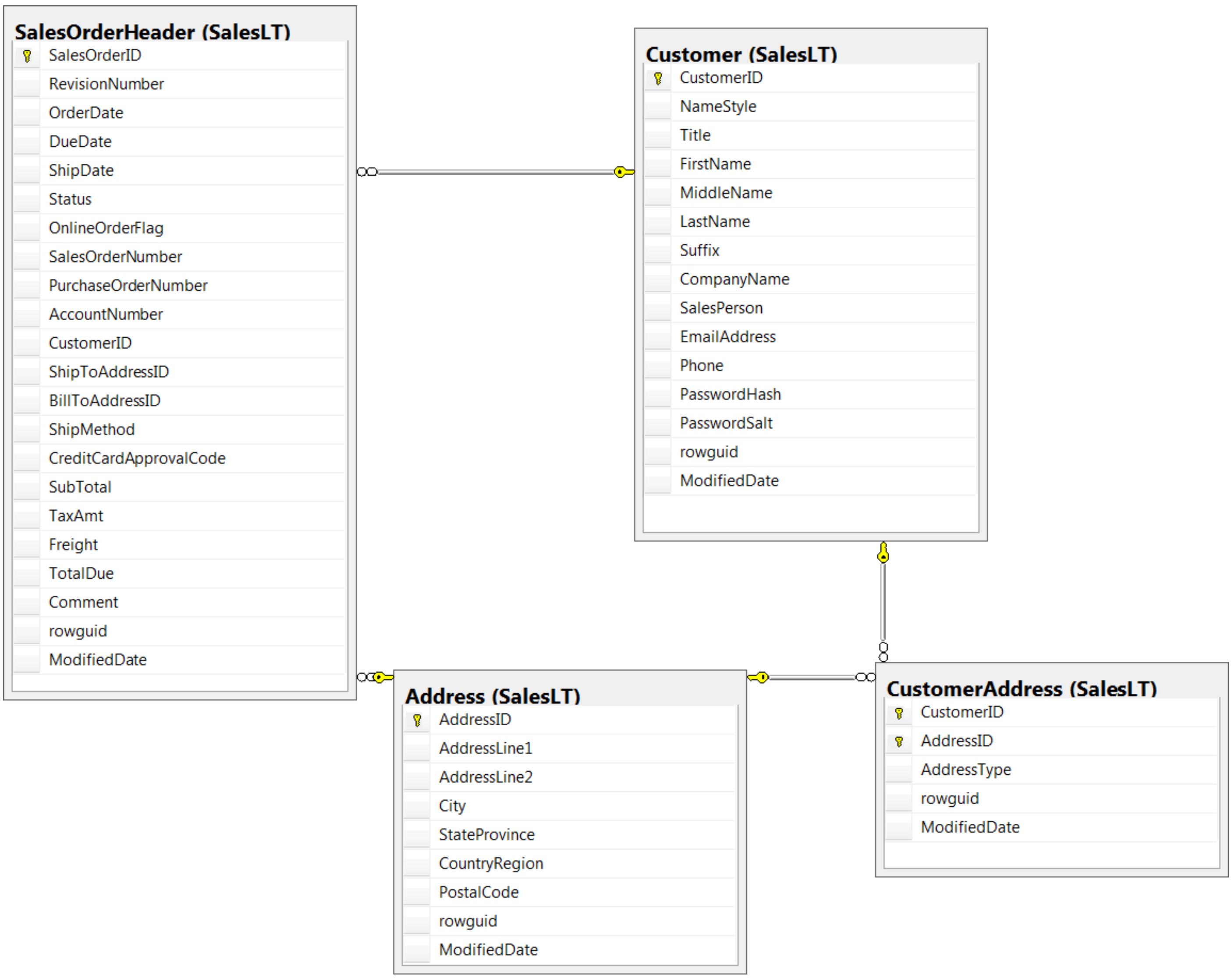
```
orderHeader_df.createOrReplaceTempView("orderHeader_table")
customer_df.createOrReplaceTempView("customer_table")

sqlcontext.sql("SELECT c.CustomerID, c.CompanyName,oh.SalesOrderID,cast(oh.TotalDue AS DEC
IMAL(10,4)) \
               FROM customer_table AS c INNER JOIN orderHeader_table AS OH ON c.CustomerID
=oh.CustomerID \
               ORDER BY TotalDue DESC LIMIT 10").show(10, truncate = False)
```

```
+----------+------------------------------+------------+-----------+
|CustomerID|CompanyName                   |SalesOrderID|TotalDue   |
+----------+------------------------------+------------+-----------+
|29736     |Action Bicycle Specialists    |71784       |119960.8240|
|30050     |Metropolitan Bicycle Supply   |71936       |108597.9536|
|29546     |Bulk Discount Store           |71938       |98138.2131 |
|29957     |Eastside Department Store     |71783       |92663.5609 |
|29796     |Riding Cycles                 |71797       |86222.8072 |
|29929     |Many Bikes Store              |71902       |81834.9826 |
|29932     |Instruments and Parts Company |71898       |70698.9922 |
|29660     |Extreme Riding Supplies       |71796       |63686.2708 |
|29938     |Trailblazing Sports           |71845       |45992.3665 |
|29485     |Professional Sales and Service|71782       |43962.7901 |
+----------+------------------------------+------------+-----------+
```

We see that the results we got using the above three methods, RDD way, DataFrame and with SparkSQL, are the same.

## 2. Retrieve customer orders with addresses

Extend your customer orders query to include the Main Office address for each customer, including the full street address, city, state or province, and country or region. Note that each customer can have multiple addressees in the SalesLTAddress.txt, so the SalesLTCustomerAddress.txt dataset enables a many-to-many relationship between customers and addresses. Your query will need to include both of these datasets, and should filter the join to SalesLTCustomerAddress.txt so that only Main Office addresses are included.




### RDD way

I am not repeating some of the steps, I did in question 1 above.

As we can see below, the datasets for this question are also tab delimited.

```
address = sc.textFile("SalesLTAddress.txt")
customer_address = sc.textFile("SalesLTCustomerAddress.txt")
```

```
customer_address.first()
```

'CustomerID\tAddressID\tAddressType\trowguid\tModifiedDate'

```
address.first()
```

'AddressID\tAddressLine1\tAddressLine2\tCity\tStateProvince\tCountryRegion\tP
ostalCode\trowguid\tModifiedDate'

Removing the column names from the RDDs.

```
address_header = address.first()
address_rdd = address.filter(lambda line: line != address_header )

customer_address_header = customer_address.first()
customer_address_rdd = customer_address.filter(lambda line: line != customer_address_heade
r)
```

Include only those with AddressType of Main Office.

Split the lines and retain only fields of interest.

```
customer_address_rdd1 = customer_address_rdd.filter(lambda line: line.split("\t")[2] =="Ma
in Office").map(lambda line: (line.split("\t")[0],     #CustomerID
                                                               line.split("\t")[1],    #Ad
dressID
                                                               ))

address_rdd1 = address_rdd.map(lambda line: (line.split("\t")[0], #AddressID
                                                               (line.split("\t")[1],  #Add
ressLine1
                                                                 line.split("\t")[3], #Cit
y
                                                                 line.split("\t")[4],  #St
ateProvince
                                                                  line.split("\t")[5] #Cou
ntryRegion
                                                               )))
```

We can now join them.

```
rdd = customer_rdd1.join(orderHeader_rdd1).join(customer_address_rdd1).map(lambda line: (l
ine[1][1], # AddressID
                                                                                        (l
ine[1][0][0], # Company
                                                                                        li
ne[1][0][1][0],# SalesOrderID
                                                                                        li
ne[1][0][1][1]# TotalDue

)))
final_rdd = rdd.join(address_rdd1)
```

```
final_rdd.first()
```

```
('993',
 (('Coalition Bike Company', '71899', 2669.3183),
  ('Corporate Office', 'El Segundo', 'California', 'United States')))
```

Let's rearrange the columns.

```
final_rdd2 = final_rdd.map(lambda line: (line[1][0][0],                  # company
                                         float(line[1][0][2]),           # TotalDue
                                         line[1][1][0],                  # Address 1
                                         line[1][1][1],                  # City
                                         line[1][1][2],                  # StateProvince
                                         line[1][1][3]                   # CountryRegion
                                             ))
```

Let's see 10 companies with the highest amount due.

```
final_rdd2.takeOrdered(10, lambda x: -x[1])
```

```
[('Action Bicycle Specialists',
  119960.824,
  'Warrington Ldc Unit 25/2',
  'Woolston',
  'England',
  'United Kingdom'),
 ('Metropolitan Bicycle Supply',
  108597.9536,
  'Paramount House',
  'London',
  'England',
  'United Kingdom'),
 ('Bulk Discount Store',
  98138.2131,
  '93-2501, Blackfriars Road,',
  'London',
  'England',
  'United Kingdom'),
 ('Eastside Department Store',
  92663.5609,
  '9992 Whipple Rd',
  'Union City',
  'California',
  'United States'),
 ('Riding Cycles',
  86222.8072,
  'Galashiels',
  'Liverpool',
  'England',
  'United Kingdom'),
 ('Many Bikes Store',
  81834.9826,
  'Receiving',
  'Fullerton',
  'California',
  'United States'),
 ('Instruments and Parts Company',
  70698.9922,
  'Phoenix Way, Cirencester',
  'Gloucestershire',
  'England',
  'United Kingdom'),
 ('Extreme Riding Supplies',
  63686.2708,
  'Riverside',
  'Sherman Oaks',
  'California',
  'United States'),
 ('Trailblazing Sports',
  45992.3665,
  '251340 E. South St.',
  'Cerritos',
  'California',
  'United States'),
 ('Professional Sales and Service',
  43962.7901,
  '57251 Serene Blvd',
```

```
  'Van Nuys',
  'California',
  'United States')]
```

**DataFrame Way**

Now, can create DataFrames from the RDDs and perform the joins.

```
address_df = sqlcontext.createDataFrame(address_rdd.map(lambda line: line.split("\t")),
                                        schema = address_header.split("\t") )

customer_address_df = sqlcontext.createDataFrame(customer_address_rdd .map(lambda line: li
ne.split("\t")),
                                        schema = customer_address_header.split("\t") )
```

We can see the schema of each DataFrame.

```
address_df.printSchema()
```

```
root
 |-- AddressID: string (nullable = true)
 |-- AddressLine1: string (nullable = true)
 |-- AddressLine2: string (nullable = true)
 |-- City: string (nullable = true)
 |-- StateProvince: string (nullable = true)
 |-- CountryRegion: string (nullable = true)
 |-- PostalCode: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: string (nullable = true)
```

```
customer_address_df.printSchema()
```

```
root
 |-- CustomerID: string (nullable = true)
 |-- AddressID: string (nullable = true)
 |-- AddressType: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: string (nullable = true)
```

Now, we can finally join the DataFrames but to order the rows based on the total amount due, we have to first convert that column to numeric.

```
joined = (customer_df.join(orderHeader_df, 'CustomerID', how = "inner")
          .join(customer_address_df,'CustomerID', how = "inner" )
          .join(address_df,'AddressID', how = "inner" ))

joined2 = joined.withColumn('Total_Due',convert(col("TotalDue"))).drop("TotalDue").filter(
joined['AddressType']=="Main Office")
```

```
joined2.select(['CompanyName','Total_Due',
                'AddressLine1','City',
                'StateProvince','CountryRegion']).orderBy('Total_Due', ascending = False).
show(10, truncate = False)
```

```
+------------------------------+-----------+-------------------------+------
---------+------------+-------------+
|CompanyName                   |Total_Due  |AddressLine1             |City
         |StateProvince|CountryRegion |
+------------------------------+-----------+-------------------------+------
---------+------------+-------------+
|Action Bicycle Specialists    |119960.824 |Warrington Ldc Unit 25/2  |Woolst
on       |England      |United Kingdom|
|Metropolitan Bicycle Supply   |108597.9536|Paramount House          |London
         |England      |United Kingdom|
|Bulk Discount Store           |98138.2131 |93-2501, Blackfriars Road,|London
         |England      |United Kingdom|
|Eastside Department Store     |92663.5609 |9992 Whipple Rd          |Union
City     |California   |United States |
|Riding Cycles                 |86222.8072 |Galashiels               |Liverp
ool      |England      |United Kingdom|
|Many Bikes Store              |81834.9826 |Receiving                |Fuller
ton      |California   |United States |
|Instruments and Parts Company |70698.9922 |Phoenix Way, Cirencester  |Glouce
stershire|England      |United Kingdom|
|Extreme Riding Supplies       |63686.2708 |Riverside                |Sherma
n Oaks   |California   |United States |
|Trailblazing Sports           |45992.3665 |251340 E. South St.      |Cerrit
os       |California   |United States |
|Professional Sales and Service|43962.7901 |57251 Serene Blvd        |Van Nu
ys       |California   |United States |
+------------------------------+-----------+-------------------------+------
---------+------------+-------------+
only showing top 10 rows
```

The answer using the RDD way is the same as the answer we got above using the RDD way.

**Running SQL Queries Programmatically**

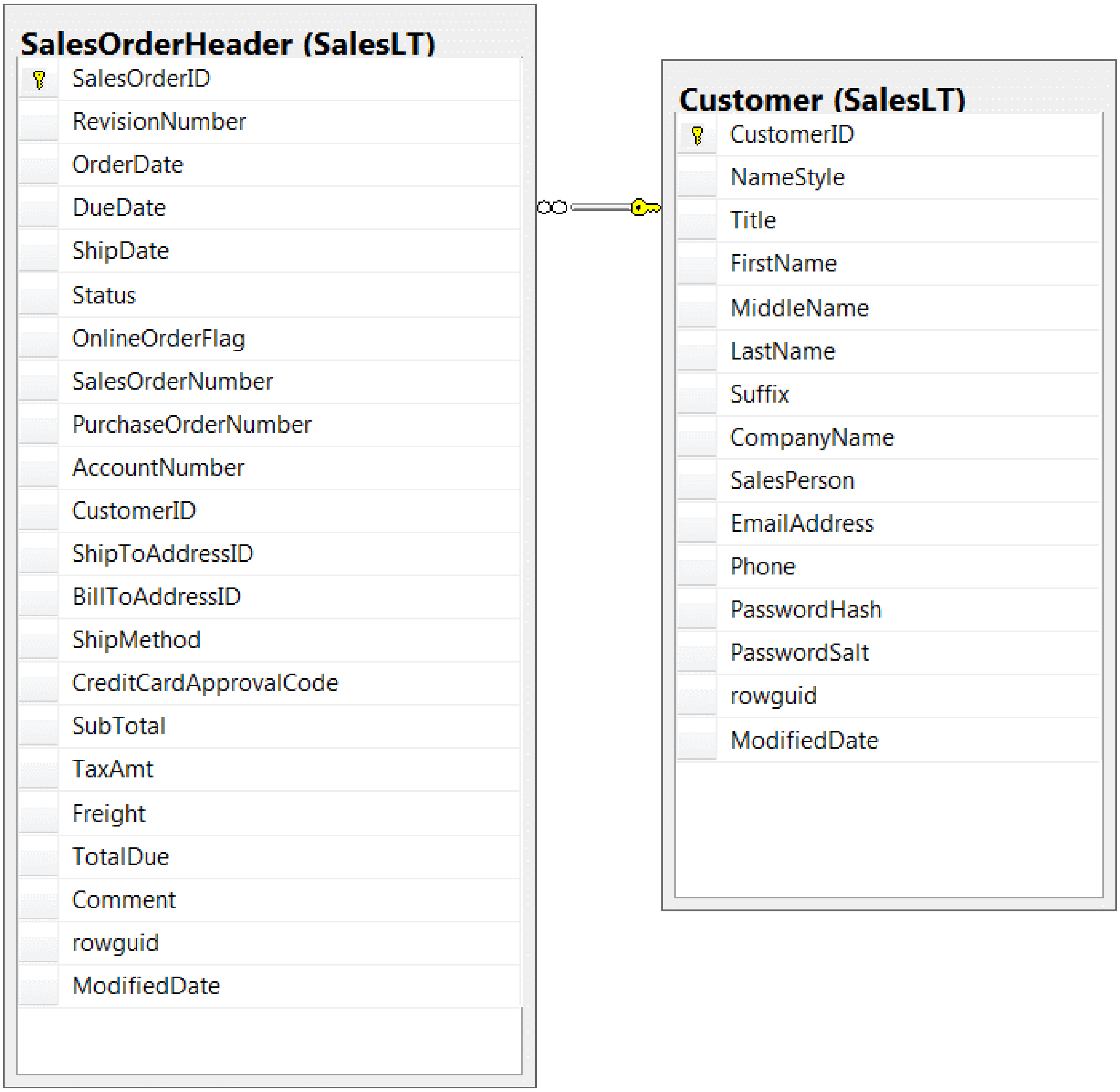
As shown below, the answer using SQL, after creating a local temporary view, also gives the same answer as the RDD way and DataFrame way above.

```
address_df.createOrReplaceTempView("address_table")

customer_address_df.createOrReplaceTempView("customer_address_table")

sqlcontext.sql("SELECT c.CompanyName,cast(oh.TotalDue AS DECIMAL(10,4)), a.AddressLine1, \
               a.City, a.StateProvince, a.CountryRegion FROM customer_table AS c \
               INNER JOIN orderHeader_table AS oh ON c.CustomerID = oh.CustomerID \
               INNER JOIN customer_address_table AS ca  ON c.CustomerID = ca.CustomerID AN
D AddressType = 'Main Office' \
              INNER JOIN address_table AS a ON ca.AddressID = a.AddressID \
              ORDER BY TotalDue DESC LIMIT 10").show(truncate = False)
```

```
+------------------------------+-----------+--------------------------+------
---------+------------+-------------+
|CompanyName                   |TotalDue   |AddressLine1              |City
         |StateProvince|CountryRegion |
+------------------------------+-----------+--------------------------+------
---------+------------+-------------+
|Action Bicycle Specialists    |119960.8240|Warrington Ldc Unit 25/2  |Woolst
on       |England      |United Kingdom|
|Metropolitan Bicycle Supply   |108597.9536|Paramount House           |London
         |England      |United Kingdom|
|Bulk Discount Store           |98138.2131 |93-2501, Blackfriars Road,|London
         |England      |United Kingdom|
|Eastside Department Store     |92663.5609 |9992 Whipple Rd           |Union
City     |California   |United States |
|Riding Cycles                 |86222.8072 |Galashiels                |Liverp
ool      |England      |United Kingdom|
|Many Bikes Store              |81834.9826 |Receiving                 |Fuller
ton      |California   |United States |
|Instruments and Parts Company |70698.9922 |Phoenix Way, Cirencester  |Glouce
stershire|England      |United Kingdom|
|Extreme Riding Supplies       |63686.2708 |Riverside                 |Sherma
n Oaks   |California   |United States |
|Trailblazing Sports           |45992.3665 |251340 E. South St.       |Cerrit
os       |California   |United States |
|Professional Sales and Service|43962.7901 |57251 Serene Blvd         |Van Nu
ys       |California   |United States |
+------------------------------+-----------+--------------------------+------
---------+------------+-------------+
```

## 3. Retrieve a list of all customers and their orders

The sales manager wants a list of all customer companies and their contacts (first name and last name), showing the sales order ID and total due for each order they have placed. Customers who have not placed any orders should be included at the bottom of the list with NULL values for the order ID and total due.




**RDD way**

Let's create the RDDs, select the fields of interest and join them

```
customer_header = customer.first()
customer_rdd = customer.filter(lambda line: line != customer_header)

orderHeader_header = orderHeader.first()
orderHeader_rdd = orderHeader.filter(lambda line: line != orderHeader_header)
```

```
orderHeader_header
```

```
'SalesOrderID\tRevisionNumber\tOrderDate\tDueDate\tShipDate\tStatus\tOnlineOr
derFlag\tSalesOrderNumber\tPurchaseOrderNumber\tAccountNumber\tCustomerID\tSh
ipToAddressID\tBillToAddressID\tShipMethod\tCreditCardApprovalCode\tSubTotal
\tTaxAmt\tFreight\tTotalDue\tComment\trowguid\tModifiedDate'
```

```
customer_rdd1 = customer_rdd.map(lambda line: (line.split("\t")[0], #CustomerID
                                              (line.split("\t")[3], #FirstName
                                               line.split("\t")[5] #LastName
                                              )))

orderHeader_rdd1 = orderHeader_rdd.map(lambda line: (line.split("\t")[10], # CustomerID
                                                    (line.split("\t")[0],  # SalesOrderID
                                                     line.split("\t")[-4]   # TotalDue
                                                    )))
```

We have to re-arrange customers that have made orders and those that have not ordered separetly and then uinon them at the end.

```
joined = customer_rdd1.leftOuterJoin(orderHeader_rdd1)
NonNulls = joined.filter(lambda line: line[1][1]!=None)
Nulls = joined.filter(lambda line: line[1][1]==None)
```

Let's see the data structures for both of them.

```
NonNulls.take(5)
```

```
[('30113', (('Raja', 'Venugopal'), ('71780', '42452.6519'))),
 ('30089', (('Michael John', 'Troyer'), ('71815', '1261.444'))),
 ('29485', (('Catherine', 'Abel'), ('71782', '43962.7901'))),
 ('29638', (('Rosmarie', 'Carroll'), ('71915', '2361.6403'))),
 ('29938', (('Frank', 'Campbell'), ('71845', '45992.3665')))]
```

Let's rearrage them.

```
NonNulls2 = NonNulls.map(lambda line: (line[0], line[1][0][0],line[1][0][1], line[1][1][0
], float(line[1][1][1])))
```

```
NonNulls2.first()
```

```
('30113', 'Raja', 'Venugopal', '71780', 42452.6519)
```

Similarly, let's rearrange the Nulls RDD.

```
Nulls.take(5)
```

```
[('190', (('Mark', 'Lee'), None)),
 ('30039', (('Robert', 'Stotka'), None)),
 ('110', (('Kendra', 'Thompson'), None)),
 ('29832', (('Jésus', 'Hernandez'), None)),
 ('473', (('Kay', 'Krane'), None))]
```

```
Nulls2 = Nulls.map(lambda line: (line[0], line[1][0][0],line[1][0][1], "NULL", "NULL"))

Nulls2.take(5)
```

```
[('190', 'Mark', 'Lee', 'NULL', 'NULL'),
 ('30039', 'Robert', 'Stotka', 'NULL', 'NULL'),
 ('110', 'Kendra', 'Thompson', 'NULL', 'NULL'),
 ('29832', 'Jésus', 'Hernandez', 'NULL', 'NULL'),
 ('473', 'Kay', 'Krane', 'NULL', 'NULL')]
```

Now, we can union them and see the top five and bottom five as below.

```
union_rdd = NonNulls2.union(Nulls2)
```

```
union_rdd.collect()[:5]
```

```
[('30113', 'Raja', 'Venugopal', '71780', 42452.6519),
 ('30089', 'Michael John', 'Troyer', '71815', 1261.444),
 ('29485', 'Catherine', 'Abel', '71782', 43962.7901),
 ('29638', 'Rosmarie', 'Carroll', '71915', 2361.6403),
 ('29938', 'Frank', 'Campbell', '71845', 45992.3665)]
```

```
union_rdd.collect()[-5:]
```

```
[('41', 'Erin', 'Hagens', 'NULL', 'NULL'),
 ('178', 'Dick', 'Dievendorff', 'NULL', 'NULL'),
 ('479', 'Lori', 'Kane', 'NULL', 'NULL'),
 ('424', 'Eli', 'Bowen', 'NULL', 'NULL'),
 ('76', 'James', 'Krow', 'NULL', 'NULL')]
```

## DataFrame

Now, we let's answer it the question the DataFrame approach.

```
customer_df = sqlcontext.createDataFrame(customer_rdd.map(lambda line: line.split("\t")),
                                        schema = customer.first().split("\t"))

orderHeader_df = sqlcontext.createDataFrame(orderHeader_rdd.map(lambda line: line.split("
\t")),
                                        schema = orderHeader.first().split("\t"))
```

```
customer_df.printSchema()
```

```
root
 |-- CustomerID: string (nullable = true)
 |-- NameStyle: string (nullable = true)
 |-- Title: string (nullable = true)
 |-- FirstName: string (nullable = true)
 |-- MiddleName: string (nullable = true)
 |-- LastName: string (nullable = true)
 |-- Suffix: string (nullable = true)
 |-- CompanyName: string (nullable = true)
 |-- SalesPerson: string (nullable = true)
 |-- EmailAddress: string (nullable = true)
 |-- Phone: string (nullable = true)
 |-- PasswordHash: string (nullable = true)
 |-- PasswordSalt: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: string (nullable = true)
```

```
orderHeader_df.printSchema()
```

```
root
 |-- SalesOrderID: string (nullable = true)
 |-- RevisionNumber: string (nullable = true)
 |-- OrderDate: string (nullable = true)
 |-- DueDate: string (nullable = true)
 |-- ShipDate: string (nullable = true)
 |-- Status: string (nullable = true)
 |-- OnlineOrderFlag: string (nullable = true)
 |-- SalesOrderNumber: string (nullable = true)
 |-- PurchaseOrderNumber: string (nullable = true)
 |-- AccountNumber: string (nullable = true)
 |-- CustomerID: string (nullable = true)
 |-- ShipToAddressID: string (nullable = true)
 |-- BillToAddressID: string (nullable = true)
 |-- ShipMethod: string (nullable = true)
 |-- CreditCardApprovalCode: string (nullable = true)
 |-- SubTotal: string (nullable = true)
 |-- TaxAmt: string (nullable = true)
 |-- Freight: string (nullable = true)
 |-- TotalDue: string (nullable = true)
 |-- Comment: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: string (nullable = true)
```

We can see samples of those that have made orders and those that have not as below.

```
joined = customer_df.join(orderHeader_df, 'CustomerID', how = "left")
joined.select(["CustomerID", 'FirstName','LastName','SalesOrderNumber','TotalDue'])\
.orderBy("TotalDue", ascending = False).show(10, truncate = False)
```

```
+----------+-----------+--------+----------------+----------+
|CustomerID|FirstName  |LastName|SalesOrderNumber|TotalDue  |
+----------+-----------+--------+----------------+----------+
|29546     |Christopher|Beck    |SO71938         |98138.2131|
|29847     |David      |Hodgson |SO71774         |972.785   |
|29957     |Kevin      |Liu     |SO71783         |92663.5609|
|30072     |Andrea     |Thomsen |SO71776         |87.0851   |
|29796     |Jon        |Grande  |SO71797         |86222.8072|
|29929     |Jeffrey    |Kurtz   |SO71902         |81834.9826|
|29531     |Cory       |Booth   |SO71935         |7330.8972 |
|29932     |Rebecca    |Laszlo  |SO71898         |70698.9922|
|30033     |Vassar     |Stern   |SO71856         |665.4251  |
|29660     |Anthony    |Chor    |SO71796         |63686.2708|
+----------+-----------+--------+----------------+----------+
only showing top 10 rows
```

```
joined.select(["CustomerID", 'FirstName','LastName','SalesOrderNumber','TotalDue'])\
.orderBy("TotalDue", ascending = True).show(10, truncate = False)
```

```
+----------+---------+-----------+----------------+--------+
|CustomerID|FirstName|LastName   |SalesOrderNumber|TotalDue|
+----------+---------+-----------+----------------+--------+
|29539     |Josh     |Barnhill   |null            |null    |
|29573     |Luis     |Bonifaz    |null            |null    |
|29865     |Lucio    |Iallo      |null            |null    |
|29978     |Ajay     |Manchepalli|null            |null    |
|451       |John     |Emory      |null            |null    |
|124       |Yuhong   |Li         |null            |null    |
|29580     |Richard  |Bready     |null            |null    |
|7         |Dominic  |Gash       |null            |null    |
|29525     |Teresa   |Atkinson   |null            |null    |
|29733     |Shannon  |Elliott    |null            |null    |
+----------+---------+-----------+----------------+--------+
only showing top 10 rows
```

#### Running SQL Queries Programmatically

Below, I have showed samples of those that have made orders and those that have not using normal SQL commands.

```
orderHeader_df.createOrReplaceTempView("orderHeader_table")
customer_df.createOrReplaceTempView("customer_table")

sqlcontext.sql("SELECT c.CustomerID, c.FirstName,c.LastName, oh.SalesOrderID,cast(oh.Total
Due AS DECIMAL(10,4)) \
               FROM customer_table AS c LEFT JOIN orderHeader_table AS oh ON c.CustomerID
 = oh.CustomerID \
               ORDER BY TotalDue DESC LIMIT 10").show(truncate = False)
```

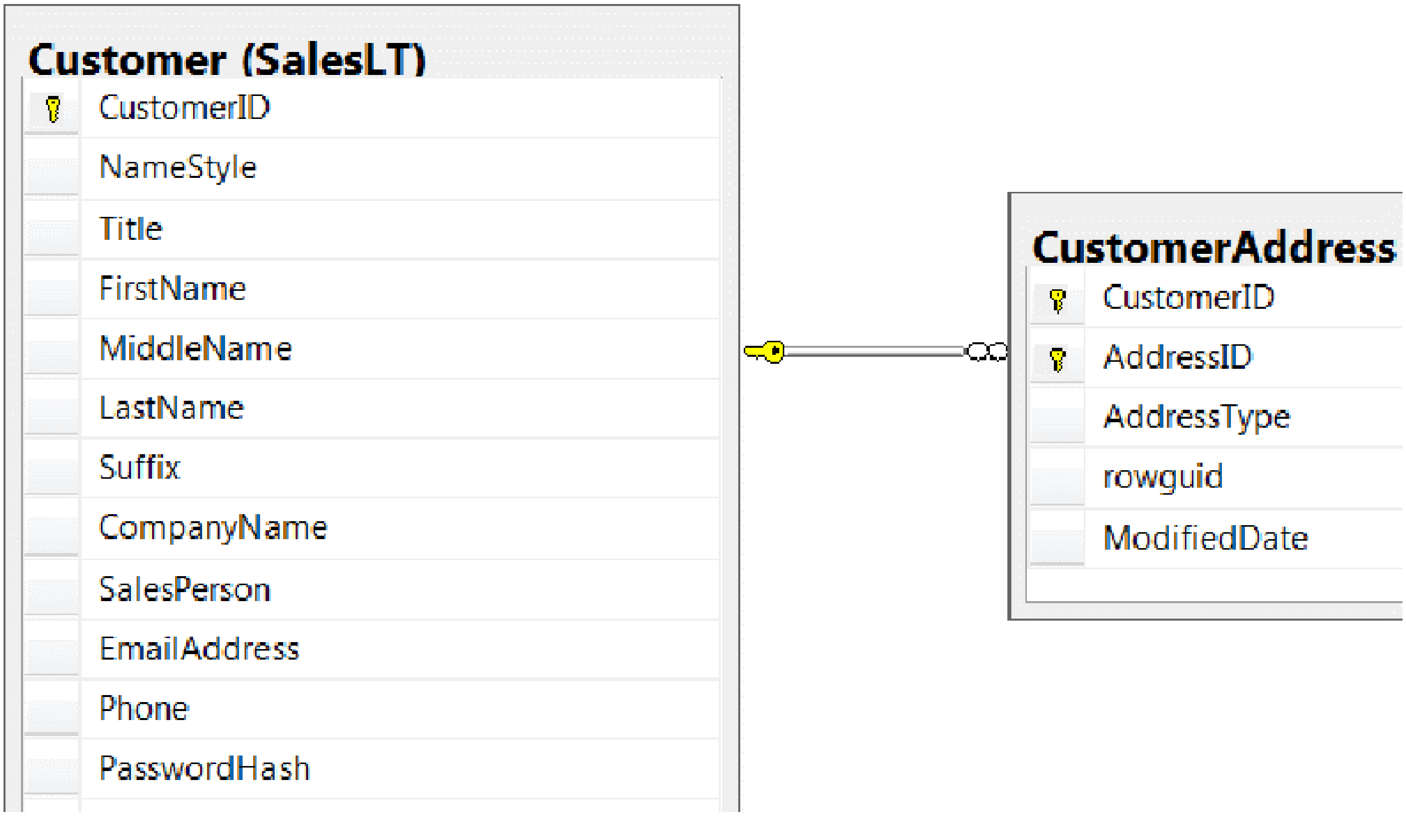
```
+----------+-----------+------------+------------+-----------+
|CustomerID|FirstName  |LastName    |SalesOrderID|TotalDue   |
+----------+-----------+------------+------------+-----------+
|29736     |Terry      |Eminhizer   |71784       |119960.8240|
|30050     |Krishna    |Sunkammurali|71936       |108597.9536|
|29546     |Christopher|Beck        |71938       |98138.2131 |
|29957     |Kevin      |Liu         |71783       |92663.5609 |
|29796     |Jon        |Grande      |71797       |86222.8072 |
|29929     |Jeffrey    |Kurtz       |71902       |81834.9826 |
|29932     |Rebecca    |Laszlo      |71898       |70698.9922 |
|29660     |Anthony    |Chor        |71796       |63686.2708 |
|29938     |Frank      |Campbell    |71845       |45992.3665 |
|29485     |Catherine  |Abel        |71782       |43962.7901 |
+----------+-----------+------------+------------+-----------+
```

```
sqlcontext.sql("SELECT c.CustomerID, c.FirstName,c.LastName, oh.SalesOrderID,cast(oh.Total
Due AS DECIMAL(10,4)) \
               FROM customer_table AS c LEFT JOIN orderHeader_table AS oh ON c.CustomerID
 = oh.CustomerID \
               ORDER BY TotalDue ASC LIMIT 10").show(truncate = False)
```

```
+----------+---------+-----------+------------+--------+
|CustomerID|FirstName|LastName   |SalesOrderID|TotalDue|
+----------+---------+-----------+------------+--------+
|7         |Dominic  |Gash       |null        |null    |
|29573     |Luis     |Bonifaz    |null        |null    |
|29539     |Josh     |Barnhill   |null        |null    |
|29978     |Ajay     |Manchepalli|null        |null    |
|451       |John     |Emory      |null        |null    |
|29865     |Lucio    |Iallo      |null        |null    |
|30005     |Nancy    |McPhearson |null        |null    |
|124       |Yuhong   |Li         |null        |null    |
|29580     |Richard  |Bready     |null        |null    |
|169       |Brenda   |Diaz       |null        |null    |
+----------+---------+-----------+------------+--------+
```

## 4. Retrieve a list of customers with no address

A sales employee has noticed that Adventure Works does not have address information for all customers. You must write a query that returns a list of customer IDs, company names, contact names (first name and last name), and phone numbers for customers with no address stored in the database.




**RDD way**

```
customer_header = customer.first()
customer_rdd = customer.filter(lambda line: line != customer_header)

customer_address_header = customer_address.first()
customer_address_rdd = customer_address.filter(lambda line: line != customer_address_heade
r)
```

```
customer_rdd1 = customer_rdd.map(lambda line: (line.split("\t")[0],  #CustomerID
                                              (line.split("\t")[3],  #FirstName
                                              line.split("\t")[5],  #LastName
                                              line.split("\t")[7],  #CompanyName
                                              line.split("\t")[9],  #EmailAddress
                                              line.split("\t")[10] #Phone
                                              )))

customer_address_rdd1 = customer_address_rdd.map(lambda line: (line.split("\t")[0],line.sp
lit("\t")[1]))
```

First, let's join the customer data to the customer address dataset.Then, we will filter the RDD to include those that do not have address information.

```
joined = customer_rdd1.leftOuterJoin(customer_address_rdd1)
joined.take(2)
```

```
[('190',
  (('Mark',
    'Lee',
    'Racing Partners',
    'mark5@adventure-works.com',
    '371-555-0112'),
   None)),
 ('30039',
  (('Robert',
    'Stotka',
    'Gift and Toy Store',
    'robert12@adventure-works.com',
    '493-555-0185'),
   '627'))]
```

```
joined.filter(lambda line: line[1][1]==None).take(5)
```

```
[('190',
  (('Mark',
    'Lee',
    'Racing Partners',
    'mark5@adventure-works.com',
    '371-555-0112'),
   None)),
 ('110',
  (('Kendra',
    'Thompson',
    'Vintage Sport Boutique',
    'kendra0@adventure-works.com',
    '464-555-0188'),
   None)),
 ('473',
  (('Kay', 'Krane', 'Racing Toys', 'kay0@adventure-works.com', '731-555-018
7'),
   None)),
 ('629',
  (('Ryan',
    'Ihrig',
    'Efficient Cycling',
    'ryan4@adventure-works.com',
    '809-555-0152'),
   None)),
 ('256',
  (('Richard',
    'Irwin',
    'Rental Bikes',
    'richard4@adventure-works.com',
    '367-555-0124'),
   None))]
```

## DataFrame way

After getting those who don't have address information, below I am diplaying 10 rows.

```
customer_df = sqlcontext.createDataFrame(customer_rdd.map(lambda line: line.split("\t")),
                                         schema = customer.first().split("\t"))

customer_address_df = sqlcontext.createDataFrame(customer_address_rdd.map(lambda line: lin
e.split("\t")),
                                         schema = customer_address_header.split("\t"))
```

```
joined = customer_df.join(customer_address_df, 'CustomerID','left')
```

```
joined.filter(col("AddressID").isNull()).\
select(['FirstName','LastName','CompanyName','EmailAddress','Phone'])\
.show(10, truncate = False)
```

```
+---------+--------+------------------------+----------------------------+---
----------------+
|FirstName|LastName|CompanyName             |EmailAddress                |Pho
ne              |
+---------+--------+------------------------+----------------------------+---
----------------+
|John     |Emory   |Roadway Bike Emporium   |john16@adventure-works.com  |691
-555-0149       |
|Yuhong   |Li      |Nearby Sporting Goods   |yuhong1@adventure-works.com |1
(11) 500 555-0176|
|Dominic  |Gash    |Associated Bikes        |dominic0@adventure-works.com|192
-555-0173       |
|Neva     |Mitchell|Riding Associates       |neva0@adventure-works.com   |992
-555-0134       |
|John     |Evans   |Real Sporting Goods     |john17@adventure-works.com  |581
-555-0172       |
|Janice   |Hows    |Area Sheet Metal Supply |janice1@adventure-works.com |1
(11) 500 555-0119|
|Jim      |Stewart |Famous Bike Shop        |jim5@adventure-works.com    |226
-555-0110       |
|Brenda   |Diaz    |Downtown Hotel          |brenda2@adventure-works.com |147
-555-0192       |
|Frank    |Mart¡nez|Rally Master Company Inc|frank5@adventure-works.com  |171
-555-0147       |
|Dora     |Verdad  |Retreat Inn             |dora0@adventure-works.com   |155
-555-0140       |
+---------+--------+------------------------+----------------------------+---
----------------+
only showing top 10 rows
```

#### Running SQL Queries Programmatically

Using SQL also gives the same answers as the DataFrame approach shown above.

```
customer_address_df.createOrReplaceTempView("customer_address_table")
customer_df.createOrReplaceTempView("customer_table")

sqlcontext.sql("SELECT  c.FirstName,c.LastName, c.CompanyName,c.EmailAddress,c.Phone \
               FROM customer_table AS c LEFT JOIN customer_address_table AS ca ON c.Custom
erID = ca.CustomerID \
               WHERE ca.AddressID IS NULL").show(10, truncate = False)
```

```
+---------+--------+-------------------------+-----------------------------+---
----------------+
|FirstName|LastName|CompanyName              |EmailAddress                 |Pho
ne              |
+---------+--------+-------------------------+-----------------------------+---
----------------+
|John     |Emory   |Roadway Bike Emporium    |john16@adventure-works.com   |691
-555-0149       |
|Yuhong   |Li      |Nearby Sporting Goods    |yuhong1@adventure-works.com  |1
(11) 500 555-0176|
|Dominic  |Gash    |Associated Bikes         |dominic0@adventure-works.com|192
-555-0173       |
|Neva     |Mitchell|Riding Associates        |neva0@adventure-works.com    |992
-555-0134       |
|John     |Evans   |Real Sporting Goods      |john17@adventure-works.com   |581
-555-0172       |
|Janice   |Hows    |Area Sheet Metal Supply  |janice1@adventure-works.com  |1
(11) 500 555-0119|
|Jim      |Stewart |Famous Bike Shop         |jim5@adventure-works.com     |226
-555-0110       |
|Brenda   |Diaz    |Downtown Hotel           |brenda2@adventure-works.com  |147
-555-0192       |
|Frank    |Mart¡nez|Rally Master Company Inc|frank5@adventure-works.com   |171
-555-0147       |
|Dora     |Verdad  |Retreat Inn              |dora0@adventure-works.com    |155
-555-0140       |
+---------+--------+-------------------------+-----------------------------+---
----------------+
only showing top 10 rows
```

**This is enough for today. In the next part of the Spark RDDs Vs DataFrames vs SparkSQL tutorial series, I will come with a different topic. If you have any questions, or suggestions, feel free to drop them below.**

# Spark RDDs Vs DataFrames vs SparkSQL - Part 3 : Web Server Log Analysis

This is the third tutorial on the Spark RDDs Vs DataFrames vs SparkSQL blog post series. The first one is available here (http://datascience-enthusiast.com/Python/DataFramesVsRDDsSpark-Part1.html). In the first part, we saw how to retrieve, sort and filter data using Spark RDDs, DataFrames and SparkSQL. In the second part (here) (http://datascience-enthusiast.com/Python/DataFramesVsRDDsVsSQLSpark-Part2.html), we saw how to work with multiple tables in Spark the RDD way, the DataFrame way and with SparkSQL. In this third part of the blog post series, we will perform web server log analysis using real-world text-based production logs. Log data can be used monitoring servers, improving business and customer intelligence, building recommendation systems, fraud detection, and much more. Server log analysis is a good use case for Spark. It's a very large, common data source and contains a rich set of information.

If you like this tutorial series, check also my other recent blos posts on Spark on Analyzing the Bible and the Quran using Spark (http://datascience-enthusiast.com/Python/analyzing_bible_quran_with_spark.html) and Spark DataFrames: Exploring Chicago Crimes (http://datascience-enthusiast.com/Python/SparkDataFrames-ExploringChicagoCrimes.html). The data and the notebooks can be downloaded from my GitHub repository (https://github.com/fissehab/Spark_certification).

The log files that we use for this assignment are in the Apache Common Log Format (CLF) (http://httpd.apache.org/docs/1.3/logs.html#common). The log file entries produced in CLF will look something like this:

**127.0.0.1 - - [01/Aug/1995:00:00:01 -0400] "GET /images/launch-logo.gif HTTP/1.0" 200 1839** Each part of this log entry is described below.

**127.0.0.1** This is the IP address (or host name, if available) of the client (remote host) which made the request to the server.

**-** The "hyphen" in the output indicates that the requested piece of information (user identity from remote machine) is not available.

**-** The "hyphen" in the output indicates that the requested piece of information (user identity from local logon) is not available.

**[01/Aug/1995:00:00:01 -0400]** the time that the server finished processing the request. The format is: [day/month/year:hour:minute:second timezone]

```
day = 2 digits
month = 3 letters
year = 4 digits
hour = 2 digits
minute = 2 digits
second = 2 digits
zone = (+ | -) 4 digits
```

**"GET /images/launch-logo.gif HTTP/1.0"** This is the first line of the request string from the client. It consists of a three components: the request method (e.g., GET, POST, etc.), the endpoint (a Uniform Resource Identifier), and the client protocol version.

**200** This is the status code that the server sends back to the client. This information is very valuable, because it reveals whether the request resulted in a successful response (codes beginning in 2), a redirection (codes beginning in 3), an error caused by the client (codes beginning in 4), or an error in the server (codes beginning in 5).

**1839** The last entry indicates the size of the object returned to the client, not including the response headers. If no content was returned to the client, this value will be "-" (or sometimes 0).

we will use a data set from NASA Kennedy Space Center WWW server in Florida. The full data set is freely available here (http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html) and contains two month's of all HTTP requests.

Let's download the data. Since I am using Jupyter Notebook, ! helps us to run a shell command

```
#! wget ftp://ita.ee.lbl.gov/traces/NASA_access_log_Jul95.gz
```

```
#! wget ftp://ita.ee.lbl.gov/traces/NASA_access_log_Aug95.gz
```

## Create Spark Context and SQL Context.

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SQLContext


conf = SparkConf().setAppName("Spark-RDD-DataFrame_SQL").setMaster("local[*]")
sc = SparkContext.getOrCreate(conf)

sqlcontext = SQLContext(sc)
```

### Create RDD

```
rdd = sc.textFile("NASA_access*")
```

### Show sample logs

```
for line in rdd.sample(withReplacement = False, fraction = 0.000001, seed = 120).collect
():
    print(line)
    print("\n")
```

```
www-b2.proxy.aol.com - - [18/Aug/1995:19:04:43 -0400] "GET /shuttle/countdow
n/ HTTP/1.0" 304 0
```

### Use regular expressions to extract the logs

```
import re

def parse_log1(line):
    match = re.search('^(\S+) (\S+) (\S+) \[(\S+) [-](\d{4})\] "(\S+)\s*(\S+)\s*(\S+)\s*
([\w\.\s*]+)?\s*"*(\d{3}) (\S+)', line)
    if match is None:
        return 0
    else:
        return 1
```

```
n_logs = rdd.count()

failed = rdd.map(lambda line: parse_log1(line)).filter(lambda line: line == 0).count()

print('Out of {} logs, {} failed to parse'.format(n_logs,failed))
```

```
Out of 3461613 logs, 1685 failed to parse
```

we see that 1685 out of the 3.5 million logs failed to parse. I took samples of the failed logs and tried to modify the above regular expression pattern as show below.

```
def parse_log2(line):

    match = re.search('^(\S+) (\S+) (\S+) \[(\S+) [-](\d{4})\] "(\S+)\s*(\S+)\s*(\S+)\s*
([/\w\.\s*]+)?\s*"* (\d{3}) (\S+)',line)
    if match is None:
        match = re.search('^(\S+) (\S+) (\S+) \[(\S+) [-](\d{4})\] "(\S+)\s*([/\w\.]+)>*
([\w/\s\.]+)\s*(\S+)\s*(\d{3})\s*(\S+)',line)
    if match is None:
        return (line, 0)
    else:
        return (line, 1)
```

```
failed = rdd.map(lambda line: parse_log2(line)).filter(lambda line: line[1] == 0).count()

print('Out of {} logs, {} failed to parse'.format(n_logs,failed))
```

```
Out of 3461613 logs, 1253 failed to parse
```

Still, 1253 of them failed to parse. However, since we have successfully parsed more than 99.9% of the data, we can work with what we have parsed. You can play with the regular expression pattern to match all of the data :).

### Extract the 11 elements from each log

```
def map_log(line):

    match = re.search('^(\S+) (\S+) (\S+) \[(\S+) [-](\d{4})\] "(\S+)\s*(\S+)\s*(\S+)\s*
([/\w\.\s*]+)?\s*"* (\d{3}) (\S+)',line)
    if match is None:
        match = re.search('^(\S+) (\S+) (\S+) \[(\S+) [-](\d{4})\] "(\S+)\s*([/\w\.]+)>*
([\w/\s\.]+)\s*(\S+)\s*(\d{3})\s*(\S+)',line)

    return(match.groups())
```

```
parsed_rdd = rdd.map(lambda line: parse_log2(line)).filter(lambda line: line[1] == 1).map(
lambda line : line[0])
```

```
parsed_rdd2 = parsed_rdd.map(lambda line: map_log(line))
```

### Show 3 lines

```
for i in parsed_rdd2.take(3):
    print(i)
    print('\n')
```

```
('199.72.81.55', '-', '-', '01/Jul/1995:00:00:01', '0400', 'GET', '/history/a
pollo/', 'HTTP/1.0"', None, '200', '6245')


('unicomp6.unicomp.net', '-', '-', '01/Jul/1995:00:00:06', '0400', 'GET', '/s
huttle/countdown/', 'HTTP/1.0"', None, '200', '3985')


('199.120.110.21', '-', '-', '01/Jul/1995:00:00:09', '0400', 'GET', '/shuttl
e/missions/sts-73/mission-sts-73.html', 'HTTP/1.0"', None, '200', '4085')
```

As shown below, each line is a log of length 11.

```
parsed_rdd2.map(lambda line: len(line)).distinct().collect()
```

```
[11]
```

Now, let's try to answer some questions.

**1. Find the 10 most common IP addresses (or host name, if available) of the client (remote host) which made the request to the server.**

**RDD way**

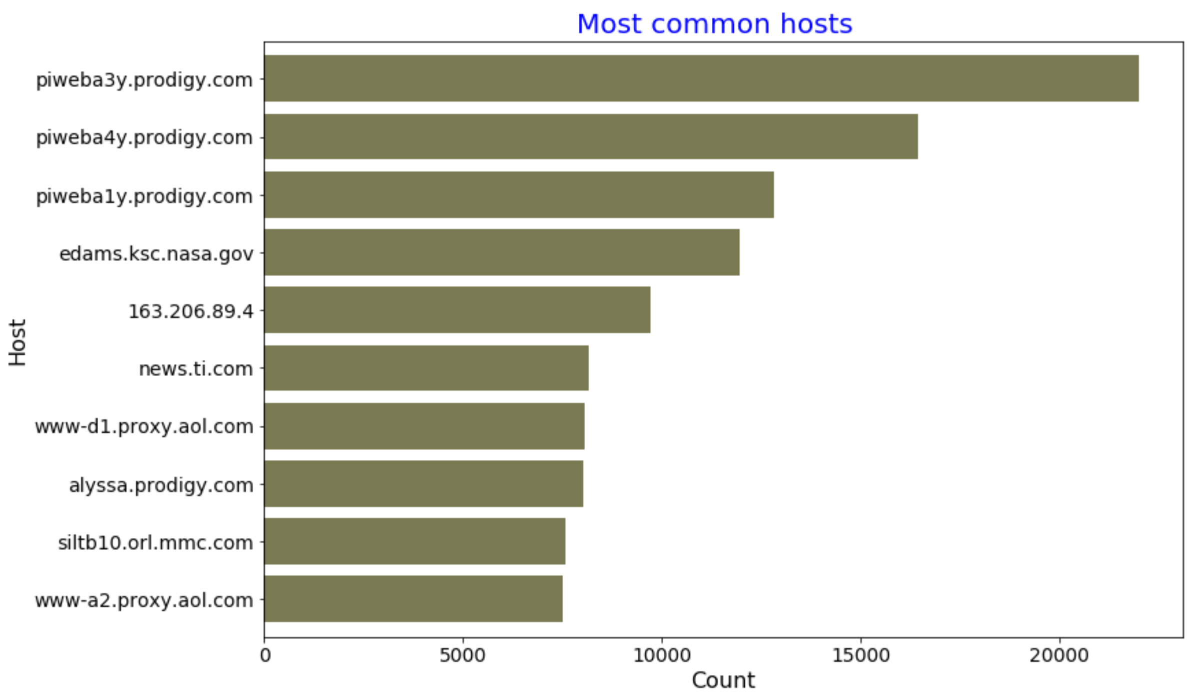
```
result = parsed_rdd2.map(lambda line: (line[0],1)).reduceByKey(lambda a, b: a + b).takeOrd
ered(10, lambda x: -x[1])
result
```

```
[('piweba3y.prodigy.com', 21988),
 ('piweba4y.prodigy.com', 16437),
 ('piweba1y.prodigy.com', 12825),
 ('edams.ksc.nasa.gov', 11964),
 ('163.206.89.4', 9697),
 ('news.ti.com', 8161),
 ('www-d1.proxy.aol.com', 8047),
 ('alyssa.prodigy.com', 8037),
 ('siltb10.orl.mmc.com', 7573),
 ('www-a2.proxy.aol.com', 7516)]
```

We can also use Pandas and Matplotlib to creata a viz.

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

```
Host = [x[0] for x in result]
count = [x[1] for x in result]
Host_count_dct = {'Host':Host, 'count':count}
Host_count_df = pd.DataFrame(Host_count_dct )


myplot = Host_count_df.plot(figsize = (12,8), kind = "barh", color = "#7a7a52", width = 0.
8,
                               x = "Host", y = "count", legend = False)

myplot.invert_yaxis()

plt.xlabel("Count", fontsize = 16)
plt.ylabel("Host", fontsize = 16)
plt.title("Most common hosts  ", fontsize = 20, color = 'b')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




### DataFrame way

```
parsed_df = sqlcontext.createDataFrame(parsed_rdd2,
                                       schema = ['host','identity_remote', 'identity_loca
l','date_time',
                                                 'time_zone','request_method','endpoint',
'client_protocol','mis',
                                                 'status_code','size_returned'], samplingR
atio = 0.1)
```

```
parsed_df.printSchema()
```

```
root
 |-- host: string (nullable = true)
 |-- identity_remote: string (nullable = true)
 |-- identity_local: string (nullable = true)
 |-- date_time: string (nullable = true)
 |-- time_zone: string (nullable = true)
 |-- request_method: string (nullable = true)
 |-- endpoint: string (nullable = true)
 |-- client_protocol: string (nullable = true)
 |-- mis: string (nullable = true)
 |-- status_code: string (nullable = true)
 |-- size_returned: string (nullable = true)
```

```
parsed_df.groupBy('host').count().orderBy('count', ascending = False).show(10, truncate = False)
```

```
+--------------------+-----+
|host                |count|
+--------------------+-----+
|piweba3y.prodigy.com|21988|
|piweba4y.prodigy.com|16437|
|piweba1y.prodigy.com|12825|
|edams.ksc.nasa.gov  |11964|
|163.206.89.4        |9697 |
|news.ti.com         |8161 |
|www-d1.proxy.aol.com|8047 |
|alyssa.prodigy.com  |8037 |
|siltb10.orl.mmc.com |7573 |
|www-a2.proxy.aol.com|7516 |
+--------------------+-----+
only showing top 10 rows
```

### SQL way

```
parsed_df.createOrReplaceTempView("parsed_table")

sqlcontext.sql('SELECT host, count(*) AS count FROM parsed_table GROUP BY\
               host ORDER BY count DESC LIMIT 10 ').show()
```

```
+--------------------+-----+
|                host|count|
+--------------------+-----+
|piweba3y.prodigy.com|21988|
|piweba4y.prodigy.com|16437|
|piweba1y.prodigy.com|12825|
|  edams.ksc.nasa.gov|11964|
|        163.206.89.4| 9697|
|         news.ti.com| 8161|
|www-d1.proxy.aol.com| 8047|
|  alyssa.prodigy.com| 8037|
| siltb10.orl.mmc.com| 7573|
|www-a2.proxy.aol.com| 7516|
+--------------------+-----+
```

### 2. Find statistics of the size of the object returned to the client.

### RDD way

```
def convert_long(x):
    x = re.sub('[^0-9]',"",x)
    if x =="":
        return 0
    else:
        return int(x)
```

```
parsed_rdd2.map(lambda line: convert_long(line[-1])).stats()
```

```
(count: 3460360, mean: 18935.441238194595, stdev: 73043.8640344, max: 682393
6.0, min: 0.0)
```

### DataFrame way

Here, we can use functions from pyspark.

```
from pyspark.sql.functions import mean, udf, col, min, max, stddev, count
from pyspark.sql.types import DoubleType, IntegerType
```

```
my_udf = udf(convert_long, IntegerType() )
(parsed_df.select(my_udf('size_returned').alias('size'))
 .select(mean('size').alias('Mean Size'),
  max('size').alias('Max Size'),
  min('size').alias('Min Size'),
  count('size').alias('Count'),
  stddev('size').alias('stddev Size')).show()
)
```

```
+-----------------+--------+--------+-------+-----------------+
|        Mean Size|Max Size|Min Size|  Count|      stddev Size|
+-----------------+--------+--------+-------+-----------------+
|18935.44123819487| 6823936|       0|3460360|73043.87458874052|
+-----------------+--------+--------+-------+-----------------+
```

**SQL way**

```
parsed_df_cleaned = parsed_rdd2.map(lambda line: convert_long(line[-1])).toDF(IntegerType
())
parsed_df_cleaned.createOrReplaceTempView("parsed_df_cleaned_table")
sqlcontext.sql("SELECT avg(value) AS Mean_size, max(value) AS Max_size, \
                min(value) AS Min_size, count(*) AS count, \
                std(value) AS stddev_size FROM parsed_df_cleaned_table").show()
```

```
+-----------------+--------+--------+-------+-----------------+
|        Mean_size|Max_size|Min_size|  count|      stddev_size|
+-----------------+--------+--------+-------+-----------------+
|18935.44123819487| 6823936|       0|3460360|73043.87458874052|
+-----------------+--------+--------+-------+-----------------+
```

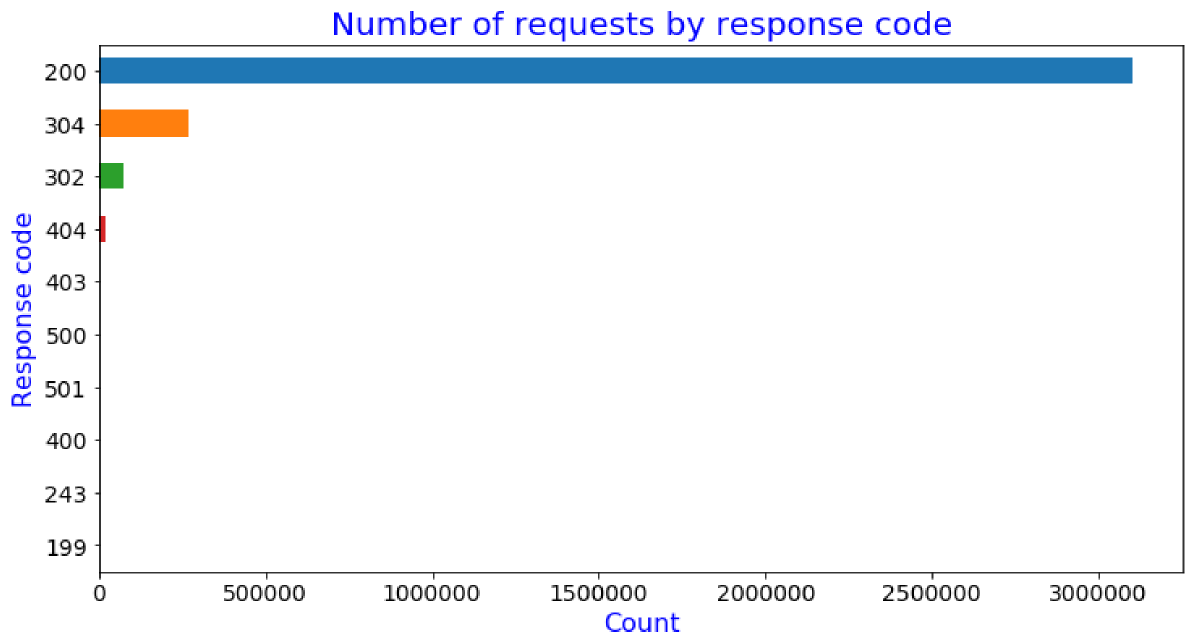
**3. Find the number of logs with each response code.**

**RDD way**

```
n_codes = parsed_rdd2.map(lambda line: (line[-2], 1)).distinct().count()

codes_count = (parsed_rdd2.map(lambda line: (line[-2], 1))
          .reduceByKey(lambda a, b: a + b)
          .takeOrdered(n_codes, lambda x: -x[1]))
codes_count
```

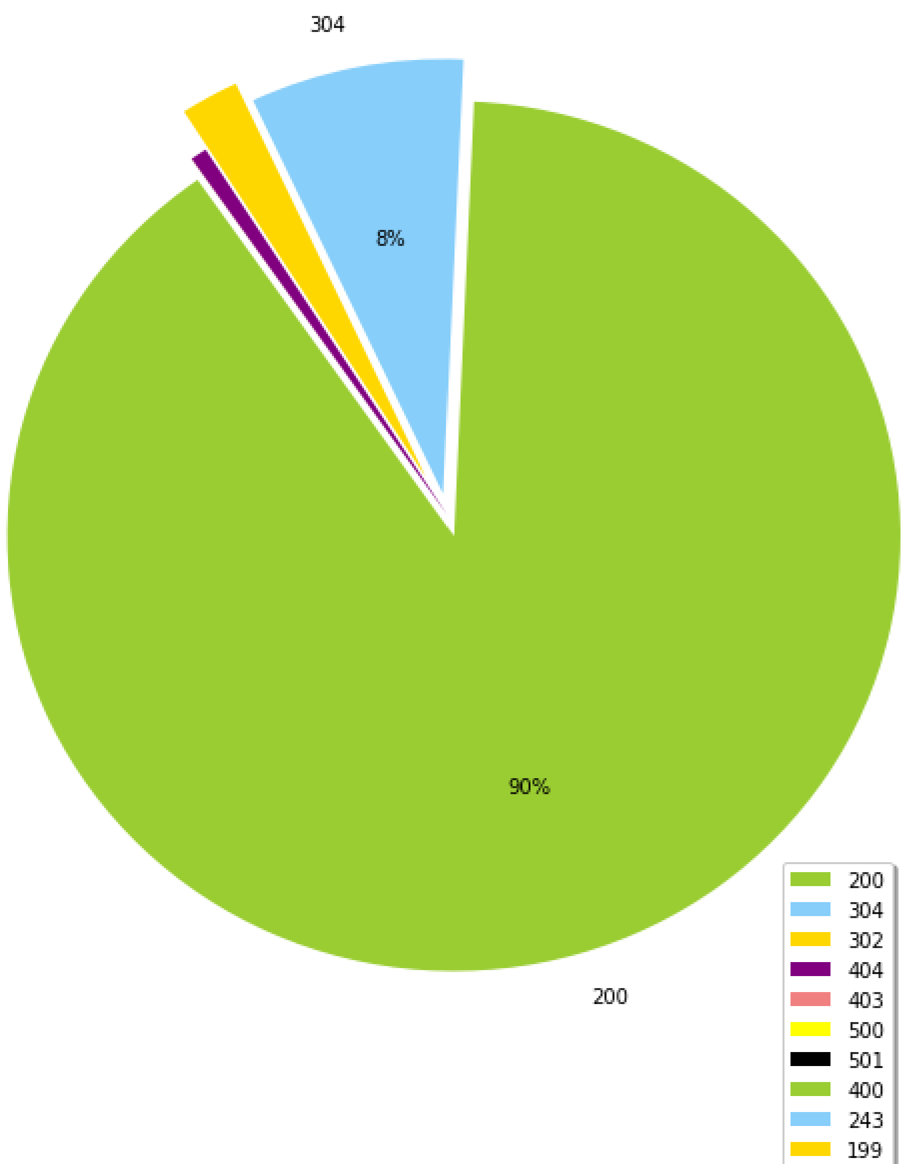
```
[('200', 3099280),
 ('304', 266773),
 ('302', 73070),
 ('404', 20890),
 ('403', 225),
 ('500', 65),
 ('501', 41),
 ('400', 14),
 ('243', 1),
 ('199', 1)]
```

```
codes =[x[0] for x in codes_count]
count =[x[1] for x in codes_count]

codes_dict = {'code':codes,'count':count}
codes_df = pd.DataFrame(codes_dict)
plot = codes_df.plot(figsize = (12, 6), kind = 'barh', y = 'count', x = 'code', legend = False)
plot.invert_yaxis()
plt.title('Number of requests by response code', fontsize = 20, color = 'b')
plt.xlabel('Count', fontsize = 16, color = 'b')
plt.ylabel('Response code', fontsize = 16, color = 'b')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




We can also create a pie chart as below.

```
def pie_pct_format(value):
    return '' if value < 7 else '%.0f%%' % value

fig = plt.figure(figsize =(10, 10), facecolor = 'white', edgecolor = 'white')
colors = ['yellowgreen', 'lightskyblue', 'gold', 'purple', 'lightcoral', 'yellow', 'black'
]
explode = (0.05, 0.05, 0.1, 0, 0, 0, 0,0,0,0)
patches, texts, autotexts = plt.pie(count, labels = codes, colors = colors,
                                    explode = explode, autopct = pie_pct_format,
                                    shadow = False,  startangle = 125)
for text, autotext in zip(texts, autotexts):
    if autotext.get_text() == '':
        text.set_text('')
plt.legend(codes, loc = (0.80, -0.1), shadow=True)
pass
```




**DataFrame way**

```
parsed_df.groupBy('status_code').count().orderBy('count', ascending = False).show()
```

```
+-----------+-------+
|status_code|  count|
+-----------+-------+
|        200|3099280|
|        304| 266773|
|        302|  73070|
|        404|  20890|
|        403|    225|
|        500|     65|
|        501|     41|
|        400|     14|
|        199|      1|
|        243|      1|
+-----------+-------+
```

**SQL way**

```
sqlcontext.sql("SELECT status_code, count(*) AS count FROM parsed_table \
GROUP BY status_code ORDER BY count DESC").show()
```

```
+-----------+-------+
|status_code|  count|
+-----------+-------+
|        200|3099280|
|        304| 266773|
|        302|  73070|
|        404|  20890|
|        403|    225|
|        500|     65|
|        501|     41|
|        400|     14|
|        199|      1|
|        243|      1|
+-----------+-------+
```

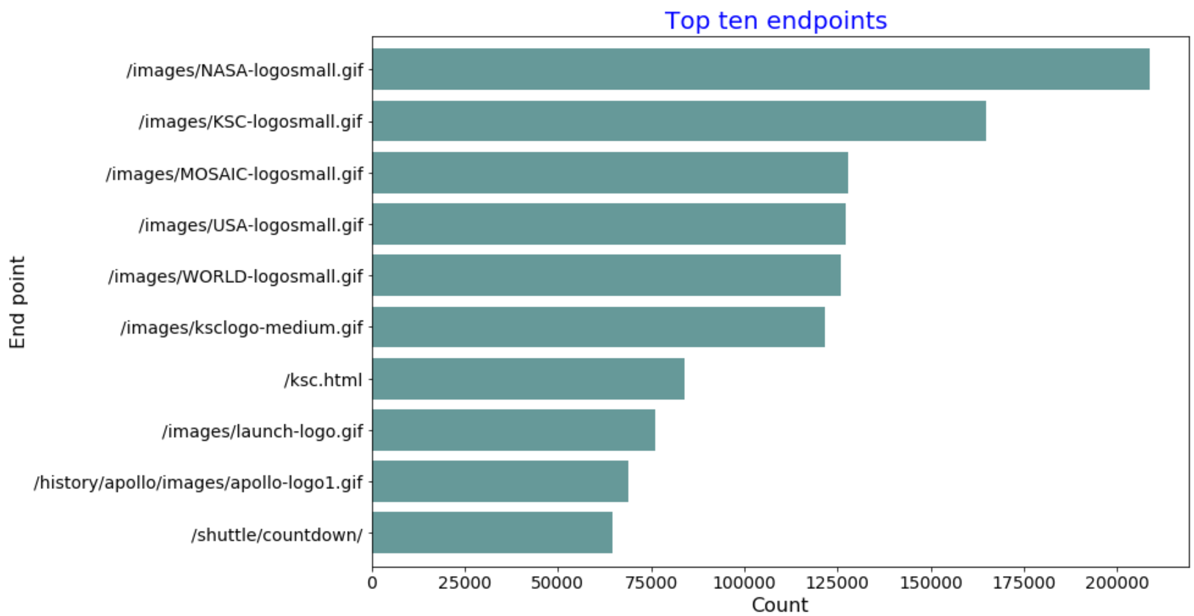
**4. What are the top ten endpoints?**

**RDD way**

```
result = parsed_rdd2.map(lambda line: (line[6],1)).reduceByKey(lambda a, b: a + b).takeOrd
ered(10, lambda x: -x[1])
result
```

```
[('/images/NASA-logosmall.gif', 208798),
 ('/images/KSC-logosmall.gif', 164976),
 ('/images/MOSAIC-logosmall.gif', 127916),
 ('/images/USA-logosmall.gif', 127082),
 ('/images/WORLD-logosmall.gif', 125933),
 ('/images/ksclogo-medium.gif', 121580),
 ('/ksc.html', 83918),
 ('/images/launch-logo.gif', 76009),
 ('/history/apollo/images/apollo-logo1.gif', 68898),
 ('/shuttle/countdown/', 64740)]
```

```
endpoint = [x[0] for x in result]
count = [x[1] for x in result]
endpoint_count_dct = {'endpoint':endpoint, 'count':count}
endpoint_count_df = pd.DataFrame(endpoint_count_dct )


myplot = endpoint_count_df .plot(figsize = (12,8), kind = "barh", color = "#669999", width
 = 0.8,
                               x = "endpoint", y = "count", legend = False)

myplot.invert_yaxis()

plt.xlabel("Count", fontsize = 16)
plt.ylabel("End point", fontsize = 16)
plt.title("Top ten endpoints ", fontsize = 20, color = 'b')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




### DataFrame way

```
parsed_df.groupBy('endpoint').count().orderBy('count', ascending = False).show(10, truncat
e = False)
```

```
+---------------------------------------+------+
|endpoint                               |count |
+---------------------------------------+------+
|/images/NASA-logosmall.gif             |208798|
|/images/KSC-logosmall.gif              |164976|
|/images/MOSAIC-logosmall.gif           |127916|
|/images/USA-logosmall.gif              |127082|
|/images/WORLD-logosmall.gif            |125933|
|/images/ksclogo-medium.gif             |121580|
|/ksc.html                              |83918 |
|/images/launch-logo.gif                |76009 |
|/history/apollo/images/apollo-logo1.gif|68898 |
|/shuttle/countdown/                    |64740 |
+---------------------------------------+------+
only showing top 10 rows
```

**SQL way**

```
sqlcontext.sql("SELECT endpoint, count(*) AS count FROM parsed_table \
GROUP BY endpoint ORDER BY count DESC LIMIT 10").show(truncate = False)
```

```
+---------------------------------------+------+
|endpoint                               |count |
+---------------------------------------+------+
|/images/NASA-logosmall.gif             |208798|
|/images/KSC-logosmall.gif              |164976|
|/images/MOSAIC-logosmall.gif           |127916|
|/images/USA-logosmall.gif              |127082|
|/images/WORLD-logosmall.gif            |125933|
|/images/ksclogo-medium.gif             |121580|
|/ksc.html                              |83918 |
|/images/launch-logo.gif                |76009 |
|/history/apollo/images/apollo-logo1.gif|68898 |
|/shuttle/countdown/                    |64740 |
+---------------------------------------+------+
```

**5. What are the top eight endpoints which did not have return code 200?**

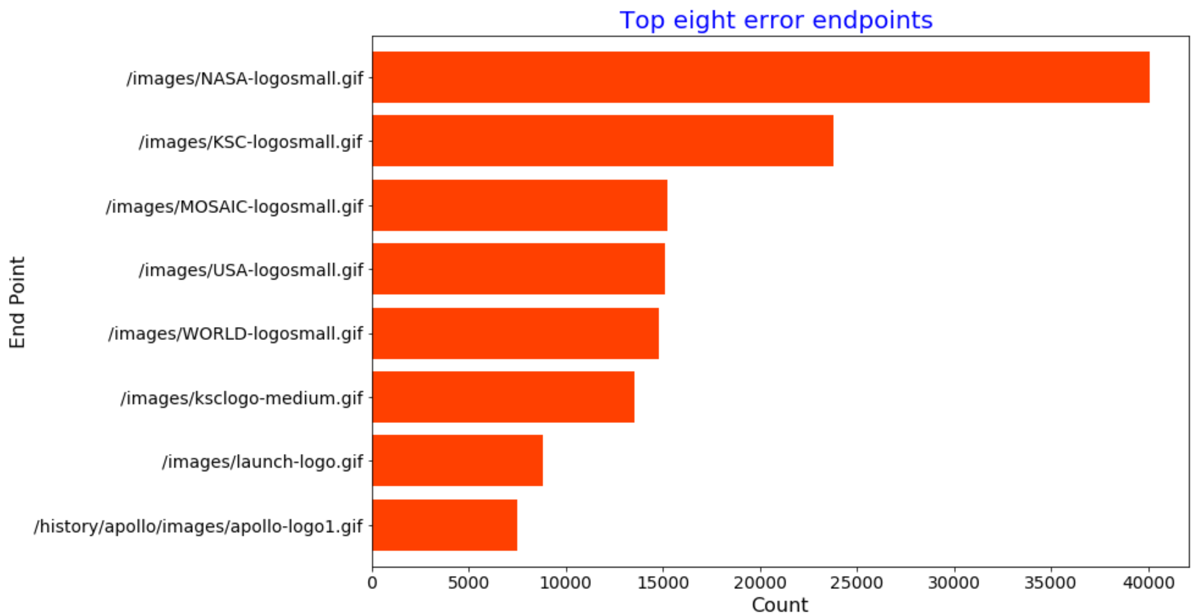
These are error endpoints

**RDD way**

```
result = (parsed_rdd2.filter(lambda line: line[9] != '200')
          .map(lambda line: (line[6], 1))
          .reduceByKey(lambda a, b: a+b)
          .takeOrdered(8, lambda x: -x[1]))
result
```

```
[('/images/NASA-logosmall.gif', 40090),
 ('/images/KSC-logosmall.gif', 23763),
 ('/images/MOSAIC-logosmall.gif', 15245),
 ('/images/USA-logosmall.gif', 15142),
 ('/images/WORLD-logosmall.gif', 14773),
 ('/images/ksclogo-medium.gif', 13559),
 ('/images/launch-logo.gif', 8806),
 ('/history/apollo/images/apollo-logo1.gif', 7489)]
```

```
endpoint = [x[0] for x in result]
count = [x[1] for x in result]
endpoint_count_dct = {'endpoint':endpoint, 'count':count}
endpoint_count_df = pd.DataFrame(endpoint_count_dct )

myplot = endpoint_count_df .plot(figsize = (12,8), kind = "barh", color = "#ff4000", width
 = 0.8,
                               x = "endpoint", y = "count", legend = False)

myplot.invert_yaxis()

plt.xlabel("Count", fontsize = 16)
plt.ylabel("End Point", fontsize = 16)
plt.title("Top eight error endpoints ", fontsize = 20, color = 'b')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




### DataFrame way

```
(parsed_df.filter(parsed_df['status_code']!=200)
 .groupBy('endpoint').count().orderBy('count', ascending = False)
   .show(8, truncate = False))
```

```
+---------------------------------------+-----+
|endpoint                               |count|
+---------------------------------------+-----+
|/images/NASA-logosmall.gif             |40090|
|/images/KSC-logosmall.gif              |23763|
|/images/MOSAIC-logosmall.gif           |15245|
|/images/USA-logosmall.gif              |15142|
|/images/WORLD-logosmall.gif            |14773|
|/images/ksclogo-medium.gif             |13559|
|/images/launch-logo.gif                |8806 |
|/history/apollo/images/apollo-logo1.gif|7489 |
+---------------------------------------+-----+
only showing top 8 rows
```

### SQL way

```
sqlcontext.sql("SELECT endpoint, count(*) AS count FROM parsed_table \
WHERE status_code != 200 GROUP BY endpoint ORDER BY count DESC LIMIT 8").show(truncate = False)
```

```
+---------------------------------------+-----+
|endpoint                               |count|
+---------------------------------------+-----+
|/images/NASA-logosmall.gif             |40090|
|/images/KSC-logosmall.gif              |23763|
|/images/MOSAIC-logosmall.gif           |15245|
|/images/USA-logosmall.gif              |15142|
|/images/WORLD-logosmall.gif            |14773|
|/images/ksclogo-medium.gif             |13559|
|/images/launch-logo.gif                |8806 |
|/history/apollo/images/apollo-logo1.gif|7489 |
+---------------------------------------+-----+
```

### 6. How many unique hosts are there in the entire log?

### RDD way

```
parsed_rdd2.map(lambda line: line[0]).distinct().count()
```

```
137978
```

### DataFrame way

```
parsed_df.select(parsed_df['host']).distinct().count()
```

```
137978
```

### SQL way

```
sqlcontext.sql("SELECT  count(distinct(host)) AS unique_host_count FROM parsed_table ").sh
ow(truncate = False)
```

```
+-----------------+
|unique_host_count|
+-----------------+
|137978           |
+-----------------+
```

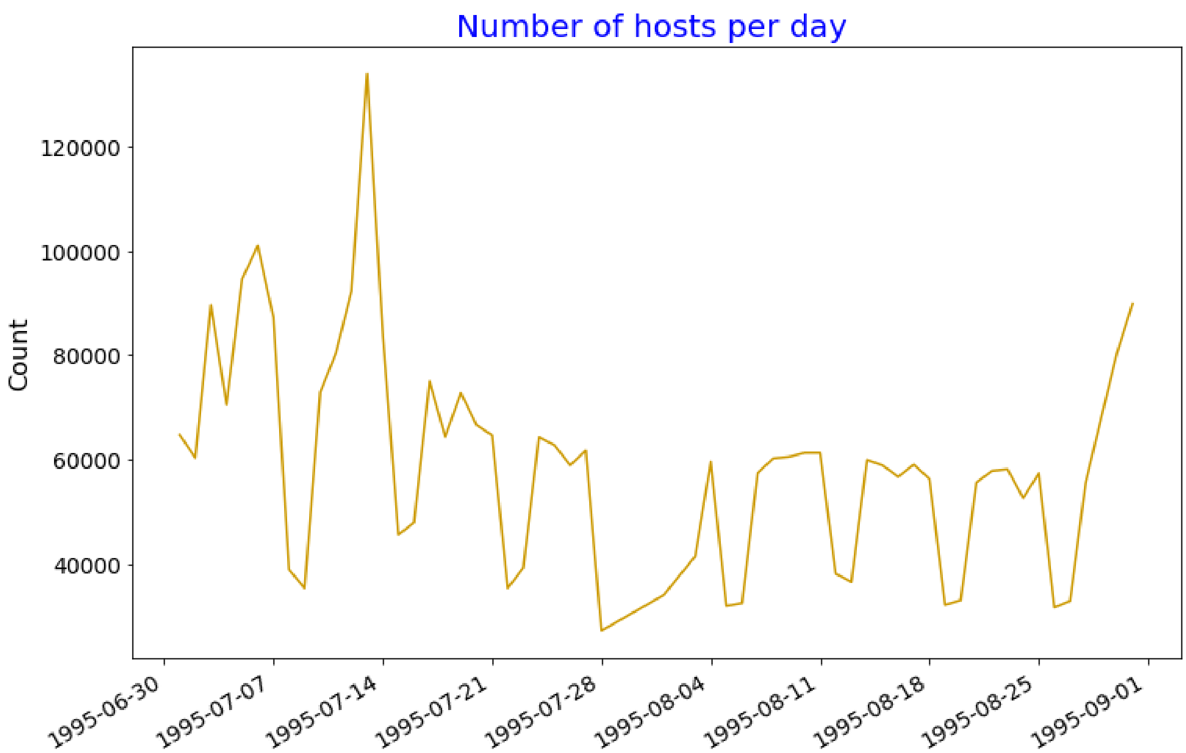
### 7. Get the number of daily hosts.

### RDD way

```
from datetime import datetime
def day_month(line):
    date_time = line[3]
    return datetime.strptime(date_time[:11], "%d/%b/%Y")
```

```
result = parsed_rdd2.map(lambda line:  (day_month(line), 1)).reduceByKey(lambda a, b: a +
b).collect()
```

```
day = [x[0] for x in result]
count = [x[1] for x in result]
day_count_dct = {'day':day, 'count':count}
day_count_df = pd.DataFrame(day_count_dct )

myplot = day_count_df.plot(figsize = (12,8), kind = "line", color = "#cc9900",
                               x = "day", y = "count", legend = False)


plt.ylabel("Count", fontsize = 16)
plt.xlabel("")
plt.title("Number of hosts per day ", fontsize = 20, color = 'b')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




Now, let's just display the first ten values to compare with results from the other methods.

```
parsed_rdd2.map(lambda line:  (day_month(line), 1)).reduceByKey(lambda a, b: a + b).takeOr
dered(10, lambda x: x[0])
```

```
[(datetime.datetime(1995, 7, 1, 0, 0), 64714),
 (datetime.datetime(1995, 7, 2, 0, 0), 60265),
 (datetime.datetime(1995, 7, 3, 0, 0), 89584),
 (datetime.datetime(1995, 7, 4, 0, 0), 70452),
 (datetime.datetime(1995, 7, 5, 0, 0), 94575),
 (datetime.datetime(1995, 7, 6, 0, 0), 100960),
 (datetime.datetime(1995, 7, 7, 0, 0), 87233),
 (datetime.datetime(1995, 7, 8, 0, 0), 38866),
 (datetime.datetime(1995, 7, 9, 0, 0), 35272),
 (datetime.datetime(1995, 7, 10, 0, 0), 72860)]
```

### DataFrame way

```
from datetime import datetime
from pyspark.sql.functions import col,udf
from pyspark.sql.types import TimestampType

myfunc =  udf(lambda x: datetime.strptime(x, '%d/%b/%Y:%H:%M:%S'), TimestampType())
parsed_df2 = parsed_df.withColumn('date_time', myfunc(col('date_time')))
```

```
from pyspark.sql.functions import date_format, month,dayofmonth

parsed_df2 = parsed_df2.withColumn("month", month(col("date_time"))).\
                                   withColumn("DayOfmonth", dayofmonth(col("date_time")))

n_hosts_by_day  = parsed_df2.groupBy(["month", "DayOfmonth"]).count().orderBy((["month", "DayOfmonth"]))
n_hosts_by_day.show(n = 10)
```

```
+-----+----------+------+
|month|DayOfmonth| count|
+-----+----------+------+
|    7|         1| 64714|
|    7|         2| 60265|
|    7|         3| 89584|
|    7|         4| 70452|
|    7|         5| 94575|
|    7|         6|100960|
|    7|         7| 87233|
|    7|         8| 38866|
|    7|         9| 35272|
|    7|        10| 72860|
+-----+----------+------+
only showing top 10 rows
```

### SQL way

```
parsed_df2.createOrReplaceTempView("parsed_df2_table")
sqlcontext.sql("SELECT month, DayOfmonth, count(*) As count FROM parsed_df2_table GROUP BY
 month, DayOfmonth\
                 ORDER BY  month, DayOfmonth LIMIT 10").show()
```

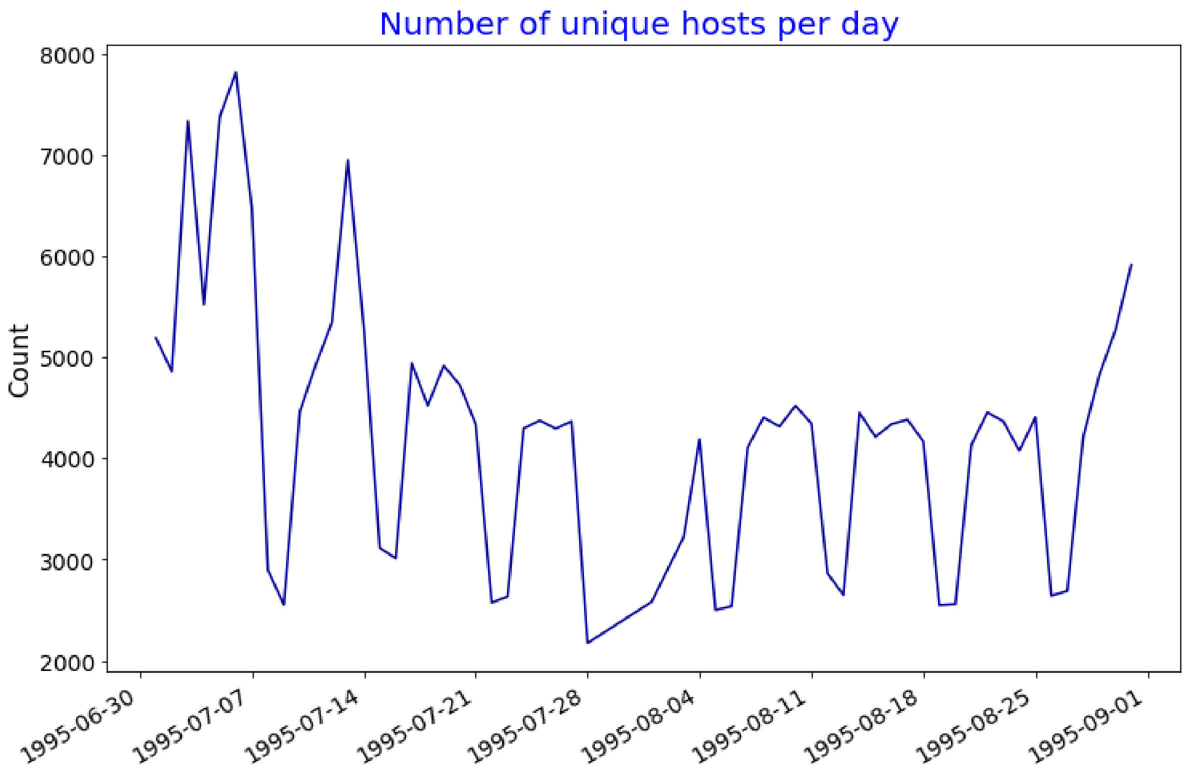
```
+-----+----------+------+
|month|DayOfmonth| count|
+-----+----------+------+
|    7|         1| 64714|
|    7|         2| 60265|
|    7|         3| 89584|
|    7|         4| 70452|
|    7|         5| 94575|
|    7|         6|100960|
|    7|         7| 87233|
|    7|         8| 38866|
|    7|         9| 35272|
|    7|        10| 72860|
+-----+----------+------+
```

#### 8. Number of unique hosts per day

#### RDD way

```
result = (parsed_rdd2.map(lambda line:  (day_month(line),line[0]))
          .groupByKey().mapValues(set)
          .map(lambda x: (x[0], len(x[1])))).collect()
```

```
day = [x[0] for x in result]
count = [x[1] for x in result]
day_count_dct = {'day':day, 'count':count}
day_count_df = pd.DataFrame(day_count_dct )

myplot = day_count_df.plot(figsize = (12,8), kind = "line", color = "#000099",
                               x = "day", y = "count", legend = False)


plt.ylabel("Count", fontsize = 16)
plt.xlabel("")
plt.title("Number of unique hosts per day ", fontsize = 20, color = 'b')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




Now, let's display 10 days with the highest values to compare with results from the other methods

```
(parsed_rdd2.map(lambda line:  (day_month(line),line[0]))
          .groupByKey().mapValues(set)
          .map(lambda x: (x[0], len(x[1])))).takeOrdered(10, lambda x: x[0])
```

```
[(datetime.datetime(1995, 7, 1, 0, 0), 5192),
 (datetime.datetime(1995, 7, 2, 0, 0), 4859),
 (datetime.datetime(1995, 7, 3, 0, 0), 7336),
 (datetime.datetime(1995, 7, 4, 0, 0), 5524),
 (datetime.datetime(1995, 7, 5, 0, 0), 7383),
 (datetime.datetime(1995, 7, 6, 0, 0), 7820),
 (datetime.datetime(1995, 7, 7, 0, 0), 6474),
 (datetime.datetime(1995, 7, 8, 0, 0), 2898),
 (datetime.datetime(1995, 7, 9, 0, 0), 2554),
 (datetime.datetime(1995, 7, 10, 0, 0), 4464)]
```

**SQL way**

```
sqlcontext.sql("SELECT  DATE(date_time) Date, COUNT(DISTINCT host) AS totalUniqueHosts FRO
M\
                parsed_df2_table GROUP   BY  DATE(date_time) ORDER BY DATE(date_time) ASC"
).show(n = 10)
```

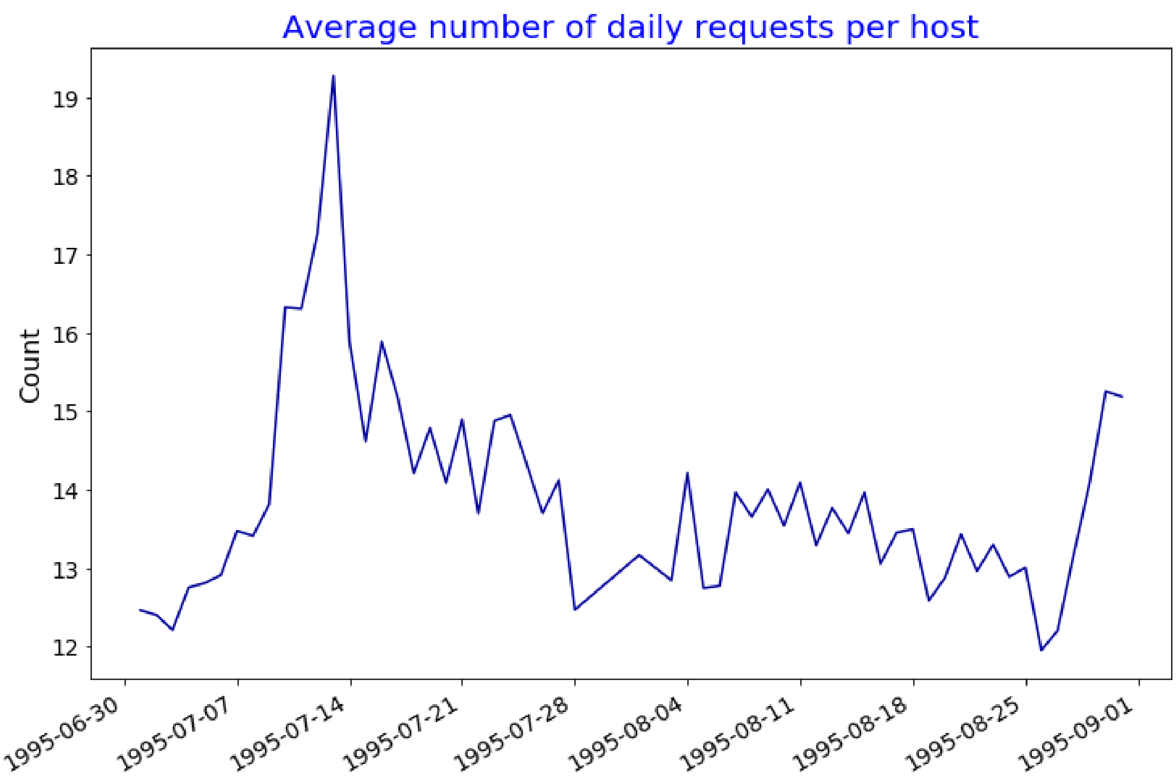
```
+----------+----------------+
|      Date|totalUniqueHosts|
+----------+----------------+
|1995-07-01|            5192|
|1995-07-02|            4859|
|1995-07-03|            7336|
|1995-07-04|            5524|
|1995-07-05|            7383|
|1995-07-06|            7820|
|1995-07-07|            6474|
|1995-07-08|            2898|
|1995-07-09|            2554|
|1995-07-10|            4464|
+----------+----------------+
only showing top 10 rows
```

**9. Average Number of Daily Requests per Hosts**

**RDD way**

```
unique_result = (parsed_rdd2.map(lambda line:  (day_month(line),line[0]))
          .groupByKey().mapValues(set)
          .map(lambda x: (x[0], len(x[1]))))

length_result = (parsed_rdd2.map(lambda line:  (day_month(line),line[0]))
          .groupByKey().mapValues(len))

joined = length_result.join(unique_result).map(lambda a: (a[0], (a[1][0])/(a[1][1]))).coll
ect()
```

```
day = [x[0] for x in joined]
count = [x[1] for x in joined]
day_count_dct = {'day':day, 'count':count}
day_count_df = pd.DataFrame(day_count_dct )

myplot = day_count_df.plot(figsize = (12,8), kind = "line", color = "#000099",
                               x = "day", y = "count", legend = False)


plt.ylabel("Count", fontsize = 16)
plt.xlabel("")
plt.title("Average number of daily requests per host", fontsize = 20, color = 'b')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




```
sorted(joined)[:10]
```

```
[(datetime.datetime(1995, 7, 1, 0, 0), 12.464175654853621),
 (datetime.datetime(1995, 7, 2, 0, 0), 12.40275776908829),
 (datetime.datetime(1995, 7, 3, 0, 0), 12.211559432933479),
 (datetime.datetime(1995, 7, 4, 0, 0), 12.753801593048516),
 (datetime.datetime(1995, 7, 5, 0, 0), 12.809833401056482),
 (datetime.datetime(1995, 7, 6, 0, 0), 12.910485933503836),
 (datetime.datetime(1995, 7, 7, 0, 0), 13.474358974358974),
 (datetime.datetime(1995, 7, 8, 0, 0), 13.411318150448585),
 (datetime.datetime(1995, 7, 9, 0, 0), 13.810493343774471),
 (datetime.datetime(1995, 7, 10, 0, 0), 16.32168458781362)]
```

### SQL way

```
sqlcontext.sql("SELECT  DATE(date_time) Date, COUNT(host)/COUNT(DISTINCT host) AS daily_re
quests_per_host FROM\
                parsed_df2_table GROUP   BY  DATE(date_time) ORDER BY DATE(date_time) ASC"
).show(n = 10)
```

```
+----------+----------------------+
|      Date|daily_requests_per_host|
+----------+----------------------+
|1995-07-01|     12.464175654853621|
|1995-07-02|      12.40275776908829|
|1995-07-03|     12.211559432933479|
|1995-07-04|     12.753801593048516|
|1995-07-05|     12.809833401056482|
|1995-07-06|     12.910485933503836|
|1995-07-07|     13.474358974358974|
|1995-07-08|     13.411318150448585|
|1995-07-09|     13.810493343774471|
|1995-07-10|      16.32168458781362|
+----------+----------------------+
only showing top 10 rows
```

#### 10. How many 404 records are in the log?

#### RDD way

```
parsed_rdd2.filter(lambda line: line[9] == '404').count()
```

```
20890
```

#### DataFrame way

```
parsed_df2.filter(parsed_df2['status_code']=="404").count()
```

```
20890
```

#### SQL way

```
sqlcontext.sql("SELECT  COUNT(*) AS logs_404_count FROM\
                parsed_df2_table WHERE status_code ==404").show(n = 10)
```

```
+--------------+
|logs_404_count|
+--------------+
|         20890|
+--------------+
```

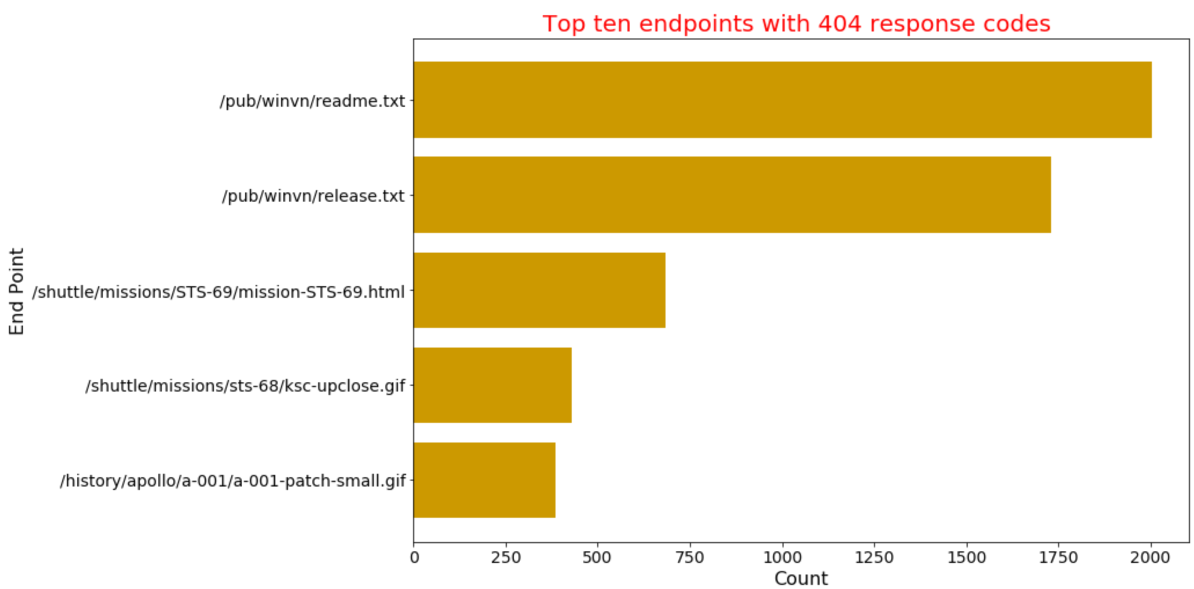
#### 11. Find the top five 404 response code endpoints

### RDD way

```
result = (parsed_rdd2.filter(lambda line: line[9] == '404')
          .map(lambda line: (line[6], 1))
          .reduceByKey(lambda a, b: a+b)
          .takeOrdered(5, lambda x: -x[1]))
result
```

```
[('/pub/winvn/readme.txt', 2004),
 ('/pub/winvn/release.txt', 1732),
 ('/shuttle/missions/STS-69/mission-STS-69.html', 683),
 ('/shuttle/missions/sts-68/ksc-upclose.gif', 428),
 ('/history/apollo/a-001/a-001-patch-small.gif', 384)]
```

```
endpoint = [x[0] for x in result]
count = [x[1] for x in result]
endpoint_count_dct = {'endpoint':endpoint, 'count':count}
endpoint_count_df = pd.DataFrame(endpoint_count_dct )

myplot = endpoint_count_df .plot(figsize = (12,8), kind = "barh", color = "#cc9900", width
 = 0.8,
                               x = "endpoint", y = "count", legend = False)

myplot.invert_yaxis()

plt.xlabel("Count", fontsize = 16)
plt.ylabel("End Point", fontsize = 16)
plt.title("Top ten endpoints with 404 response codes ", fontsize = 20, color = 'r')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




### DataFrame way

```
(parsed_df2.filter(parsed_df2['status_code']=="404")
.groupBy('endpoint').count().orderBy('count', ascending = False).show(5, truncate = False
))
```

```
+-----------------------------------------+-----+
|endpoint                                 |count|
+-----------------------------------------+-----+
|/pub/winvn/readme.txt                    |2004 |
|/pub/winvn/release.txt                   |1732 |
|/shuttle/missions/STS-69/mission-STS-69.html|683  |
|/shuttle/missions/sts-68/ksc-upclose.gif    |428  |
|/history/apollo/a-001/a-001-patch-small.gif |384  |
+-----------------------------------------+-----+
only showing top 5 rows
```

**SQL way**

```
sqlcontext.sql("SELECT endpoint, COUNT(*) AS count FROM\
                parsed_df2_table WHERE status_code ==404 GROUP BY endpoint\
                ORDER BY count DESC LIMIT 5").show(truncate = False)
```

```
+-----------------------------------------+-----+
|endpoint                                 |count|
+-----------------------------------------+-----+
|/pub/winvn/readme.txt                    |2004 |
|/pub/winvn/release.txt                   |1732 |
|/shuttle/missions/STS-69/mission-STS-69.html|683  |
|/shuttle/missions/sts-68/ksc-upclose.gif    |428  |
|/history/apollo/a-001/a-001-patch-small.gif |384  |
+-----------------------------------------+-----+
```

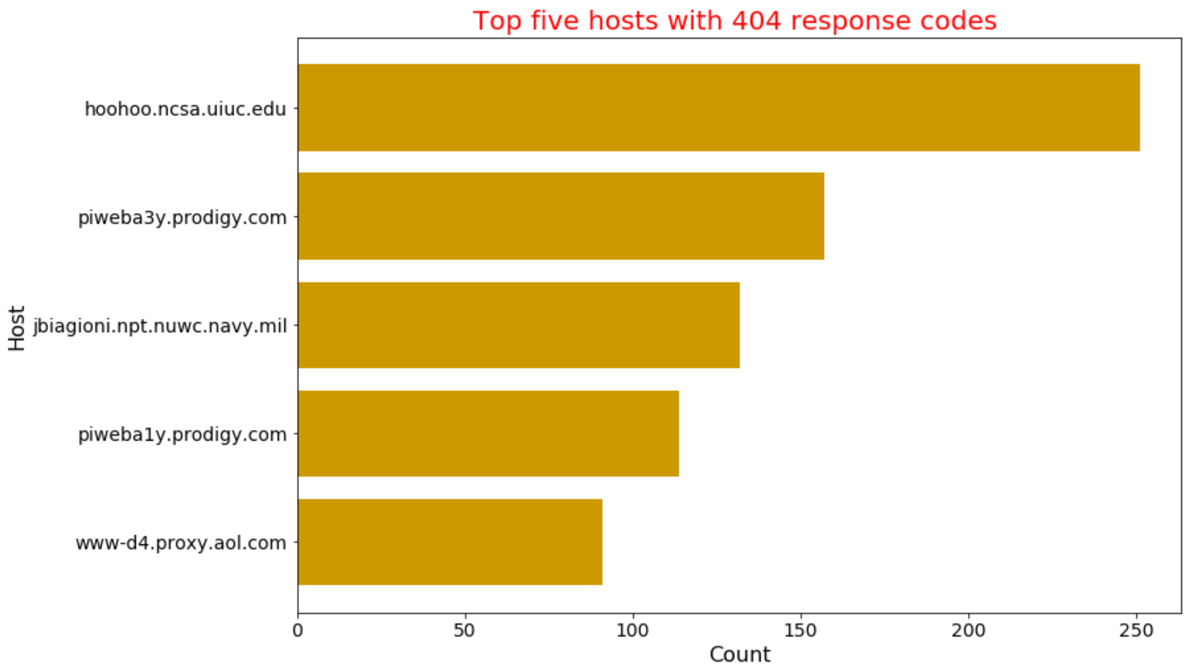
**12. Find the top five 404 response code hosts**

**RDD way**

```
result = (parsed_rdd2.filter(lambda line: line[9] == '404')
          .map(lambda line: (line[0], 1))
          .reduceByKey(lambda a, b: a+b)
          .takeOrdered(5, lambda x: -x[1]))
result
```

```
[('hoohoo.ncsa.uiuc.edu', 251),
 ('piweba3y.prodigy.com', 157),
 ('jbiagioni.npt.nuwc.navy.mil', 132),
 ('piweba1y.prodigy.com', 114),
 ('www-d4.proxy.aol.com', 91)]
```

```
host = [x[0] for x in result]
count = [x[1] for x in result]
host_count_dct = {'host':host, 'count':count}
host_count_df = pd.DataFrame(host_count_dct )

myplot = host_count_df .plot(figsize = (12,8), kind = "barh", color = "#cc9900", width = 0.8,
                                x = "host", y = "count", legend = False)

myplot.invert_yaxis()

plt.xlabel("Count", fontsize = 16)
plt.ylabel("Host", fontsize = 16)
plt.title("Top five hosts with 404 response codes ", fontsize = 20, color = 'r')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




### DataFrame way

```
(parsed_df2.filter(parsed_df2['status_code']=="404")
.groupBy('host').count().orderBy('count', ascending = False).show(5, truncate = False))
```

```
+---------------------------+-----+
|host                       |count|
+---------------------------+-----+
|hoohoo.ncsa.uiuc.edu       |251  |
|piweba3y.prodigy.com       |157  |
|jbiagioni.npt.nuwc.navy.mil|132  |
|piweba1y.prodigy.com       |114  |
|www-d4.proxy.aol.com       |91   |
+---------------------------+-----+
only showing top 5 rows
```

### SQL way

```
sqlcontext.sql("SELECT host, COUNT(*) AS count FROM\
                parsed_df2_table WHERE status_code ==404 GROUP BY host\
                ORDER BY count DESC LIMIT 5").show(truncate = False)
```

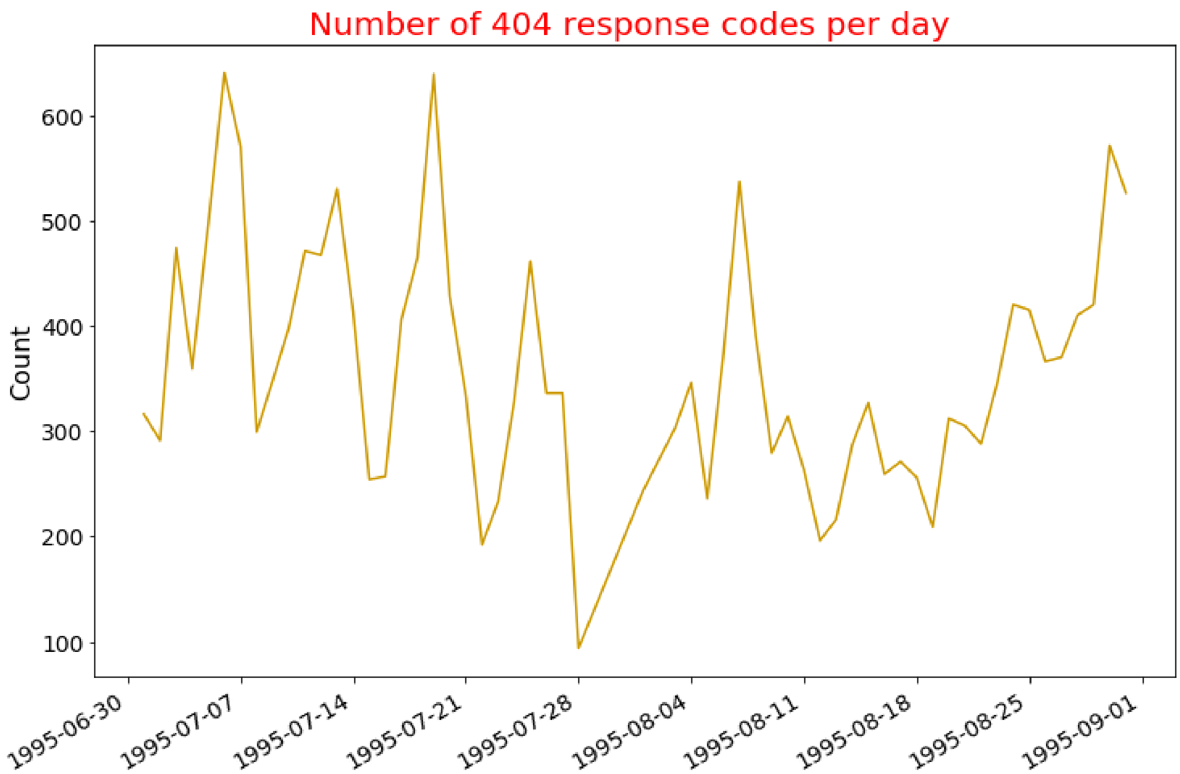
```
+----------------------------+-----+
|host                        |count|
+----------------------------+-----+
|hoohoo.ncsa.uiuc.edu        |251  |
|piweba3y.prodigy.com        |157  |
|jbiagioni.npt.nuwc.navy.mil|132  |
|piweba1y.prodigy.com        |114  |
|www-d4.proxy.aol.com        |91   |
+----------------------------+-----+
```

### 13. Create a viz of 404 response codes per day

### RDD way

```
result = (parsed_rdd2.filter(lambda line: line[9] == '404')
          .map(lambda line:  (day_month(line), 1))
          .reduceByKey(lambda a, b: a+b).collect())
```

```
day = [x[0] for x in result]
count = [x[1] for x in result]
day_count_dct = {'day':day, 'count':count}
day_count_df = pd.DataFrame(day_count_dct )

myplot = day_count_df.plot(figsize = (12,8), kind = "line", color = "#cc9900",
                               x = "day", y = "count", legend = False)


plt.ylabel("Count", fontsize = 16)
plt.xlabel("")
plt.title("Number of 404 response codes per day ", fontsize = 20, color = 'r')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




Now, let's display 10 days with the highest number of 404 errors to compare results from the other methods.

```
day_count_df.sort_values('count', ascending = False)[:10]
```

| | count | day |
|---|---|---|
| **42** | 640 | 1995-07-06 |
| **4** | 639 | 1995-07-19 |
| **44** | 571 | 1995-08-30 |
| **5** | 570 | 1995-07-07 |
| **34** | 537 | 1995-08-07 |
| **47** | 530 | 1995-07-13 |
| **13** | 526 | 1995-08-31 |
| **6** | 497 | 1995-07-05 |
| **21** | 474 | 1995-07-03 |
| **38** | 471 | 1995-07-11 |

#### DataFrame way

```
(parsed_df2.filter(parsed_df2['status_code']=="404")
.groupBy(["month", "DayOfmonth"]).count()
.orderBy('count', ascending = False).show(10)
)
```

```
+-----+----------+-----+
|month|DayOfmonth|count|
+-----+----------+-----+
|    7|         6|  640|
|    7|        19|  639|
|    8|        30|  571|
|    7|         7|  570|
|    8|         7|  537|
|    7|        13|  530|
|    8|        31|  526|
|    7|         5|  497|
|    7|         3|  474|
|    7|        11|  471|
+-----+----------+-----+
only showing top 10 rows
```

#### SQL way

```
sqlcontext.sql("SELECT  DATE(date_time) AS Date, COUNT(*) AS daily_404_erros FROM\
                parsed_df2_table WHERE status_code = 404 \
                GROUP   BY  DATE(date_time) ORDER BY daily_404_erros DESC LIMIT 10").show
()
```

```
+----------+---------------+
|      Date|daily_404_erros|
+----------+---------------+
|1995-07-06|            640|
|1995-07-19|            639|
|1995-08-30|            571|
|1995-07-07|            570|
|1995-08-07|            537|
|1995-07-13|            530|
|1995-08-31|            526|
|1995-07-05|            497|
|1995-07-03|            474|
|1995-07-11|            471|
+----------+---------------+
```

#### 14. Top five days for 404 response codes

#### RDD way

```
(parsed_rdd2.filter(lambda line: line[9] == '404')
          .map(lambda line:  (day_month(line), 1))
          .reduceByKey(lambda a, b: a+b).takeOrdered(5, lambda x: -x[1]))
```

```
[(datetime.datetime(1995, 7, 6, 0, 0), 640),
 (datetime.datetime(1995, 7, 19, 0, 0), 639),
 (datetime.datetime(1995, 8, 30, 0, 0), 571),
 (datetime.datetime(1995, 7, 7, 0, 0), 570),
 (datetime.datetime(1995, 8, 7, 0, 0), 537)]
```

This has been solved the SQL way and the RDD way in No. 13 above.

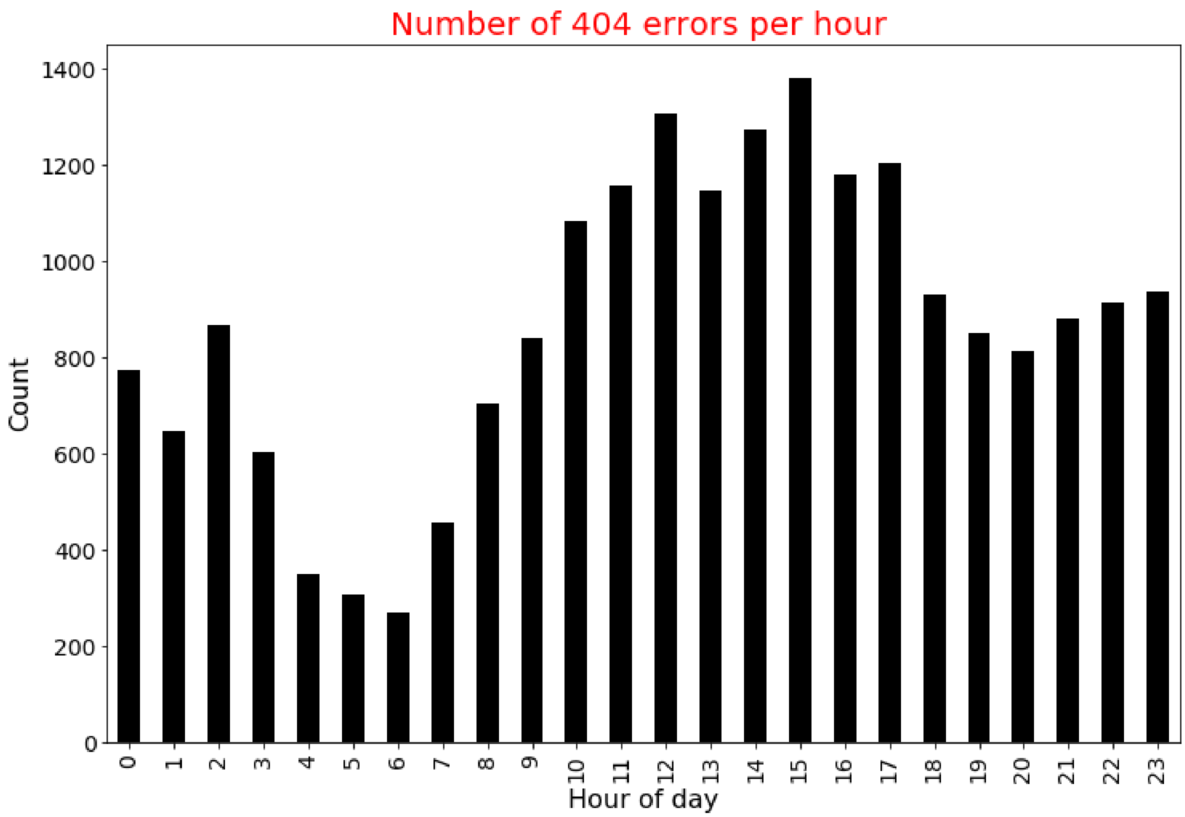
#### 15. Create an hourly 404 response codes line chart

#### RDD way

```
def date_time(line):
    date_time = line[3]
    return datetime.strptime(date_time, "%d/%b/%Y:%H:%M:%S")
```

```
result = (parsed_rdd2.filter(lambda line: line[9] == '404').map(lambda line:  (date_time(l
ine).hour, 1))
          .reduceByKey(lambda a, b: a + b)).collect()
result = sorted(result)
```

```
hour = [x[0] for x in result]
count = [x[1] for x in result]
hour_count_dct = {'hour': hour, 'count':count}
hour_count_df = pd.DataFrame(hour_count_dct )

myplot = hour_count_df.plot(figsize = (12,8), kind = "bar", color = "#000000",x ='hour',
                                y = "count", legend = False)


plt.ylabel("Count", fontsize = 16)
plt.xlabel("Hour of day", fontsize = 16)
plt.title("Number of 404 errors per hour ", fontsize = 20, color = 'r')
plt.xticks(size = 14)
plt.yticks(size = 14)
plt.show()
```




```
result[:10] # Just displaying the first five to compare results from the other methods
```

```
[(0, 774),
 (1, 648),
 (2, 868),
 (3, 603),
 (4, 351),
 (5, 306),
 (6, 269),
 (7, 458),
 (8, 705),
 (9, 840)]
```

**DataFrame way**

```
from pyspark.sql.functions import hour
parsed_df3 = parsed_df2.withColumn('hour_of_day', hour(col('date_time')))

(parsed_df3.filter(parsed_df3['status_code']=="404")
.groupBy("hour_of_day").count()
.orderBy("hour_of_day", ascending = True).show(10))
```

```
+-----------+-----+
|hour_of_day|count|
+-----------+-----+
|          0|  774|
|          1|  648|
|          2|  868|
|          3|  603|
|          4|  351|
|          5|  306|
|          6|  269|
|          7|  458|
|          8|  705|
|          9|  840|
+-----------+-----+
only showing top 10 rows
```

**SQL way**

```
sqlcontext.sql("SELECT  HOUR(date_time) AS hour, COUNT(*) AS hourly_404_erros FROM\
                parsed_df2_table WHERE status_code = 404 \
                GROUP BY HOUR(date_time) ORDER BY HOUR(date_time) LIMIT 10").show(n = 100)
```

```
+----+----------------+
|hour|hourly_404_erros|
+----+----------------+
|   0|             774|
|   1|             648|
|   2|             868|
|   3|             603|
|   4|             351|
|   5|             306|
|   6|             269|
|   7|             458|
|   8|             705|
|   9|             840|
+----+----------------+
```

**This is enough for today. See you in the next part of the DataFrames Vs RDDs in Spark tutorial series.**

# Spark RDDs Vs DataFrames vs SparkSQL - Part 4 : Set Operators

This is the fourth tutorial on the Spark RDDs Vs DataFrames vs SparkSQL blog post series. The first one is available here (http://datascience-enthusiast.com/Python/DataFramesVsRDDsSpark-Part1.html). In the first part, we saw how to retrieve, sort and filter data using Spark RDDs, DataFrames and SparkSQL. In the second part (here) (http://datascience-enthusiast.com/Python/DataFramesVsRDDsVsSQLSpark-Part2.html), we saw how to work with multiple tables in Spark the RDD way, the DataFrame way and with SparkSQL. In the third part (available here) (http://datascience-enthusiast.com/Python/DataFramesVsRDDsVsSQLSpark-Part3.html) of the blog post series, we performed web server log analysis using real-world text-based production logs. In this fourth part, we will see set operators in Spark the RDD way, the DataFrame way and the SparkSQL way.

Also, check out my other recent blog posts on Spark on Analyzing the Bible and the Quran using Spark (http://datascience-enthusiast.com/Python/analyzing_bible_quran_with_spark.html) and Spark DataFrames: Exploring Chicago Crimes (http://datascience-enthusiast.com/Python/SparkDataFrames-ExploringChicagoCrimes.html).

The data and the notebooks can be downloaded from my GitHub repository (https://github.com/fissehab/Spark_certification).

The three types of set operators in RDD, DataFrame and SQL approach are shown below.

**RDD**

- union
- intersection
- subtract

**DataFrame**

- unionAll
- intersect
- subtract

**SparkSQL**

- union all
- intersect
- except

The inputs set operations expect have to have the same variables (columns).

For this tutorial, we will work with the **SalesLTCustomer.txt**, and **SalesLTCustomerAddress.txt** datasets. Let's answer a couple of questions using Spark Resilient Distiributed (RDD) way, DataFrame way and SparkSQL by employing set operators.

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SQLContext


conf = SparkConf().setAppName("RDD Vs DataFrames Vs SparkSQL -part 4").setMaster("local
[*]")
sc = SparkContext.getOrCreate(conf)

sqlcontext = SQLContext(sc)
```

## Create RDD

```
customer = sc.textFile("SalesLTCustomer.txt")
customer_address = sc.textFile("SalesLTCustomerAddress.txt")
```

#### Understand the data

```
customer.first()
```

```
'CustomerID\tNameStyle\tTitle\tFirstName\tMiddleName\tLastName\tSuffix\tCompa
nyName\tSalesPerson\tEmailAddress\tPhone\tPasswordHash\tPasswordSalt\trowguid
\tModifiedDate'
```

```
customer_address.first()
```

```
'CustomerID\tAddressID\tAddressType\trowguid\tModifiedDate'
```

As shown above, the data is tab delimited.

#### Remove the header from the RDD

```
customer_header = customer.first()
customer_rdd = customer.filter(lambda line: line != customer_header)

customer_address_header = customer_address.first()
customer_address_rdd = (customer_address.filter(lambda line: line != customer_address_head
er))
```

#### Create DataFrames, understand the schema and show sample data

```
customer_df = sqlcontext.createDataFrame(customer_rdd.map(lambda line: line.split("\t")),
                                         schema = customer_header.split("\t") )
```

```
customer_df.printSchema()
```

```
root
 |-- CustomerID: string (nullable = true)
 |-- NameStyle: string (nullable = true)
 |-- Title: string (nullable = true)
 |-- FirstName: string (nullable = true)
 |-- MiddleName: string (nullable = true)
 |-- LastName: string (nullable = true)
 |-- Suffix: string (nullable = true)
 |-- CompanyName: string (nullable = true)
 |-- SalesPerson: string (nullable = true)
 |-- EmailAddress: string (nullable = true)
 |-- Phone: string (nullable = true)
 |-- PasswordHash: string (nullable = true)
 |-- PasswordSalt: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: string (nullable = true)
```

```
customer_df.select(['CustomerID','FirstName','MiddleName','LastName','CompanyName']).show(
10, truncate = False)
```

```
+----------+---------+----------+----------+-------------------------+
|CustomerID|FirstName|MiddleName|LastName  |CompanyName              |
+----------+---------+----------+----------+-------------------------+
|1         |Orlando  |N.        |Gee       |A Bike Store             |
|2         |Keith    |NULL      |Harris    |Progressive Sports       |
|3         |Donna    |F.        |Carreras  |Advanced Bike Components |
|4         |Janet    |M.        |Gates     |Modular Cycle Systems    |
|5         |Lucy     |NULL      |Harrington|Metropolitan Sports Supply|
|6         |Rosmarie |J.        |Carroll   |Aerobic Exercise Company |
|7         |Dominic  |P.        |Gash      |Associated Bikes         |
|10        |Kathleen |M.        |Garza     |Rural Cycle Emporium     |
|11        |Katherine|NULL      |Harding   |Sharp Bikes              |
|12        |Johnny   |A.        |Caprio    |Bikes and Motorbikes     |
+----------+---------+----------+----------+-------------------------+
only showing top 10 rows
```

```
customer_address_df = sqlcontext.createDataFrame(customer_address_rdd.map(lambda line: lin
e.split("\t")),
                                                 schema = customer_address_header.split("
\t") )
```

```
customer_address_df.printSchema()
```

```
root
 |-- CustomerID: string (nullable = true)
 |-- AddressID: string (nullable = true)
 |-- AddressType: string (nullable = true)
 |-- rowguid: string (nullable = true)
 |-- ModifiedDate: string (nullable = true)
```

```
customer_address_df.show(10, truncate = False)
```

```
+----------+---------+-----------+------------------------------------+------
-----------------+
|CustomerID|AddressID|AddressType|rowguid                             |Modifi
edDate           |
+----------+---------+-----------+------------------------------------+------
-----------------+
|29485     |1086     |Main Office|16765338-DBE4-4421-B5E9-3836B9278E63|2003-0
9-01 00:00:00.000|
|29486     |621      |Main Office|22B3E910-14AF-4ED5-8B4D-23BBE757414D|2001-0
9-01 00:00:00.000|
|29489     |1069     |Main Office|A095C88B-D7E6-4178-A078-2ECA44214801|2001-0
7-01 00:00:00.000|
|29490     |887      |Main Office|F12E1702-D897-4035-B614-0FE2C72168A9|2002-0
9-01 00:00:00.000|
|29492     |618      |Main Office|5B3B3EB2-3F43-47ED-A20C-23697DABF23B|2002-1
2-01 00:00:00.000|
|29494     |537      |Main Office|492D92B6-31AF-47EA-8E00-7C11C7CCC20D|2001-0
9-01 00:00:00.000|
|29496     |1072     |Main Office|0A66B0F3-24BC-4148-9661-26E935CEC99A|2003-0
9-01 00:00:00.000|
|29497     |889      |Main Office|7E0B56FD-7324-4898-BEDB-2F56A843BB0F|2001-0
7-01 00:00:00.000|
|29499     |527      |Main Office|C90CB0C3-976A-4075-A2B9-13136F4F1A92|2002-0
9-01 00:00:00.000|
|29502     |893      |Main Office|8ACB1C6A-7CDF-417B-AFD5-9F4C642F3C7E|2003-0
7-01 00:00:00.000|
+----------+---------+-----------+------------------------------------+------
-----------------+
only showing top 10 rows
```

#### Register the DataFrames as Tables so as to excute SQL over the tables

```
customer_df.createOrReplaceTempView("customer_table")
customer_address_df.createOrReplaceTempView("customer_address_table")
sales_address_df.createOrReplaceTempView("sales_address_table")
```

## 1. Retrieve customers with only a main office address

```
Write a query that returns the company name of each company that appears in a table
 of customers with a ‘Main Office’ address, but not in a table of customers with a
 ‘Shipping’ address.
```

#### SparkSQL way

```
sql1 = sqlcontext.sql("SELECT c.CompanyName \
                FROM customer_table AS c INNER JOIN customer_address_table AS ca \
                ON c.CustomerID = ca.CustomerID INNER JOIN sales_address_table AS sa \
                ON ca.AddressID = sa.AddressID\
                WHERE ca.AddressType = 'Main Office'\
                EXCEPT\
                SELECT c.CompanyName \
                FROM customer_table AS c INNER JOIN customer_address_table AS ca \
                ON c.CustomerID = ca.CustomerID INNER JOIN sales_address_table AS sa \
                ON ca.AddressID = sa.AddressID\
                WHERE ca.AddressType = 'Shipping' ORDER BY CompanyName")
sql1.show(5, truncate = False)
```

```
+---------------------------+
|CompanyName                |
+---------------------------+
|A Bike Store               |
|A Great Bicycle Company    |
|A Typical Bike Shop        |
|Acceptable Sales & Service|
|Action Bicycle Specialists|
+---------------------------+
only showing top 5 rows
```

### DataFrame way

```
df1 = ( (
         customer_df.join(customer_address_df.filter(customer_address_df.AddressType == 'M
ain Office'),
                          'CustomerID', 'inner')
         .join(sales_address_df, "AddressID", 'inner').select("CompanyName")
        )

        .subtract

        (
            (customer_df.join(customer_address_df.filter(customer_address_df.AddressType =
= 'Shipping'),
                              'CustomerID', 'inner')
             .join(sales_address_df, "AddressID", 'inner').select("CompanyName")
            )
        )
    ).orderBy('CompanyName')

df1.show(5, truncate = False)
```

```
+--------------------------+
|CompanyName               |
+--------------------------+
|A Bike Store              |
|A Great Bicycle Company   |
|A Typical Bike Shop       |
|Acceptable Sales & Service|
|Action Bicycle Specialists|
+--------------------------+
only showing top 5 rows
```

### RDD way

```
rdd1 = (
           (customer_rdd.map(lambda line: (line.split("\t")[0],line.split("\t")[7]))

           .join(
             customer_address_rdd.filter(lambda line: line.split("\t")[2] =='Main Office')
             .map(lambda line: (line.split("\t")[0], (line.split("\t")[1],
                                                             line.split("\t")[2])))
               )
            .map(lambda line: line[1][0]).distinct()

              )
         .subtract(
                (customer_rdd.map(lambda line: (line.split("\t")[0],line.split("\t")[7]))

            .join(
                 customer_address_rdd.filter(lambda line: line.split("\t")[2] =='Shipping'
)
                 .map(lambda line: (line.split("\t")[0], (line.split("\t")[1],
                                                                 line.split("\t")[2])))
                 )
             .map(lambda line: line[1][0]).distinct()

              )
           )
        ).collect()
```

```
sorted(rdd1)[:5]
```

```
['A Bike Store',
 'A Great Bicycle Company',
 'A Typical Bike Shop',
 'Acceptable Sales & Service',
 'Action Bicycle Specialists']
```

We see that the first five companies from the SparkSQL way, RDD way and DataFrame way are the same but let's compare all the results.

The results from the SQL and DataFrame are of type **pyspark.sql.types.Row**. So, let's make them orginary Python lists.

```
df1.collect()[:5]
```

```
[Row(CompanyName='A Bike Store'),
 Row(CompanyName='A Great Bicycle Company'),
 Row(CompanyName='A Typical Bike Shop'),
 Row(CompanyName='Acceptable Sales & Service'),
 Row(CompanyName='Action Bicycle Specialists')]
```

```
df = [i[0] for i in  df1.collect()]
df[:5]
```

```
['A Bike Store',
 'A Great Bicycle Company',
 'A Typical Bike Shop',
 'Acceptable Sales & Service',
 'Action Bicycle Specialists']
```

```
sql1.collect()[:5]
```

```
[Row(CompanyName='A Bike Store'),
 Row(CompanyName='A Great Bicycle Company'),
 Row(CompanyName='A Typical Bike Shop'),
 Row(CompanyName='Acceptable Sales & Service'),
 Row(CompanyName='Action Bicycle Specialists')]
```

```
sql = [i[0] for i in  sql1.collect()]
sql[:5]
```

```
['A Bike Store',
 'A Great Bicycle Company',
 'A Typical Bike Shop',
 'Acceptable Sales & Service',
 'Action Bicycle Specialists']
```

Now, let's see if they have the same length.

```
[len(sql), len(rdd1), len(df)]
```

```
[396, 396, 396]
```

Next, let's check if they have the same elements. First, we have to soft our lists.

```
sorted(sql) ==sorted(rdd1)
```

```
True
```

```
sorted(sql) ==sorted(df)
```

```
True
```

```
sorted(df) ==sorted(rdd1)
```

```
True
```

Therefore, we see that the results from the SparkSQL appraoch, DataFrame approach and RDD approach are the same.

**2.Retrieve only customers with both a main office address and a shipping address**

Write a query that returns the company name of each company that appears in a table of customers with a ‘Main Office’ address, and also in a table of customers with a ‘Shipping’ address.

**SparkSQL way**

```
sqlcontext.sql("SELECT c.CompanyName \
                FROM customer_table AS c INNER JOIN customer_address_table AS ca \
                ON c.CustomerID = ca.CustomerID\
                WHERE ca.AddressType = 'Main Office'\
                INTERSECT\
                SELECT c.CompanyName \
                FROM customer_table AS c INNER JOIN customer_address_table AS ca \
                ON c.CustomerID = ca.CustomerID \
                WHERE ca.AddressType = 'Shipping'ORDER BY CompanyName").show(truncate = Fa
lse)
```

```
+---------------------------+
|CompanyName                |
+---------------------------+
|All Cycle Shop             |
|Center Cycle Shop          |
|Elite Bikes                |
|Family's Favorite Bike Shop|
|Hardware Components        |
|Modular Cycle Systems      |
|Progressive Sports         |
|Racing Toys                |
|Safe Cycles Shop           |
|Sample Bike Store          |
+---------------------------+
```

There are only ten companies that have ‘Main Office’ address and ‘Shipping’ address.

**DataFrame way**

```
(  (customer_df.join(customer_address_df.filter(customer_address_df.AddressType == 'Main O
ffice'),
                          'CustomerID', 'inner')
         .select("CompanyName")
    )

    .intersect

    (

    customer_df.join(customer_address_df.filter(customer_address_df.AddressType == 'Shippi
ng'),
                      'CustomerID', 'inner')
     .select("CompanyName")
    )
   .orderBy("CompanyName")
   .show(truncate = False)
)
```

```
+---------------------------+
|CompanyName                |
+---------------------------+
|All Cycle Shop             |
|Center Cycle Shop          |
|Elite Bikes                |
|Family's Favorite Bike Shop|
|Hardware Components        |
|Modular Cycle Systems      |
|Progressive Sports         |
|Racing Toys                |
|Safe Cycles Shop           |
|Sample Bike Store          |
+---------------------------+
```

As shown above, the results from the SparkSQL approach and DataFrame approach are the same.

**RDD way**

```
result = (
           (customer_rdd.map(lambda line: (line.split("\t")[0],line.split("\t")[7]))

           .join(
             customer_address_rdd.filter(lambda line: line.split("\t")[2] =='Main Office')
             .map(lambda line: (line.split("\t")[0], (line.split("\t")[1],
                                                      line.split("\t")[2])))
               )
            .map(lambda line: line[1][0])

              )
         .intersection(
                (customer_rdd.map(lambda line: (line.split("\t")[0],line.split("\t")[7]))

            .join(
                 customer_address_rdd.filter(lambda line: line.split("\t")[2] =='Shipping'
)
                 .map(lambda line: (line.split("\t")[0], (line.split("\t")[1],
                                                          line.split("\t")[2])))
                 )
             .map(lambda line: line[1][0])

              )

           )

        ).collect()

sorted(result)
```

```
['All Cycle Shop',
 'Center Cycle Shop',
 'Elite Bikes',
 "Family's Favorite Bike Shop",
 'Hardware Components',
 'Modular Cycle Systems',
 'Progressive Sports',
 'Racing Toys',
 'Safe Cycles Shop',
 'Sample Bike Store']
```

The results from the RDD way are also the same to the DataFrame way and the SparkSQL way.

**This is enough for today. In the next part of the Spark RDDs Vs DataFrames vs SparkSQL tutorial series, I will come with a different topic. If you have any questions, or suggestions, feel free to drop them below.**

## Leveraging Hive with Spark using Python

In this blog post, we will see how to use Spark with Hive, particularly:

```
- how to create and use Hive databases
- how to create and use Hive tables
- how to load data to Hive tables
- how to insert data to Hive tables
- how to read data from Hive tables
- we will also see how to save dataframes to any Hadoop supported file system
```

To work with hive, we have to instantiate SparkSession with Hive support, including connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions if we are using Spark 2.0.0 and later. If we are using earleir Spark versions, we have to use **HiveContext** which is variant of Spark SQL that integrates with data stored in Hive. Even when we do not have an existing Hive deployment, we can still enable Hive support.

In this tutorial, I am using stand alone Spark. When not configured by the hive-site.xml, the context automatically creates metastore_db in the current directory. As shown below, initially, we do not have metastore_db but after we instantiate SparkSession with Hive support, we see that metastore_db has been created. Further, when we excute create database command, **spark-warehouse** is created.

First, let's see what we have in the current working directory.

In [2]:
```python
import os
os.listdir(os.getcwd())
```

Out[2]:
```
['Leveraging Hive with Spark using Python.ipynb',
 'metastore_db',
 'output',
 '.ipynb_checkpoints',
 'ratings.csv',
 'spark-warehouse',
 'movies.csv',
 'yahoo_stocks.csv',
 'tags.csv',
 'links.csv',
 'derby.log']
```

Initially, we do not have **metastore_db**.

In [3]:
```
from pyspark.sql import SparkSession
spark = SparkSession.builder.enableHiveSupport().getOrCreate()
```

Now, let's check if metastore_db has been created.

In [4]:
```
os.listdir(os.getcwd())
```

Out[4]:
```
['Leveraging Hive with Spark using Python.ipynb',
 'metastore_db',
 '.ipynb_checkpoints',
 'derby.log']
```

Now, as you can see above, **metastore_db** has been created.

Now, we can use Hive commands to see databases and tables. However, at this point, we do not have any database or table. We will create them below.

In [5]:
```
spark.sql('show databases').show()
```

```
+------------+
|databaseName|
+------------+
|     default|
+------------+
```

In [6]:
```
spark.sql('show tables').show()
```

```
+--------+---------+-----------+
|database|tableName|isTemporary|
+--------+---------+-----------+
+--------+---------+-----------+
```

We can see the functions in Spark.SQL using the command below. At the time of this writing, we have about 250 functions.

In [7]:
```
fncs =  spark.sql('show functions').collect()
len(fncs)
```

Out[7]:
```
252
```

Let's see some of them.

In [8]:
```
for i in fncs[100:111]:
    print(i[0])
```

```
initcap
inline
inline_outer
input_file_block_length
input_file_block_start
input_file_name
instr
int
isnan
isnotnull
isnull
```

By the way, we can see what a function is used for and what the arguments are as below.

In [9]:
```
spark.sql("describe function instr").show(truncate = False)
```

```
+-----------------------------------------------------------------------------
-------------------------+
|function_desc
                         |
+-----------------------------------------------------------------------------
-------------------------+
|Function: instr
                         |
|Class: org.apache.spark.sql.catalyst.expressions.StringInstr
                         |
|Usage: instr(str, substr) - Returns the (1-based) index of the first occurre
nce of `substr` in `str`.|
+-----------------------------------------------------------------------------
-------------------------+
```

Now, let's create a database. The data we will use is MovieLens 20M Dataset (http://files.grouplens.org/datasets/movielens/). We will use movies, ratings and tags data sets.

In [10]:
```
spark.sql('create database movies')
```

Out[10]: DataFrame[]

Let's check if our database has been created.

In [12]:
```
spark.sql('show databases').show()
```

```
+------------+
|databaseName|
+------------+
|     default|
|      movies|
+------------+
```

Yes, movies database has been created.

Now, let's download the data. I am using Jupyter Notebook so **!** enabes me to use shell commands.

In [12]:
```
! wget http://files.grouplens.org/datasets/movielens/ml-latest.zip
```

```
--2018-01-10 22:07:23--  http://files.grouplens.org/datasets/movielens/ml-lat
est.zip
Resolving files.grouplens.org (files.grouplens.org)... 128.101.34.235
Connecting to files.grouplens.org (files.grouplens.org)|128.101.34.235|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 248434223 (237M) [application/zip]
Saving to: ‘ml-latest.zip’

ml-latest.zip       100%[===================>] 236.92M  1.02MB/s    in 2m 40s


2018-01-10 22:10:04 (1.48 MB/s) - ‘ml-latest.zip’ saved [248434223/248434223]
```

Now, let's create tables: in textfile format, in ORC and in AVRO format. But first, we have to make sure we are using the movies database by switching to it using the command below.

In [13]:
```
spark.sql('use movies')
```

Out[13]: DataFrame[]

The movies dataset has movieId, title and genres fields. The ratings dataset, on the other hand, as userId, movieID, rating and timestamp fields. Now, let's create the tables.

Please refer to the [Hive manual (http://files.grouplens.org/datasets/movielens/ml-latest.zip)](http://files.grouplens.org/datasets/movielens/ml-latest.zip) for details on how to create tables and load/insert data into the tables.

In [ ]:
```
spark.sql('create table movies \
         (movieId int,title string,genres string) \
         row format delimited fields terminated by ","\
         stored as textfile')                                              # i
n textfile format
```

In [129]:
```
spark.sql("create table ratings\
           (userId int,movieId int,rating float,timestamp string)\
           stored as ORC" )                                                # i
n ORC format
```

Out[129]: DataFrame[]

Let's create another table in AVRO format. We will insert count of movies by generes into it later.

In [18]:
```
spark.sql("create table genres_by_count\
           ( genres string,count int)\
           stored as AVRO" )                                               # i
n AVRO format
```

Out[18]: DataFrame[]

Now, let's see if the tables have been created.

In [19]:
```
spark.sql("show tables").show()
```

```
+--------+---------------+-----------+
|database|      tableName|isTemporary|
+--------+---------------+-----------+
|  movies|genres_by_count|      false|
|  movies|         movies|      false|
|  movies|        ratings|      false|
|  movies|           tags|      false|
+--------+---------------+-----------+
```

We see all the tables we created above.

We can get information about a table as below. If we do not include formatted or extended in the command, we see only information about the columns. But now, we see even its location, the database and other attributes.

In [23]:
```
spark.sql("describe formatted ratings").show(truncate = False)
```

```
+----------------------------+---------------------------------------------------
-------------------+-------+
|col_name                    |data_type
                    |comment|
+----------------------------+---------------------------------------------------
-------------------+-------+
|userId                      |int
                    |null   |
|movieId                     |int
                    |null   |
|rating                      |float
                    |null   |
|timestamp                   |string
                    |null   |
|                            |
                    |       |
|# Detailed Table Information|
                    |       |
|Database                    |movies
                    |       |
|Table                       |ratings
                    |       |
|Owner                       |fish
                    |       |
|Created                     |Thu Jan 11 20:28:31 EST 2018
                    |       |
|Last Access                 |Wed Dec 31 19:00:00 EST 1969
                    |       |
|Type                        |MANAGED
                    |       |
|Provider                    |hive
                    |       |
|Table Properties            |[transient_lastDdlTime=1515720511]
                    |       |
|Location                    |file:/home/fish/MySpark/HiveSpark/spark-warehou
se/movies.db/ratings|       |
|Serde Library               |org.apache.hadoop.hive.ql.io.orc.OrcSerde
                    |       |
|InputFormat                 |org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
                    |       |
|OutputFormat                |org.apache.hadoop.hive.ql.io.orc.OrcOutputForma
t                   |       |
|Storage Properties          |[serialization.format=1]
                    |       |
|Partition Provider          |Catalog
                    |       |
+----------------------------+---------------------------------------------------
-------------------+-------+
```

Now let's load data to the movies table. We can load data from a local file system or from any hadoop supported file system. If we are using a hadoop directory, we have to remove **local** from the command below. Please refer the hive manual (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML#LanguageManualDML-Loadingfilesintotables) for details. If we are loading it just one time, we do not need to include **overwrite**. However, if there is possiblity that we could run the code more than one time, including **overwrite** is important not to append the same dataset to the table again and again. Hive does not do any transformation while loading data into tables. Load operations are currently pure copy/move operations that move datafiles into locations corresponding to Hive tables. Hive does some minimal checks to make sure that the files being loaded match the target table. So, pay careful attention to your code.

In [24]:
```
spark.sql("load data local inpath '/home/fish/MySpark/HiveSpark/movies.csv'\
                 overwrite into table movies")
```

Out[24]:
```
DataFrame[]
```

Rather than loading the data as a bulk, we can pre-process it and create a dataframe and insert our dataframe to the table. Let's insert the ratings data by first creating a dataframe.

We can create dataframes in two ways.

- by using the Spark SQL read function such as spark.read.csv, spark.read.json, spark.read.orc, spark.read.avro, spark.rea.parquet, etc.
- by reading it in as an RDD and converting it to a dataframe after pre-processing it

Let's specify schema for the ratings dataset.

In [4]:
```
from pyspark.sql.types import *

schema = StructType([
             StructField('userId', IntegerType()),
             StructField('movieId', IntegerType()),
             StructField('rating', DoubleType()),
             StructField('timestamp', StringType())
            ])
```

Now, we can read it in as dataframe using dataframe reader as below.

In [178]:
```
ratings_df = spark.read.csv("/home/fish/MySpark/HiveSpark/ratings.csv", schema
 = schema, header = True)
```

We can see the schema of the dataframe as:

In [179]:
```
ratings_df.printSchema()
```

```
root
 |-- userId: integer (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- rating: double (nullable = true)
 |-- timestamp: string (nullable = true)
```

We can also display the first five records from the dataframe.

In [121]:
```
ratings_df.show(5)
```

```
+------+-------+------+------------+
|userId|movieId|rating|   timestamp|
+------+-------+------+------------+
|     1|    110|   1.0|1.425941529E9|
|     1|    147|   4.5|1.425942435E9|
|     1|    858|   5.0|1.425941523E9|
|     1|   1221|   5.0|1.425941546E9|
|     1|   1246|   5.0|1.425941556E9|
+------+-------+------+------------+
only showing top 5 rows
```

The second option to create a dataframe is to read it in as RDD and change it to dataframe by using the **toDF** dataframe function or createDataFrame from SparkSession . Remember, we have to use the **Row** function from pyspark.sql to use **toDF**.

In [6]:
```
from pyspark.sql import Row
from pyspark import SparkContext, SparkConf

conf = SparkConf().setMaster("local[*]")
sc = SparkContext.getOrCreate(conf)

rdd = sc.textFile("/home/fish/MySpark/HiveSpark/ratings.csv")
header = rdd.first()
ratings_df2 = rdd.filter(lambda line: line != header).map(lambda line: Row(use
rId = int(line.split(",")[0]),
                                                                      movieId =
 int(line.split(",")[1]),
                                                                      rating =
float(line.split(",")[2]),
                                                                      timestamp
 = line.split(",")[3]
                                                                     )).toDF()
```

We can also do as below:

In [7]:
```
rdd2 = rdd.filter(lambda line: line != header).map(lambda line:line.split(","
))
ratings_df2_b =spark.createDataFrame(rdd2, schema = schema)
```

We see the schema and the the first five records from ratings_df and ratings_df2 are the same.

In [123]:
```
ratings_df2.printSchema()
```

```
root
 |-- movieId: long (nullable = true)
 |-- rating: double (nullable = true)
 |-- timestamp: string (nullable = true)
 |-- userId: long (nullable = true)
```

In [125]:
```
ratings_df2.show(5)
```

```
+-------+------+----------+------+
|movieId|rating| timestamp|userId|
+-------+------+----------+------+
|    110|   1.0|1425941529|     1|
|    147|   4.5|1425942435|     1|
|    858|   5.0|1425941523|     1|
|   1221|   5.0|1425941546|     1|
|   1246|   5.0|1425941556|     1|
+-------+------+----------+------+
only showing top 5 rows
```

To insert a dataframe into a Hive table, we have to first create a temporary table as below.

In [130]:
```
ratings_df.createOrReplaceTempView("ratings_df_table") # we can also use regis
terTempTable
```

Now, let's insert the data to the ratings Hive table.

In [131]:
```
spark.sql("insert into table ratings select * from ratings_df_table")
```

Out[131]:
```
DataFrame[]
```

Next, let's check if the movies and ratings hive tables have the data.

In [25]:
```
spark.sql("select * from movies limit 10").show(truncate = False)
```

```
+-------+-----------------------------------+-------------------------------
----------+
|movieId|title                              |genres
          |
+-------+-----------------------------------+-------------------------------
----------+
|null   |title                              |genres
          |
|1      |Toy Story (1995)                   |Adventure|Animation|Children|Come
dy|Fantasy|
|2      |Jumanji (1995)                     |Adventure|Children|Fantasy
          |
|3      |Grumpier Old Men (1995)            |Comedy|Romance
          |
|4      |Waiting to Exhale (1995)           |Comedy|Drama|Romance
          |
|5      |Father of the Bride Part II (1995)|Comedy
          |
|6      |Heat (1995)                        |Action|Crime|Thriller
          |
|7      |Sabrina (1995)                     |Comedy|Romance
          |
|8      |Tom and Huck (1995)                |Adventure|Children
          |
|9      |Sudden Death (1995)                |Action
          |
+-------+-----------------------------------+-------------------------------
----------+
```

In [132]:
```
spark.sql("select * from ratings limit 10").show(truncate = False)
```

```
+------+-------+------+-------------+
|userId|movieId|rating|timestamp    |
+------+-------+------+-------------+
|52224 |51662  |3.5   |1.292347002E9|
|52224 |54286  |4.0   |1.292346944E9|
|52224 |56367  |3.5   |1.292346721E9|
|52224 |58559  |4.0   |1.292346298E9|
|52224 |59315  |3.5   |1.292346497E9|
|52224 |60069  |4.5   |1.292346644E9|
|52224 |60546  |4.5   |1.292346916E9|
|52224 |63082  |4.0   |1.292347049E9|
|52224 |68157  |3.5   |1.292347351E9|
|52224 |68358  |4.0   |1.292347043E9|
+------+-------+------+-------------+
```

We see that we can put our data in hive tables by either directly loading data in a local or hadoop file system or by creating a dataframe and registering the dataframe as a temporary table.

We can also query data in hive table and save it another hive table. Let's calculate number of movies by genres and insert those genres which occur more than 500 times to genres_by_count AVRO hive table we created above.

In [26]:
```
spark.sql("select genres, count(*) as count from movies\
          group by genres\
          having count(*) > 500 \
          order by count desc").show()
```

```
+-------------------+-----+
|             genres|count|
+-------------------+-----+
|              Drama| 5521|
|             Comedy| 3604|
|        Documentary| 2903|
| (no genres listed)| 2668|
|       Comedy|Drama| 1494|
|      Drama|Romance| 1369|
|     Comedy|Romance| 1017|
|             Horror|  944|
|Comedy|Drama|Romance|  735|
|     Drama|Thriller|  573|
|        Crime|Drama|  567|
|    Horror|Thriller|  553|
|           Thriller|  530|
+-------------------+-----+
```

In [65]:
```
spark.sql("insert into table genres_by_count \
          select genres, count(*) as count from movies\
          group by genres\
          having count(*) >= 500 \
          order by count desc")
```

Out[65]: `DataFrame[]`

Now, we can check if the data has been inserted to the hive table appropriately:

In [66]:
```
spark.sql("select * from genres_by_count order by count desc limit 3").show()
```

```
+-----------+-----+
|     genres|count|
+-----------+-----+
|      Drama| 5521|
|     Comedy| 3604|
|Documentary| 2903|
+-----------+-----+
```

We can also use data in hive tables with other dataframes by first registering the dataframes as temporary tables.

Now, let's create a temporary table from the tags dataset and then we will join it with movies and ratings tables which are in hive.

In [180]:
```
schema = StructType([
             StructField('userId', IntegerType()),
             StructField('movieId', IntegerType()),
             StructField('tag', StringType()),
             StructField('timestamp', StringType())
            ])

tags_df = spark.read.csv("/home/fish/MySpark/HiveSpark/tags.csv", schema = sch
ema, header = True)
tags_df.printSchema()
```

```
root
 |-- userId: integer (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- tag: string (nullable = true)
 |-- timestamp: string (nullable = true)
```

Next, register the dataframe as temporary table.

In [77]:
```
tags_df.registerTempTable('tags_df_table')
```

From the **show tables** hive command below, we see that three of them are permanent but two of them are temporary tables.

In [78]:
```
spark.sql('show tables').show()
```

```
+--------+----------------+-----------+
|database|       tableName|isTemporary|
+--------+----------------+-----------+
|  movies| genres_by_count|      false|
|  movies|          movies|      false|
|  movies|         ratings|      false|
|        |ratings_df_table|       true|
|        |   tags_df_table|       true|
+--------+----------------+-----------+
```

Now, lets' join the three tables by using inner join. The result is a dataframe.

In [133]:
```
joined = spark.sql("select m.title, m.genres, r.movieId, r.userId,  r.rating,
 r.timestamp as ratingTimestamp, \
              t.tag, t.timestamp as tagTimestamp from ratings as r inner join
 tags_df_table as t\
              on r.movieId = t.movieId and r.userId = t.userId inner join mov
ies as m on r.movieId = m.movieId")
```

In [134]:
```
type(joined)
```

Out[134]: pyspark.sql.dataframe.DataFrame

We can see the first five records as below.

In [135]:
```
joined.select(['title','genres','rating']).show(5, truncate = False)
```

```
+-------------------------------------------------------------+--------------
-------------+------+
|title                                                        |genres
              |rating|
+-------------------------------------------------------------+--------------
-------------+------+
|Star Wars: Episode IV - A New Hope (1977)                    |Action|Adventu
re|Sci-Fi     |4.0   |
|Star Wars: Episode IV - A New Hope (1977)                    |Action|Adventu
re|Sci-Fi     |4.0   |
|She Creature (Mermaid Chronicles Part 1: She Creature) (2001)|Fantasy|Horror
|Thriller     |2.5   |
|The Veil (2016)                                              |Horror
              |2.0   |
|A Conspiracy of Faith (2016)                                 |Crime|Drama|My
stery|Thriller|3.5   |
+-------------------------------------------------------------+--------------
-------------+------+
only showing top 5 rows
```

We can also save our dataframe in other file system.

Let's create a new directory and save the dataframe in csv, json, orc and parquet formats.

Let's see two ways to do that:

In [136]:
```
!pwd
```

```
/home/fish/MySpark/HiveSpark
```

In [96]:
```
!mkdir output
```

In [169]:
```
joined.write.csv("/home/fish/MySpark/HiveSpark/output/joined.csv", header = True)

joined.write.json("/home/fish/MySpark/HiveSpark/output/joined.json")

joined.write.orc("/home/fish/MySpark/HiveSpark/output/joined_orc")

joined.write.parquet("/home/fish/MySpark/HiveSpark/output/joined_parquet" )
```

Now, let's check if the data is there in the formats we specified.

In [170]:
```
! ls output
```

```
joined.csv  joined.json  joined_orc  joined_parquet
```

The second option to save data:

In [171]:
```
joined.write.format('csv').save("/home/fish/MySpark/HiveSpark/output/joined2.csv" , header = True)

joined.write.format('json').save("/home/fish/MySpark/HiveSpark/output/joined2.json" )

joined.write.format('orc').save("/home/fish/MySpark/HiveSpark/output/joined2_orc" )

joined.write.format('parquet').save("/home/fish/MySpark/HiveSpark/output/joined2_parquet" )
```

Now, let's see if we have data from both oprions.

In [172]:
```
! ls output
```

```
joined2.csv   joined2_orc      joined.csv   joined_orc
joined2.json  joined2_parquet  joined.json  joined_parquet
```

Similarly, let's see two ways to read the data.

First option:

In [175]:
```
read_csv = spark.read.csv('/home/fish/MySpark/HiveSpark/output/joined.csv', header = True)

read_orc = spark.read.orc('/home/fish/MySpark/HiveSpark/output/joined_orc')

read_parquet = spark.read.parquet('/home/fish/MySpark/HiveSpark/output/joined_parquet')
```

In [176]:
```
read_orc.printSchema()
```

```
root
 |-- title: string (nullable = true)
 |-- genres: string (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- userId: integer (nullable = true)
 |-- rating: float (nullable = true)
 |-- ratingTimestamp: string (nullable = true)
 |-- tag: string (nullable = true)
 |-- tagTimestamp: double (nullable = true)
```

second option:

In [186]:
```
read2_csv = spark.read.format('csv').load('/home/fish/MySpark/HiveSpark/outpu
t/joined.csv', header = True)

read2_orc = spark.read.format('orc').load('/home/fish/MySpark/HiveSpark/outpu
t/joined_orc')

read2_parquet = spark.read.format('parquet').load('/home/fish/MySpark/HiveSpar
k/output/joined_parquet')
```

In [187]:
```
read2_parquet.printSchema()
```

```
root
 |-- title: string (nullable = true)
 |-- genres: string (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- userId: integer (nullable = true)
 |-- rating: float (nullable = true)
 |-- ratingTimestamp: string (nullable = true)
 |-- tag: string (nullable = true)
 |-- tagTimestamp: double (nullable = true)
```

We can also write a dataframe into a hive table by using **insertInto**. This requires that the schema of the DataFrame is the same as the schema of the table.

Let's see the schema of the **joined** dataframe and create two hive tables: one in ORC and one in PARQUET formats to insert the dataframe into.

In [148]:
```
joined.printSchema()
```

```
root
 |-- title: string (nullable = true)
 |-- genres: string (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- userId: integer (nullable = true)
 |-- rating: float (nullable = true)
 |-- ratingTimestamp: string (nullable = true)
 |-- tag: string (nullable = true)
 |-- tagTimestamp: double (nullable = true)
```

Create ORC Hive Table:

In [152]:
```
spark.sql("create table joined_orc\
           (title string,genres string, movieId int, userId int,  rating floa
t, \
           ratingTimestamp string,tag string, tagTimestamp string )\
           stored as ORC" )
```

Out[152]: DataFrame[]

Create PARQUET Hive Table:

In [153]:
```
spark.sql("create table joined_parquet\
          (title string,genres string, movieId int, userId int,  rating float,
 \
           ratingTimestamp string,tag string, tagTimestamp string )\
           stored as PARQUET")
```

Out[153]: DataFrame[]

Let's see if the tables have been created.

In [144]:
```
spark.sql('show tables').show()
```

```
+--------+----------------+-----------+
|database|       tableName|isTemporary|
+--------+----------------+-----------+
|  movies| genres_by_count|      false|
|  movies|      joined_orc|      false|
|  movies|  joined_parquet|      false|
|  movies|          movies|      false|
|  movies|         ratings|      false|
|        |ratings_df_table|       true|
|        |   tags_df_table|       true|
+--------+----------------+-----------+
```

They are there. Now, let's insert dataframe into the tables.

In [154]:
```
joined.write.insertInto('joined_orc')
```

In [155]:
```
joined.write.insertInto('joined_parquet')
```

Finally, let's check if the data has been inserted into the hive tbales.

In [185]:
```
spark.sql('select title, genres, rating from joined_orc order by rating desc l
imit 5').show(truncate = False)
```

```
+----------------------------+--------------------------------------------+----
--+
|title                       |genres                                      |rati
ng|
+----------------------------+--------------------------------------------+----
--+
|To Die For (1995)           |Comedy|Drama|Thriller                       |5.0
  |
|Seven (a.k.a. Se7en) (1995)|Mystery|Thriller                            |5.0
  |
|Seven (a.k.a. Se7en) (1995)|Mystery|Thriller                            |5.0
  |
|Seven (a.k.a. Se7en) (1995)|Mystery|Thriller                            |5.0
  |
|Toy Story (1995)            |Adventure|Animation|Children|Comedy|Fantasy|5.0
  |
+----------------------------+--------------------------------------------+----
--+
```

In [184]:
```
spark.sql('select title, genres, rating  from joined_parquet order by rating d
esc limit 5').show(truncate = False)
```

```
+------------------------------------------+------------------------+------+
|title                                     |genres                  |rating|
+------------------------------------------+------------------------+------+
|Beautiful Girls (1996)                    |Comedy|Drama|Romance    |5.0   |
|Before Sunrise (1995)                     |Drama|Romance           |5.0   |
|Beautiful Girls (1996)                    |Comedy|Drama|Romance    |5.0   |
|Twelve Monkeys (a.k.a. 12 Monkeys) (1995)|Mystery|Sci-Fi|Thriller|5.0   |
|"Bridges of Madison County                | The (1995)"            |5.0   |
+------------------------------------------+------------------------+------+
```

Everything looks great! See you in my next tutorial on Apache Spark.