Leveraging Apache Spark for Data Analytics

Jacob Brionez, Damon Resnick, Trace Smith
Southern Methodist University
April 4, 2017

Abstract

In this study, the core functionalities of Apache Spark are discussed. A brief description of Spark and the core functionalities are presented in this work followed by an instruction on how to connect MySQL and PySpark, an API for interfacing with Python 3.5. Apache Spark was connected to an external relational database, MySQL, and evaluated for computational performance of airline flight data from the United States in 2008. Additionally, the 680 MB data file was cached in memory and queried with traditional SQL in order to implement analytics and visualizations using Python.

Introduction

The ability to process large data in an efficient fashion can be handled through traditional MapReduce in the Hadoop File System over a series of distributed nodes. However, as the era of Big Data continues to advance, a more accelerated approach for analytical applications is required. An open-source, scalable framework commonly used by Data Scientist for processing vast amounts of data over a cluster of high performing computers is Apache Spark. Whether the data is streamed in real-time such as tweets or whether an archive of unstructured data recorded from SCADA systems, Spark can be a valuable tool Data Scientist can leverage for performing data analytics at lighting-fast speeds.

Apache Spark is an open source distributed data processing framework developed specifically for Big Data. According to Spark documentation, one of the main distinguishing functionalities of Spark from Hadoop is the ability to process data extremely fast by caching the dataset in memory and processed in parallel over a cluster of nodes.[1] This allows Spark to perform up to 100x faster than Hadoop. The Spark ecosystem can run in Hadoop clusters through YARN or Spark's standalone mode. It can also process data in HDFS, HBase, Cassandra, Hive, and any Hadoop Input Format. It is designed to perform both batch processing (similar to MapReduce) and new workloads like streaming, interactive queries, and machine learning. Spark offers a Python API for standard CPython implementation, and can call into existing C libraries for Python such as NumPy and Pandas.

Spark SQL is Apache Spark's module for working with structured and unstructured data. Spark SQL lets you query structured data inside Spark programs using either SQL or a familiar DataFrame API, usable in Java, Scala, Python, and R. Spark SQL includes a cost-based optimizer, columnar storage, and code generation to make queries fast. It also scales to thousands of nodes and multiple hour long queries using the Spark engine. This provides full mid-query fault tolerance, eliminating the need for using a different engine for historical data. Spark also uses a micro-batch execution model that does not have much impact on applications, because the batches can be as short as 0.5 seconds. In most applications of streaming big data, the analytics is done over a larger window (say 10 minutes), or the latency to get data in is higher (e.g. sensors collect readings every 10 seconds). Spark's model enables “exactly-once” semantics and consistency, meaning the system gives correct results despite slow nodes or failures.[2]

Spark was created by Matei Zahria at UC Berkely in the AMPLab in 2009 and open-sourced in 2010. By 2013 it was donated to the Apache Software Foundation, and today is one of the most active projects in the Apache Software Foundation for big data. Spark is essentially a replacement for the MapReduce function of Hadoop. Where MapReduce shuffles things in and out of storage, Spark is designed to keep things in memory during processing. This simple change enables Spark to achieve speeds up to 100 times faster than MapReduce.

Spark benefits the field of Data Science by taking advantage of extending the MapReduce model to efficiently support more computations, interactive queries, and stream processing. With large datasets speed is key. It defines the difference between exploring interactive data and waiting hours for results. One of the main efficiencies Spark offers is the ability to run computations in memory, it is also more efficient for complex applications running on a local disk. Additionally, it provides a inexpensive speed boost to a wide range of processing types and is highly accessible with its offering of APIs for well known programs like Python, Java, Scala, and SQL.

Spark is designed to be easier to use in development. It is more powerful and expressive in terms of how you give it instructions to crunch data. Along with the Map and Reduce like functions from MapReduce, it also as others like Filter, Join, and Group-by, making it much easier to develop in Spark. Spark provides for lots of instructions that are a higher level of abstraction than MapReduce, and can consist of more than one single map and reduce (i.e. batch processing).

The core application programming interface of Spark is called Resilient Distributed Dataset (RDD).[3] RDD is a read-only set of objects partitioned across multiple machines that are able to be rebuilt if a partition is lost. RDD’s can also be a collection of immutable objects that can be operated on in parallel. Each RDD can be split into multiple partitions and distributed over a series of nodes in a cluster. Resilient refers to fault-tolerance with the help of a RDD lineage graph which enables it to recompute missing or damaged partitions due to node failures. Distributed refers to data residing on multiple nodes in a cluster. Dataset is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects (that represent records of the data you work with).

It is good to think of Spark as an add-on to Hadoop as a replacement of MapReduce. Basically it does what MapReduce does but in-memory. Because of this it can produce real-time analytics, from the streaming of big data resources. It has a few different modules such as Spark SQL, which enables Spark users to query structured data from HDFS using SQL. Other modules include MlLib for data analytics, statistics, and machine learning, GraphX for graphical data processing, and Streaming for the conversion of data into small data streams.

Resilient Distributed Dataset forms the basic abstraction on which Spark programming model works. RDD abstracts us away from traditional map-reduce style programs, giving us the interface of a collection (which is distributed), and hence operations requiring a boilerplate in MapReduce are now a collection of operations (e.g. groupby, joins, count, distinct, max, min etc). It also allows us to do iterative processing quite easily, by sharing RDD between operations. RDD can also be optionally cached giving a performance boost.[3]

The main benefit of the MapReduce paradigm is simplicity. MapReduce consists of two functions: Map and Reduce. These functions are defined by the user. They take as input a key/value pair and output a key/value pair. Hidden from the user, the MapReduce implementation handles parallelism, failures of nodes, data distribution and load balancing. In contrast to the loading and indexing phase of parallel DBMSs, MapReduce only loads the data in a distributed file system and then processes the data on-the-fly. MapReduce divides the dataset into smaller chunks of data and distributes them among to the cluster. While parallel DBMS make use of the declarative language SQL, MapReduce tasks can be programmed using procedural, object oriented languages, such as Java.[4]

PySpark

PySpark is the Python API for interfacing with Spark to perform big data analytics. The following section will outline a series of examples that implement Spark using Python. Initially, the objective was geared towards connecting PySpark to a relational database (i.e. MySQL) in order to examine the computational time to execute complex queries through Spark, as opposed to SQL. This is interesting because long running queries as a parallel system of Spark can perform much faster than traditional SQL which consists of extensive indices for large scale datasets. Moreover, databases such as MySQL can only use one CPU core per query, whereas Spark can use all cores on all cluster nodes and can perform up to 10 times as fast as MySQL.

Several complications were encountered during this process, mostly modifying the PATH location for PySpark to locate the executable JDBC binary Java files to establish a connection with MySQL. Once the PATH was successfully set and connection was made, the ensuing issues involved extensive processing time for PySpark to perform a simple select all (limit 5) from a single relational table. Several sources in the literature raised similar concerns of excessive computational time running PySpark on top of relational databases.[1,5,7] Since this work is centered around running Spark locally, processing the data through a distributed network such as Amazon Web Services EC2 cluster would have alleviated the resulting bottleneck. The following section outlines the work undertaken to load the data into MySQL by interfacing with the database through Python, however the end product resulted in directly reading into memory the dataset as a temporary solution to bypass the issues mentioned above.

Data Overview

In this work, the ‘flights’ dataset, containing roughly 7 million recorded flights at airports located across the United States in 2008, will be the focal point of the analysis. The source of the dataset is from the American Statistical Association (ASA)[11], however the original reference is from the U.S. Department of Transportation’s Bureau of Transportation Statistics, which tracks the on-time and delayed domestic flights. In the initial attempt we tried to import the over 7 million valued data file directly into MySQL via MySQL Workbench using the data import wizard. This method was ultimately terminated due to the inefficient import time of several consecutive days.

To handle the vastness of the data file (i.e. 680 MB), an alternative approach consisted of interfacing directly to MySQL through Python for importing the dataset. The following code below is written in Python 3.5 and was developed to interface with the database for importing and querying the ‘flights’ dataset. Moreover, the table schema will consist of two separate tables, ‘flight_info’ and ‘airports’, and each table contains 29 and 12 entities, respectively. It should be noted that in order to establish a connection with the MySQL database through Python, installing the *mysql-connector-python* package is required.

The following Python libraries will be loaded:

```
import mysql.connector
import csv
import time
```

Build connection object to interface with the MySQL database; enter root password and create ‘flights’ database

```
# Connect/Load Database
db = mysql.connector.connect(user='root',passwd="*******")
c = db.cursor()
c.execute('''CREATE DATABASE IF NOT EXISTS Flights;''')
```

Create the flight_info table with 29 entities and pass in the ‘sql’ instance for execution

```
sql='''CREATE TABLE IF NOT EXISTS flight_info(Year INT, Month Int, \
        DayofMonth INT, DayOfWeek INT, DepTime INT, CRSDepTime INT, ArrTime INT, \
        CRSArrTime INT, UniqueCarrier TEXT, FlightNum INT, TailNum CHAR(8),\
        ActualElapsedTime INT, CRSElapsedTime INT, AirTime INT, ArrDelay INT, \
        DepDelay INT, Origin TEXT, Dest TEXT, Distance INT, TaxiIn INT, TaxiOut INT, \
        Cancelled INT, CancellationCode TEXT, Diverted INT, CarrierDelay INT, \
        WeatherDelay INT, NASDelay INT, SecurityDelay INT, LateAircraftDelay INT);'''
c.execute(sql)
```

Load in the external flight data csv file (i.e. 2008.csv) for each entity; run the commit method using the connection object and then close connection to database

```
c.execute('''
    LOAD DATA LOCAL INFILE '2008.csv'
    into Table flight_info FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n' IGNORE 1 LINES
    (Year, Month, DayofMonth, DayOfWeek,
    DepTime, CRSDepTime, ArrTime, CRSArrTime,
    UniqueCarrier, FlightNum, TailNum, ActualElapsedTime,
    CRSElapsedTime, AirTime, ArrDelay, DepDelay, Origin, Dest,
    Distance, TaxiIn, TaxiOut, Cancelled, CancellationCode, Diverted,
    CarrierDelay, WeatherDelay, NASDelay, SecurityDelay,LateAircraftDelay)
''')
db.commit()
db.close()
print("Done")
```

The following query is to verify if the data was correctly imported. If so, a row of tuples will be the output format populated with the corresponding data. To control the output, only the first 5 rows will be examined.

```
# Connect/Load Database
#Enter your root password
db = mysql.connector.connect(user='root',passwd="*******",db='Flights')
c = db.cursor()
c.execute('''SELECT * FROM flight_info limit 5''')
start = time.time()
total = 0
for row in c.fetchall():
    print(row)
    total+=1
end = time.time()
print(end-start)
print("Total Observations:",total)
db.close()
```

```
(2008, 1, 3, 4, 2003, 1955, 2211, 2225, 'WN', 335, 'N712SW', 128, 150, 116, -14, 8, 'IAD', 'TPA', 810, 4, 8, 0, '',
 0, 0, 0, 0, 0, 0)
(2008, 1, 3, 4, 754, 735, 1002, 1000, 'WN', 3231, 'N772SW', 128, 145, 113, 2, 19, 'IAD', 'TPA', 810, 5, 10, 0, '', 0,
0, 0, 0, 0, 0)
(2008, 1, 3, 4, 628, 620, 804, 750, 'WN', 448, 'N428WN', 96, 90, 76, 14, 8, 'IND', 'BWI', 515, 3, 17, 0, '', 0, 0, 0,
0, 0, 0)
(2008, 1, 3, 4, 926, 930, 1054, 1100, 'WN', 1746, 'N612SW', 88, 90, 78, -6, -4, 'IND', 'BWI', 515, 3, 7, 0, '', 0, 0,
0, 0, 0, 0)
(2008, 1, 3, 4, 1829, 1755, 1959, 1925, 'WN', 3920, 'N464WN', 90, 90, 77, 34, 34, 'IND', 'BWI', 515, 3, 10, 0, '', 0,
2, 0, 0, 0, 32)
0.0005898475646972656
Total Observations: 5
```

Reconnect to the ‘flights’ database to import the second table, ‘airports’.

```
sql2 = '''CREATE TABLE IF NOT EXISTS airports(AirportID INT, Name TEXT, City TEXT, \
        Country TEXT, FAACODE TEXT, ICAO TEXT, LATITUDE INT, LONGITUDE INT, \
        ALTITUDE INT, TimeZ INT, DST TEXT, TZ TEXT);'''
c.execute(sql2)
```

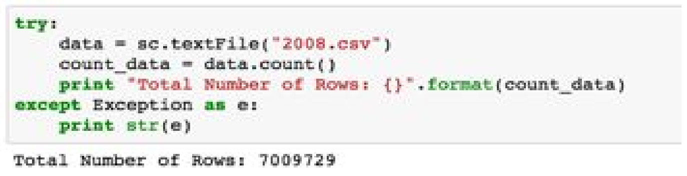
```
db = mysql.connector.connect(user='root',passwd="********",db='Flights')
c = db.cursor()
c.execute('''
     LOAD DATA LOCAL INFILE 'airport_location.csv'
     into Table airports fields terminated by ',' lines terminated by '\n' ignore 1 lines
     (AirportID,Name,City,Country,FAACODE,ICAO,LATITUDE,LONGITUDE,ALTITUDE,TimeZ,DST,TZ)
     ''')
db.commit()
db.close()
print("Done")
```

Getting Setup with Apache Spark

As referenced earlier, Pyspark is a Python API interface to Apache Spark. Navigating to http://spark.apache.org/downloads, the latest version of Spark can be downloaded. Next, unpack .tgz file to root directory and modify PATH variables in the .bash_profile along with adding the following alias to run PySpark from the Jupyter Notebook by simply entering snotebook in the command line or terminal for Mac users:

- export SPARK_PATH=~/spark-2.1.0-bin-hadoop2.7
- export PYSPARK_DRIVER_PYTHON="jupyter" export
- PYSPARK_DRIVER_PYTHON_OPTS="notebook" alias
- snotebook='$SPARK_PATH/bin/pyspark --master local[2]’

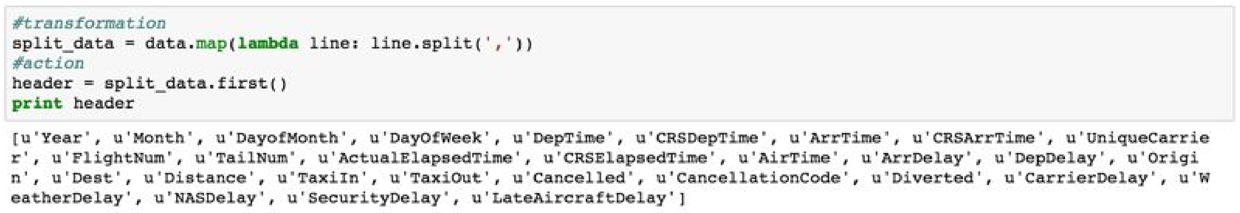
Given the circumstances of running PySpark in conjunction with MySQL as discussed in the previous section, the ‘flights’ dataset was instead loaded into memory by creating an RDD and reading in each line from the dataset. Using the textFile method, the file can be assigned to the ‘data’ variable and from here, the number of rows can be returned.



```
try:
    data = sc.textFile("2008.csv")
    count_data = data.count()
    print "Total Number of Rows: {}".format(count_data)
except Exception as e:
    print str(e)
```

```
Total Number of Rows: 7009729
```

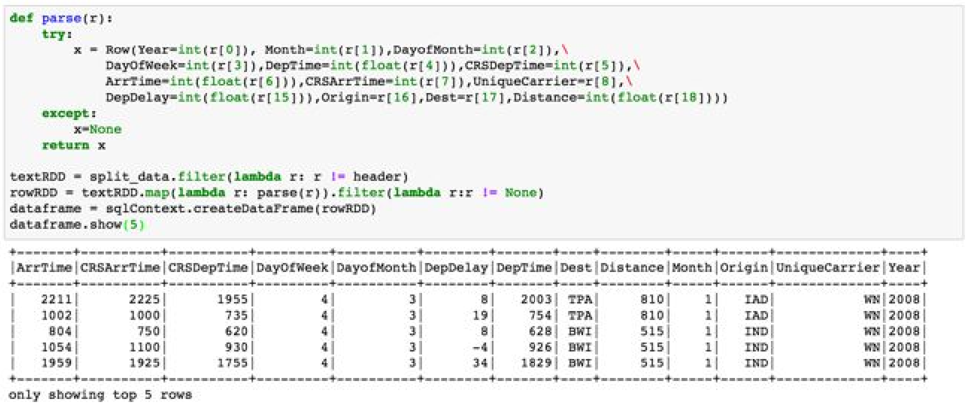
RDD is a collection of elements that are segmented across multiple nodes in a distributed cluster and can be computed in parallel. The two main functions that can be performed on a RDD is Transformation and Action. Any operation applied on a RDD which creates new RDDs (i.e. map) is referred to as a transformation, while aggregating the output of the transformation by applying any function (i.e reduce) is called an action. This is essentially what was performed when the data set was read into memory. Loading the file row by row (i.e. sc.textFile,) and then counting the number of rows, is an example of a transformation and action, respectively. Another instance of these two operations is when a function is mapped to each row where the resulting row is split by a comma in the csv file. To return the column headers, ‘first()’ method is applied to the transformation.



```
#transformation
split_data = data.map(lambda line: line.split(','))
#action
header = split_data.first()
print header
```

```
[u'Year', u'Month', u'DayofMonth', u'DayOfWeek', u'DepTime', u'CRSDepTime', u'ArrTime', u'CRSArrTime', u'UniqueCarrie
r', u'FlightNum', u'TailNum', u'ActualElapsedTime', u'CRSElapsedTime', u'AirTime', u'ArrDelay', u'DepDelay', u'Origi
n', u'Dest', u'Distance', u'TaxiIn', u'TaxiOut', u'Cancelled', u'CancellationCode', u'Diverted', u'CarrierDelay', u'W
eatherDelay', u'NASDelay', u'SecurityDelay', u'LateAircraftDelay']
```

Next, we need to assign the data type for each column (i.e. integer or character) and filter out the rows that do not contain the header. Then the parse function can be mapped to each row. In Spark, a Data-Frame is essentially equivalent to a table in a relational database or a data frame similar to Pandas (i.e. Python). Data-Frames can be constructed from other sources such as structured data files, tables in Hive, external databases, or existing RDDs. Data-Frames can be constructed by calling the createDataFrame method as shown below.

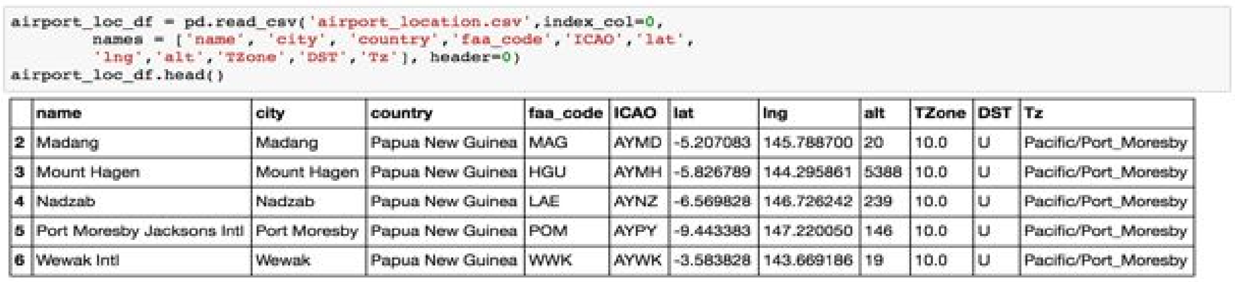


```
def parse(r):
    try:
        x = Row(Year=int(r[0]), Month=int(r[1]),DayofMonth=int(r[2]),\
            DayOfWeek=int(r[3]),DepTime=int(float(r[4])),CRSDepTime=int(r[5]),\
            ArrTime=int(float(r[6])),CRSArrTime=int(r[7]),UniqueCarrier=r[8],\
            DepDelay=int(float(r[15])),Origin=r[16],Dest=r[17],Distance=int(float(r[18])))
    except:
        x=None
    return x

textRDD = split_data.filter(lambda r: r != header)
rowRDD = textRDD.map(lambda r: parse(r)).filter(lambda r:r != None)
dataframe = sqlContext.createDataFrame(rowRDD)
dataframe.show(5)
```

```
+-------+----------+----------+---------+----------+--------+-------+----+--------+-----+------+-------------+----+
|ArrTime|CRSArrTime|CRSDepTime|DayOfWeek|DayofMonth|DepDelay|DepTime|Dest|Distance|Month|Origin|UniqueCarrier|Year|
+-------+----------+----------+---------+----------+--------+-------+----+--------+-----+------+-------------+----+
|   2211|      2225|      1955|        4|         3|       8|   2003| TPA|     810|    1|   IAD|           WN|2008|
|   1002|      1000|       735|        4|         3|      19|    754| TPA|     810|    1|   IAD|           WN|2008|
|    804|       750|       620|        4|         3|       8|    628| BWI|     515|    1|   IND|           WN|2008|
|   1054|      1100|       930|        4|         3|      -4|    926| BWI|     515|    1|   IND|           WN|2008|
|   1959|      1925|      1755|        4|         3|      34|   1829| BWI|     515|    1|   IND|           WN|2008|
+-------+----------+----------+---------+----------+--------+-------+----+--------+-----+------+-------------+----+
only showing top 5 rows
```

In the preceding code blocks, the objective is to aggregate the data in such a way to create a visualization that identifies the airports with the highest delayed flight time on average and scaled by the magnitude of flights arriving and departing from the airports in the United States. To setup the geographical plot, the MatplotLib toolkit 'Basemap' is used to construct the map with the corresponding shape-files for each state. Simply type 'sudo install basemap' in the command line to install the Python package. Before getting to the visualization, the airport location csv file is loaded into memory and is stored in the Data-Frame 'airport_loc_df.'



```
airport_loc_df = pd.read_csv('airport_location.csv',index_col=0,
        names = ['name', 'city', 'country','faa_code','ICAO','lat',
        'lng','alt','TZone','DST','Tz'], header=0)
airport_loc_df.head()
```

	name	city	country	faa_code	ICAO	lat	lng	alt	TZone	DST	Tz
2	Madang	Madang	Papua New Guinea	MAG	AYMD	-5.207083	145.788700	20	10.0	U	Pacific/Port_Moresby
3	Mount Hagen	Mount Hagen	Papua New Guinea	HGU	AYMH	-5.826789	144.295861	5388	10.0	U	Pacific/Port_Moresby
4	Nadzab	Nadzab	Papua New Guinea	LAE	AYNZ	-6.569828	146.726242	239	10.0	U	Pacific/Port_Moresby
5	Port Moresby Jacksons Intl	Port Moresby	Papua New Guinea	POM	AYPY	-9.443383	147.220050	146	10.0	U	Pacific/Port_Moresby
6	Wewak Intl	Wewak	Papua New Guinea	WWK	AYWK	-3.583828	143.669186	19	10.0	U	Pacific/Port_Moresby

As part of the Spark Ecosystem, one of the core functionalities of Spark is the ability to write SQL queries. For this example, the data has been loaded directly into a Data-Frame, but if a connection is made to the MySQL database or even to a NoSQL database such as MongoDB, SQL can still be utilized. For purposes of illustration, to establish a connection to the Flights database in MySQL, executing the following code should yield the connection. Thus, a temporary view can be created and SQL syntax can be invoked for extracting and writing data to and from a database.

```python
from pyspark.sql import DataFrameReader
df = sqlContext.read.format("jdbc").options(
        url ="jdbc:mysql://localhost:3306/Flights?createDatabaseIfNotExist=true",
        driver="com.mysql.jdbc.Driver",
        dbtable="flight_info",
        user="root",
        partition=20,
        password="******").load()
```

```python
df.createOrReplaceTempView("flights")
sqlDF = spark.sql("SELECT * FROM flights")
sqlDF.show()
```

Another example is depicted below in which the number of delayed flights are computed using a ‘groupby’ clause:

```python
delays = sqlContext.sql("SELECT Origin, count(*) Num_Flights,avg(DepDelay)\
                         Delay FROM dataframe GROUP BY Origin")
```

	Origin	Num_Flights	Delay
0	BGM	699	5.915594
1	PSE	742	0.057951
2	DLG	111	16.495495
3	INL	71	-4.802817
4	MSY	38510	8.891587

The subsequent steps are used to structure the data into the necessary format in order to abstract the flight information and construct a visualization. The code below joins the two Data-Frames (flight_infor and airport locations) on the keys “Origin” and “FAA_Code”

```python
#Join origin_df with airports
df_airports = pd.merge(origin_df,airport_loc_df,left_on = 'Origin',right_on = 'faa_code')
df_airports.head()
```

One important consideration to make is standardizing the delay time given the volume of flights vary significantly for smaller metropolitan areas. This is taken into account in the code below, along with storing the coordinates of the airports in x-y variables.

```python
#standardize delay times
def zscore(x):
    return (x-np.average(x))/np.std(x)

#Plot Airport Delay
countrange=max(df_airports['Num_Flights'])-min(df_airports['Num_Flights'])
standarize = (zscore(df_airports['Delay']))
x,y = map(np.asarray(df_airports['lng']),np.asarray(df_airports['lat']))
volume=df_airports['Num_Flights']*4000.0/countrange
```

The two code blocks below are designated for building a map of the U.S. using the Basemap toolkit in addition to creating a scatter plot of latitude and longitude locations of airport. The size of each data point is based on the magnitude of delayed flight times.

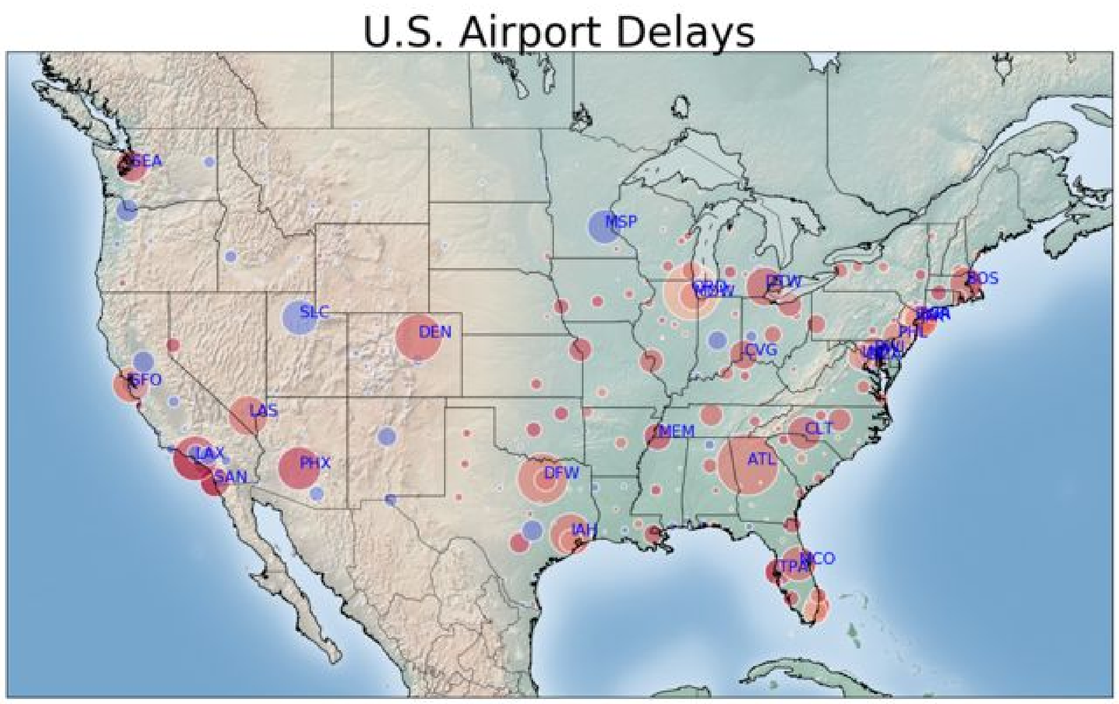
```python
fig = plt.figure(figsize=(20,20))
map = Basemap(projection = 'merc',area_thresh = 4000,resolution ='i',
            rsphere=6371200.,llcrnrlon=-130,llcrnrlat=21,urcrnrlon=-62, urcrnrlat=52)
```

```
#Add color to map:
color = pl.get_cmap('coolwarm')(np.linspace(0.0,1.0,70))
color = np.flipud(color)
map.scatter(x, y,  marker='o', s= volume, linewidths=1.5,
    edgecolors='white', alpha = .7, color=color[(standarize*10)])
```

The final step consists of providing labels to airports and for visualization purposes, Hawaii will be omitted. Also, airports where the number of flights exceeds 70,000 will be considered.

```
#Add Labels to Map and ignoring Hawaii
df_text=df_airports[(df_airports['Num_Flights']>70000) &
                    (df_airports['faa_code'] != 'HNL')]
xtext,ytext = map(np.asarray(df_text['lng']),
                    np.asarray(df_text['lat']))
txt=np.asarray(df_text['faa_code'])
zp=zip(xtext,ytext,txt)
for row in zp:
    plt.text(row[0],row[1],row[2], fontsize=16, color='blue',)
plt.title("U.S. Airport Delays", fontsize = 42)
plt.show()
```

In summary, Apache Spark was implemented to load into memory a large dataset (i.e. 7 million rows) of delayed flights recorded in 2008. By using the data functionalities of PySpark, parsing, querying, and filtering, we are able to arrive at the outcome shown below. One can observe the magnitude of delayed flights throughout the main lands of the U.S. with the highest magnitude of delays occurring in major cities such as Atlanta, Dallas, Denver, Houston, Chicago, and Los Angeles.



Conclusion

In this study, the core functionalities of Apache Spark are discussed. A brief description of Spark and the core functionalities are presented in this work followed by an instruction on how to connect MySQL and PySpark, an API for interfacing with Python 3.5. Apache Spark was connected to an external relational database, MySQL, and evaluated for computational performance of airline flight data from the United States in 2008. Additionally, the 680 MB data file was cached in memory and queried with traditional SQL in order to implement analytics and visualizations using Python.

Overall the results obtained were satisfactory with the methods available to us. Given additional resources one option would have been to configure Spark-Hadoop together since there is more supporting documentation available for setup and the setup of the configuration used was where the majority of time was spent on this project. It is debatable whether Python or R would have performed better or have been more user friendly but we were pleased with the performance of PySpark. That being said we believe that the version that we chose was the best option due to its low cost, high performance value, and usability of resulting data for visualization. As a result we were able to configure a useable tool we can use on a continuing basis in our academic and professional lives.

References

[1] Zaharia M., et al., “Learning Spark”, O'Reilly, 2015, 274s

[2] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica, “Discretized Streams: Fault-Tolerant Streaming Computation at Scale”, SOSP’13, Nov. 3–6, 2013, (http://people.csail.mit.edu/matei/papers/2013/sosp_spark_streaming.pdf)

[3] Rahul Kavale, Scrap your MapReduce! (Or, Introduction to Apache Spark), Nov 16, 2014 http://rahulkavale.github.io/blog/2014/11/16/scrap-your-map-reduce/

[4] Livio Hobi, SQL versus MapReduce, A comparison between two approaches to large scale data analysys, May 30, 2013, http://www.chinacloud.cn/upload/2013-06/13061707031382.pdf

[5] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica, “Spark: Cluster Computing with Working Sets”, https://amplab.cs.berkeley.edu/wp-content/uploads/2011/06/Spark-Cluster-Computing-with-Working-Sets.pdf

[6] Copyright © 2017 The Apache Software Foundation, http://spark.apache.org/

[7] Github site of The Apache Software Foundation, https://github.com/apache/spark

[8] Data Scientist Workbench is a Big Data University's Virtual Lab Environment © Copyright IBM Corp. 2016 https://datascientistworkbench.com/

[9] Get the data, Dataset Source, Data Expo 2009 http://stat-computing.org/dataexpo/2009/the-data.html

[10] Alexander Rubin, How Apache Spark makes your slow MySQL queries 10x faster (or more), August 17, 2016, https://www.percona.com/blog/2016/08/17/apache-spark-makes-slow-mysql-queries-10x-faster/

[11] American Statistical Association -- 2008 Airline Flight Dataset http://stat-computing.org/dataexpo/2009/the-data.html