CSCI 5408, Winter 2017

Assignment-3:

Apache Spark, Real Time Data Pipelines and Analysis

Instructor: Dr. Qigang Gao

Date of Submission: February 21, 2017

Submitted By:

Kundan Kumar (B00763592)

Shalav Verma (B00756911)

DALHOUSIE UNIVERSITY

Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia

Task Distribution

Serial No.	Tasks	Execution
1	Apache Spark installation and Configuration	Kundan Kumar/Shalav Verma
2	Application 1: Application queries for Word count Operations	Kundan Kumar
3	Application 2.1: Total number of birth registered in a year	Shalav Verma
4	Application 2.2: Total number of births registered for a year by gender	Kundan Kumar
5	Application 2.3: Top 5 most popular names registered for a year	Shalav Verma
6	Application 2.4: Total number of birth registrations for a name	Kundan Kumar
7	Application 3.1: Total injuries and fatalities	Shalav Verma
8	Application 3.2: Total incident in a year	Shalav Verma
9	Application 3.3 Total injuries grouped by year and quarter	Kundan Kumar
10	Application 3.4: Total incidents grouped by borough, year and month	Kundan Kumar

Table of Contents

Section 1: Task Description

In this assignment, we will learn the concepts of Big Data Systems running on Clouds. We will use Apache Spark to build Real Time Data Pipelines which can be used over data-warehouse and distributed databases. Apache Spark is a cluster computing framework which allows lighting fast computing through in-memory computing, along with features of implicit data parallelism and fault tolerance [1]. It has Application Programming Interfaces in R, Python, Java and Scala through which it offers distributed task scheduling, dispatching and basic I/O functions [1]. The following diagram illustrates an overview of Apache Spark.

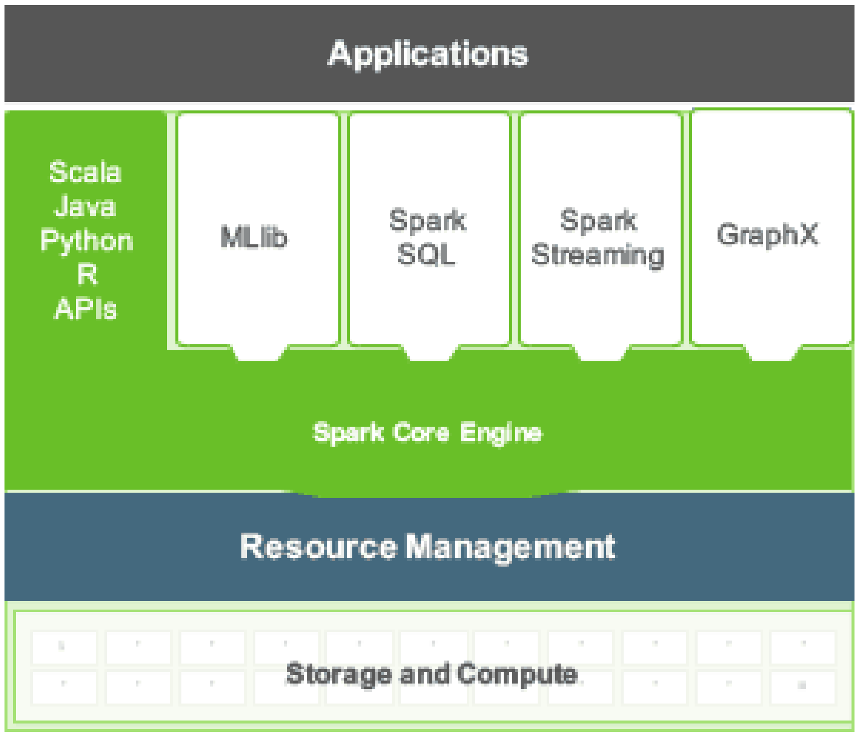

Fig 1.1: Apache Spark Architecture overview. [2]

Apache Spark can run Spark locally with one or more worker thread. It can also connect to a Spark Standalone or a Meso cluster using IP and Port for cluster computing. Apache Spark uses RDD (Resilient Distributed Dataset) to have immutable, partitioned and distributed in-memory storage of data from different sources to perform data transformations and actions on them.

In this assignment, we will use local standalone configuration to perform following data-analysis:

1. **Word Count**: Count the number of distinct words in a text document.
2. **SQL like operations on Big Datasets**: Loading data into Spark and performing SQL like operations on it using Spark SQL and Dataframes.
3. **Advanced SQL operations (aggregations, Roll-up, drill down etc.)**: Loading data into Spark, performing required pre-processing and further deriving insights using aggregations, Roll-up and drill down like operations on the dataset.

1.1 Task Deliverable Details

The following table lists all other resources submitted along with this report:

Serial No	Resource	Path
1.	Word Count Application and output for analysis on Divine Comedy paragraph [3].	./src/word_count
2.	Application and sample output for analysis on National Names dataset [4].	./src/baby_names_analysis
3.	Application and sample output for analysis on NYPD Motor Vehicle collision dataset [5].	./src/nypd_mv_collision_analysis
4.	Readme File	./README.txt

Table 1.1.1: Task Deliverable Details

Section 2: Spark Design

For the current assignment, a standalone deployment of Spark on local machine was used. We used a Spark deployment on a Windows 7 operating system and local file system for storing data from where the data was loaded to generate RDD. The diagram below shows the deployment overview:

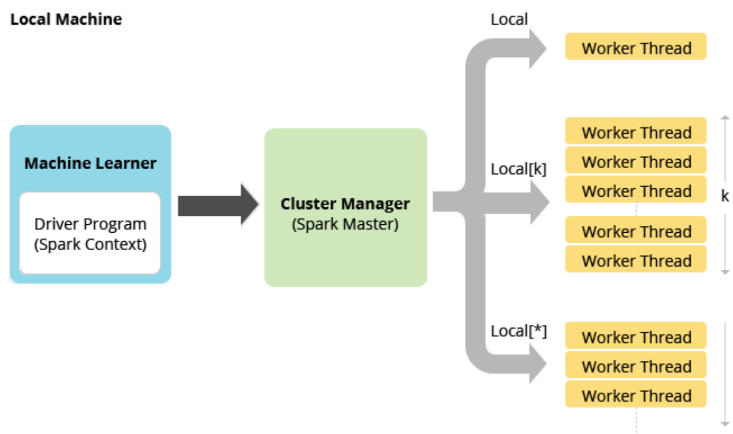

Fig 2.1: Standalone deployment mode of Apache Spark [6]

For the above deployment scenario, Apache Spark runs in local mode and can have one or more than one worker threads on the same machine. The number of worker threads should be ideally set to the number of cores on the machine and is configured as follows while creating the Spark context.

```
#Spark context in Python
from pyspark import SparkContext
# 1 Worker thread. No parallelism
sc = SparkContext("local","App")
```

```
#Spark context in Python
from pyspark import SparkContext
# 2 Worker thread.
sc = SparkContext("local[2]","App")
```

Fig 2.2: Configuring Spark Context on standalone deployment

We used Spark version 2.1.0. The following table shows the addition configurations done to setup Spark on Windows operating system.

Serial No.	Path Variable	Value
1	JAVA_HOME	C:\Program Files\Java\jdk1.7.0_79
2	SCALA_HOME	C:\Program Files\scala
3	SBT_HOME	C:\Program Files\sbt
4	SPARK_HOME	C:\spark-2.1.0-bin-hadoop2.7\
5	HADOOP_HOME	E:\winutils
6	PATH	%SCALA_HOME%\bin; %SBT_HOME%\bin; %SPARK_HOME %\bin;%JAVA_HOME%\bin;%HADOOP_HOME%\bin

Table 2.1: Configurations to set-up Spark on Windows [7]

We used Jupyter Notebook [8] provided by Anaconda [9] installation package as IDE to write program.

Section 3: Application Queries and Outputs

In the following sub-sections, we will see the applications queries and outputs for different datasets.

3.1 Word Count

Requirement: Write an application which can count distinct words and number of occurrences of each word in the dataset.

The following approach was adopted:

1. We used the content of '*WordCountData.txt*' as the input dataset.
2. Data Preprocessing:
    a. Removed special characters like comma, empty spaces etc.
    b. Removed stop words using Natural Language Tool Kit.
3. Further we used Apache Spark's *flatMap* and *reduceByKey* methods to build a dictionary of unique words.
4. We also used Pandas package to generate a bar plot for 20 most frequent important words.

Below is the snapshot of the application for word count:

```
import re
import nltk
import time
import pandas as pd
from pyspark import SparkContext
from nltk.corpus import stopwords

# Pre-process each line, to remove noise and stopwords.
def preprocess_to_get_words(text_file_rdd):
    words=re.sub('[^a-z| |0-9]', '', text_file_rdd.strip().lower()).split(" ")
    eng_stopwords=stopwords.words('english')
    return [word for word in words if word not in eng_stopwords and len(word)>1]

if __name__=="__main__":

    #sc = SparkContext("local","Application")

    # Get File RDD
    text_file_rdd = sc.textFile("C:\\Users\\6910P\\Google Drive\\Dalhousie\\term_1\\data_management_analytics\\assignment_3\\Word

    start_time = time.time()

    # Pre-process, build FlatMap and reduce by key to get count of each unique word
    allWords = text_file_rdd.flatMap(preprocess_to_get_words).map(lambda aWord: (aWord,1))
    unqWord_freq_tuples_list = allWords.reduceByKey(lambda v1,v2: v1+v2)

    print "Processing Time:"+str((time.time() - start_time)*1000)+" msec\nTotal distinct words:"+str(unqWord_freq_tuples_list.cou

    # Get the words locally, print in tabular format using Pandas
    df = pd.DataFrame(sorted(unqWord_freq_tuples_list.collect()), columns=['Word', 'Count'])
    df.sort_values([ 'Count'], ascending=[False], inplace=True)
    pd.set_option('display.max_rows', unqWord_freq_tuples_list.count())
    print df[:].to_string(index=False)

    # Print bar chart for top 20 words using Pandas
    df_top_n = df.head(20)
    %matplotlib inline
    df_top_n.plot(kind='bar', x=df_top_n['Word'])
```

Fig 3.1.1: Application to count unique words

Below is the snapshot of the partial-output of the application along with the processing time:

```
Processing Time:27.0001888275 msec
Total distinct words:2580

  Word  Count
  thou     88
   thy     54
  thee     51
  thus     43
   one     42
 light     33
   yet     31
  hath     29
   may     28
whence     23
nature     23
  love     22
 whose     22
  well     22
  made     21
```

Fig 3.1.2: Output of application for word count

Below is the snapshot of the bar-chart for the top 20 words generated using Pandas.

Query Processing Time is **27 Milliseconds.**

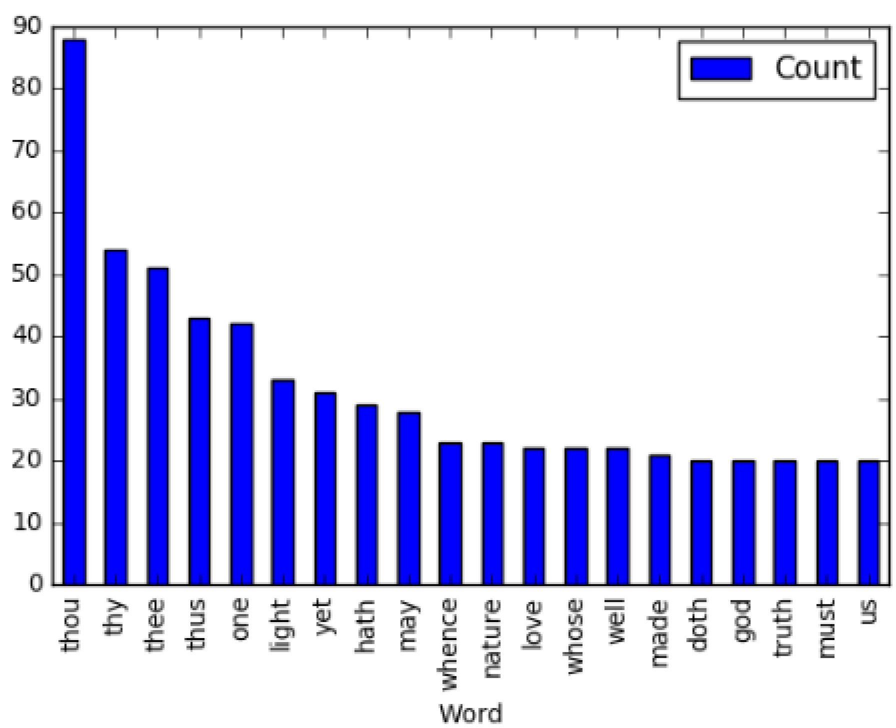

Fig 3.1.3: Bar Chart for top 20 words generated using Pandas.

3.2 Analysis of Baby Names Dataset

Requirement: Import the data from baby names dataset and write an application which can be used to perform queries.

The following approach was adopted:

1. NationalNames.csv has been used as the input dataset.
2. Data has been imported into dataframe using *sqlContext.read.load* function.
3. *sqlContext.sql* was used to query data from the dataframe.

3.2.1 Total number of birth registered in a year.

Below is the snapshot of the application:

```
import time
from pyspark.sql import Row
from pyspark import SparkContext
from pyspark.sql import SQLContext

if __name__=="__main__":

    # Create Spark Context
    #sc = SparkContext("local[2]","Application")
    sqlContext = SQLContext(sc)
    filepath = "C:\\Users\\6910P\\Google Drive\\Dalhousie\\term_1\\data_management_analytics\\assignment_3\\NationalNames.csv"

    # Load data.
    df = sqlContext.read.load(filepath,format='com.databricks.spark.csv', header='true', inferSchema='true')
    df.registerTempTable("babynames")

    start_time = time.time()
    # Total number of birth registered in a year
    birth_count_by_year = sqlContext.sql("SELECT year, SUM(COUNT) FROM babynames GROUP BY year ORDER BY year ASC")
    print "Processing Time:"+str((time.time() - start_time)*1000)+" msec.\nTotal number of birth registered in a year:"
    birth_count_by_year.show(birth_count_by_year.count(),False)
```

Fig 3.2.1.1: Application to count total birth registered in a year.

Below is the snapshot of the partial output. Complete output is shared in *src/baby_names_analysis/2.1.txt* file.

```
Processing Time:13.9999389648 msec.
Total number of birth registered in a year:
+----+----------+
|year|sum(COUNT)|
+----+----------+
|1880|201484    |
|1881|192699    |
|1882|221538    |
|1883|216950    |
|1884|243467    |
|1885|240855    |
|1886|255319    |
|1887|247396    |
|1888|299480    |
|1889|288950    |
|1890|301402    |
|1891|286678    |
|1892|334383    |
|1893|325223    |
|1894|338694    |
```

Fig 3.2.1.2: Partial output having count of total birth registered in a year.

Query processing time is **14 Milliseconds.**

3.2.2 Total number of births registered for a year by gender

Below is the snapshot of the application:

```
from pyspark.sql import Row
from pyspark import SparkContext
from pyspark.sql import SQLContext

if __name__=="__main__":

    # Create Spark Context
    sc = SparkContext("local[2]","Application")
    sqlContext = SQLContext(sc)
    filepath = "C:\\Users\\6910P\\Google Drive\\Dalhousie\\term_1\\data_management_analytics\\assignment_3\\NationalNames.csv"

    # Load data.
    df = sqlContext.read.load(filepath,format='com.databricks.spark.csv', header='true', inferSchema='true')
    df.registerTempTable("babynames")

    start_time = time.time()
    # Total number of births registered in a year by gender
    birth_count_by_year_gender = sqlContext.sql("SELECT year, gender , SUM(count) FROM babynames GROUP BY year, gender ORDER BY y
    print  "Processing Time:"+str((time.time() - start_time)*1000)+" msec.\nTotal number of births registered in a year by gender
    birth_count_by_year_gender.show(birth_count_by_year_gender.count(),False)
```

Fig 3.2.2.1 Sample application to count births registered for a year by gender

Below is the snapshot of the partial output. Complete output is shared in *src/baby_names_analysis/2.2.txt* file:

```
Processing Time:15.0001049042 msec.
Total number of births registered in a year by gender:
+----+------+----------+
|year|gender|sum(count)|
+----+------+----------+
|1880|F     |90993     |
|1880|M     |110491    |
|1881|F     |91954     |
|1881|M     |100745    |
|1882|F     |107850    |
|1882|M     |113688    |
|1883|F     |112321    |
|1883|M     |104629    |
|1884|F     |129022    |
|1884|M     |114445    |
|1885|F     |133055    |
|1885|M     |107800    |
|1886|F     |144535    |
|1886|M     |110784    |
|1887|F     |145982    |
|1887|M     |101414    |
|1888|F     |178627    |
|1888|M     |120853    |
|1889|F     |178366    |
|1889|M     |110584    |
|1890|F     |190377    |
```

Fig 3.2.2.2: Output for the above query

Query processing time is **15 Milliseconds**

3.2.3 Top 5 most popular names registered for a year

Below is the snapshot of the application and the result:

```
from pyspark.sql import Row
from pyspark import SparkContext
from pyspark.sql import SQLContext

if __name__=="__main__":

    #sc = SparkContext("local[2]","Application")
    sqlContext = SQLContext(sc)
    filepath = "C:\\Users\\6910P\\Google Drive\\Dalhousie\\term_1\\data_management_analytics\\assignment_3\\NationalNames.csv"

    df = sqlContext.read.load(filepath,format='com.databricks.spark.csv', header='true', inferSchema='true')
    df.registerTempTable("babynames")
    start_time = time.time()
    # Input a year and populate top 5 most popular names registered that year
    top_5_names_in_year = sqlContext.sql("SELECT DISTINCT name, count FROM babynames WHERE year=1880 ORDER BY count DESC")
    print  "Processing Time:"+str((time.time() - start_time)*1000)+" msec.\nInput a year and populate top 5 most popular names re
    top_5_names_in_year.show(5)
```

```
Processing Time:49.0000247955 msec.
Input a year and populate top 5 most popular names registered that year
+-------+-----+
|   name|count|
+-------+-----+
|   John| 9655|
|William| 9532|
|   Mary| 7065|
|  James| 5927|
|Charles| 5348|
+-------+-----+
only showing top 5 rows
```

Fig 3.2.3.1: Sample application and output for 5 most popular names of year 1880

Query processing time is **49 Milliseconds.**

3.2.4 Total number of birth registrations for a name

Below is the snapshot of the application:

```
from pyspark.sql import Row
from pyspark import SparkContext
from pyspark.sql import SQLContext

if __name__=="__main__":

    #sc = SparkContext("local[2]","Application")
    sqlContext = SQLContext(sc)
    filepath = "C:\\Users\\6910P\\Google Drive\\Dalhousie\\term_1\\data_management_analytics\\assignment_3\\NationalNames.csv"

    df = sqlContext.read.load(filepath,format='com.databricks.spark.csv', header='true', inferSchema='true')
    df.registerTempTable("babynames")

    start_time = time.time()
    # Input a child name and populate total number of birth registrations throughout the dataset for that name
    birth_count_by_Name = sqlContext.sql("SELECT SUM(Count) FROM babynames WHERE Name='Mary'")
    print  "Processing Time:"+str((time.time() - start_time)*1000)+" msec.\nInput a child name and populate total number of birth
    birth_count_by_Name.show(birth_count_by_Name.count(),False)
```

Fig 3.2.4.1: Sample Application to get total birth registered for a name

Below is the snapshot of the output:

```
Processing Time:9.0000629425 msec.
Input a child name and populate total number of birth registrations throughout the dataset for that name:
+----------+
|sum(Count)|
+----------+
|4130441   |
+----------+
```

Fig: 3.2.4.2: Output for total birth registered for the name 'Mary'

Query processing time is **9 Milliseconds**.

3.3 Analysis of NYPD Motor Vehicle Collision Dataset

Requirement: Import the data from NYPD dataset and write an application which can be used to perform the queries.

The following approach was adopted:

1. Data preprocessing
    a. Updated the column header names where empty spaces in names were replaced by "_".
    b. Corrected a corrupted row where the data for one column was split across two lines.
2. Data frame was updated with additional columns to hold year, month and quarter (as required by 2nd query) and month (as required by 3rd and 4th query) from the date column of the provided dataset.
    a. A function to get year from date was implemented.
    b. A function to get month from date was implemented.
    c. A function to get quarter from date was implemented.
3. *sqlContext.read.load* was used to load data into the dataframe.
4. Application also provides the feature to load the dataset from local machine or from web hosted resource.
5. *sqlContext.sql* was used to query data from the dataframe.

3.3.1 Total injuries and fatalities

Requirement: Capture total injuries and fatalities associated with each motor collision record, identified by a unique incident key.

Below is the snapshot of the application:

```
import time
import urllib
from pyspark.sql import Row
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType


def get_data_file(isLocal):
    if(isLocal):
        filepath = "C:\\Users\\6910P\\Google Drive\\Dalhousie\\term_1\\data_management_analytics\\assignment_3\\NYPD_Motor_Vehicl
        return filepath
    else:
        resource_url ="https://data.cityofnewyork.us/api/views/h9gi-nx95/rows.csv"
        urllib.urlretrieve("https://data.cityofnewyork.us/api/views/h9gi-nx95/rows.csv","NYPD_Motor_Vehicle_Collisions.csv")
        return "NYPD_Motor_Vehicle_Collisions.csv"


if __name__=="__main__":

    #sc = SparkContext("local[2]","Application")
    data_file  = get_data_file(True)
    sqlContext = SQLContext(sc)

    df = sqlContext.read.load(data_file,format='com.databricks.spark.csv', header='true', inferSchema='true')
    df.registerTempTable("nypdmvcollisions")

    print "Count of total records:"+str(df.count())

    start_time = time.time()
    # Capture total injuries and fatalities associated with each motor collision record(identified by a unique incident key)
    query =  """SELECT UNIQUE_KEY, (  NUMBER_OF_PERSONS_KILLED  + NUMBER_OF_PEDESTRIANS_KILLED  + NUMBER_OF_CYCLIST_KILLED +
        NUMBER_OF_MOTORIST_KILLED) AS All_Fatalities_Count,
        (NUMBER_OF_PERSONS_INJURED+NUMBER_OF_PEDESTRIANS_INJURED+NUMBER_OF_CYCLIST_INJURED+NUMBER_OF_MOTORIST_INJURED) As All_Inj
        FROM nypdmvcollisions"""
    record_by_key = sqlContext.sql(query)
    print "Processing Time:"+str((time.time() - start_time)*1000)+" msec.\nTotal injuries and fatalities associated with each mot
    record_by_key.show(1000, False)
```

Fig 3.3.1.1: Application to count total injuries and fatalities

Below is the snapshot of the partial output. Detailed output is shared in *./src/nypd_mv_collision_analysis/3.1.txt* .

```
Processing Time:28.9998054504 msec.
Total injuries and fatalities associated with each motor collision record:
Wall time: 29 ms
+----------+--------------------+-----------------+
|UNIQUE_KEY|All_Fatalities_Count|All_Injured_Count|
+----------+--------------------+-----------------+
|3559084   |0                   |0                |
|3527641   |0                   |0                |
|3527134   |0                   |0                |
|3527090   |0                   |0                |
|3527168   |0                   |2                |
|3526991   |0                   |3                |
|3440410   |0                   |0                |
|3437710   |0                   |0                |
|3461182   |0                   |0                |
|3440704   |0                   |0                |
|3438113   |0                   |2                |
|3439653   |0                   |0                |
|3284922   |0                   |0                |
|2833714   |0                   |0                |
|336679    |0                   |0                |
|3557464   |0                   |0                |
|3557999   |0                   |0                |
|3559576   |0                   |0                |
|3558765   |0                   |0                |
|3558014   |0                   |0                |
|3527684   |0                   |2                |
|3526631   |0                   |4                |
|3439570   |0                   |2                |
|3437745   |0                   |0                |
|3438552   |0                   |0                |
|3437577   |0                   |0                |
```

Fig: 3.3.1.2: Output for count of total injuries and fatalities

Query processing time is **28.99 Milliseconds.**

3.3.2 Total incident in a year

Requirement: Capture total incident counts in a year (grouped by year)

Below is the snapshot of the application:

```
import time
import urllib
from pyspark.sql import Row
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

def get_data_file(isLocal):
    if(isLocal):
        filepath = "C:\\Users\\6910P\\Google Drive\\Dalhousie\\term_1\\data_management_analytics\\assignment_3\\NYPD_Motor_Vehicl
        return filepath
    else:
        resource_url ="https://data.cityofnewyork.us/api/views/h9gi-nx95/rows.csv"
        urllib.urlretrieve("https://data.cityofnewyork.us/api/views/h9gi-nx95/rows.csv","NYPD_Motor_Vehicle_Collisions.csv")
        return "NYPD_Motor_Vehicle_Collisions.csv"

def get_year(date):
    return date[-4:]
    

if __name__=="__main__":

    #sc = SparkContext("local[2]","Application")
    data_file  = get_data_file(True)
    sqlContext = SQLContext(sc)
    
    df = sqlContext.read.load(data_file,format='com.databricks.spark.csv', header='true', inferSchema='true')
    df.registerTempTable("nypdmvcollisions")
    
    print "Count of total records:"+str(df.count())
    
    start_time = time.time()
    
    #Capture total incident counts in a year (grouped by year)
    udfDateToYear=udf(get_year, StringType())
    df_with_year = df.withColumn("year", udfDateToYear("DATE"))
    df_with_year.registerTempTable("nypdmvcollisions_year")
    start_time = time.time()
    query_2 = "SELECT year, COUNT(*) as cnt from nypdmvcollisions_year GROUP BY year ORDER BY year DESC"
    query_2_sql = sqlContext.sql(query_2)
    print "Processing Time:"+str((time.time() - start_time)*1000)+" msec."
    query_2_sql.show(query_2_sql.count(),False)
```

Fig: 3.3.2.1: Sample application to count incidents in a year

Below is the snapshot of the output:

```
Count of total records:970702
Processing Time:13.9999389648 msec.
+----+------+
|year|cnt   |
+----+------+
|2017|15760 |
|2016|227263|
|2015|217539|
|2014|205929|
|2013|203689|
|2012|100522|
+----+------+
```

Fig: 3.3.2.2: Output for count of incidents in a year

Query processing time is **13.99 Milliseconds**.

3.3.3 Total injuries grouped by year and quarter

Requirement: Capture total injuries (can be sum of injuries and fatalities) grouped by year and quarter

Below is the snapshot of the application:

```python
def get_year(date):
    return date[-4:]

def get_quarter(date):
    month = int(date[:2])
    if(month <4):
        return "1"
    elif (month > 3) and (month <7):
        return "2"
    elif (month > 6) and (month <10):
        return "3"
    else:
        return "4"

if __name__=="__main__":

    #sc = SparkContext("local[2]","Application")
    data_file  = get_data_file(True)
    sqlContext = SQLContext(sc)

    df = sqlContext.read.load(data_file,format='com.databricks.spark.csv', header='true', inferSchema='true')
    df.registerTempTable("nypdmvcollisions")

    print "Count of total records:"+str(df.count())

    start_time = time.time()

    #Capture total incident counts in a year (grouped by year)
    udfDateToYear=udf(get_year, StringType())
    df_with_year = df.withColumn("year", udfDateToYear("DATE"))
    df_with_year.registerTempTable("nypdmvcollisions_year")

    #Capture total injuries(can be sum of injuries and fatalities) grouped by year and quarter
    udfDateToQuarter=udf(get_quarter, StringType())
    df_with_year_quart = df_with_year.withColumn("quarter", udfDateToQuarter("DATE"))
    df_with_year_quart.registerTempTable("nypdmvcollisions_year_quart")
    query_3 = """SELECT year , quarter,
    (SUM(NUMBER_OF_PERSONS_KILLED) + SUM(NUMBER_OF_PEDESTRIANS_KILLED) +SUM(NUMBER_OF_CYCLIST_KILLED)+ SUM(NUMBER_OF_MOTORIST_KIL
    (SUM(NUMBER_OF_PERSONS_INJURED)+SUM(NUMBER_OF_PEDESTRIANS_INJURED)+SUM(NUMBER_OF_CYCLIST_INJURED)+ SUM(NUMBER_OF_MOTORIST_INJ
    from nypdmvcollisions_year_quart GROUP BY year, quarter ORDER BY year DESC, quarter ASC"""
    start_time = time.time()
    query_3_sql = sqlContext.sql(query_3)
    print "Processing Time:"+str((time.time() - start_time)*1000)+" msec.\nCapture total injuries(can be sum of injuries and fata
    query_3_sql.show()
```

Fig 3.3.3.1: Sample application to count injuries grouped by year and quarter

Below is the snapshot of the output:

```
Count of total records:970702
Processing Time:33.9999198914 msec.
Capture total injuries(can be sum of injuries and fatalities) grouped by year and quarter
+----+-------+-------------------+----------------+
|year|quarter|All_Fatalities_Count|All_Injured_Count|
+----+-------+-------------------+----------------+
|2017|      1|                 38|            8580|
|2016|      1|                 96|           23650|
|2016|      2|                123|           31213|
|2016|      3|                138|           41602|
|2016|      4|                142|           32240|
|2015|      1|                 90|           20334|
|2015|      2|                140|           26798|
|2015|      3|                112|           28088|
|2015|      4|                144|           27456|
|2014|      1|                108|           21758|
|2014|      2|                140|           27379|
|2014|      3|                158|           27088|
|2014|      4|                118|           26176|
|2013|      1|                144|           23580|
|2013|      2|                110|           29144|
|2013|      3|                170|           29838|
|2013|      4|                170|           27662|
|2012|      3|                150|           28584|
|2012|      4|                124|           26306|
+----+-------+-------------------+----------------+
```

Fig: 3.3.3.2: Output for count of injuries grouped by year and quarter

Query processing time is **33.99 Milliseconds**.

3.3.4 Total incidents grouped by borough, year and month

Requirement: Capture total injuries (sum of injuries and fatalities) and incident count grouped by Borough, year and month

Below is the snapshot of the application:

```
def get_month(date):
    return int(date[:2])

if __name__=="__main__":

    #sc = SparkContext("local[2]","Application")
    data_file  = get_data_file(True)
    sqlContext = SQLContext(sc)
    
    df = sqlContext.read.load(data_file,format='com.databricks.spark.csv', header='true', inferSchema='true')
    df.registerTempTable("nypdmvcollisions")
    
    print "Count of total records:"+str(df.count())
    
    #Capture total incident counts in a year (grouped by year)
    udfDateToYear=udf(get_year, StringType())
    df_with_year = df.withColumn("year", udfDateToYear("DATE"))
    df_with_year.registerTempTable("nypdmvcollisions_year")

    #Capture total injuries(can be sum of injuries and fatalities) grouped by year and quarter
    udfDateToQuarter=udf(get_quarter, StringType())
    df_with_year_quart = df_with_year.withColumn("quarter", udfDateToQuarter("DATE"))
    df_with_year_quart.registerTempTable("nypdmvcollisions_year_quart")

    #Capture total injuries and incident count(sum of injuries and fatalities) grouped by Borough, year and month
    udfDateToQuarterMonth=udf(get_month, StringType())
    df_with_year_quart_month = df_with_year.withColumn("month", udfDateToQuarterMonth("DATE"))
    df_with_year_quart_month.registerTempTable("nypdmvcollisions_year_quart_month")
    query_4 = """SELECT BOROUGH , year , month,
    (SUM(NUMBER_OF_PERSONS_KILLED) + SUM(NUMBER_OF_PEDESTRIANS_KILLED) +SUM(NUMBER_OF_CYCLIST_KILLED)+ SUM(NUMBER_OF_MOTORIST_KIL
    SUM(NUMBER_OF_PERSONS_INJURED)+SUM(NUMBER_OF_PEDESTRIANS_INJURED)+SUM(NUMBER_OF_CYCLIST_INJURED)+ SUM(NUMBER_OF_MOTORIST_INJU
    AS All_Injuries_Incidents_Count from nypdmvcollisions_year_quart_month GROUP BY BOROUGH, year, month ORDER BY
    year DESC, month ASC"""
    start_time = time.time()
    query_4_sql = sqlContext.sql(query_4)
    print "Processing Time:"+str((time.time() - start_time)*1000)+" msec.\nCapture total injuries and incident count(sum of injur
    query_4_sql.show(query_4_sql.count(),False)
```

Fig: 3.3.4.1: Sample application to get total incidents grouped by borough, year and month

Below is the snapshot of the output:

```
Count of total records:970702
Processing Time:23.0000019073 msec.
Capture total injuries and incident count(sum of injuries and fatalities) grouped by Borough, year and month
+-------------+----+-----+--------------------------+
|BOROUGH      |year|month|All_Injuries_Incidents_Count|
+-------------+----+-----+--------------------------+
|null         |2017|1    |3465                      |
|MANHATTAN    |2017|1    |779                       |
|BROOKLYN     |2017|1    |1794                      |
|STATEN ISLAND|2017|1    |293                       |
|QUEENS       |2017|1    |1444                      |
|BRONX        |2017|1    |843                       |
|MANHATTAN    |2016|1    |986                       |
|BROOKLYN     |2016|1    |2060                      |
|QUEENS       |2016|1    |1506                      |
|null         |2016|1    |1934                      |
|BRONX        |2016|1    |798                       |
|STATEN ISLAND|2016|1    |164                       |
|MANHATTAN    |2016|10   |1102                      |
|null         |2016|10   |4613                      |
|STATEN ISLAND|2016|10   |288                       |
|BROOKLYN     |2016|10   |2202                      |
|BRONX        |2016|10   |977                       |
|QUEENS       |2016|10   |1936                      |
|STATEN ISLAND|2016|11   |342                       |
|BROOKLYN     |2016|11   |2189                      |
|BRONX        |2016|11   |985                       |
```

Fig: 3.3.4.2: Sample output for total incidents grouped by borough, year and month

Query processing time is **23 Milliseconds**.

Section 4: Summary

4.1 Comments on Apache Spark

We used Apache Spark in standalone deployment mode on a Windows 7 operating system. It has a nice integration with development and analytics packages like Anaconda, Jupyter and Pandas which simplifies the data analysis by providing interfaces for development, debugging and visualization to gain quick and better insights in the data. Data of varied formats (CSV, JSON etc) can be easily imported and analyzed. Also, with its API in a variety of languages like R, Java, Scala and Python, it is easy to adopt.

4.2 Observations and Recommendations

We used Apache Spark to perform data analysis. Using Apache Spark, we were able to perform operations like couniting unique words, perform basic SQL like operations and advanced operations like Roll-up and drill down in real time. The following table summarizes the performance observed.

Serial No.	Data Set	Dataset Size	Performance Measure
1.	WordCountData.txt	60KB, ~1500 Lines	Counted all words in **27 milli-seconds**
2.	NationalNames.csv	42MB, ~2 Million Records	Average Query Time: **11 milli-seconds** (For 4 queries discussed in Section 3)
3.	NYPD_Motor_Vehicle_Collisions.csv	176 MB, ~1 Million Records	Average Query Time: **22 milli-seconds** (For 4 queries discussed in Section 3)

Table 4.1: Evaluation of query performance on Apache Spark

On the basis of above observations, we can conclude that Apache Spark can be used for real time data analysis. Through its in-memory computation the data analysis is quick and efficient. We could also use it to manipulate data like adding additional columns required for aggregations (like roll up by quarter or month) which are very important to gain in-depth insights.

References

[1] "Apache Spark," 20 February 2017. [Online]. Available: https://en.wikipedia.org/wiki/Apache_Spark.

[2] "What is Apache Spark," Hortonworks, 2017. [Online]. Available: http://hortonworks.com/apache/spark/. [Accessed 20 February 2017].

[3] "Divine Comedy: Paradiso E-Text | Canto 1," Gradesaver.com, [Online]. Available: http://www.gradesaver.com/divine-comedy-paradiso/e-text/canto-1. [Accessed 21 February 2017].

[4] "US Baby Names | Kaggle," Kaggle.com, [Online]. Available: https://www.kaggle.com/kaggle/us-baby-names. [Accessed 21 February 2017].

[5] "NYPD Motor Vehicle Collisions | NYC OpenData," Data.cityofnewyork.us, [Online]. Available: https://data.cityofnewyork.us/Public-Safety/NYPD-Motor-Vehicle-Collisions/h9gi-nx95. [Accessed 21 February 2017].

[6] "Deployment Patterns - Machine Learner 1.1.1 - WSO2 Documentation," Docs.wso2.com, [Online]. Available: https://docs.wso2.com/display/ML111/Deployment+Patterns. [Accessed 20 February 2017].

[7] N. Tayal, "How to run Apache Spark on Windows7 in standalone mode," Nishutayaltech.blogspot.ca, [Online]. Available: http://nishutayaltech.blogspot.ca/2015/04/how-to-run-apache-spark-on-windows7-in.html. [Accessed 20 February 2017].

[8] "Installing Jupyter Notebook — Jupyter Documentation 4.1.1 alpha documentation," Jupyter.readthedocs.io, [Online]. Available: http://jupyter.readthedocs.io/en/latest/install.html. [Accessed 20 February 2017].

[9] "Anaconda Overview," Continuum, [Online]. Available: https://www.continuum.io/anaconda-overview. [Accessed 20 February 2017].