Chapter 6. Big Data Processing with Apache Spark

Exercise Workbook

Contents

Lab 1: Working with the PySpark Shell

In this lab, we will use the PySpark shell to explore working with PySpark. First get hands-on experience with the generic shell. In the next lab, we will configure the shell to work with Jupyter and explore how to work with PySpark on Jupyter.

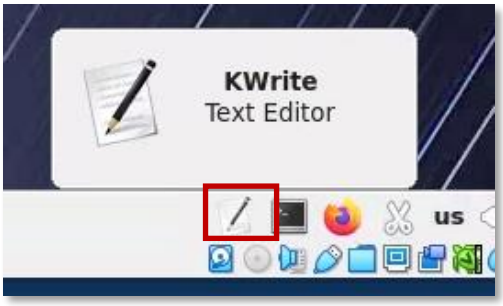
1. Setting up the PySpark environment and starting the shell

   Set up the environmental variables to open the PySpark shell in its native setting

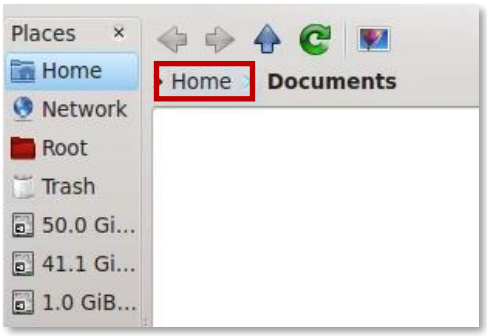
   1.1 Modify your bash shell start up settings

      1.1.1 Open .bashrc file for editing from your home directory. You may use any editing tool of your choice. Here, we shall show using KWrite.

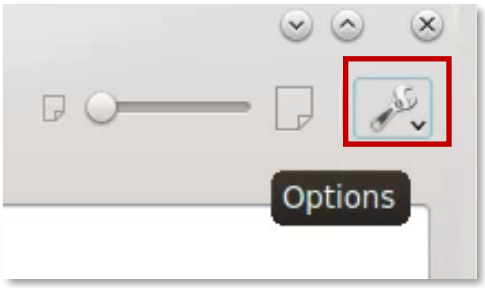
      1.1.2 Open KWrite by clicking on the KWrite text edition icon in the tools panel from the bottom right screen



      1.1.3 Navigate to your home directory



      1.1.4 Click on the wrench icon to modify the settings



1.1.5 Check the Show Hidden Files

1.1.6 Select .bashrc file and edit the file to enable ipython mode

```
# Uncheck the following two lines to run PySpark from ipython
export PYSPARK_DRIVER_PYTHON="ipython"
export PYSPARK_DRIVER_PYTHON_OPTS=""

# Uncheck the following two lines to run PySpark from jupyter
# export PYSPARK_DRIVER_PYTHON="jupyter"
# export PYSPARK_DRIVER_PYTHON_OPTS="notebook --port 3333"
```

The # in the beginning of the line indicates that the line is a comment. Remove the # comment from the two lines of setting for ipython mode. When the # is removed, KWrite will change the color of the text as shown above to indicate that the commands will be executed. Make sure that only the ipython settings are enabled. The Jupyter settings should remain commented with # in the beginning of the line

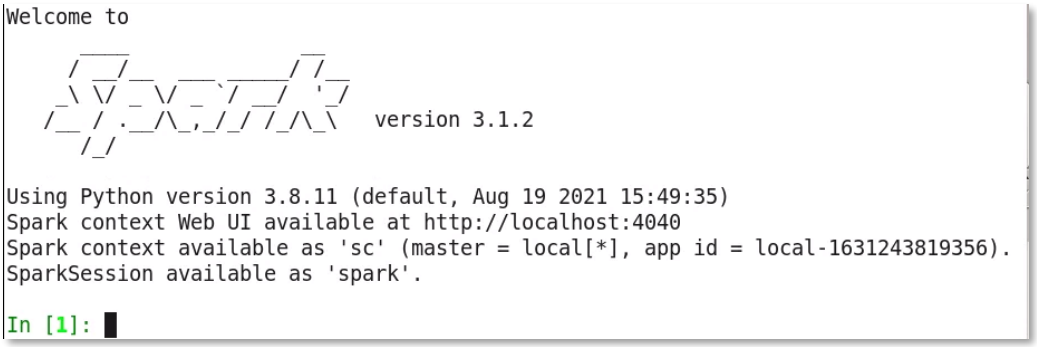
1.1.7 Normally, the .bashrc file gets executed whenever you start a new terminal session and the settings are associated with that terminal only. In order to apply the changed settings, close the current terminal and open a new terminal. Alternatively, you may also execute the following command from the current terminal to execute the .bashrc file

```
[student@localhost ~]$ source .bashrc
```

1.2 Start the pyspark shell

```
[student@localhost ~]$ pyspark
```

1.3 The pyspark command is actually a script which will start up various settings, including creating the SparkContext and the SparkSession. Your screen should be similar to below:



1.4 Check and verify the SparkContext and SparkSession are available

1.4.1 Run sc and spark from the shell

```
In [ ]:  sc
In [ ]:  spark
```

```
In [1]: sc
Out[1]: <SparkContext master=local[*] appName=PySparkShell>

In [2]: spark
Out[2]: <pyspark.sql.session.SparkSession at 0x7f4cfe091f40>
```

1.5 Exit the shell using either the exit() command or Ctrl-d

```
In [3]:  exit()
```

2. Explore a text file on HDFS using Hue and HDFS CLI

2.1 Explore and copy alice_in_wonderland.txt file to your HDFS home directory

2.1.1 Navigate to /home/student/Data directory

```
[student@localhost ~]$ cd /home/student/Data
```

2.1.2 Open alice_in_wonderland.txt with an editor of your choice. KWrite or vim are readily available choices. Review the file and its content. Notice that it is a plain text file.

2.1.3 Copy the file to your HDFS home directory. Name the new file alice.txt

```
[student@localhost ~]$ hdfs dfs -put \
alice_in_wonderland.txt alice.txt
```

2.2 Verify that the file has been uploaded properly. We will use the hdfs command line.

```
[student@localhost ~]$ hdfs dfs -ls
[student@localhost ~]$ hdfs dfs -cat alice.txt
```

2.3 We will use the Hue file browser interface this time

2.3.1 Make sure Zookeeper is running. If not start Zookeeper.

```
[student@localhost ~]$ sudo systemctl status zookeeper
```

If zookeeper is not running or failed, you will see a screen similar to below

```
[student@localhost ~]$ sudo systemctl status zookeeper
● zookeeper.service
   Loaded: loaded (/etc/systemd/system/zookeeper.service; disabled; vendor prese
t: disabled)
   Active: failed (Result: exit-code) since Sat 2021-09-11 16:54:47 KST; 10s ago
  Process: 6526 ExecStop=/home/kafka/kafka/bin/zookeeper-server-stop.sh (code=ex
ited, status=0/SUCCESS)
  Process: 6075 ExecStart=/home/kafka/kafka/bin/zookeeper-server-start.sh /home/
kafka/kafka/config/zookeeper.properties (code=exited, status=143)
 Main PID: 6075 (code=exited, status=143)
```

If zookeeper is running properly, you will see screen similar to below.

```
[student@localhost ~]$ sudo systemctl status zookeeper
● zookeeper.service
   Loaded: loaded (/etc/systemd/system/zookeeper.service; disabled; vendor prese
t: disabled)
   Active: active (running) since Sat 2021-09-11 17:00:13 KST; 5s ago
  Process: 6526 ExecStop=/home/kafka/kafka/bin/zookeeper-server-stop.sh (code=ex
ited, status=0/SUCCESS)
 Main PID: 6602 (java)
   CGroup: /system.slice/zookeeper.service
           └─6602 java -Xmx512M -Xms512M -server -XX:+UseG1GC -XX:MaxGCPauseM...
```

Restart Zookeeper if necessary:

```
[student@localhost ~]$   sudo systemctl restart zookeeper
```

2.4 Follow similar step as above for Hue.

```
[student@localhost ~]$   sudo systemctl status hue
```

Once Hue is running properly, will get a screen similar to below after checking its status

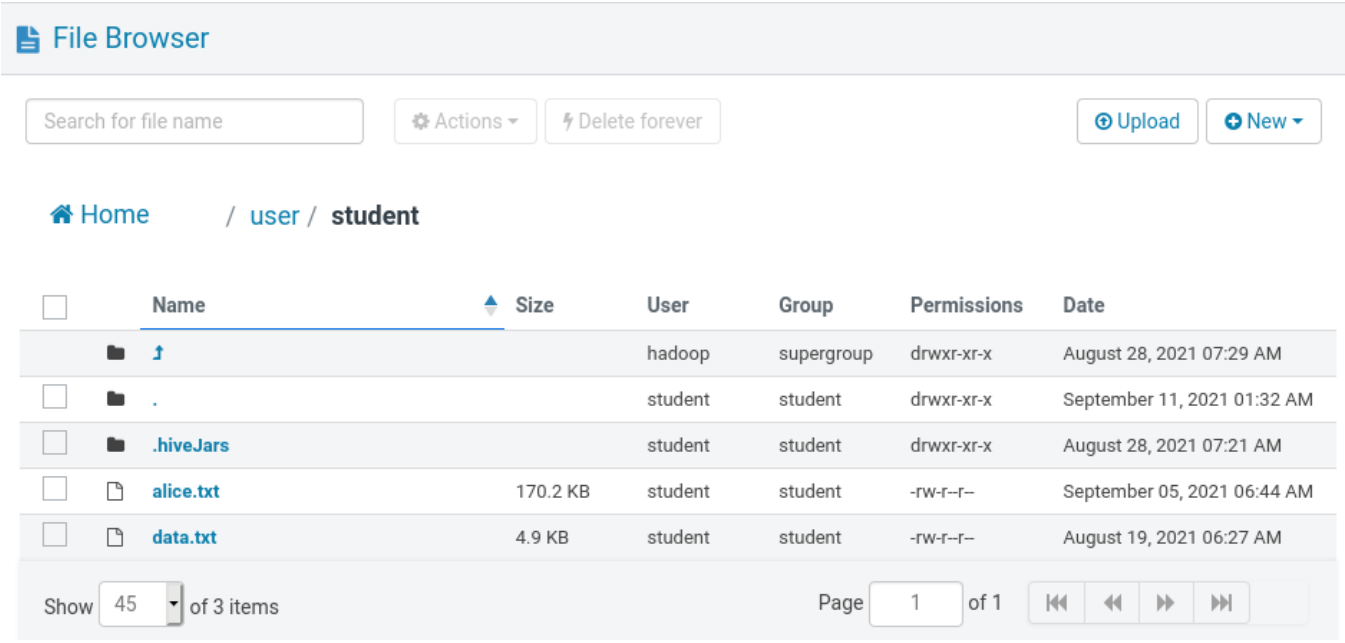
```
[student@localhost ~]$ sudo systemctl status hue
● hue.service
   Loaded: loaded (/etc/systemd/system/hue.service; disabled; vendor preset: dis
abled)
   Active: inactive (dead)
[student@localhost ~]$ sudo systemctl start hue
[student@localhost ~]$ sudo systemctl status hue
● hue.service
   Loaded: loaded (/etc/systemd/system/hue.service; disabled; vendor preset: dis
abled)
   Active: active (running) since Sat 2021-09-11 17:03:55 KST; 2s ago
 Main PID: 7089 (sh)
   CGroup: /system.slice/hue.service
           ├─7089 /bin/sh -c /usr/local/hue/build/env/bin/supervisor > /usr/l...
           ├─7091 /usr/local/hue/build/env/bin/python2.7 /usr/local/hue/build...
           ├─7099 /usr/local/hue/build/env/bin/python2.7 /usr/local/hue/build...
           └─7100 /usr/local/hue/build/env/bin/python2.7 /usr/local/hue/build...
```

Restart Hue if necessary:

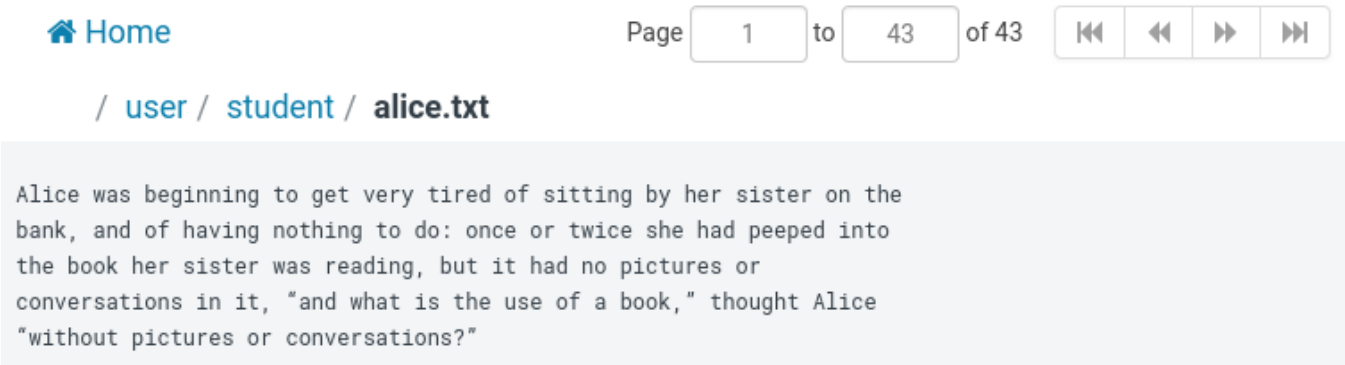
```
[student@localhost ~]$   sudo systemctl restart hue
```

2.5 Open Hue from Firefox browser. Use the available bookmark or use the following URL: http://localhost:8888. Use the following authorization information
Username: student
Password: student

2.6 Select Files from the left tab menu and navigate to the file browser pane.



2.7 Select alice.txt file. Hue will display the content of the file.



If the file is a text file, including JSON, CSV or Plain Text files, Hue is able to display the contents. If the file is in a binary format such as Parquet file format, use the parquet-tools command line tool.

3. Run our first Spark transformations and save to HDFS

3.1 Start PySpark from a terminal

```
[student@localhost ~]$ pyspark
```

3.2 Read alice.txt and verify its content

3.2.1 Create aliceRDD by reading the alice.txt file from our home HDFS directory

```
In [ ]:  aliceRDD = sc.textFile("alice.txt")
```

3.2.2 Display 5 rows of aliceRDD. Use take(n) action operator to return n rows from aliceRDD. Use a for loop to iterate through each of the row items and finally, use print() to display the content of each row.

```
In [ ]:  for line in aliceRDD.take(5):
             print(line)
```

```
In [3]: for line in aliceRDD.take(5):
   ...:     print(line)
   ...:
The Project Gutenberg eBook of Alice's Adventures in Wonderland, by Lewis Carroll

This eBook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever. You may copy it, give it away or re-use it under the terms
```

3.3 Transform alice.txt, selecting only lines that contain rabbit in them.

3.3.1 Create a new RDD with lines that contain "rabbit" in it. Do not distinguish between uppercase and lowercase. Convert all the lines to lowercase first and then filter for lines that contain "rabbit"

```
In [ ]:  rabbitRDD = aliceRDD \
             .map(lambda line: line.lower()) \
             .filter(lambda line: "rabbit" in line)
```

3.3.2 Print out 5 rows of the resulting rabbitRDD and verify that they all contain "rabbit"

3.3.3 Use the count() action command to count how many lines contain "rabbit" How may lines contain "rabbit"?

```
In [8]: rabbitRDD.count()
Out[8]: 53
```

3.4 Save rabbitRDD as a text file to your HDFS home directory and verify that it was properly saved. Name the file "rabbit.txt".

3.4.1 Use saveAsTextFile(<path>) action to save rabbitRDD

```
In [ ]:  rabbitRDD.saveAsTextFile("rabbit.txt")
```

3.4.2 Use the HDFS command line to verify that rabbit.txt has been saved. Notice that rabbit.txt is a directory rather than a text file. This is because Apache Spark is a distributed parallel system with multiple Executors processing the data. Each Executor saves its partition to the designated output path.

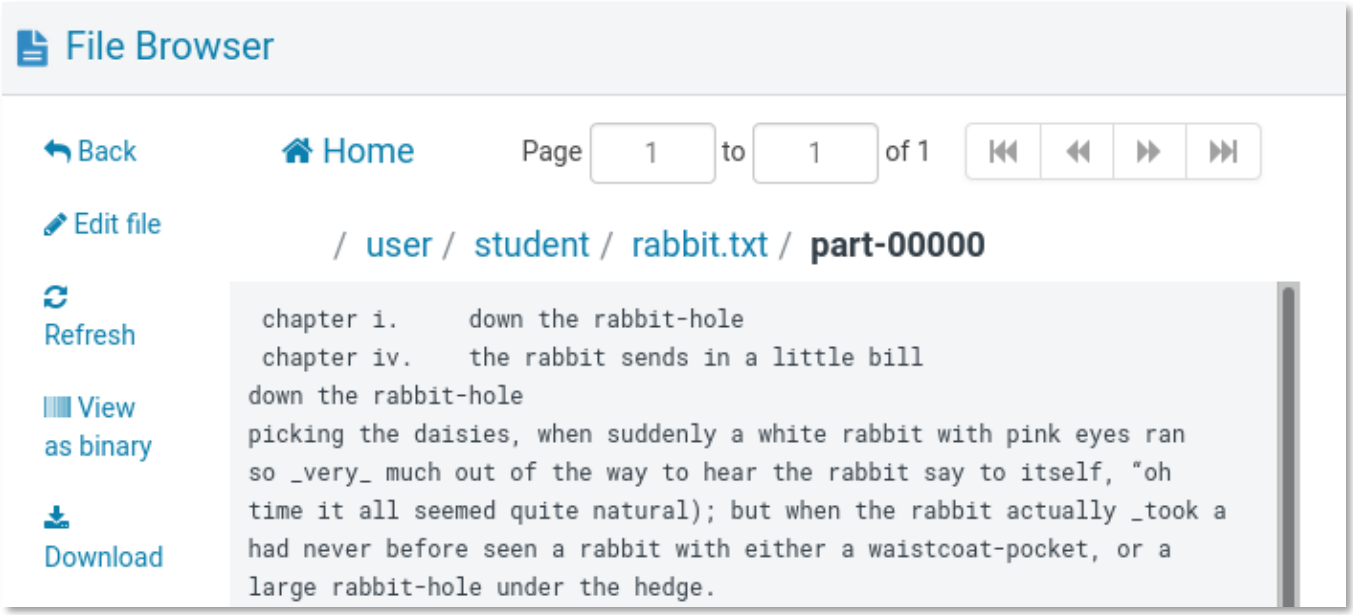
```
[student@localhost ~]$ hdfs dfs -ls rabbit.txt
Found 3 items
-rw-r--r--   1 student student          0 2021-09-11 18:34 rabbit.txt/_SUCCESS
-rw-r--r--   1 student student       1958 2021-09-11 18:34 rabbit.txt/part-00000
-rw-r--r--   1 student student       1614 2021-09-11 18:34 rabbit.txt/part-00001
```

3.4.3 Use the -cat subcommand option to view the contents of part-00000

```
[student@localhost ~]$ hdfs dfs -cat rabbit.txt/part-00000
```

Verify that all of the lines contains the word "rabbit" and all the text is in lower case.

3.4.4 Now use HUE to do the same.



4. Merging HDFS results to a local file

   4.1 Sometimes it might be desirable to merge the output results that are partitioned into multiple files into a single local file. Use the HDFS command line with the --getmerge option to create a single file

```
[student@localhost ~]$ hdfs dfs -getmerge rabbit.txt rabbit.txt
```

   4.2 Verify that a local file with the merged content has been saved to the local drive

```
[student@localhost ~]$ ls
[student@localhost ~]$ cat rabbit.txt
```

Lab 2: Basic Python

1. Working with Jupyter

   1.1 Edit the .bashrc file and change to Jupyter mode

```
# Uncheck the following two lines to run PySpark from ipython
# export PYSPARK_DRIVER_PYTHON="ipython"
# export PYSPARK_DRIVER_PYTHON_OPTS=""

# Uncheck the following two lines to run PySpark from Jupyter
export PYSPARK_DRIVER_PYTHON="jupyter"
export PYSPARK_DRIVER_PYTHON_OPTS="notebook --port 3333"
export PYSPARK_PYTHON=python3
```

   1.2 Start from a new terminal or execute .bashrc file with the source command

   1.3 Start the PySpark shell. The shell should launch Firefox automatically and start Jupyter. In case, the browser does not start automatically, launch from the link provided on the command line.

   1.4 Start a new Notebook and select the Python 3 (ipykernel)

```
[student@localhost ~]$ jupyter notebook
```

   1.5 In Jupyter, enter a command in a cell, press "shift Enter" to execute the command and move to the next cell. Test the interface by checking for the instantiation of a SparkContext and SparkSession object.

```
In [1]: sc
```

```
Out[1]: SparkContext

        Spark UI
        Version
        v3.1.2
        Master
        local[*]
        AppName
        PySparkShell
```

```
In [2]: spark
```

```
Out[2]: SparkSession - hive
        SparkContext

        Spark UI
        Version
        v3.1.2
        Master
        local[*]
        AppName
        PySparkShell
```

2. Applying Python functions to our data

   2.1 Review kv1.txt in /home/student/Data directory. Create an RDD from the file. View the contents of the RDD to verify that the data has been properly loaded.

      2.1.1 Navigate to /home/student/Data and review kv1.txt. You may use any tool of your choice to review the contents of the file. Some options are to use KWrite, use the Linux `cat <filename>` command, etc.

      2.1.2 Use readTextFile() with a local file. The path to a local file must be provide with a full url such as file:/home/student/Data/kv1.txt

      2.1.3 Use the take(n) action to view the contents of the file. Your output should be similar to below

```
myrdd = sc.textFile("file:/home/student/Data/kv1.txt")
myrdd.take(5)
```

```
['238\x01val_238',
 '86\x01val_86',
 '311\x01val_311',
 '27\x01val_27',
 '165\x01val_165']
```

   2.2 Create a 2 element List for each line of the RDD

2.2.1 Use map() to transform each row. Use the String split(<delimiter>) to split the string using "\x01" as the delimiter.

2.2.2 Use take(n) to verify that you have a List for each row in the RDD

```
myrdd = sc.textFile("file:/home/student/Data/kv1.txt") \
          .map(lambda line: line.split("\x01"))
myrdd.take(5)
```

```
[['238', 'val_238'],
 ['86', 'val_86'],
 ['311', 'val_311'],
 ['27', 'val_27'],
 ['165', 'val_165']]
```

2.3 Create a Python function that takes a string, and replaces a pattern string with a replacement string. Apply function to the second element in the List.

2.3.1 Use def to create a function. Use the string replace(<string>,<pattern>,<replacement>) method. Replace the "val_" with "value is " for the second element in the List

2.3.2 Use map() and apply above function to transform the RDD.

2.3.3 Use take(n) to verify your RDD. Your results should be similar to below

```
def replaceStr(string, srch, repl):
    return string.replace(srch, repl)

myrdd = sc.textFile("file:/home/student/Data/kv1.txt") \
        .map(lambda line: line.split("\x01")) \
        .map(lambda array: replaceStr(array[1], "val_", "value is "))
myrdd.take(5)
```

```
['value is 238', 'value is 86', 'value is 311', 'value is 27', 'value is 165']
```

2.4 Create a Python function that will slice a string and return only part of the string. Appy the function to the remove the "value is " prefix in the string. You may choose to create a named function and apply it or use the lambda notation with an anonymous function.

2.4.1 To slice a string, use the [start_index:end_index_non_inclusive] syntax. If the ending index is left blank, Python will start at the start_index and include the rest of the string in its output. "value is " occupies index 0 through 8. We want to slice the string from index 9 to the end.

2.4.2 Use take(n) to verify your RDD. Your results should be similar to below.

```
def replaceStr(string, srch, repl):
    return string.replace(srch, repl)

myrdd = sc.textFile("file:/home/student/Data/kv1.txt") \
        .map(lambda line: line.split("\x01")) \
        .map(lambda array: replaceStr(array[1], "val_", "value is ")) \
        .map(lambda string: string[9:])
myrdd.take(5)
```

```
['238', '86', '311', '27', '165']
```

2.5 Each row in the current RDD is a number represented in a String datatype. Convert each row to a Tuple. The tuple will have two elements each. The first element will be an integer representation of the string number and the second element will be the string number.

2.5.1 Use the `(<first element, <second element>)` operator to create a Tuple. Use the int(<string number>) function to cast a string into an integer.

2.5.2 Use take(n) to verify your RDD. Your results should be similar to below.

```
def replaceStr(string, srch, repl):
    return string.replace(srch, repl)

myrdd = sc.textFile("file:/home/student/Data/kv1.txt") \
        .map(lambda line: line.split("\x01")) \
        .map(lambda array: replaceStr(array[1], "val_", "value is ")) \
        .map(lambda string: string[9:]) \
        .map(lambda string: (int(string), string))
myrdd.take(5)
```

```
[(238, '238'), (86, '86'), (311, '311'), (27, '27'), (165, '165')]
```

2.6 Create a Python function that takes a number and returns True if the number is an even number. Use the function to filter the RDD.

2.6.1 Use the % operator to check if a number is even. The % operator returns the remainder of a division. After dividing the number by 2, if the remainder is 0, the number is an even number.

2.6.2 Use the if statement to return either True if even number or False, otherwise.

2.6.3 Use .filter(<Boolean function>) to select rows where the first element of the Tuple is an even number.

2.6.4 Use take(n) to verify your RDD. Your results should be similar to below.

```
def evenNum(num):
    if (num%2 == 0): return True
    else: return False

evenRDD = myrdd.filter(lambda tupl: evenNum(tupl[0]))
evenRDD.take(5)
```

```
[(238, '238'), (86, '86'), (278, '278'), (98, '98'), (484, '484')]
```

2.7 In Python, every datatype has a True or False value. For integers, every integer other than 0 is true. The only integer that is false is 0. Redo above using a short lambda function instead.

2.7.1 Use the `%` operator with the `not` logical operator.

2.7.2 Use take(n) to verify your RDD. Your results should be similar to below.

```
evenRDD = myrdd.filter(lambda tupl: not (tupl[0]%2))
evenRDD.take(5)
```

```
[(238, '238'), (86, '86'), (278, '278'), (98, '98'), (484, '484')]
```

2.8 Create a function that navigates through a collection of Tuples. For each element of the Tuple, if the element is an integer type, add 1000 to it and print it. If the element is a string type, print two copies of it. If the data type is neither, print "ERROR"

2.8.1 Use the `for` loop to navigate to each element in the collection

2.8.2 Use a nested `for` loop to navigate to each element in the Tuple

2.8.3 Use the type(<variable>) to determine type. To test for integer, use the `is` operator with `int`. To test for string, use the `is` operator with `string`.

2.9 Get the first 5 rows of the evenRDD created above. Pass this collection to the function created in step 2.8 above.

2.9.1 The output of your result should look similar to below.

```
def prTuple(coll):
    for tupl in coll:
        for item in tupl:
            if type(item) is int:
                print(item + 1000)
            elif type(item) is str:
                print("\t", item*2)
            else:
                print("Not int nor string")

small_list = evenRDD.take(5)
prTuple(small_list)
```

```
1238
        238238
1086
        8686
1278
        278278
1098
        9898
1484
        484484
```

3. Practicing Python Basics

   3.1 Create an RDD by reading the alice_in_wonderland.txt file

   3.2 Create a function that converts all words in a sentence to start with uppercase. This is different from converting the entire word into uppercase

      3.2.1 Create an empty string: capWords = "". As each word is capitalized, use the + operator to concat the newly capitalized word to capWords.

      3.2.2 Use the string split(<delimiter>) method to separate the words in the string

      3.2.3 Use a for loop to iterate over each word

      3.2.4 For each word, capitalize the first letter with the string upper() method

      3.2.5 Use string slicing to add the rest of the word. Hint: word(1:) - If the second parameter is empty, end of string is assumed and the rest of the string is returned

```
def makeCap(string):
    capWords = ""
    for word in string.split(" "):
        capWords += (word[0:1].upper() + word[1:])
        capWords += " "
    return capWords

print(makeCap("test this string to see all words in cap"))
```

```
Test This String To See All Words In Cap
```

3.3 Convert all words in the Alice in Wonderland to capitalized words

3.3.1 Use the map transformation with the function you created above

```
capRDD = aliceRDD.map(makeCap)
capRDD.take(5)
```

```
['The Project Gutenberg EBook Of Alice’s Adventures In Wonderland, By Lewis Carroll ',
 ' ',
 'This EBook Is For The Use Of Anyone Anywhere In The United States And ',
 'Most Other Parts Of The World At No Cost And With Almost No Restrictions ',
 'Whatsoever. You May Copy It, Give It Away Or Re-use It Under The Terms ']
```

3.4 As it turns out, Python has a nice string method called capitalize() that does the same thing. Redo step 3.2 above using the capitalize method. Redo step 3.3 using your new function

3.5 This time, we will do the inverse capitalize. Here, we capitalize all the letters other than the first letter.

Lab 3: Working with Core API Transformations

In this lab, we will practice working with the various Core API transformations that were introduced in the lectures. Start a new PySpark shell that runs on Jupyter to begin the labs.

1. Creating RDDs from Source Files

   1.1 Make sure that Hue is up and running. Follow the steps from Lab 1:2.3.

   1.2 Creating RDDs from text files

      1.2.1 Verify that "alice.txt" file exists on your HDFS home directory. It was created from a previous lab. Either use the hdfs dfs -ls command or Hue to do this.

      1.2.2 Use the SparkContext.textFile(<path>) command to read the alice.txt file. Name the new RDD as aliceRDD.

      1.2.3 Verify that aliceRDD has been created properly using take(5)

```
aliceRDD.take(5)
```

```
['The Project Gutenberg eBook of Alice's Adventures in Wonderland, by Lewis Carroll',
 '',
 'This eBook is for the use of anyone anywhere in the United States and',
 'most other parts of the world at no cost and with almost no restrictions',
 'whatsoever. You may copy it, give it away or re-use it under the terms']
```

   1.3 Creating RDDs from s3 buckets

      1.3.1 Make sure an AWS account with free tier access is available. One was created in Hands-On C4U1 Lab1. If not, create one now by following the instructions.

      1.3.2 Make sure the necessary AWS credentials is available to work with AWS CLI. If not, follow the instructions in Hands-On C4U1 Lab 1, Step 3.5 and create a new security credential. AWS only allows two credentials to be created at a time. If two credentials already exist and the Secret Key associated with the Access Key ID had been misplaced, a new one will have to be created. Delete one of the credentials and create a new one.

      1.3.3 Make sure that AWS CLI installed on your computer. If not, follow the instructions from Hands-On C4U1 Lab 1, Step 3.1.

      1.3.4 Make sure that AWS CLI is configured with the proper Access Key ID and Secret Key. Follow the steps from Hands-On C4U1 Lab 1, Step 3.5 if unsure how to do this.

      1.3.5 Use the AWS CLI to create a new bucket. Bucket names become part of the web access URL. Therefore, a globally unique name must be set for the bucket. For the rest of this lab, "<bucket-name>" will be used as the bucket name.

1.3.6 Upload weblog.log file provided to by instructor to s3://<bucket-name>/weblogs/weblog.log using the AWS CLI. Make sure weblog.log file is in current directory before executing the following command. If weblog.log is not in current directory, either navigate to the directory where the file resides or provide a full path to the aws s3 cp command.

```
aws s3 cp weblog.log s3://<bucket-name>/weblogs/weblog.log
```

1.3.7 From Jupyter, run the following command to read weblog.log from the s3 bucket. Make sure to replace the Access Key ID, Secret Key and bucket name with your own information.

```
access_key = "<AWS Access Key ID>"
secret_key = "<AWS Secret Key>"
hadoop_conf=sc._jsc.hadoopConfiguration()
hadoop_conf.set("fs.s3a.impl",
"org.apache.hadoop.fs.s3a.S3AFileSystem")
hadoop_conf.set("fs.s3a.access.key", access_key)
hadoop_conf.set("fs.s3a.secret.key", secret_key)
s3RDD = sc.textFile("s3a://<bucket-name>/weblogs/weblog.log")
```

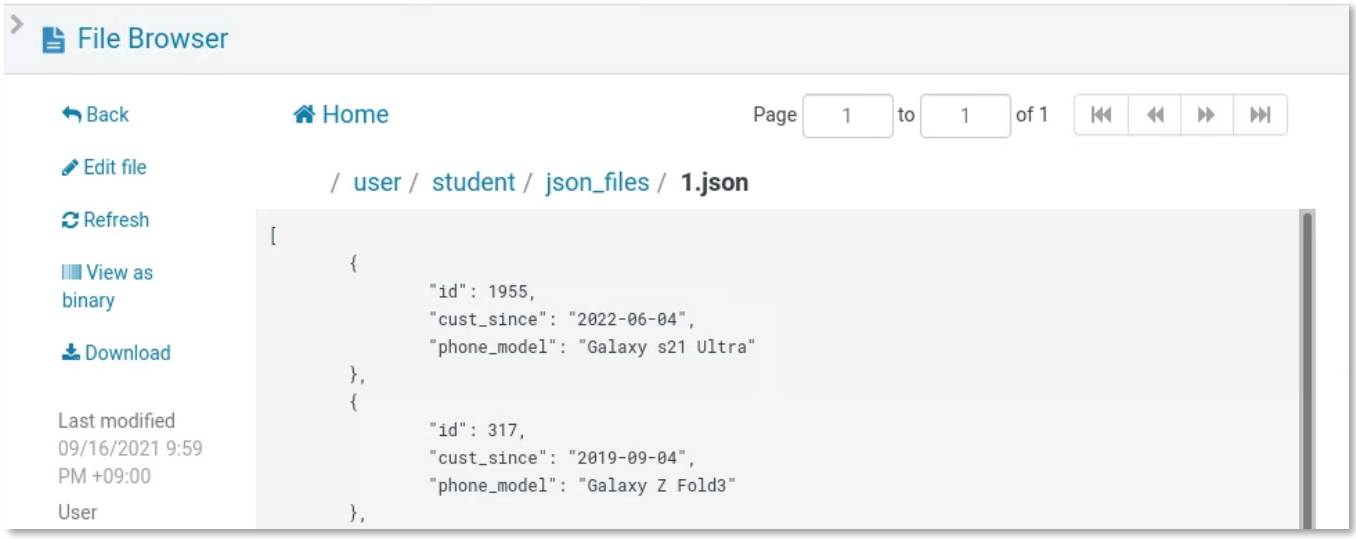
1.3.8 Verify that weblog.log has been read from the s3 bucket by calling .take(5) on the s3RDD

1.4 Creating RDDs from whole text files

1.4.1 Navigate to /home/student/Data directory

1.4.2 Copy the json_files directory to the HDFS home directory

1.4.3 Verify that the files have been copied properly. Use Hue to verify and review the file contents.



1.4.4 From Jupyter use SparkContext.wholeTextFiles(<path to source files>) to read each JSON file as a single element in the newly created RDD. In the path to source, enter an absolute path. It is not necessary to specify the file system as Spark is currently configured to read from HDFS by default. The full path can be observed from Hue above. It is `/user/student/json_files`. Name the new RDD as jsonRDD.

1.4.5 Verify that an RDD has been properly created using take(1). Observe the format of the element in the RDD. The "\t" is a tab. Each element is a pair tuple where the first element is the path to the file and the second element is the content of the file.

```
json_src = "/user/student/json_files/"
jsonRDD = sc.wholeTextFiles(json_src)
jsonRDD.take(1)
```

```
[('hdfs://localhost:9000/user/student/json_files/11.json',
  '[\n\t{\n\t\t"id": 1955,\n\t\t"cust_since": "2022-06-04",\n\t\t"phone_model": "Galaxy s2
1 Ultra"\n\t},\n\t{\n\t\t"id": 317,\n\t\t"cust_since": "2019-09-04",\n\t\t"phone_model": "
Galaxy Z Fold3"\n\t},\n\t{\n\t\t"id": 101,\n\t\t"cust_since": "2019-01-02",\n\t\t"phone_mo
del": "Galaxy A52s"\n\t},\n\t{\n\t\t"id": 5265,\n\t\t"cust_since": "2020-04-11",\n\t\t"pho
ne_model": "Galaxy A52s"\n\t},\n\t{\n\t\t"id": 6219,\n\t\t"cust_since": "2019-06-15",\n\t\
t"phone_model": "Galaxy s21 Ultra"\n\t},\n\t{\n\t\t"id": 1892,\n\t\t"cust_since": "2019-01
-01",\n\t\t"phone_model": "Galaxy A12"\n\t}\n]')]
```

2. Single dataset transformations

2.1 Using the map transformation to partially parse the s3RDD from above.

2.1.1 Use s3RDD.take(5) to observe the data format. The Apache Web log data shows the IP address and the UserID in the locations shown below. In addition, there is a timestamp, followed by activity information.

```
s3RDD.take(5)
```

```
['210.229.232.134 - 7392 [16/Sep/2021:17:52:11 +0900] "GET /posts/posts/explore HTTP/1.0"
200 5065 "http://patton.com/" "Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10 10_3; rv:1.9.5.2
0) Gecko/2020-12-27 01:44:54 Firefox/13.0"',
 '191.117.210.84 - 1850 [16/Sep/2021:17:54:41 +0900] "DELETE /wp-content HTTP/1.0" 200 496
6 "https://hernandez.com/author.jsp" "Mozilla/5.0 (Windows NT 5.2) AppleWebKit/536.0 (KHTM
L, like Gecko) Chrome/17.0.845.0 Safari/536.0"',
 '112.84.194.172 - 1633 [16/Sep/2021:17:58:41 +0900] "GET /explore HTTP/1.0" 200 5071 "htt
ps://www.macdonald-miller.com/privacy/" "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10 11_
1) AppleWebKit/531.0 (KHTML, like Gecko) Chrome/13.0.859.0 Safari/531.0"',
 '108.72.115.92 - 3207 [16/Sep/2021:18:01:57 +0900] "GET /wp-admin HTTP/1.0" 301 4955 "htt
ps://webb.com/" "Mozilla/5.0 (Android 2.3.2; Mobile; rv:60.0) Gecko/60.0 Firefox/60.0"',
 '156.183.43.195 - 5578 [16/Sep/2021:18:03:54 +0900] "GET /app/main/posts HTTP/1.0" 301 49
17 "https://www.quinn.com/author/" "Mozilla/5.0 (X11; Linux x86_64; rv:1.9.5.20) Gecko/201
1-01-03 10:13:07 Firefox/7.0"']
```

2.1.2 Use .map() transformation with String.split(<delimiter>) method to split the string

2.1.3 Observe the result with take(5). Notice that the result is a List with several elements. Also notice that the first element is the IP address and the third element is the UserID.

2.1.4 Use .map() transformation to extract the IP address and the UserID. Create a new tuple with from the IP address and UserID. In Python, create tuples with the (<element>, <element>, <element>, ...) operator. Name the new RDD as IpUserRDD.

2.1.5 Observe the result with take(5). Each element is a tuple of form (IP, UserID)

```
IpUserRDD = s3RDD \
    .map(lambda line: line.split(" ")) \
    .map(lambda lst: (lst[0], lst[2]))
IpUserRDD.take(5)
```

```
[('210.229.232.134', '7392'),
 ('191.117.210.84', '1850'),
 ('112.84.194.172', '1633'),
 ('108.72.115.92', '3207'),
 ('156.183.43.195', '5578')]
```

2.2 Exploring the flatMap transformation

2.2.1 In a new cell, import the JSON library. The library contains methods that helps parse JSON files.

```
import json
```

2.2.2 Use the json.loads(<JSON content) to parse the individual records. The jsonRDD from above step is in (<path to file>, <JSON content>) format. Use map() transformation with json.loads(<JSON content>) to parse the JSON strings. In Python, access each element of a Tuple has the same syntax as accessing List items. To access the second element of a Tuple, use tupleName[1]

2.2.3 Use take(2) to observe the result. How did json.loads() parse the JSON content?

In the previous step, when the JSON files were read using wholeTextFiles(), each JSON file became the JSON content of the (<path> , <JSON content>) tuple that is created. Since each file contains multiple JSON records, when json.loads() is applied to each JSON content, many JSON records are parsed as Python dictionaries. A .Python dictionary is contains Key:Value information. The format is {key:value, key:value, key:value, ...}. Multiple JSON records within each JSON content is parsed as individual dictionaries and all records are returned in a List.

[NEW PAGE INSERTED ON PURPOSE]

```
import json
myrdd = jsonRDD \
    .map(lambda tup: json.loads(tup[1]))
myrdd.take(2)
```

```
[[{'id': 1955, 'cust_since': '2022-06-04', 'phone_model': 'Galaxy s21 Ultra'},
  {'id': 317, 'cust_since': '2019-09-04', 'phone_model': 'Galaxy Z Fold3'},
  {'id': 101, 'cust_since': '2019-01-02', 'phone_model': 'Galaxy A52s'},
  {'id': 5265, 'cust_since': '2020-04-11', 'phone_model': 'Galaxy A52s'},
  {'id': 6219, 'cust_since': '2019-06-15', 'phone_model': 'Galaxy s21 Ultra'},
  {'id': 1892, 'cust_since': '2019-01-01', 'phone_model': 'Galaxy A12'}],
 [{'id': 5449, 'cust_since': '2021-10-05', 'phone_model': 'Galaxy S10'},
  {'id': 1046, 'cust_since': '2021-06-23', 'phone_model': 'Galaxy Z Fold3'},
  {'id': 8295, 'cust_since': '2022-01-11', 'phone_model': 'Galaxy s21 Ultra'},
  {'id': 8549, 'cust_since': '2019-12-25', 'phone_model': 'Galaxy S8'},
  {'id': 3913, 'cust_since': '2020-10-22', 'phone_model': 'Galaxy A52s'},
  {'id': 1450, 'cust_since': '2022-06-20', 'phone_model': 'Galaxy A03'},
  {'id': 7184, 'cust_since': '2021-07-10', 'phone_model': 'Galaxy Note20'},
  {'id': 2070, 'cust_since': '2021-09-13', 'phone_model': 'Galaxy S10'},
  {'id': 546, 'cust_since': '2020-01-05', 'phone_model': 'Galaxy S8'},
  {'id': 4789, 'cust_since': '2021-08-08', 'phone_model': 'Galaxy A03'},
  {'id': 7117, 'cust_since': '2022-06-17', 'phone_model': 'Galaxy Z Fold3'},
  {'id': 4943, 'cust_since': '2020-12-08', 'phone_model': 'Galaxy Note10'},
  {'id': 6862, 'cust_since': '2020-05-19', 'phone_model': 'Galaxy A12'},
  {'id': 5993, 'cust_since': '2020-12-05', 'phone_model': 'Galaxy Note20'},
  {'id': 9087, 'cust_since': '2019-08-24', 'phone_model': 'Galaxy Z Fold3'},
  {'id': 9656, 'cust_since': '2019-06-17', 'phone_model': 'Galaxy S10'},
  {'id': 3720, 'cust_since': '2020-08-01', 'phone_model': 'Galaxy A22'}]]
```

2.2.4 The JSON file was successfully parsed but the result is not in the format that is necessary. Each JSON record should be individual elements. Notice above that we requested 2 rows with take(2) and got 2 Lists, each containing multiple records. Use the flatMap() transformation instead to create a new RDD where each element of the List created by the transformation function is placed in its own row. Modify the code and use flatMap() instead of map().

2.2.5 To return the value portion of a key:value in a dictonary, use the get("field name") method. However, it is good practice to use get("field name", None) instead. This syntax returns None, which is Python's equivalent of Null in case the dictionary does not contain the request Key. Use map transformation to create RDD of (<"id">, (<"cust_since">,<"phone_model">)). The output is a nested Tuple.

```
import json
myrdd = jsonRDD \
    .flatMap(lambda tup: json.loads(tup[1])) \
    .map(lambda kv: (kv.get("id", None), \
                     (kv.get("cust_since", None), kv.get("phone_model", None))))
myrdd.take(5)
```

```
[(1955, ('2022-06-04', 'Galaxy s21 Ultra')),
 (317, ('2019-09-04', 'Galaxy Z Fold3')),
 (101, ('2019-01-02', 'Galaxy A52s')),
 (5265, ('2020-04-11', 'Galaxy A52s')),
 (6219, ('2019-06-15', 'Galaxy s21 Ultra'))]
```

2.3 Use distinct() to remove any duplicate elements in a RDD

2.3.1 Start with aliceRDD that was created in above step. Use flatMap() transformation with the String.split() method to create an RDD of words

2.3.2 Use the RDD.count() action to count number of words.

2.3.3 Apply the distinct() transformation.

2.3.4 Use RDD.count() on the new RDD to count number of distinct words in Alice In Worderland.

```
words = aliceRDD \
    .flatMap(lambda line: line.split(' ')) \
    .count()
print(words)
```

```
31192
```

```
distinct_words = aliceRDD \
    .flatMap(lambda line: line.split(' ')) \
    .distinct() \
    .count()
print(distinct_words)
```

```
5981
```

3. Working with Set operator transformations

3.1 Creating RDDs from a collection.

3.1.1 Create a List with the following elements and name it fruit1: ["Banana", "Pear", "Kiwi", "Peach", "Grape"]

3.1.2 Use SparkContext.parallelize(fruit1) to create fruit1RDD

3.1.3 Use collect() action to verify the RDD

```
fruit1 = ["Banana", "Pear", "Kiwi", "Peach", "Grape"]
fruit1RDD = sc.parallelize(fruit1)
fruit1RDD.collect()
```

```
['Banana', 'Pear', 'Kiwi', 'Peach', 'Grape']
```

3.1.4 Create a List with the following elements and name it fruit2:
["Strawberry", "Kiwi", "Watermelon", "Banana", "Apple"]

3.1.5 Use SparkContext.parallelize(fruit2) to create fruit2RDD

3.1.6 Use collect() action to verify the RDD

```
fruit2 = ["Strawberry", "Kiwi", "Watermelon", "Banana", "Apple"]
fruit2RDD = sc.parallelize(fruit2)
fruit2RDD.collect()
```

```
['Strawberry', 'Kiwi', 'Watermelon', 'Banana', 'Apple']
```

3.2 Create unionRDD by using dataset1.union(dataset2) transformation

```
unionRDD = fruit1RDD.union(fruit2RDD)
unionRDD.collect()
```

```
['Banana',
 'Pear',
 'Kiwi',
 'Peach',
 'Grape',
 'Strawberry',
 'Kiwi',
 'Watermelon',
 'Banana',
 'Apple']
```

3.3 Create intersectRDD by using dataset1. intersection(dataset2) transformation

```
intersectRDD = fruit1RDD.intersection(fruit2RDD)
intersectRDD.collect()
```

```
['Kiwi', 'Banana']
```

3.4 Create subtractRDD by using dataset1. subtract(dataset2) transformation

```
subtractRDD = fruit1RDD.subtract(fruit2RDD)
subtractRDD.collect()
```

```
['Peach', 'Pear', 'Grape']
```

3.5 Create cartesianRDD by using dataset1. cartesian(dataset2) transformation

```
cartesianRDD = fruit1RDD.cartesian(fruit2RDD)
cartesianRDD.collect()
```

```
[('Banana', 'Strawberry'),
 ('Banana', 'Kiwi'),
 ('Banana', 'Watermelon'),
 ('Banana', 'Banana'),
 ('Banana', 'Apple'),
 ('Pear', 'Strawberry'),
 ('Pear', 'Kiwi'),
 ('Pear', 'Watermelon'),
 ('Pear', 'Banana'),
 ('Pear', 'Apple'),
 ('Kiwi', 'Strawberry'),
 ('Kiwi', 'Kiwi'),
 ('Kiwi', 'Watermelon'),
 ('Kiwi', 'Banana'),
 ('Kiwi', 'Apple'),
 ('Peach', 'Strawberry'),
 ('Peach', 'Kiwi'),
 ('Peach', 'Watermelon'),
 ('Peach', 'Banana'),
 ('Peach', 'Apple'),
 ('Grape', 'Strawberry'),
 ('Grape', 'Kiwi'),
 ('Grape', 'Watermelon'),
 ('Grape', 'Banana'),
 ('Grape', 'Apple')]
```

4. Working with Partition based transformations and actions

   4.1 Using mapPartitions() to increase performance.
   mapPartitions() is very similar to map(), however, with a key difference. When using mapPartitions(), Spark passes an iterator that may be used to iterate over the rows in a particular partition. This allows developers to create functions that perform some heavy-duty operations such as creating a connection to a Database just once per partition. The iterator is then used to map some transformation. This could be an operation to update or insert a record into a database using the connection created.

      4.1.1 Create the following function: This function receives an iterator as its input parameter. Think of an iterator as any type of collection that contains multiple items and has a next() method that can be used by the for loop to traverse the collection. The function prints a message and then uses the for loop with the iterator to scan each element. Python's yield operator is an easy way to create a new iterator. Alternatively, an empty List could have been created and each item appended to the empty list.

```
def perPartition(it):
    print("I just did a heavy operation once per partition")
    for item in it:
        yield "updated_" + item
```

4.1.2 Create the following list:
my_records = ["record1", "record2", "record3", "record4", "record5"]

4.1.3 Use SparkContext.parallelize(<collection>, <number of partitions>) to create an RDD with 2 partitions using the my_records collection created above. Name the RDD, recordsRDD.

4.1.4 Transform recordsRDD with mapPartitions(). The mapPartition(<func>) transformation provides an iterator that may be used by <func> function. Use the perPartition() function created above for this purpose.

4.1.5 Use collect() to view the results. Notice that the print statement has been executed twice for each partition. The iterator was used to update each element in each of the partitions.

```
def perPartition(it):
    print("I just did a heavy operation once per partition")
    for item in it:
        yield "updated_" + item
```

```
my_records = ["record1", "record2", "record3", "record4", "record5"]
recordsRDD = sc \
    .parallelize(my_records,2) \
    .mapPartitions(lambda iterator: perPartition(iterator))
recordsRDD.collect()
```

```
I just did a heavy operation once per partition
I just did a heavy operation once per partition
```

```
['updated_record1',
 'updated_record2',
 'updated_record3',
 'updated_record4',
 'updated_record5']
```

4.2 The mapPartitionsWithIndex() transformation is very similar to mapPartitions(). The only real difference is that instead of just an iterators, an index number of the current partition is also passed. The signature for the transformation is mapPartitionsWithIndex(lambda index, iterator: <some function(index, iterator>)

4.2.1 Modify perPartition above to now accept an index number for the partition. This time change the print statement to include which partition is performing the

print by using the partition index number passed. Name the new function, perPartitionIndex.

4.2.2 Modify perPartitionIndex above so that the updated records include the partition index number.

4.2.3 Use collect() to view the results.

```
def perPartitionIndex(index, it):
    print("I just did a heavy operation in partition:", index)
    for item in it:
        yield "updated_" + str(index) + "_" + item
```

```
my_records = ["record1", "record2", "record3", "record4", "record5"]
records2RDD = sc \
    .parallelize(my_records,2) \
    .mapPartitionsWithIndex(lambda index, iterator: perPartitionIndex(index, iterator))
records2RDD.collect()
```

```
I just did a heavy operation in partition: 0
I just did a heavy operation in partition: 1
```

```
['updated_0_record1',
 'updated_0_record2',
 'updated_1_record3',
 'updated_1_record4',
 'updated_1_record5']
```

4.3 Use foreachPartition() action to print a statement per partition

4.3.1 Create the following function:

```
def actionPerPartition1(it):
    print("I just did a foreachPartition action")
```

4.3.2 Use the recordsRDD created above with foreachPartition(<func>). Use actionPerPartition for the <func> input parameter. Since we created 2 partitions, actionPerPartition will be executed twice. The output will be similar to the following:

```
recordsRDD.foreachPartition(actionPerPartition1)
```

```
I just did a foreachPartition action
I just did a foreachPartition action
```

4.4 Use foreachPartition() action to print number of rows in each partition.

4.4.1 Sometimes, after transforming the data, one of the partitions can get overloaded with more data compared to other partitions. This is called a

skewing problem. Create the following function that will print the number of elements in each partition.

```
def actionPerPartition2(it):
    print("I have", len(list(it)), "elements")
```

The iterator passed by foreachPartition is actually a generator. A generator can be thought of as a lazy list. The list is not actualized and in fact may continue to grow. A lazy list is useful to create a list from a streaming source, for example, when the end of the list is unknown. In order to get the current length of the lazy iterator, we cast it into an actual list and then take the length of the list.

4.4.2 Use foreachPartition() on recordsRDD with actionPerPartition2. The output will be similar to below.

```
recordsRDD.foreachPartition(actionPerPartition2)
```

```
I just did a heavy operation once per partition
I have 2 elements
I just did a heavy operation once per partition
I have 3 elements
```

Notice that the output includes the output of the perPartition() function created above. When the lazy iterator is actualized by casting into a list, Spark follows its dependency lineage to get the actual list. Along this dependency lineage was the mapPartitions(perPartition) transformation. This causes Spark to perform this transformation as necessary.

4.4.3 This time, use foreachPartition on records2RDD with actionPerPartiton2. How is the output different? What is going on?

The same thing is actually happening, however, records2RDD's dependency lineage is different than recordsRDD's lineage. records2RDD has the mapPartitionWithIndex(perPartionIndex) in its dependency lineage.

```
records2RDD.foreachPartition(actionPerPartition2)
```

```
I just did a heavy operation in partition: 1
I have 3 elements
I just did a heavy operation in partition: 0
I have 2 elements
```

5. Transformations that changes the number of RDD partitions

5.1 Use coalesce() to reduce number of partitions.

5.1.1 Define function printElements as follows:

```
def printElements(iterator):
    for item in iterator: print(item)
    print("**********")
```

5.1.2 Create an RDD consisting of the numbers 1 through 9. Create the RDD with 4 partitions.

5.1.3 Use getNumPartitions() to print the number of partitions.

5.1.4 Use foreachPartition with printElements on the RDD to print elements in each partition

5.1.5 Use coalesce() to reduce the number of partitions to 2

5.1.6 Use getNumPartitions() again to print the new number of partitions.

5.1.7 Use foreachPartition with printElements on the new RDD to print elements in each partition

5.2 Use repartition() to change number of partition

5.2.1 Create an RDD consisting of the numbers 1 through 9. Create the RDD with 4 partitions.

5.2.2 Use getNumPartitions() to print the number of partitions.

5.2.3 Use foreachPartition with printElements on the RDD to print elements in each partition

5.2.4 Use repartition(4) on the RDD. Normally, repartition is used to change the number of partitions and re-shuffle the data at the same time. By keeping the number of partitions the same, it is easier to observe the data being shuffled by the transformation.

5.2.5 Use getNumPartitions() again to print the new number of partitions.

5.2.6 Use foreachPartition with printElements on the new RDD to print elements in each partition

```
orderNums = [1, 2, 3, 4, 5, 6, 7, 8, 9]
orderRDD = sc.parallelize(orderNums, 4)
print("Number of partitions:", orderRDD.getNumPartitions())
orderRDD.foreachPartition(printElements)
```

```
Number of partitions: 4
```

```
7
8
9
**********
5
6
**********
3
4
**********
1
2
**********
```

```
unitRDD = orderRDD.coalesce(2)
print("Number of partitions:", unitRDD.getNumPartitions())
unitRDD.foreachPartition(printElements)
```

```
Number of partitions: 2
```

```
5
6
7
8
9
**********
1
2
3
4
**********
```

```
repartRDD = orderRDD.repartition(4)
print("Number of partitons:", repartRDD.getNumPartitions())
repartRDD.foreachPartition(printElements)
```

```
Number of partitons: 4
3
4
**********
**********
1
2
**********
5
6
7
8
9
**********
```

6. Miscellaneous transformations

6.1 Use sample() transformation to sample an RDD and create a partial RDD. This transformation is very useful when doing data exploration on a large dataset. sample() can be used with replacement or not. When set to True, sampled data can be repeated. When set to False, once a datapoint has been sampled, it is not replaced with another one, and therefore data is not repeated

6.1.1 Create an RDD consisting of numbers from 1 to 99. Use the Python range(99) function to generate the numbers. Use SparkContext.parallellize to create the RDD. Name the RDD, to100RDD.

6.1.2 Use sample(<with Replacement?>,<fraction to sample>,<seed>) to sample the to100RDD. Sample 20% (0.2) of the data without replacement and use a seed of your choice.

6.1.3 Repeat above, but this time, generate a random number for the seed. Use the following code to generate a random number.

```
import random
seed = int(random.random())
```

6.1.4 This time, take a sample with replacement. Notice that when with replacement is set to True, sampled data can be repeated.

```
import random
seed = int(random.random())
to100RDD = sc.parallelize(range(100))
print(to100RDD.sample(False, 0.2, seed).collect())
```

```
[14, 25, 27, 29, 48, 49, 55, 61, 65, 69, 76, 80, 85, 89, 93, 96, 98]
```

```
to100RDD = sc.parallelize(range(100))
print(to100RDD.sample(True, 0.2, 3654).collect())
```

```
[0, 1, 1, 12, 18, 21, 22, 38, 44, 48, 52, 52, 54, 55, 60, 62, 72, 78, 80, 82, 87, 92, 98]
```

Lab 4: Working with Pair RDDs

In this lab, we will create pair rdds and work with various pair rdd transformations

1. Creating Pair RDDs

   1.1 Using keyBy() to create pair rdds

      1.1.1 Create a List of strings

```
mydata1 = ["Henry, 42, M", "Jessica, 16, F",
           "Sharon, 21, F", "Jonathan, 27, M",
           "Shaun, 11, M", "Jasmine, 62, F"]
```

      1.1.2 Create a new RDD from mydata1 using SparkContext.parallelize(<list>) method

```
myrdd = sc.parallelize(mydata1)
```

      1.1.3 Use take(5) to make sure the RDD has been created

      1.1.4 Parse each string into individual words using the String.split(<delimiter>) function. In the above data, the delimiter is a comma (",")

```
.map(lambda line: line.split(","))
```

      1.1.5 Use take(5) to check your transformation

      1.1.6 Use keyBy(<function to determine key>) to create a pair rdd. keyBy() will use the parameter function to create the key. It will use the data passed to it as the value.

```
.keyBy(lambda collection: collection[0])
```

      1.1.7 Use take(5) to check your transformation. What is the result? Is it as expected?

```
mydata1 = ["Henry, 42, M", "Jessica, 16, F",
           "Sharon, 21, F", "Jonathan, 27, M",
           "Shaun, 11, M", "Jasmine, 62, F"]
```

```
myrdd = sc \
    .parallelize(mydata1)  \
    .map(lambda line: line.split(",")) \
    .keyBy(lambda collection: collection[0])
myrdd.take(5)
```

```
[('Henry', ['Henry', ' 42', ' M']),
 ('Jessica', ['Jessica', ' 16', ' F']),
 ('Sharon', ['Sharon', ' 21', ' F']),
 ('Jonathan', ['Jonathan', ' 27', ' M']),
 ('Shaun', ['Shaun', ' 11', ' M'])]
```

1.2 This time use the map() transformation to accomplish the same thing. Some developers prefer to use map() because it is much more direct as to the transformation. Some developers prefer to use keyBy() because the transformation name itself loudly states that a (Key, Pair) tuple is expected as its output. Trying doing this on your own before turning the page.

```
myrdd = sc \
    .parallelize(mydata1)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[0], collection))
myrdd.take(5)
```

```
[('Henry', ['Henry', ' 42', ' M']),
 ('Jessica', ['Jessica', ' 16', ' F']),
 ('Sharon', ['Sharon', ' 21', ' F']),
 ('Jonathan', ['Jonathan', ' 27', ' M']),
 ('Shaun', ['Shaun', ' 11', ' M'])]
```

Notice that when using the map() transformation, a tuple is explicitly created, whereas with the keyBy() transformation, it is an expected result since the method name alludes that a pair tuple will be created. Which transformation to use is your choice. There isn't any actual performance difference between the two methods.

1.3 Create a more complicated nested pair rdd. Take the data source from above and create a pair rdd of form ("name", ("age", "gender")). "name" and "gender" are strings while "age" is an integer.

1.3.1 Transform each row of strings to a row of List containing all the elements. Use the String.split(",") method to parse the string into individual items.

1.3.2 From each List, create the required complex tuple, thus creating a pair rdd.

1.3.3 take(5) to test your transformations.

```
myrdd = sc \
    .parallelize(mydata1)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[0], (int(collection[1]), collection[2])))
myrdd.take(5)
```

```
[('Henry', (42, ' M')),
 ('Jessica', (16, ' F')),
 ('Sharon', (21, ' F')),
 ('Jonathan', (27, ' M')),
 ('Shaun', (11, ' M'))]
```

1.4 Using flatMapValues to create pair rdds

1.4.1 Expand mydata by adding a list of favorite colors for each person. Each person has chosen two favorite colors. The two colors are delimited with a colon (:)

```
mydata2 = ["Henry,red:blue",
           "Jessica,pink:turquoise",
           "Sharon,blue:pink",
           "Jonathan,blue:green",
           "Shaun,sky blue:red",
           "Jasmine,yellow:orange"]
```

1.4.2 Create a RDD from the List using SparkContext.parallelize()

1.4.3 From each string, parse each of the items and create a List

1.4.4 From the List, select the first element (name) and second element (favorite colors) to create a pair rdd

1.4.5 Use RDD.flatMapValues(<function to create collection from values>) to transform above rdd to form (name, favorite_color). The flatMapValues() transformation expects its input to be in (key, value) pair tuple. Such an RDD was created in above step. Since each person has been allowed to choose two favorite colors, the value portion of the pair tuple will be a string of two colors delimited by a colon (:). Pass flatMapValues a function that will split this string into two colors. Hint: Use the String.split(<delimiter>) function. flatMapValues() will apply the passed function on the values portion of the (key, value) input. The function is expected to create a collection. flatMapValues() will then "flatten" each of the colors and place in its separate (key,value) pair. It will duplicate the original key for each new row created.

1.4.6 Use take(5) to test the transformations.

```
mydata2 = ["Henry,red:blue",
           "Jessica,pink:turquoise",
           "Sharon,blue:pink",
           "Jonathan,blue:green",
           "Shaun,sky blue:red",
           "Jasmine,yellow:orange"]
```

```
favColorRDD = sc \
    .parallelize(mydata2)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[0], collection[1])) \
    .flatMapValues(lambda colors: colors.split(":"))
favColorRDD.take(5)
```

```
[('Henry', 'red'),
 ('Henry', 'blue'),
 ('Jessica', 'pink'),
 ('Jessica', 'turquoise'),
 ('Sharon', 'blue')]
```

2. Aggregation transformations with Pair RDDs

2.1 Calculate the sum of ages for all males and the sum of ages for all females.

2.1.1 Use mydata1 from previous step to create a pair RDD of form (gender, age)

```
genderSumAge = sc \
    .parallelize(mydata1)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[2], int(collection[1])))
genderSumAge.take(6)
```

```
[(' M', 42), (' F', 16), (' F', 21), (' M', 27), (' M', 11), (' F', 62)]
```

2.1.2 The gender is key and the age is the value of the pair rdd created in above step. Use reduceByKey(with v1+v2 function) to add all the ages of rows with the same key. The gender is currently the key, so all values of rows with the same gender will be added.

2.1.3 Use the collect() action to view the results

```
genderSumAge = sc \
    .parallelize(mydata1)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[2], int(collection[1]))) \
    .reduceByKey(lambda v1, v2: v1+v2)
genderSumAge.collect()
```

```
[(' M', 80), (' F', 99)]
```

2.2 Calculate the maximum age for each gender. Change the function passed to reduceByKey() to calculate the maximum. Python has a max() function.

2.3 Repeat above to calculate the minimum this time. Python has a min() function

```
genderMaxAge = sc \
    .parallelize(mydata1)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[2], int(collection[1]))) \
    .reduceByKey(lambda v1, v2: max(v1,v2))
genderMaxAge.take(6)
```

```
[(' M', 42), (' F', 62)]
```

```
genderMinAge = sc \
    .parallelize(mydata1)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[2], int(collection[1]))) \
    .reduceByKey(lambda v1, v2: min(v1,v2))
genderMinAge.take(6)
```

```
[(' M', 11), (' F', 16)]
```

2.4 This time, instead of reduceByKey(), use the countByKey () action to produce an output of dictionary datatype. Print out the dictionary.

```
countGender = sc \
    .parallelize(mydata1)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[2], int(collection[1]))) \
    .countByKey()
print(countGender)
```

```
defaultdict(<class 'int'>, {' M': 3, ' F': 3})
```

A dictionary in Python is a data structure consisting of key:value elements. The output should have two elements, one for each gender and the count for that gender.

2.5 Print out the 3 oldest persons from mydata1.

2.5.1 Create a pair rdd of (age, name) from mydata1

2.5.2 Use sortByKey to sort by descending order

2.5.3 Flip the (age, name) row items to (name, age)

2.5.4 Use take(3) to print out the 3 oldest persons

```
sortAgeRDD = sc \
    .parallelize(mydata1)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (int(collection[1]), collection[0])) \
    .sortByKey(ascending = False) \
    .map(lambda tup: (tup[1], tup[0]))
sortAgeRDD.take(3)
```

```
[('Jasmine', 62), ('Henry', 42), ('Jonathan', 27)]
```

2.6 Using mydata2, produce a report that shows for each color, all persons whose favorite color it is.

2.6.1 Follow steps 1.4 from above to create a pair rdd tuple with (person, color) information for all records of a person and their favorite colors in mydata2.

2.6.2 Swap the tuple so that each row shows (color, person)

2.6.3 Use groupByKey to group each color and create a list of persons whose favorite color it.

2.6.4 Use a nested for loop to print each color and all persons whose favorite color it is. Add a tab or some spacing on the inner loop so that a tabulated output it produced.

```
for color in colorLikers.collect():
    print(color[0])
    for person in color[1]:
        print("  ", person)
```

```
colorLikers = sc \
    .parallelize(mydata2)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[0], collection[1])) \
    .flatMapValues(lambda colors: colors.split(":")) \
    .map(lambda tup: (tup[1], tup[0])) \
    .groupByKey()

for color in colorLikers.collect():
    print(color[0])
    for person in color[1]:
        print("   ", person)
```

```
orange
   Jasmine
blue
   Henry
   Sharon
   Jonathan
green
   Jonathan
sky blue
   Shaun
yellow
   Jasmine
red
   Henry
   Shaun
pink
   Jessica
   Sharon
turquoise
   Jessica
```

2.7 The groupByKey() is a very expensive operation because it requires all partitions to exchange their entire data sets with each other. There is no opportunity to aggregate the data before the exchange occurs.

A less expensive method is to use the aggregateByKey(<initial value>, <aggregation function within partition>, <aggregation function between partitions>) transformation instead. The aggregateByKey method is provided a function that allows partitions to aggregate the data while still within the partition. This produces a local aggregation dataset within the partition. This aggregated dataset is typically much smaller and can dramatically reduce the amount of data that has to be exchanged amongst partitions. Finally, another function is provided that aggregates the data amongst the partitions, producing a global aggregated dataset.

Redo the transformations to produce the report from the previous step 2.6. Instead of groupByKey in the last step, use aggregateByKey().

2.7.1 Initialize the starting aggregation value to an empty list. The aggregation functions will append to this empty list, all persons who like a key value color.

```
zeroValue = []
```

2.7.2 Create a `seqOp(accumulator, element)` function that performs the local aggregation. The `accumulator` will initially hold the `zeroValue`, in other words, an empty list. Each person whose key is equal to the key being aggregated, will be passes as `element`. Append an element to a List using the List.append() method.

```
def seqOp(accumulator, element):
    accumulator.append(element)
    return accumulator
```

2.7.3 Create `combOp(accumulator1, accumulator2)` function that performs the global aggregation. Each partition will pass its local accumulated dataset to this function. This function, must therefore, combine each List from each Partition that contains the persons who like the key color being aggregated. In Python, the plus (+) operator concatenates two Lists.

```
def combOp(accumulator1, accumulator2):
    return accumulator1 + accumulator2
```

2.7.4 Create a RDD from mydata2 with a two (2) partitions. A second parameter can be added to the parallelize() method to manually set the number of partitions.

```
sc.parallelize(mydata2, 2)
```

2.7.5 Check the output result with the collect() action.

```
colorLikers2.collect()
```

```
zeroValue = []

def seqOp(accumulator, element):
    accumulator.append(element)
    return accumulator

def combOp(accumulator1, accumulator2):
    return accumulator1 + accumulator2

colorLikers2 = sc \
    .parallelize(mydata2, 2)  \
    .map(lambda line: line.split(",")) \
    .map(lambda collection: (collection[0], collection[1])) \
    .flatMapValues(lambda colors: colors.split(":")) \
    .map(lambda tup: (tup[1], tup[0])) \
    .aggregateByKey(zeroValue, seqOp, combOp)

colorLikers2.collect()
```

```
[('green', ['Jonathan']),
 ('sky blue', ['Shaun']),
 ('yellow', ['Jasmine']),
 ('orange', ['Jasmine']),
 ('red', ['Henry', 'Shaun']),
 ('blue', ['Henry', 'Sharon', 'Jonathan']),
 ('pink', ['Jessica', 'Sharon']),
 ('turquoise', ['Jessica'])]
```

2.8 Joining Pair RDDs

2.8.1 Create a pair rdd from mydata1. The pair rdd should be in the following form: (name, [age, gender]). Name the rdd, data1RDD.

2.8.2 Create a pair rdd from mydata2. The pair rdd should be in the following form: (name, [color1, color2]). Name the rdd, data2RDD.

2.8.3 Join data1RDD with data2RDD. Observe the output format.

```
data1RDD.join(data2RDD)
```

2.8.4 Transform the joined RDD to the following form: "name, age, gender, color1, color2." To access each item within the nested data structure, chain the indexes as follows: [index][inside index][way inside index]. For example, if the data is in the following form, access gender and color2 with the following syntax

```
myTuple = (name,(List[age, gender],   List[color1, color2]))
# indexes [0]   [1][0]     [1][0][1]  [1][1]       [1][1][1]
myGender = myTuple[1][0][1]
```

```
myColor2 = myTuple[1][1][1]
```

2.8.5 Save the resulting RDD to /user/student/persons directory.

[THIS PAGE LEFT BLANK ON PURPOSE]

```
data1RDD = sc.parallelize(mydata1) \
    .map(lambda line: (line.split(",")[0], [line.split(",")[1],line.split(",")[2]]))

data2RDD = sc.parallelize(mydata2) \
    .map(lambda line: (line.split(",")[0], line.split(",")[1])) \
    .map(lambda tup: (tup[0], [tup[1].split(":")[0], tup[1].split(":")[1]]))

joinRDD = data1RDD.join(data2RDD)

outRDD = joinRDD \
    .map(lambda tup: [tup[0], tup[1][0][0], tup[1][0][1], tup[1][1][0], tup[1][1][1]]) \
    .map(lambda lst: ",".join(lst))

outRDD.saveAsTextFile("/user/students/persons/")

outRDD.take(5)
```

```
['Jessica, 16, F,pink,turquoise',
 'Jonathan, 27, M,blue,green',
 'Henry, 42, M,red,blue',
 'Shaun, 11, M,sky blue,red',
 'Sharon, 21, F,blue,pink']
```

Lab 5: Putting it all together

In our MySQL database, there are records of authors who have posted messages. There is an XML file which shows the latitude and longitude of the location when the message was posted. There is a json file for each author which shows the type of phones owned by the author.

Put together all this information and produce a report that shows Author (first name and last name) posted message (first few words of the title of post) using phone (a list of phones owned by author) at location (latitude, longitude).

1. Prepare the data sources

   1.1 Examine authors tables in MySQL

      1.1.1 Use the following command to access MySQL:

```
$  mysql -u student -p
Enter password: # type student when prompted for the password
```

      1.1.2 From mysql, use the following commands to examine the schema for the authors tables

```
show databases;
use labs;
show tables;
desc authors;
```

```
MariaDB [labs]> desc authors;
+------------+-------------+------+-----+-------------------+----------------+
| Field      | Type        | Null | Key | Default           | Extra          |
+------------+-------------+------+-----+-------------------+----------------+
| id         | int(11)     | NO   | PRI | NULL              | auto_increment |
| first_name | varchar(50) | NO   |     | NULL              |                |
| last_name  | varchar(50) | NO   |     | NULL              |                |
| email      | varchar(100)| NO   | UNI | NULL              |                |
| birthdate  | date        | NO   |     | NULL              |                |
| added      | timestamp   | NO   |     | CURRENT_TIMESTAMP |                |
+------------+-------------+------+-----+-------------------+----------------+
6 rows in set (0.00 sec)
```

   1.2 Use Sqoop to import the authors table to /user/student/authors HDFS directory

      1.2.1 Use sqoop import subcommand

1.2.2 Set the --connect string to jdbc:mysql://localhost/<database name>

1.2.3 Set the authorizations: --username and -password are both "student"

1.2.4 Set the --table to import to authors

1.2.5 Set the --target-dir where the imported data will be saved to /user/student/authors

1.2.6 Save as text file using --as-textfile

```
sqoop import \
--connect jdbc:mysql://localhost/labs \
--username student --password student \
--table authors --target-dir /user/student/authors \
--as-textfile
```

1.2.7 Use HDFS command line or Hue to verify that the data has been imported

1.3 Examine the posts table from MySQL. Use DESC command to get a printout of the posts table schema

```
MariaDB [labs]> desc posts;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| id          | int(11)      | NO   | PRI | NULL    | auto_increment |
| author_id   | int(11)      | NO   |     | NULL    |                |
| title       | varchar(255) | NO   |     | NULL    |                |
| description | varchar(500) | NO   |     | NULL    |                |
| content     | text         | NO   |     | NULL    |                |
| date        | date         | NO   |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)
```

1.4 Use Sqoop to import the posts table to /user/student/posts HDFS directory

1.4.1 Use sqoop import subcommand

1.4.2 Set the --connect string to jdbc:mysql://localhost/<database name>

1.4.3 Set the authorizations: --username and -password are both "student"

1.4.4 Set the --table to import to posts

1.4.5 Set the --target-dir where the imported data will be saved to /user/student/posts

1.4.6 Save as text file using --as-textfile

```
sqoop import \
--connect jdbc:mysql://localhost/labs \
--username student --password student \
--table posts --target-dir /user/student/posts \
--hive-drop-import-delims \
--as-textfile
```

1.4.7 Use HDFS command line or Hue to verify that the data has been imported

1.5 Copy author_phone.json file to /user/student/author_phone.json in HDFS

1.5.1 Navigate to /home/student/Data in the Linux directory

1.5.2 Examine author_phone.json file and review its content and data schema

1.5.3 Use HDFS -put subcommand to make a copy on HDFS.

1.6 Copy post_records directory to /user/student/post_records folder in HDFS.

1.6.1 Navigate to /home/student/Data/post_records in the Linux directory

1.6.2 Examine any of the XML files and review its content and data schema

1.6.3 Navigate back up to /home/student/Data

1.6.4 Use HDFS -put subcommand to make a copy of the entire post_records directory in HDFS

2. Create RDDs from the four (4) data sources in HDFS

2.1 Create authorNameRDD, matching the First Name and Last Name with the author <id>

2.1.1 Using SparkContext.textFile(), read the data from the imported authors table

2.1.2 Transform the RDD to form (<id>, (first_name, last_name))
The pair rdd shows first name and last name for each author <id>. Make sure to cast <id> to an integer from string. The resulting rdd should look similar to below

```
[(1, ('Walton', 'Adams')),
 (2, ('Marietta', 'Walsh')),
 (3, ('Lily', 'Wintheiser')),
 (4, ('Estevan', 'Gleason')),
 (5, ('Thaddeus', 'Rowe'))]
```

2.2 Create postsRDD, showing the post <id> with the author_id of the post and first few letters of the title of each post

2.2.1 Using SparkContext.textFile(), read the data from the imported posts table

2.2.2 Transform the RDD to form (<id>, (author_id, <first 10 letters of the title>)) The pair rdd shows for each post <id>, the author_id of the post and the first 10 letters of the title of the post. The resulting rdd should look similar to below:

```
[('1', ('1', 'Cupiditate')),
 ('2', ('2', 'Excepturi ')),
 ('3', ('3', 'Enim rerum')),
 ('4', ('4', 'Labore ips')),
 ('5', ('5', 'Placeat ex'))]
```

2.3 Create phoneRDD from authors_phone.json file. Some of the authors have multiple phones and the phone model for some of the authors is not known. Unfortunately, rather than showing "Unknown, the data shows an empty string"

2.3.1 Import the json library in order to parse the JSON records

2.3.2 authors_phone.json file does not have records delimited by a newline. Use wholeTextFiles to read the source file

2.3.3 The JSON library has a json.loads(<JSON string>) method that creates a collection of JSON records. Use flatMap() transformation with the json.loads() function to create a new row for each JSON record.

2.3.4 Each JSON record is in the form of a Python dictionary. A dictionary is a key:value data structure. To get the value of a key, use the Dictionary.get(<key name>, None) method. This method returns either the Value for the matching Key or None if not found. Using the get() method, transform the JSON record into a pair tuple of form (author_id, phone_model)

2.3.5 Create a function that examines a string and returns "Unkown" if empty, replaces "," with " or", or simply returns the string if and "," is not in the string

```
def setPhoneName(s):
    if s == "": return "Unknown"
    elif "," in s: return s.replace(",", " or")
    else: return s
```

2.3.6 Using the setPhoneName() function, transform the (author_id, phone_model) by applying setPhoneName to the phone_model. The result should still be a

pair tuple of form (author_id, <modified phone model after applying setPhoneName>). The resulting rdd should look similar to below:

```
[(1, 'Samsung Galaxy s21 Ultra'),
 (2, 'Samsung Galaxy A52s'),
 (3, 'Samsung Galaxy Z Fold3'),
 (4, 'Apple iPhone 7'),
 (5, 'Apple iPhone 11'),
 (6, 'Samsung Galaxy A12'),
 (7, 'Samsung Galaxy A52 or Samsung Galaxy s21 Ultra'),
 (8, 'Samsung Galaxy A52s'),
 (9, 'Samsung Galaxy Note 20 or Samsung Galaxy Note20'),
 (10, 'Unknown'),
```

2.4 Create latlongRDD from the XML files in post_records. The output should be of form (post_id, (latitude, longitude))

2.4.1 The XML files contain post_id and location fields. The location field is a comma delimited field showing the latitude and longitude. Use wholeTextFiles() to read the XML files. wholeTextFiles() returns a tuple of form (<path to file>, <XML content>). The XML content contains many XML records.

2.4.2 Define getPosts(XML string) helper functions This function will parse the XML content and return a collection of XML records.

```
import xml.etree.ElementTree as ET
def getPosts(s):
    posts = ET.fromstring(s)
    return posts.iter("record")
```

2.4.3 getPosts(<XML Content>) will return a collection of XML records. The <XML Content> is in the Value portion of the tuple created by wholeTextFiles(). In order to create a new row for each XML record in the collection returned by getPosts(), use flatMap() instead of map() to apply the transformation.

2.4.4 Define getPostID(element) helper function. This function will read a XML element and return the <post_id> field as a string.

```
def getPostID(elem):
    return elem.find("post_id").text
```

2.4.5 Define getLocation(element) helper function. This function will read a XML element and return the <location> field as a string.

```
def getPostLocation(elem):
    return elem.find("location").text
```

2.4.6 Using the two help functions just created, parse the post_id and location information from each XML record. Transform the RDD to form (<post_id>, <location>) pair rdd.

2.4.7 The <location> information contains latitude, longitude information with a comma (",") delimiter. Use the String.split(<delimiter>) method to transform the (<post_id>, <location>) pair rdd to (<post_id>, (<latitude>,<logitude>)). The resulting rdd should look similar to below:

```
[('1', ('-78.67343', ' 146.15251')),
 ('2', ('-24.8449', ' 71.73862')),
 ('3', ('33.15167', ' 90.2584')),
 ('4', ('78.53576', ' -113.2306')),
 ('5', ('37.09904', ' 141.78509'))]
```

3. Combine and join data for insight

3.1 Join authorNameRDD with phoneRDD to match the author's name with their phone(s)

3.1.1 Use the RDD1.join(RDD2) transformation to join authorNameRDD with phoneRDD on the common author_id key. Name the new RDD, authorPhoneRDD. authorPhoneRDD will be of form(<author_id>, ((<first_name>,<last_name>), <phone model names>))

3.1.2 take(5) to verify the output. The resulting output should be similar to below.

```
[(5, (('Thaddeus', 'Rowe'), 'Apple iPhone 11')),
 (10, (('Dewitt', 'Smitham'), 'Unknown')),
 (15, (('Brenda', 'Mayer'), 'Apple iPhone 8')),
 (20, (('Wilfredo', 'Yundt'), 'Samsung Galaxy A12 or Apple iPhone 6')),
 (25, (('Westley', 'Rempel'), 'Unknown'))]
```

3.1.3 Use count() to make sure that there are 10000 records

3.1.4 Transform the format of authorPhoneRDD to (<author_id>, [<first_name>,<last_name>,<phone model names>]). Name the new RDD authorNamePhoneRDD which should look similar to below:

```
[(5, ['Thaddeus', 'Rowe', 'Apple iPhone 11']),
 (10, ['Dewitt', 'Smitham', 'Unknown']),
 (15, ['Brenda', 'Mayer', 'Apple iPhone 8']),
 (20, ['Wilfredo', 'Yundt', 'Samsung Galaxy A12 or Apple iPhone 6']),
 (25, ['Westley', 'Rempel', 'Unknown'])]
```

3.2 Join postsRDD with latlongRDD to match each post information with the location of the post.

3.2.1 Use the RDD1.join(RDD2) transformation to join postsRDD with latlongRDD. Name the new RDD, postLocationRDD. It will be of form (<post_id>, ((<author_id>,<title>), (<latitude>,<longitude>))). Its output should be similar to below:

```
[('4', (('4', 'Labore ips'), ('78.53576', ' -113.2306'))),
 ('16', (('16', 'Eius ea ha'), ('41.55404', ' 7.26911'))),
 ('20', (('20', 'Facere atq'), ('71.33242', ' 7.33176'))),
 ('22', (('22', 'Sint iusto'), ('2.78725', ' 124.55732'))),
 ('24', (('24', 'Dolorem ve'), ('56.27864', ' -115.38594')))]
```

3.2.2 Transform the format of postLocationRDD to (<author_id>, [<title>,<latitude>,<longitude>]) and name it authorPostLocationRDD. Its output should be similar to below:

```
[(4, ['Labore ips', '78.53576', ' -113.2306']),
 (16, ['Eius ea ha', '41.55404', ' 7.26911']),
 (20, ['Facere atq', '71.33242', ' 7.33176']),
 (22, ['Sint iusto', '2.78725', ' 124.55732']),
 (24, ['Dolorem ve', '56.27864', ' -115.38594'])]
```

3.3 Join authorNamePhoneRDD with authorPostLocationRDD.

3.3.1 Name the output of the join to nameTitlePhoneLocRDD. The resulting output will have form (<author_id>, ([<first_name>,<last_name>,<phone_model_names>], [<title>,<latitude>,<longitude>])).

3.3.2 Transform above output to a tuple of ("string", "string") data type. Create the two strings by concatenating various pieces of information. Use the Python plus (+) operator to concat string literals with values. ("<first_name> <last_name> on <phone model names>" , "Posted <title> from lat: <latitude> lon: <longitude>") The resulting output should look similar to below:

```
[('Brigitte Shanahan on Unknown',
  'Posted Et sit har from lat: 70.80241 lon: 32.37947'),
 ('Brigitte Shanahan on Unknown',
  'Posted Voluptates from lat: -64.13821 lon: -129.31313'),
 ('Brigitte Shanahan on Unknown',
  'Posted Vel debiti from lat: -15.29791 lon: -110.17868'),
 ('Brigitte Shanahan on Unknown',
  'Posted Debitis am from lat: 30.79221 lon: 73.33461'),
 ('Brigitte Shanahan on Unknown',
  'Posted Possimus n from lat: 56.51724 lon: 19.96479')]
```

3.4 Group all the posts from the same author.

3.4.1 First Use .groupByKey() on nameTitlePhoneLocRDD. This is a very expensive operation. In fact, because it is so expensive, Spark performs the action lazily. Instead of returning the actual results, Spark returns Resultiterable objects. These objects are lazy iterables and their values have not yet been calculated.

```
[('Laurianne Jerde on Samsung Galaxy S21',
  <pyspark.resultiterable.ResultIterable at 0x7f348cd02460>),
 ('Theodora Nicolas on Unknown',
  <pyspark.resultiterable.ResultIterable at 0x7f348cd024c0>),
 ('Rodolfo Will on Unknown',
  <pyspark.resultiterable.ResultIterable at 0x7f348cd02520>),
 ('Filomena Weimann on Samsung Galaxy A11',
  <pyspark.resultiterable.ResultIterable at 0x7f348cd025b0>),
 ('Marcus Osinski on Apple iPhone 6',
  <pyspark.resultiterable.ResultIterable at 0x7f348cd025e0>)]
```

3.4.2 Try again, using the aggregateByKey() transformation. What should the sequence function look like? How about the combination function? The final output will look similar to below.

```
[('Laurianne Jerde on Samsung Galaxy S21',
  ['Posted Id repudia from lat: -70.76073 lon: 150.41541',
   'Posted Eos nihil  from lat: -60.27312 lon: 0.28329',
   'Posted Neque nam  from lat: -15.1393 lon: -153.41288',
   'Posted Quam exped from lat: 59.64538 lon: 162.21816',
   'Posted Vero enim  from lat: 75.63085 lon: -179.84681',
   'Posted Explicabo  from lat: 28.53925 lon: 39.01707',
   'Posted Quis corru from lat: 18.68474 lon: -134.40318',
   'Posted Repellat i from lat: -21.51617 lon: -178.06651',
   'Posted Magni blan from lat: -63.58662 lon: -126.2323',
   'Posted Qui non as from lat: 59.41561 lon: 140.52858',
   'Posted Est at exc from lat: -19.77205 lon: -22.35178']),
 ('Theodora Nicolas on Unknown',
  ['Posted Id odio di from lat: 60.95745 lon: 165.47261',
   'Posted Vel consec from lat: -0.87067 lon: 98.3268',
   'Posted Ea ea corr from lat: 24.36141 lon: -16.24',
   'Posted Dolor dolo from lat: -10.56048 lon: -177.68041',
   'Posted Delectus c from lat: -19.3932 lon: -116.58581',
   'Posted Dicta nesc from lat: 6.52134 lon: 74.41023',
   'Posted Tempora qu from lat: 65.80853 lon: -96.85641',
   'Posted Quis verit from lat: 12.82261 lon: 33.00807',
   'Posted Ipsam quae from lat: -8.92964 lon: -108.6568',
   'Posted Accusantiu from lat: 30.05355 lon: -83.10026',
   'Posted Iure velit from lat: -67.15977 lon: 16.55862']),
 ('Rodolfo Will on Unknown',
  ['Posted Numquam fu from lat: -44.8536 lon: 148.99115',
   'Posted Laboriosam from lat: -80.42673 lon: -158.10129',
   'Posted Consectetu from lat: -70.57578 lon: -113.24809',
   'Posted Et esse ni from lat: -41.59669 lon: 76.07817',
   'Posted Totam est  from lat: 10.97619 lon: 169.78154',
   'Posted Et vel exp from lat: 4.52976 lon: -2.34918',
   'Posted Est sunt r from lat: 37.36874 lon: -29.10476',
   'Posted Optio non  from lat: 28.49852 lon: 123.30139',
   'Posted Quia in se from lat: 54.48167 lon: -114.4119',
   'Posted Inventore  from lat: 76.3897 lon: -99.48951',
   'Posted Autem rati from lat: -43.75018 lon: -144.92193']),
```

```
src = "/user/student/authors/"
authorNameRDD = sc \
    .textFile(src) \
    .map(lambda line: (int(line.split(",")[0]), (line.split(",")[1],line.split(",")[2])))
authorNameRDD.take(5)
```

```
[(1, ('Walton', 'Adams')),
 (2, ('Marietta', 'Walsh')),
 (3, ('Lily', 'Wintheiser')),
 (4, ('Estevan', 'Gleason')),
 (5, ('Thaddeus', 'Rowe'))]
```

```
src = "/user/student/posts/"
postsRDD = sc \
    .textFile(src) \
    .map(lambda line: (line.split(",")[0],
                       (line.split(",")[1],line.split(",")[2][0:10])))
postsRDD.take(5)
```

```
[('1', ('1', 'Cupiditate')),
 ('2', ('2', 'Excepturi ')),
 ('3', ('3', 'Enim rerum')),
 ('4', ('4', 'Labore ips')),
 ('5', ('5', 'Placeat ex'))]
```

```
import json

def setPhoneName(s):
    if s == "": return "Unknown"
    elif "," in s: return s.replace(",", " or")
    else: return s

src = "/user/student/author_phone.json"
phoneRDD = sc \
    .wholeTextFiles(src) \
    .flatMap(lambda tup: json.loads(tup[1])) \
    .map(lambda kv: (kv.get("author_id", None), kv.get("phone_model", None))) \
    .map(lambda kv: (kv[0], setPhoneName(kv[1])))
phoneRDD.take(20)
```

```
[(1, 'Samsung Galaxy s21 Ultra'),
 (2, 'Samsung Galaxy A52s'),
 (3, 'Samsung Galaxy Z Fold3'),
 (4, 'Apple iPhone 7'),
 (5, 'Apple iPhone 11'),
 (6, 'Samsung Galaxy A12'),
```

```
import xml.etree.ElementTree as ET
def getPosts(s):
    posts = ET.fromstring(s)
    return posts.iter("record")

def getPostID(elem):
    return elem.find("post_id").text

def getPostLocation(elem):
    return elem.find("location").text
```

```
latlongRDD = sc \
    .wholeTextFiles("post_records/*") \
    .flatMap(lambda xmls: getPosts(xmls[1])) \
    .map(lambda element: (getPostID(element),
                          tuple(getPostLocation(element).split(","))))
latlongRDD.take(5)
```

```
[('1', ('-78.67343', ' 146.15251')),
 ('2', ('-24.8449', ' 71.73862')),
 ('3', ('33.15167', ' 90.2584')),
 ('4', ('78.53576', ' -113.2306')),
 ('5', ('37.09904', ' 141.78509'))]
```

```
authorPhoneRDD = authorNameRDD.join(phoneRDD)
authorPhoneRDD.take(5)
```

```
[(5, (('Thaddeus', 'Rowe'), 'Apple iPhone 11')),
 (10, (('Dewitt', 'Smitham'), 'Unknown')),
 (15, (('Brenda', 'Mayer'), 'Apple iPhone 8')),
 (20, (('Wilfredo', 'Yundt'), 'Samsung Galaxy A12 or Apple iPhone 6')),
 (25, (('Westley', 'Rempel'), 'Unknown'))]
```

```
authorPhoneRDD.count()
```

```
10000
```

```
authorNamePhoneRDD = authorPhoneRDD \
    .map(lambda tup: (tup[0], [tup[1][0][0], tup[1][0][1], tup[1][1]]))
authorNamePhoneRDD.take(5)
```

```
[(5, ['Thaddeus', 'Rowe', 'Apple iPhone 11']),
 (10, ['Dewitt', 'Smitham', 'Unknown']),
 (15, ['Brenda', 'Mayer', 'Apple iPhone 8']),
 (20, ['Wilfredo', 'Yundt', 'Samsung Galaxy A12 or Apple iPhone 6']),
 (25, ['Westley', 'Rempel', 'Unknown'])]
```

```
postLocationRDD = postsRDD.join(latlongRDD)
postLocationRDD.take(5)
```

```
[('4', (('4', 'Labore ips'), ('78.53576', ' -113.2306'))),
 ('16', (('16', 'Eius ea ha'), ('41.55404', ' 7.26911'))),
 ('20', (('20', 'Facere atq'), ('71.33242', ' 7.33176'))),
 ('22', (('22', 'Sint iusto'), ('2.78725', ' 124.55732'))),
 ('24', (('24', 'Dolorem ve'), ('56.27864', ' -115.38594')))]
```

```
postLocationRDD.count()
```

```
110000
```

```
authorPostLocationRDD = postLocationRDD \
    .map(lambda tup: (int(tup[1][0][0]), [tup[1][0][1], tup[1][1][0], tup[1][1][1]]))
authorPostLocationRDD.take(5)
```

```
[(4, ['Labore ips', '78.53576', ' -113.2306']),
 (16, ['Eius ea ha', '41.55404', ' 7.26911']),
 (20, ['Facere atq', '71.33242', ' 7.33176']),
 (22, ['Sint iusto', '2.78725', ' 124.55732']),
 (24, ['Dolorem ve', '56.27864', ' -115.38594'])]
```

```
nameTitlePhoneLocRDD = authorNamePhoneRDD.join(authorPostLocationRDD) \
    .map(lambda tup:
         (tup[1][0][0]+" "+tup[1][0][1]+" on "+tup[1][0][2],
          "Posted "+tup[1][1][0]+" from lat: "+tup[1][1][1]+" lon:"+tup[1][1][2]))
nameTitlePhoneLocRDD.take(5)
```

```
[('Brigitte Shanahan on Unknown',
  'Posted Et sit har from lat: 70.80241 lon: 32.37947'),
 ('Brigitte Shanahan on Unknown',
  'Posted Voluptates from lat: -64.13821 lon: -129.31313'),
 ('Brigitte Shanahan on Unknown',
  'Posted Vel debiti from lat: -15.29791 lon: -110.17868'),
 ('Brigitte Shanahan on Unknown',
  'Posted Debitis am from lat: 30.79221 lon: 73.33461'),
 ('Brigitte Shanahan on Unknown',
  'Posted Possimus n from lat: 56.51724 lon: 19.96479')]
```

```
zeroValue = []

def seqOp(accumulator, element):
    accumulator.append(element)
    return accumulator

def combOp(accumulator1, accumulator2):
    return accumulator1 + accumulator2

finalRDD = nameTitlePhoneLocRDD \
    .aggregateByKey(zeroValue, seqOp, combOp)

finalRDD.take(5)
```

```
[('Laurianne Jerde on Samsung Galaxy S21',
  ['Posted Id repudia from lat: -70.76073 lon: 150.41541',
   'Posted Eos nihil  from lat: -60.27312 lon: 0.28329',
   'Posted Neque nam  from lat: -15.1393 lon: -153.41288',
   'Posted Quam exped from lat: 59.64538 lon: 162.21816',
   'Posted Vero enim  from lat: 75.63085 lon: -179.84681',
   'Posted Explicabo  from lat: 28.53925 lon: 39.01707',
   'Posted Quis corru from lat: 18.68474 lon: -134.40318',
   'Posted Repellat i from lat: -21.51617 lon: -178.06651',
   'Posted Magni blan from lat: -63.58662 lon: -126.2323',
   'Posted Qui non as from lat: 59.41561 lon: 140.52858',
   'Posted Est at exc from lat: -19.77205 lon: -22.35178']),
 ('Theodora Nicolas on Unknown',
  ['Posted Id odio di from lat: 60.95745 lon: 165.47261',
   'Posted Vel consec from lat: -0.87067 lon: 98.3268',
   'Posted Ea ea corr from lat: 24.36141 lon: -16.24',
   'Posted Dolor dolo from lat: -10.56048 lon: -177.68041',
   'Posted Delectus c from lat: -19.3932 lon: -116.58581',
   'Posted Dicta nesc from lat: 6.52134 lon: 74.41023',
   'Posted Tempora qu from lat: 65.80853 lon: -96.85641',
   'Posted Quis verit from lat: 12.82261 lon: 33.00807',
   'Posted Ipsam quae from lat: -8.92964 lon: -108.6568',
   'Posted Accusantiu from lat: 30.05355 lon: -83.10026',
   'Posted Iure velit from lat: -67.15977 lon: 16.55862']),
```

Lab 6: Working with the DataFrame API

In this lab, we will use the PySpark shell to explore working with PySpark DataFrame API. Create DataFrames from various data sources and perform basic transformations.

1. Creating a DataFrame from various data sources

   1.1 Create a DataFrame from JSON files

      1.1.1 Navigate to /home/student/Data directory

      1.1.2 Copy people.json file to the HDFS home directory

      1.1.3 Use Hue or the HDFS command line command to verify that people.json has been copied to HDFS

      1.1.4 From Jupyter, use the spark.read.json(<file_path>) command to create a dataframe from people.json

      1.1.5 Print the schema using the .printSchema() action.

      1.1.6 Print out the contents of the dataframe using .show() action.

```
jsonDF = spark.read.json("people.json")
jsonDF.printSchema()
jsonDF.show(5)
```

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

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

   1.2 Filter the dataframe for persons who are older than 20

      1.2.1 Use where(<condition>) transformation with "age >=30" as the condition

      1.2.2 Print the schema using the .printSchema() action.

      1.2.3 Print out the contents of the dataframe using .show() action.

```
olderDF = jsonDF.where("age >= 30")
olderDF.printSchema()
olderDF.show(5)
```

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

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

1.3 Create a new dataframe that only contains the name columns.

1.3.1 Use the select() transformation

1.3.2 Print the schema of the new dataframe

1.3.3 Use show(5) to verify the output

```
nameDF = olderDF.select("name")
nameDF.printSchema()
nameDF.show(5)
```

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

+----+
|name|
+----+
|Andy|
+----+
```

1.4 Create dataframe from CSV file

1.4.1 Copy the people.csv file in /home/student/Data directory to the HDFS home directory

1.4.2 Verify the file has been copied

1.4.3 Use spark.read.csv(<path_to_file>) to create a dataframe from a CSV file.

```
workerDF = spark.read.csv("people.csv")
workerDF.printSchema()
workerDF.show()
```

The output is not quite what was expected. The dataframe consists of a single column named _c0 of type string. Look at the underlying data and fix the problem.

```
workerDF = spark.read.csv("people.csv")
workerDF.printSchema()
workerDF.show()
```

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

+-----------------+
|              _c0|
+-----------------+
|      name;age;job|
|Jorge;30;Developer|
|  Bob;32;Developer|
+-----------------+
```

There are several problems. The first is that Spark has gobbled all the data into a single column. Look carefully at the data. Notice that the delimiter is a semi-colon (;). Spark's default delimiter for CSV files is a comma (,). An option must be given to specify the non-default delimiter. This can be set using the "sep" option.

The next problem is the column names. Currently there is a _c0 for the single column name. Spark defaults to _c0, _c1, etc when it does not know the column names. However, look carefully. The first row contains all the column names. Set the "header" option to "true" to let Spark know that the first row is a header.

1.4.4 Fix the two problems and print the schema and show the contents to verify the problem has been fixed.

```
workerDF = spark.read.option("sep",";").option("header", "true").csv("people.csv")
workerDF.printSchema()
workerDF.show()
```

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

+-----+---+---------+
| name|age|      job|
+-----+---+---------+
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+---+---------+
```

1.5 Create dataframe from Parquet file

1.5.1 Navigate to /home/student/Data directory

1.5.2 Use parquet-tools to inspect the schema of users.parquet file

```
parquet-tools inspect users.parquet
```

```
[student@localhost Data]$ parquet-tools inspect users.parquet

############ file meta data ############
created_by: parquet-mr version 1.4.3
num_columns: 3
num_rows: 2
num_row_groups: 1
format_version: 1.0
serialized_size: 490


############ Columns ############
name
favorite_color
array

############ Column(name) ############
name: name
path: name
max_definition_level: 0
max_repetition_level: 0
physical_type: BYTE_ARRAY
logical_type: String
converted_type (legacy): UTF8

############ Column(favorite_color) ############
name: favorite_color
path: favorite_color
max_definition_level: 1
max_repetition_level: 0
physical_type: BYTE_ARRAY
logical_type: String
converted_type (legacy): UTF8

############ Column(array) ############
name: array
path: favorite_numbers.array
```

1.5.3 Use parquet-tools show option to view the contents of users.parquet

```
parquet-tools show users.parquet | more
```

```
[student@localhost Data]$ parquet-tools show users.parquet | more
+--------+------------------+--------------------+
| name   | favorite_color   | favorite_numbers   |
|--------+------------------+--------------------|
| Alyssa |                  | [ 3.  9. 15. 20.]  |
| Ben    | red              | []                 |
+--------+------------------+--------------------+
```

1.5.4 Copy users.parquet to the HDFS home directory

1.5.5 Use Hue or hdfs command line to verify the file has been copied

1.5.6 Create a dataframe for users.parquet:
Spark's default format for dataframes is parquet. In addition, because parquet files have their schema embedded in the binary file, there isn't any need for options. Simply use spark.read.load(<path_to_file>)

```
usersDF = spark.read.load("users.parquet")
usersDF.printSchema()
usersDF.show()
```

```
root
 |-- name: string (nullable = true)
 |-- favorite_color: string (nullable = true)
 |-- favorite_numbers: array (nullable = true)
 |    |-- element: integer (containsNull = true)

+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

1.6 Creating a dataframe from an Avro file
While the Avro file format has been supported since Spark 2.4, the JAR file must be made available to Spark. It is possible to use --package org.apache.spark:spark-avro_2.12:3.1.2 when starting the PySpark shell. However, this method is often prone to connection problems. An easier solution is to simply download the org.apache.spark:spark-avro_2.12:3.1.2.jar file and place it in the Spark JAR classpath.

1.6.1 Navigate to /home/student/Data

1.6.2 Copy spark-avro_2.12-3.1.2.jar file to $SPARK_HOME/jars

```
cp spark-avro_2.12-3.1.2.jar $SPARK_HOME/jars/.
```

1.6.3 Restart PySpark and Jupyter

1.6.4 Copy users.avro to the HDFS home directory

1.6.5 Use Hue or hdfs command line to verify the file has been copied

1.6.6 Create a dataframe for users.avro using

```
spark.read.format("avro").load(<path_to_file>)
```

1.6.7 Verify the schema

1.6.8 Verify the content

```
avroDF = spark.read.format("avro").load("users.avro")
avroDF.printSchema()
avroDF.show()
```

```
root
 |-- name: string (nullable = true)
 |-- favorite_color: string (nullable = true)
 |-- favorite_numbers: array (nullable = true)
 |    |-- element: integer (containsNull = true)

+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

2. Saving DataFrames

2.1 Save the Users data in Avro format to JSON format

2.1.1 Use spark.write.option(<options>).<format_shortcut>(<path>) to save a dataframe in the desired format using the indicated options

```
avroDF.write.json("/user/student/users_json")
```

2.1.2 Use HDFS command line or Hue to verify that the data has been properly saved and in the correct format

/ user / student / users_json /

part-00000-71a3441c-1761-484e-be94-f050606e13da-c000.json

```
{"name":"Alyssa","favorite_numbers":[3,9,15,20]}
{"name":"Ben","favorite_color":"red","favorite_numbers":[]}
```

2.2 Save the dataframe created from people.json as a CSV file

2.2.1 Save jsonDF created in previous step as a CSV file format

2.2.2 Set the delimiter to "|" using .option("sep", <delimiter>)

2.2.3 Include a header row with the column names using .option("header", "true")

2.2.4 Save the dataframe in /user/student/people_csv

2.2.5 Use HDFS command line or Hue to verify the data has been saved and in the correct format.

/ user / student / people_csv /
part-00000-cb3387aa-e6bb-48c1-8393-baf53d4dbfdb-c000.csv

```
age|name
""|Michael
30|Andy
19|Justin
```

2.3 Try saving avroDF as a CSV file. Choose a delimiter of your choice and save to /user/student/users_csv.

2.3.1 What happened? What is the error message received? Not all data formats can be saved in every format. In this case, the Avro file has a column that is an Array (similar to a List in Python). Unfortunately, a CSV format cannot support a complex column such as an Array.

2.4 Save workerDF in CSV format.

2.4.1 Save the CSV file to /user/student/workers

2.4.2 Set the option to not include a header row. In other words, set the header option to false.

2.4.3 Verify that the file has been created as expected

/ user / student / workers /
part-00000-3f39dc36-d1ef-4260-93dc-f03a1f1fa12b-c000.csv

```
Jorge,30,Developer
Bob,32,Developer
```

3. DataFrame write modes

3.1 Create a new dataframe from a collection

3.1.1 Create the following List and name it new_workers

```
new_workers = [("Henry", '18', "Mail Clerk"),
               ("Sharon", '24', "Marketing"),
               ("Shaun", '32', "Attorney")]
```

3.1.2 Use SparkSession.createDataFrame(<collection>) to create a new dataframe from the new_workers list

```
newWorkersDF = spark.createDataFrame(new_workers)
```

3.2 Save newWorkersDF to /user/student/workers in CSV format

3.2.1 First, try saving using the same syntax as before.

```
newWorkersDF.write \
    .option("header", "false") \
    .csv("/user/student/workers")
```

An ERROR is raised. What is the error message? Why did the error message occur?

```
AnalysisException: path hdfs://localhost:9000/user/student/workers already exists.
```

We previously saved workerDF to the designated directory in step 2.4 above. It is not possible to simply save to the same directory. Spark default setting is to raise an ERROR when attempting to write to an existing directory. This is a good think, since we don't want to accidently overwrite some previously saved work, accidently.

3.2.2 Use the .mode(<write_mode>) method to change the write mode to append instead. Use the following command instead, this time:

3.2.3 newWorkersDF.write \

3.2.4 .option("header", "false") \

3.2.5 .mode("append") \

3.2.6 .csv("/user/student/workers")

3.2.7 Verify using Hue that the additional worker data has been appended to the designated directory.

3.2.8 Finally, create a dataframe by reading the CSV file in /user/student/workers and use show to view its content. Verify that the additional workers have been appended.

```
spark.read.csv("/user/student/workers").show()
```

```
+------+---+----------+
|   _c0|_c1|       _c2|
+------+---+----------+
| Jorge| 30| Developer|
|   Bob| 32| Developer|
| Henry| 18|Mail Clerk|
|Sharon| 24| Marketing|
| Shaun| 32|  Attorney|
+------+---+----------+
```

The new workers have been added but the column names have been lost. This is because we explicitly set the header option to false when the data was saved. In another lab, we will learn how to explicitly create schema information and apply it to dataframes.

Lab 7: Working with Hive from Spark

In this lab, we will use Spark to access Hive tables, modify them and save them as both managed and external Hive tables.

1. Create a dataframe from Hive table

   1.1 Check to make sure authors table exists in Hive. If not, import from MySQL.

      1.1.1 In a previous lab, the authors table in MySQL was imported to Hive in mydb database. Make sure mydb database exists and authors table exists: Start beeline with the following command and use show databases from within it.

```
$ beeline -u jdbc:hive2://
```

```
0: jdbc:hive2://> show databases;
OK
+----------------+
| database_name  |
+----------------+
| default        |
| mydb           |
+----------------+
```

if the database does not exist, use the following command to create mydb database

```
jdbc:hive2://> CREATE DATABASE mydb;
```

If mydb does not exist, the authors table most likely does not exist, neither. From another terminal, use Sqoop to Import authors table to Hive. Use the following sqoop command.

```
$  sqoop import \
--connect jdbc:mysql://localhost/labs \
--username student --password student \
--fields-terminated-by '\t' --table authors \
--hive-import --hive-database 'mydb' \
--hive-table 'authors' --split-by id
```

Sqoop might give an error if /user/student/authors directory already exists from a previous lab. Sqoop imports and stages the authors table in the user's home directory before calling the Hive

engine to create and load the data. If this is the case, rename /user/student/authors to /user/student/my_authors and call the sqoop command again. To rename a HDFS directory, use the hdfs dfs -mv <original_name> <new_name> command

```
$ hdfs dfs -mv authors my_authors
```

1.1.2 Check to make sure authors table has been properly imported, either from a previous lab or from the previous step above. Run the following command from beeline to check.

```
jdbc:hive2://> use mydb;
jdbc:hive2://> show tables;
```

```
0: jdbc:hive2://> use mydb;
OK
No rows affected (0.261 seconds)
0: jdbc:hive2://> show tables;
OK
+-----------+
| tab_name  |
+-----------+
| authors   |
+-----------+
```

1.2 From Jupyter, read the authors Hive table

1.2.1 Use spark.read.table("<database_name>.<table_name>") command to read the table and save to authorsDF.

1.2.2 Print schema of authorsDF using the printSchema() dataframe action

```
authorDF = spark.read.table("mydb.authors")
authorDF.printSchema()
authorDF.show(5)
```

```
root
 |-- id: integer (nullable = true)
 |-- first_name: string (nullable = true)
 |-- last_name: string (nullable = true)
 |-- email: string (nullable = true)
 |-- birthdate: string (nullable = true)
 |-- added: string (nullable = true)
```

1.2.3 Show the first 5 rows of authorsDF with .show(5)

```
+---+----------+---------+-------------------+----------+-------------------+
| id|first_name| last_name|              email| birthdate|              added|
+---+----------+---------+-------------------+----------+-------------------+
|  1|    Walton|    Adams|barmstrong@exampl...|1989-03-01|1997-01-02 04:18:...|
|  2|  Marietta|    Walsh|hand.stella@examp...|2018-05-30|2010-08-26 18:20:...|
|  3|      Lily|Wintheiser|darren.blanda@exa...|1981-08-21|1973-06-11 07:28:...|
|  4|   Estevan|  Gleason|shanahan.aliyah@e...|2013-07-17|1995-01-29 16:08:...|
|  5|  Thaddeus|     Rowe|bednar.robin@exam...|2019-02-26|2017-01-05 04:13:...|
+---+----------+---------+-------------------+----------+-------------------+
```

2. Create Hive managed table from Spark

   2.1 Starting from authorDF, create a new dataframe with only the "id", "email", and "birthdate"

      2.1.1 Use select("<column_name1>","<column_name2>",. . .) and name the new dataframe, bdayDF.

      2.1.2 Print the schema and show the first 5 rows to confirm the transformation. By default, show() prints 20 rows and truncates columns. To control number of rows printed and to not truncate, use the following parameters: show(<num_rows>, <truncate = True/False>)

```
bdayDF = authorDF.select("id", "email", "birthdate")
bdayDF.printSchema()
```

```
root
 |-- id: integer (nullable = true)
 |-- email: string (nullable = true)
 |-- birthdate: string (nullable = true)
```

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

```
+---+--------------------------+----------+
|id |email                     |birthdate |
+---+--------------------------+----------+
|1  |barmstrong@example.com    |1989-03-01|
|2  |hand.stella@example.net   |2018-05-30|
|3  |darren.blanda@example.org |1981-08-21|
|4  |shanahan.aliyah@example.net|2013-07-17|
|5  |bednar.robin@example.net  |2019-02-26|
+---+--------------------------+----------+
only showing top 5 rows
```

   2.2 Save the Hive table as a Managed table

      2.2.1 Use spark.write.saveAsTable(<db_name.table_name>) to save bdayDF as a new managed Hive table

```
bdayDF.write.saveAsTable("mydb.author_bday")
```

2.3 Verify author_bday Hive table

2.3.1 Return to beeline or start beeline again

2.3.2 Navigate to mydb database and print all tables in mydb database

```
jdbc:hive2://> USE mydb;
jdbc:hive2://> SHOW TABLES;
```

2.3.3 Execute the following command to view the first 5 rows of author_bday

```
jdbc:hive2://> SELECT * FROM author_bday LIMIT 5;
```

```
0: jdbc:hive2://> SELECT * FROM author_bday LIMIT 5;
OK
+-----------------+-------------------------------+------------------------+
| author_bday.id  |      author_bday.email        | author_bday.birthdate  |
+-----------------+-------------------------------+------------------------+
| 1               | barmstrong@example.com        | 1989-03-01             |
| 2               | hand.stella@example.net       | 2018-05-30             |
| 3               | darren.blanda@example.org     | 1981-08-21             |
| 4               | shanahan.aliyah@example.net   | 2013-07-17             |
| 5               | bednar.robin@example.net      | 2019-02-26             |
+-----------------+-------------------------------+------------------------+
```

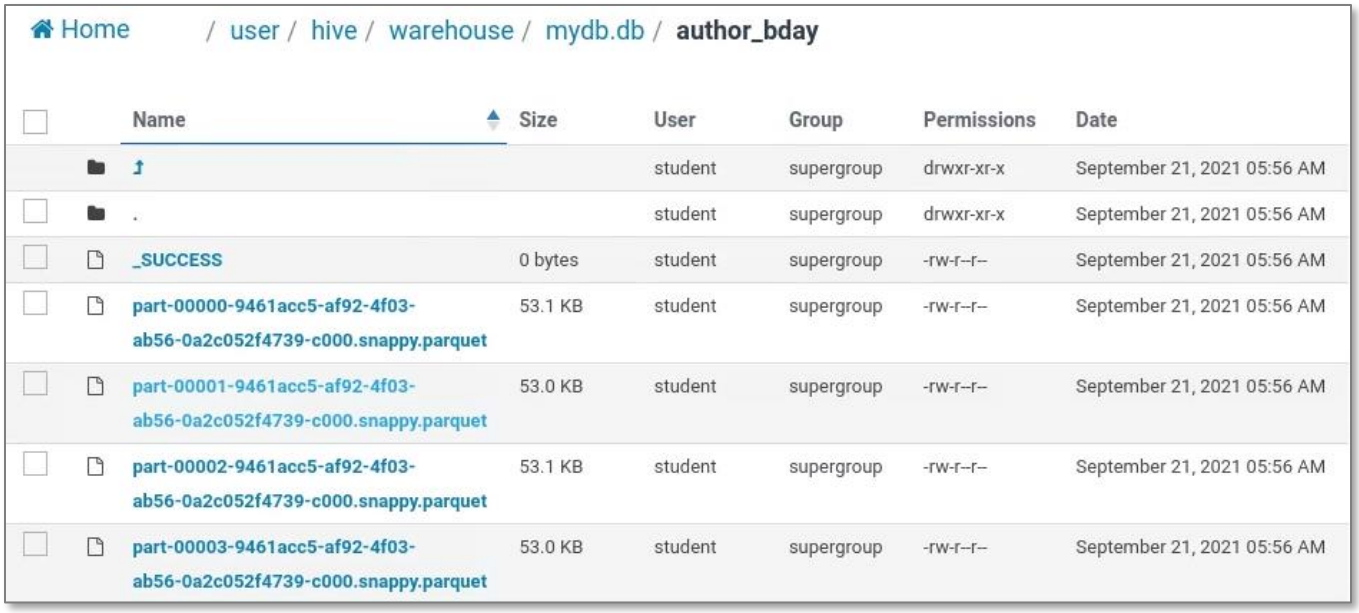
2.3.4 Use the describe formatted Hive command to review the metadata for author_bday table

```
jdbc:hive2://> DESC FORMATTED author_bday;
```

As you scroll through the output, find the section that shows the location of the data stored and the table type. Notice that author_bday is a managed hive table, and the data is stored in the Hive warehouse directory as expected.

```
| Location:                     | hdfs://localhost:9000/user/hive/warehouse/mydb
.db/author_bday | NULL                                                        |
| Table Type:                   | MANAGED_TABLE
```

2.3.5 Use Hue to navigate to the location where the data has been stored. Verify the location and content.



Notice that the data has been saved as a parquet file. Spark's default format for saving a dataframe is Parquet format

2.3.6 Use hdfs dfs -get subcommand to copy part-00000-xxxxx file to local disk. Use parquet-tools to view the schema and content of the parquet file. To make getting the file easier, use the * wildcard to match the ending after the initial part-00000.

```
$ hdfs dfs -ls /user/hive/warehouse/mydb.db/author_bday
$ hdfs dfs -get /user/hive/warehouse/mydb.db/author_bday/part-00000*
$ parquet-tools inspect part-xxxxx
$ parquet-tools show part-xxxxx
```

3. Create Hive External table from Spark

3.1 Create a new dataframe with only the "id", "first_name", and "last_name" columns

3.2 Use printSchema() and show() to verify the new dataframe

```
nameDF.printSchema()
nameDF.show(5)
```

```
root
 |-- id: integer (nullable = true)
 |-- first_name: string (nullable = true)
 |-- last_name: string (nullable = true)

+---+----------+----------+
| id|first_name| last_name|
+---+----------+----------+
|  1|    Walton|     Adams|
|  2|  Marietta|     Walsh|
|  3|      Lily|Wintheiser|
|  4|   Estevan|   Gleason|
|  5|  Thaddeus|      Rowe|
+---+----------+----------+
only showing top 5 rows
```

3.3 Save the Hive table as an External table in CSV format, using tab separator, and that includes a header row. Spark saves Hive tables as an External table if an explicit path is provided. Otherwise, the table is saved as a managed table in the Hive warehouse directory.

3.3.1 Use spark.write.format(<format>).option(<options>).saveAsTable(<db>.<table>)

```
nameDF.write \
    .format("csv") \
    .option("path", "/user/student/author_names") \
    .option("sep", "\t") \
    .option("header", "true") \
    .saveAsTable("mydb.author_names")
```

3.4 Use beeline to verify that the new external table has been created. Make sure to use the desc formatted command to verify that the table is an external table and the location of the data is in the expected location. Notice that Hive is keeping a "place holder" under the mydb.db directory since the table belongs to this database.

```
| Location:                     | hdfs://localhost:9000/user/hive/warehouse/mydb
.db/author_names-__PLACEHOLDER__ | NULL
      |
| Table Type:                   | EXTERNAL_TABLE
```

However, as you scroll further down, there is another "path" information where the actual data is stored.

```
|                                  | path
    | hdfs://localhost:9000/user/student/author_names    |
```

3.5 Use Hue to verify that the data has been written in the correct format and in the designated CSV format.

Lab 8: Spark SQL Transformations

In this lab, we will explore various Spark SQL transformations. We will go beyond basic transformations and begin working with Column objects to create Column expression that allows more powerful transformations.

1. Querying DataFrames with Spark SQL transformation

   1.1 Print the first name, email address and birthdate of the 3 oldest authors

   When working with Spark transformations as a novice or beginner, it is much easier to develop the final code by observing the result of a transformation at each step. In Jupyter, set up 2 cells. On the top cell, add and chain transformations. On the second cell, use the show() action to immediately review the result of the transformation.

   1.1.1 Create a new dataframe by reading the authors table in mydb databases in Hive. Use show(5) on the second cell to review.

```
authorDF = spark.read.table("mydb.authors")
```

```
authorDF.show(5)
```

```
+---+----------+----------+--------------------+----------+--------------------+
| id|first_name| last_name|               email| birthdate|               added|
+---+----------+----------+--------------------+----------+--------------------+
|  1|    Walton|     Adams|barmstrong@exampl...|1989-03-01|1997-01-02 04:18:...|
|  2|  Marietta|     Walsh|hand.stella@examp...|2018-05-30|2010-08-26 18:20:...|
|  3|      Lily|Wintheiser|darren.blanda@exa...|1981-08-21|1973-06-11 07:28:...|
|  4|   Estevan|   Gleason|shanahan.aliyah@e...|2013-07-17|1995-01-29 16:08:...|
|  5|  Thaddeus|      Rowe|bednar.robin@exam...|2019-02-26|2017-01-05 04:13:...|
+---+----------+----------+--------------------+----------+--------------------+
only showing top 5 rows
```

   1.1.2 Transform the dataframe by selecting only the first_name, email, and birthdate. Chain this transformation to the end and execute both cells.

   When chaining transformations, PySpark requires a "\" character if the code spills over to a new line. Actually, it is good practice to place one transformation on each line and use the "\" character to separate then into individual lines. This makes the code much more legible and clearer.

```
authorDF = spark.read.table("mydb.authors") \
    .select("first_name", "email", "birthdate")
```

```
authorDF.show(5)
```

```
+----------+--------------------+----------+
|first_name|               email| birthdate|
+----------+--------------------+----------+
|    Walton|barmstrong@exampl...|1989-03-01|
|  Marietta|hand.stella@examp...|2018-05-30|
|      Lily|darren.blanda@exa...|1981-08-21|
|   Estevan|shanahan.aliyah@e...|2013-07-17|
|  Thaddeus|bednar.robin@exam...|2019-02-26|
+----------+--------------------+----------+
only showing top 5 rows
```

1.1.3 Use orderBy(<column name>) on the "birthdate" column. Chain the transformation and execute both cells to view the results

```
authorDF = spark.read.table("mydb.authors") \
    .select("first_name", "email", "birthdate") \
    .orderBy("birthdate")
```

```
authorDF.show(5)
```

```
+----------+--------------------+----------+
|first_name|               email| birthdate|
+----------+--------------------+----------+
|  Roderick|bobby.conroy@exam...|1970-01-01|
|   Jeffery| verla17@example.org|1970-01-02|
|   Mikayla|  xwalsh@example.org|1970-01-05|
|       Lew|  emie19@example.net|1970-01-06|
|   Shannon|michele48@example...|1970-01-10|
+----------+--------------------+----------+
only showing top 5 rows
```

1.1.4 Use limit(<num>) to limit the number of rows to <num> = 3. Chain this transformation and execute both cells to view the results.

```
authorDF = spark.read.table("mydb.authors") \
    .select("first_name", "email", "birthdate") \
    .orderBy("birthdate") \
    .limit(3)
```

```
authorDF.show(5)
```

```
+----------+--------------------+----------+
|first_name|               email| birthdate|
+----------+--------------------+----------+
|  Roderick|bobby.conroy@exam...|1970-01-01|
|   Jeffery| verla17@example.org|1970-01-02|
|   Mikayla|  xwalsh@example.org|1970-01-05|
+----------+--------------------+----------+
```

Notice that although show(5) was executed, only 3 rows are displayed. This is because, limit(3) has reduced the dataframe to 3 rows. There isn't any more rows to print, so even though show(5) was called, Spark returned the maximum number of rows available.

1.2 Print the first name, email address and birthdate of the 3 oldest authors. However, this time modify the transformation from above such that the query selects the 3 oldest authors born after the new millennium. That is, we want to only choose from authors born on or after January 1, 2001.

1.2.1 Use the where(<condition>) to select only authors whose birthdate is greater than or equal to '2001-01-01'. Try using the string "birthdate >= '2001-01-01'" for the condition.

The where() transformation should immediately follow the select transformation from above and before the orderBy transformation. The orderBy transformation is an expensive operation requiring partitions to exchange information in order to globally sort the dataframe. We want to reduce the amount of data that needs to be exchanged before and not after. By reducing the dataset to those born in the new millennium, before orderBy, the amount of data that needs to be exchanged is reduced significantly.

Remove the limit(3) transformation for now. We want to make sure that we view all the authors whose birthdate satisfies the condition, BEFORE limiting the output, in order to verify the filter. An easy way to do this is to add a # in front of it. The # indicates that what follows is a comment. In essence, we are commented out the limit() transformation.

```
authorDF = spark.read.table("mydb.authors") \
    .select("first_name", "email", "birthdate") \
    .where("birthdate >= '2000-01-01'") \
    .orderBy("birthdate") \
#    .limit(3)
```

```
authorDF.show(5)
```

```
+----------+--------------------+----------+
|first_name|               email| birthdate|
+----------+--------------------+----------+
|Earnestine|junius98@example.net|2000-01-01|
|      Zoie|    jorn@example.org|2000-01-04|
|  Izabella|xmacejkovic@examp...|2000-01-07|
|    Maryse|nella.grant@examp...|2000-01-07|
|   Xzavier|josefina.lynch@ex...|2000-01-10|
+----------+--------------------+----------+
only showing top 5 rows
```

1.2.2 Now that the code has been tested and verified, add back the limit(3) transformation to display the 3 oldest authors born after the new millennium.

```
authorDF = spark.read.table("mydb.authors") \
    .select("first_name", "email", "birthdate") \
    .where("birthdate >= '2000-01-01'") \
    .orderBy("birthdate") \
    .limit(3)
```

```
authorDF.show(5)
```

```
+----------+--------------------+----------+
|first_name|               email| birthdate|
+----------+--------------------+----------+
|Earnestine|junius98@example.net|2000-01-01|
|      Zoie|    jorn@example.org|2000-01-04|
|    Maryse|nella.grant@examp...|2000-01-07|
+----------+--------------------+----------+
```

2. Using Column objects and Column expressions

It was possible to perform a fairly simple comparison operation using string notion in the where clause above. However, for more complex operations, column objects and column expressions can deploy a rich set of operators and functions, including the ability to define user-defined functions.

2.1 Create dataframe from sales.csv

2.1.1 Navigate to /home/student/Data local directory

2.1.2 Inspect sales.csv and determine its schema and delimiter

2.1.3 Copy sales.csv to the HDFS home directory

2.1.4 Create a dataframe from sales.csv and name it salesDF. Does the source file have a header row? What is the delimiter? Is it the default "," separator?

2.1.5 Print the schema and show(5) to verify the dataframe has been created

```
root
 |-- Region: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- ItemType: string (nullable = true)
 |-- SalesChannel: string (nullable = true)
 |-- OrderPriority: string (nullable = true)
 |-- UnitsSold: string (nullable = true)
 |-- UnitPrice: string (nullable = true)
 |-- UnitCost: string (nullable = true)
 |-- TotalRevenue: string (nullable = true)
 |-- TotalCost: string (nullable = true)
 |-- TotalProfit: string (nullable = true)

+--------------------+-------+----------+------------+------------+---------+---------+--
------+------------+----------+-----------+
|              Region|Country|  ItemType|SalesChannel|OrderPriority|UnitsSold|UnitPrice|Un
itCost|TotalRevenue| TotalCost|TotalProfit|
+--------------------+-------+----------+------------+------------+---------+---------+--
------+------------+----------+-----------+
|Middle East and N...|  Libya| Cosmetics|     Offline|           M|     8446|    437.2|
263.33|   3692591.2|2224085.18| 1468506.02|
|       North America| Canada|Vegetables|      Online|           M|     3018|   154.06|
90.93|   464953.08| 274426.74|  190526.34|
|Middle East and N...|  Libya| Baby Food|     Offline|           C|     1517|   255.28|
159.42|   387259.76| 241840.14|  145419.62|
|                Asia|  Japan|    Cereal|     Offline|           C|     3322|    205.7|
117.11|    683335.4| 389039.42|  294295.98|
|  Sub-Saharan Africa|   Chad|    Fruits|     Offline|           H|     9845|     9.33|
6.92|    91853.85|   68127.4|   23726.45|
+--------------------+-------+----------+------------+------------+---------+---------+--
------+------------+----------+-----------+
only showing top 5 rows
```

2.2 Using column objects, select just the Country, ItemType, UnitsSold, UnitPrice, and UnitCost

2.2.1 Use salesDF.<column_name> notation to create a column object.

2.2.2 Print the schema and use show to actualize the transformation and to verify

```
root
 |-- Country: string (nullable = true)
 |-- UnitsSold: string (nullable = true)
 |-- UnitPrice: string (nullable = true)
 |-- UnitCost: string (nullable = true)

+-------+---------+---------+--------+
|Country|UnitsSold|UnitPrice|UnitCost|
+-------+---------+---------+--------+
|  Libya|     8446|    437.2|  263.33|
| Canada|     3018|   154.06|   90.93|
|  Libya|     1517|   255.28|  159.42|
|  Japan|     3322|    205.7|  117.11|
|   Chad|     9845|     9.33|    6.92|
+-------+---------+---------+--------+
only showing top 5 rows
```

2.3 Use column objects cast function to change datatype and perform calculation

2.3.1 Column objects may be cast to a different datatype using the cast(<new_type>) function. Use cast to convert UnitsSold and UnitPrice to float and calculate the total amount of the order by multiplying the two values

```
calcRevDF = salesInfoDF \
    .select(salesDF.Country,
            salesDF.UnitsSold.cast("float")*salesDF.UnitPrice.cast("float"))
calcRevDF.printSchema()
calcRevDF.show(5)
```

```
root
 |-- Country: string (nullable = true)
 |-- (CAST(UnitsSold AS FLOAT) * CAST(UnitPrice AS FLOAT)): float (nullable = true)

+-------+-----------------------------------------------------+
|Country|(CAST(UnitsSold AS FLOAT) * CAST(UnitPrice AS FLOAT))|
+-------+-----------------------------------------------------+
|  Libya|                                            3692591.2|
| Canada|                                            464953.06|
|  Libya|                                            387259.75|
|  Japan|                                             683335.4|
|   Chad|                                             91853.85|
+-------+-----------------------------------------------------+
only showing top 5 rows
```

2.3.2 The resulting datatype after performing an operation on column objects is a column object. A column object may be save to a variable just like any other object. This time, create a variable and save the result of the operation on the column objects. Print the variable.

```
calcRevenue = salesDF.UnitsSold.cast("float")*salesDF.UnitPrice.cast("float")
print(calcRevenue)
```

```
Column<'(CAST(UnitsSold AS FLOAT) * CAST(UnitPrice AS FLOAT))'>
```

Notice the output shows that the variable is of type Column with the actual operations as the value.

2.3.3 The output of step 2.3.1 shows a column name that is literally the operation performed to obtain the column result. Use the alias column object method to give it a more appropriate name. Redo callcRevDF. This time use calcRevenue column object that was saved along with alias method to give name the column "Sales_Revenue"

```
calcRevDF = salesInfoDF \
    .select(salesDF.Country,
            calcRevenue.alias("Sales_Revenue"))
calcRevDF.printSchema()
calcRevDF.show(5)
```

```
root
 |-- Country: string (nullable = true)
 |-- Sales_Revenue: float (nullable = true)

+-------+-------------+
|Country|Sales_Revenue|
+-------+-------------+
|  Libya|    3692591.2|
| Canada|    464953.06|
|  Libya|    387259.75|
|  Japan|     683335.4|
|   Chad|     91853.85|
+-------+-------------+
only showing top 5 rows
```

2.4 Notice that there are multiple entries for a country. For example, in above output, there is at least two row items for Libya. This is because, each sale is further classified by ItemType. Redo above calculation but this time, include the ItemType

```
salesDF = spark.read.option("header", "true").csv("sales.csv")
calcRevenue = salesDF.UnitsSold.cast("float")*salesDF.UnitPrice.cast("float")
revItemCountryDF = salesDF \
    .select(salesDF.Country, salesDF.ItemType,
           calcRevenue.alias("Sales_Revenue"))
revItemCountryDF.printSchema()
revItemCountryDF.show(5)
```

```
root
 |-- Country: string (nullable = true)
 |-- ItemType: string (nullable = true)
 |-- Sales_Revenue: float (nullable = true)

+-------+----------+-------------+
|Country|  ItemType|Sales_Revenue|
+-------+----------+-------------+
|  Libya| Cosmetics|    3692591.2|
| Canada|Vegetables|    464953.06|
|  Libya| Baby Food|    387259.75|
|  Japan|    Cereal|     683335.4|
|   Chad|    Fruits|     91853.85|
+-------+----------+-------------+
only showing top 5 rows
```

2.5 Modify revItemCountryDF from above. Use a comparative operator to check if quota has been met

2.5.1 Create a new column that will either be true or false depending on if a quota has been met. The DataFrame API offers the withColumn(<column_name>, <operation to calculate value of column>) method. Set a variable to a quota of 3 million.

```
quota = 3000000
```

2.5.2 Create a column object that inspects the "Sales_Revenue" column from above and checks if the quota has been met

```
quotaMet = (revItemCountryDF.Sales_Revenue > quota)
```

2.5.3 Use withColumn() and create a new column and name it "Quota_Met." Use the quotaMet column object for the operation

```
withColumn("Quota_Met", quotaMet)
```

2.5.4 Print the schema and show(5) to verify the transformations.

```
quota = 3000000
quotaMet = (revItemCountryDF.Sales_Revenue > quota)
revItemCountryQuotaDF = revItemCountryDF \
    .withColumn("Quota_Met", quotaMet)
revItemCountryQuotaDF.printSchema()
revItemCountryQuotaDF.show(5)
```

```
root
 |-- Country: string (nullable = true)
 |-- ItemType: string (nullable = true)
 |-- Sales_Revenue: float (nullable = true)
 |-- Quota_Met: boolean (nullable = true)

+-------+----------+-------------+---------+
|Country|  ItemType|Sales_Revenue|Quota_Met|
+-------+----------+-------------+---------+
|  Libya| Cosmetics|    3692591.2|     true|
| Canada|Vegetables|    464953.06|    false|
|  Libya| Baby Food|    387259.75|    false|
|  Japan|    Cereal|     683335.4|    false|
|   Chad|    Fruits|     91853.85|    false|
+-------+----------+-------------+---------+
only showing top 5 rows
```

2.6 This time, change the definition of what it means for the quota to have been met. The quota will be met if either the calculated amount is greater than the quota or UnitsSold exceeds a set number

2.6.1 Create a new dataframe from sales.csv and select "Country", "ItemType", "UnitsSold" and "UnitPrice"

2.6.2 Set variable amtQuota = 3000000

2.6.3 Set variable cntQuota = 5000

2.6.4 Redo the quotaMet column object. It should now create a new column object after checking for both amtQuota and cntQuota. If either of the quota has been met, return True

2.6.5 Using withColumn, add "Sales_Revenue" column. The column value will be the amount calculated by multiplying "UnitsSold" with "UnitPrice" This time, do not cast to float as we did before.

2.6.6 Using withColumn, add "Quota_met" column. The column value will be based on the modified quotaMet column object

```
# Read the CSV data
salesDF = spark.read.option("header", "true").csv("sales.csv") \
    .select("Country","ItemType", "UnitsSold", "UnitPrice")

# set the quota constants
amtQuota = 3000000
cntQuota = 5000

# Create the conditions
salesRevenue = salesDF.UnitsSold * salesDF.UnitPrice
quotaMet = (salesRevenue > amtQuota) | (salesDF.UnitsSold > cntQuota)

# Create the new dataframe
revItemCountryQuotaDF = salesDF \
    .withColumn("Sales_Revenue", salesRevenue) \
    .withColumn("Quota_Met", quotaMet)
revItemCountryQuotaDF.printSchema()
revItemCountryQuotaDF.show(5)
```

```
root
 |-- Country: string (nullable = true)
 |-- ItemType: string (nullable = true)
 |-- UnitsSold: string (nullable = true)
 |-- UnitPrice: string (nullable = true)
 |-- Sales_Revenue: double (nullable = true)
 |-- Quota_Met: boolean (nullable = true)

+-------+----------+---------+---------+------------------+---------+
|Country|  ItemType|UnitsSold|UnitPrice|     Sales_Revenue|Quota_Met|
+-------+----------+---------+---------+------------------+---------+
|  Libya| Cosmetics|     8446|    437.2|3692591.1999999997|     true|
| Canada|Vegetables|     3018|   154.06|         464953.08|    false|
|  Libya| Baby Food|     1517|   255.28|         387259.76|    false|
|  Japan|    Cereal|     3322|    205.7| 683335.3999999999|    false|
|   Chad|    Fruits|     9845|     9.33|          91853.85|     true|
+-------+----------+---------+---------+------------------+---------+
only showing top 5 rows
```

Notice that Spark was smart enough to know that the salesReveue should be cast to some number. In fact, it has automatically cast it to double datatype. Separating the conditions and saving them to a variable makes the withColumn() transformation much more legible. In PySpark the ("|") character is used to "OR" conditions.

3. Using aggregate functions with column objects

   3.1 List all the ItemTypes sold

      3.1.1 Discover all the ItemTypes by grouping on ItemType. Use groupBy(<column>) to do this.

      3.1.2 Use count() to count number of rows for each ItemType. The output should be similar to below:

```
spark.read.option("header", "true").csv("sales.csv") \
    .select("ItemType") \
    .groupBy("ItemType") \
    .count() \
    .show()
```

```
+---------------+-----+
|       ItemType|count|
+---------------+-----+
|      Baby Food|   87|
|         Cereal|   79|
|           Meat|   78|
|      Household|   77|
|     Vegetables|   97|
|      Beverages|  101|
|Office Supplies|   89|
|      Cosmetics|   75|
|  Personal Care|   87|
|         Fruits|   70|
|         Snacks|   82|
|        Clothes|   78|
+---------------+-----+
```

Can you guess why the code selects just the "ItemType" column before grouping by that column? groupBy is a very expensive operation that requires all partitions to exchange information in order to generate a global grouping. Therefore, it is important to reduce the amount of data that needs to be shuffled around before calling the groupBy transformation. The only required column is "ItemType" itself so it is the only column selected.

3.2 What is the average number of items sold for each ItemType?

3.2.1 From salesDF, select "ItemType" and "UnitsSold". Cast the "UnitsSold" to an integer. This column will have to be a number in order to calculate the average later on.

3.2.2 Use groupBy for each ItemType

3.2.3 Calculate the average using the mean(<column>) transformation.

3.2.4 Print the schema and show() to verify the transformations and the output

```
salesDF = spark.read.option("header", "true").csv("sales.csv")

aggDF = salesDF \
    .select("ItemType", salesDF.UnitsSold.cast("int")) \
    .groupBy("ItemType") \
    .mean("UnitsSold")

aggDF.printSchema()
aggDF.show()
```

```
root
 |-- ItemType: string (nullable = true)
 |-- avg(UnitsSold): double (nullable = true)

+--------------+-----------------+
|      ItemType|   avg(UnitsSold)|
+--------------+-----------------+
|     Baby Food|5018.597701149425|
|        Cereal|4908.215189873417|
|          Meat|5229.679487179487|
|     Household|4818.077922077922|
|    Vegetables|4858.515463917526|
|     Beverages|4999.059405940594|
|Office Supplies|4994.179775280899|
|     Cosmetics|          5680.96|
| Personal Care|5468.091954022989|
|        Fruits|5073.214285714285|
|        Snacks|4818.829268292683|
|       Clothes|4845.974358974359|
+--------------+-----------------+
```

3.3 What is the average sales revenue for each ItemType?

3.3.1 From salesDF, select "ItemType", "UnitPrice" and "UnitsSold".

3.3.2 Using withColumn, create a Sales_Revenue column whose value is UnitPrice * UnitsSold. The result will automatically be cast to double datatype.

3.3.3 Use groupBy for each ItemType

3.3.4 Calculate the average revenue using the mean(<column>) transformation.

3.3.5 Print the schema and show() to verify the transformations and the output

```
salesDF = spark.read.option("header", "true").csv("sales.csv") \
    .select("ItemType", "UnitsSold", "UnitPrice")

salesRevenue = salesDF.UnitsSold * salesDF.UnitPrice

avgSalesRevenueDF = salesDF \
    .withColumn("Sales_Revenue", salesRevenue) \
    .groupBy("ItemType") \
    .mean("Sales_Revenue")

avgSalesRevenueDF.printSchema()
avgSalesRevenueDF.show()
```

```
root
 |-- ItemType: string (nullable = true)
 |-- avg(Sales_Revenue): double (nullable = true)

+---------------+------------------+
|       ItemType|avg(Sales_Revenue)|
+---------------+------------------+
|      Baby Food|1281147.6211494252|
|         Cereal| 1009619.864556962|
|           Meat|2206349.4788461546|
|      Household| 3219776.932987012|
|     Vegetables|  748502.892371134|
|      Beverages|237205.36881188114|
|Office Supplies| 3252259.811460674|
|      Cosmetics|2483715.7120000003|
|  Personal Care| 446907.1554022989|
|         Fruits|  47333.0892857143|
|         Snacks|  735256.969756098|
|        Clothes|  529568.077948718|
+---------------+------------------+
```

4. Creating User-Defined functions

4.1 Our current quotaMet condition does not really provide very good information. It does not consider the underlying ItemType in testing if quota has been met. Modify the condition to reflect the ItemType.

4.1.1 Create a Python dictionary as shown below:

```
cntQuotaDict = {"Baby Food" : 5018,
                "Cereal" :4908,
                "Household" : 5229,
                "Vegetables" : 4818,
                "Beverages" : 4858,
                "Office Supplies" : 4994,
                "Cosmetics" : 5680,
                "Personal Care" : 5468,
```

```
                "Fruits" : 5073,
                "Snacks" : 4818,
                "Clothes" : 4845,}
```

A dictionary in Python is a Key:Pair item. Dictionaries are created using the {item, item,….} operator. The value of a Key:Pair is accessed using the get(<key>) method of a dictionary. This dictionary shows the ItemType as the Key and the average sales count as the Value.

4.1.2 Create a user-defined function (UDF) that takes two parameters. The first parameter is the ItemType. The second parameter is a count of the ItemType. The UDF will reference cntQuotaDict to check whether the count is greater than the count is the dictionary

```
def cntQuota(item, cnt):
    meet_this = cntQuotaDict.get(item, None)
    if meet_this == None: meet_this = 0
    my_cnt = int(cnt)
    return my_cnt > meet_this
```

4.1.3 In order to use a UDF, the function must first be registered. Use the udf function. The return type of the function must also be registered

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


quotaUDF = udf(lambda item, cnt: cntQuota(item, cnt),
BooleanType())
```

4.2 Use the UDF to determine if quota has been met

4.2.1 Create a new dataframe by reading sales.csv and name is quotaDF.

4.2.2 Use select method to include "Country" and "ItemType" columns.

4.2.3 A UDF may be called within a select statement. Continue within the select transformation from above step to include the result of the UDF. The UDF expects the parameters to be passed as Column objects. Use the col method to explicitly create column objects and pass it to the UDF. The UDF requires the ItemType and UnitsSold columns

4.2.4 Use the alias function to rename the result of the UDF to "Met_Quota"

4.2.5 Print schema and show() to verify the transformation and output

```
quotaDF = spark.read.option("header", "true").csv("sales.csv") \
    .select("Country",
            quotaUDF(col("ItemType"), col("UnitsSold")).alias("Met_Quota"))
quotaDF.printSchema()
quotaDF.show()
```

```
root
 |-- Country: string (nullable = true)
 |-- Met_Quota: boolean (nullable = true)

+----------+---------+
|   Country|Met_Quota|
+----------+---------+
|     Libya|     true|
|    Canada|    false|
|     Libya|    false|
|     Japan|    false|
|      Chad|     true|
|   Armenia|     true|
|   Eritrea|    false|
|Montenegro|     true|
|   Jamaica|    false|
|      Fiji|    false|
|      Togo|    false|
|Montenegro|    false|
|    Greece|    false|
|     Sudan|    false|
|  Maldives|     true|
|Montenegro|    false|
|   Estonia|    false|
| Greenland|    false|
|Cape Verde|    false|
|   Senegal|     true|
+----------+---------+
only showing top 20 rows
```

Lab 9: Working with Spark SQL

Spark SQL allows Spark developers to use ISO standard SQL to create queries from DataFrames.

1. Using SparkSession.sql() transformation

   1.1 Create a query using the sql method

      1.1.1 Execute the following code from Jupyter.

```
authorsDF = spark.sql(" SELECT * FROM mydb.authors LIMIT 10 ")
authorsDF.printSchema()
authorsDF.show()
```

As can be seen, spark.sql(<SQL query>) returns a dataframe. The query returns all the columns from authors table in mydb database in Hive.

1.2 Try another more complex SQL query where part of the query includes some string to filter on.

```
authorsDF = spark.sql(""" SELECT first_name, last_name, email
                          FROM mydb.authors
                          WHERE email LIKE '%org'
                          LIMIT 10  """ )
authorsDF.printSchema()
authorsDF.show(truncate=False)
```

This SQL query returns the first_name, last_name and email address of authors whose email address ends with "org." When a query includes strings such as '%org', use triple quotes (""") to surround the SQL query. This makes it easy to include strings within the query without having to escape them.

```
authorsDF = spark.sql(""" SELECT first_name, last_name, email
                          FROM mydb.authors
                          WHERE email LIKE '%org'
                          LIMIT 10  """ )
authorsDF.printSchema()
authorsDF.show(truncate=False)
```

```
root
 |-- first_name: string (nullable = true)
 |-- last_name: string (nullable = true)
 |-- email: string (nullable = true)

+----------+----------+-------------------------+
|first_name|last_name |email                    |
+----------+----------+-------------------------+
|Lily      |Wintheiser|darren.blanda@example.org  |
|Dewitt    |Smitham   |yfadel@example.org         |
|Willard   |Wilderman |virgil45@example.org       |
|Jazlyn    |Osinski   |philip.schaden@example.org |
|America   |Marquardt |ulockman@example.org       |
|Alvis     |Crist     |kennith25@example.org      |
|Thurman   |Lesch     |mauricio.harris@example.org|
|Tad       |Bechtelar |jada.grant@example.org     |
|Dudley    |Kirlin    |lauretta82@example.org     |
|Marcelino |Tremblay  |lenora.fritsch@example.org |
+----------+----------+-------------------------+
```

1.3 Create a query and show all authors who are born in the new millennium.

1.3.1 Select only the first_name, last_name and birthdate

1.3.2 Filter the results for birthdates that fall in the new millennium

1.3.3 Sort the output by birthdate

1.3.4 Print the schema and show() to verify

```
authorsDF = spark.sql(""" SELECT first_name, last_name, birthdate
                          FROM mydb.authors
                          WHERE birthdate >= '2000-01-01'
                          ORDER BY birthdate
                          LIMIT 10  """ )
authorsDF.printSchema()
authorsDF.show(truncate=False)
```

```
root
 |-- first_name: string (nullable = true)
 |-- last_name: string (nullable = true)
 |-- birthdate: string (nullable = true)

+----------+----------+----------+
|first_name|last_name |birthdate |
+----------+----------+----------+
|Earnestine|Thiel     |2000-01-01|
|Zoie      |Gusikowski|2000-01-04|
|Maryse    |West      |2000-01-07|
|Izabella  |Hilll     |2000-01-07|
|Xzavier   |Flatley   |2000-01-10|
|Eva       |Beatty    |2000-01-15|
|Bailee    |Pacocha   |2000-01-17|
|Roma      |Torp      |2000-01-17|
|Gustave   |Rippin    |2000-01-19|
|Mariana   |Cremin    |2000-01-19|
+----------+----------+----------+
```

As can be seen, using SQL directly can be much more convenient for those who are comfortable with the Structured Query Language. However, many developers find combining Column expressions and SQL statements in their code, to be most productive.

2. Using DataFrames as table/view in a query

So far, we have queried existing Hive tables. However, it is possible to use any dataframe within SparkSession.sql(). In order to do so, temporary views must be created.

2.1 Create dataframe from sales.csv in the HDFS home directory

2.1.1 Make sure that sales.csv is in the HDFS home directory. If it is not, navigate to /home/student/Data and copy sales.csv to HDFS.

2.1.2 Use SparkSession.read() to create a dataframe. Include all the columns. Name the dataframe, salesDF.

2.2 Create a temporary view

2.2.1 Use DataFrame.createTempView(<view_name>) or DataFrame.createOrReplaceTempView(<view_name>) to create a temporary view. The DataFrame.createTempView() will generate an error if the temporary view already exists. DataFrame.createOrReplaceTempView() is a better

alternative when developing applications since code is repeatedly executed, and an error will be generated after the first temporary view has been generated when using DataFrame.createTempView().

```
salesDF.createOrReplaceTempView("sales")
```

2.3 Using SparkSession.sql(), query sales table and print the first 10 rows of all columns

2.4 Using SparkSession.sql() to create aggregate queries

2.4.1 Create a SQL query that groups the sales by each Country, ItemType and SalesChannel. Print the sum of TotalRevenue as Revenue_Sum and sum of TotalProfit as Profit_Sum for each group. The output should be ordered by the highest profit first and in descending order.

2.5 Using full address of source file syntax in a query

Sometimes it is desirable to simply give the address of a source file to query it without having to create a dataframe, create a temporary view and then using the temporary view within a query.

2.5.1 Use <format>.`<path to source file>` syntax to directly address a source file inside a SQL query. Create a query to view all the columns of author_phone.json file. The source file is surrounded by back ticks. This is the key below the ESC key and to the left of 1 on the keyboard.

```
spark.sql(""" SELECT * FROM
              json.`/user/student/author_phone.json`
              LIMIT 10 """).show()
```

When using this option, there isn't any opportunity to provide any options nor transform the data before querying it. This syntax should only be used when such operations are unnecessary.

3. Using Spark SQL Magic in Jupyter

3.1 Python provides sparksql-magic which allows using cell magic within Jupyter

3.1.1 Install sparksql-magic with the following command:

```
!pip install sparksql-magic
```

3.1.2 After the installation is complete, load the external library

```
%load_ext sparksql_magic
```

3.1.3 Now use %%sparksql to indicate that we will use sparksql magic on this cell. Follow this with a simple query.

```
%%sparksql
select * from mydb.authors limit 10;
```

id	first_name	last_name	email	birthdate	added
1	Walton	Adams	barmstrong@example.com	1989-03-01	1997-01-02 04:18:41.0
2	Marietta	Walsh	hand.stella@example.net	2018-05-30	2010-08-26 18:20:14.0
3	Lily	Wintheiser	darren.blanda@example.org	1981-08-21	1973-06-11 07:28:12.0
4	Estevan	Gleason	shanahan.aliyah@example.net	2013-07-17	1995-01-29 16:08:31.0
5	Thaddeus	Rowe	bednar.robin@example.net	2019-02-26	2017-01-05 04:13:48.0
6	Cortez	Russel	kennedi.stokes@example.com	1977-06-14	2007-03-02 10:19:18.0
7	Deion	Yundt	lindgren.timmy@example.com	1970-09-02	1988-06-22 16:18:23.0
8	Caterina	Cartwright	lschroeder@example.com	1986-10-04	2001-08-05 00:39:02.0
9	Rylee	Morar	barton.wuckert@example.net	2003-11-23	1991-09-23 17:09:22.0
10	Dewitt	Smitham	yfadel@example.org	1995-12-17	1997-05-09 04:35:50.0

4. Joining DataFrames

4.1 Join dataframes using the .join() transformation

4.1.1 Create authorsDF dataframe from the authors Hive table

4.1.2 Create authorPhoneDF dataframe from author_phone.json file in the HDFS home directory

4.1.3 Use the dataframe.join(<dataframe>, <join condition>, <join type>) transformation to join authorsDF with authorPhoneDF. Use the default inner join. Use column expressions for the <join condition>.

```
joinDF = authorDF \
    .join(authorPhoneDF,
          authorDF.id == authorPhoneDF.author_id)
```

4.1.4 Print the schema and use show() to verify the transformation and output

4.2 Join dataframes using SparkSession.sql() transformation

4.2.1 Create a temporary view for authorsDF. Name the view, author

4.2.2 Create a temporary view for authorPhoneDF. Name the view, author_phone

4.2.3 Use the sql() transformation to join the two tables/views. Select only the first_name, last_name, and phone_model as output columns

4.2.4 Limit the output to 10 rows

```
spark.sql(""" SELECT a.first_name, a.last_name, b.phone_model
              FROM author a
              JOIN author_phone b
              ON a.id = b.author_id
              LIMIT 10 """).show()
```

```
+----------+----------+--------------------+
|first_name| last_name|         phone_model|
+----------+----------+--------------------+
|    Walton|     Adams|Samsung Galaxy s2...|
|  Marietta|     Walsh| Samsung Galaxy A52s|
|      Lily|Wintheiser|Samsung Galaxy Z ...|
|   Estevan|   Gleason|      Apple iPhone 7|
|  Thaddeus|      Rowe|     Apple iPhone 11|
|    Cortez|    Russel|  Samsung Galaxy A12|
|     Deion|     Yundt|Samsung Galaxy A5...|
|  Caterina|Cartwright| Samsung Galaxy A52s|
|     Rylee|     Morar|Samsung Galaxy No...|
|    Dewitt|   Smitham|                    |
+----------+----------+--------------------+
```

4.3 Use sparksql-magic to run a query that prints 10 youngest authors who do not have any phones. We will have a giveaway event for the lucky candidates.

4.3.1 Join author and author_phone. These are the names of the temporary view created earlier

4.3.2 Select first_name, last_name, phone_model and birthdate columns

4.3.3 Filter for records with empty string in phone_model

4.3.4 Order by birthdate in descending order. Since the birthdate is ordered in descending order, the youngest authors will appear first.

4.3.5 Limit the output to 10 rows

```
%%sparksql

SELECT a.first_name, a.last_name, b.phone_model, a.birthdate
    FROM author a
    JOIN author_phone b
    ON a.id = b.author_id
    WHERE b.phone_model = ""
    ORDER BY a.birthdate DESC
    LIMIT 10
```

first_name	last_name	phone_model	birthdate
Kristin	Murray		2019-04-18
Ladarius	Parker		2019-04-17
Elliot	Miller		2019-04-08
Cassandra	Sporer		2019-04-07
Howell	Hane		2019-04-03
Virginia	Murray		2019-04-02
Freda	Pacocha		2019-03-26
Brandon	Donnelly		2019-03-14
Julianne	Schaefer		2019-03-11
Virgie	Wyman		2019-03-10

5. Running DDL and DML commands from Spark

In addition to SQL, developers can run DDL (Dynamic Definition Language) and DML (Data Manipulation Language) to define and create Hive tables as well as modify them.

5.1 Use either spark.sql() or a sparksql-magic cell to execute these DDL commands

5.1.1 Show all the databases in Hive

5.1.2 Navigate to mydb database with the USE <database> command

5.1.3 Show all the tables in mydb

5.1.4 Create a new managed table and name it spark_test. This table will contain two columns: name as string and age as integer. Leave all the rest to the default setting.

5.1.5 Review details of spark_test using DESC FORMATTED command

5.1.6 Use INSERT command to add a few rows to spark_test

5.1.7 Print all the columns from spark_test to verify that the rows have been inserted

```
%%sparksql

SHOW DATABASES;
```

namespace
default
mydb

```
spark.sql("USE mydb")
```

```
DataFrame[]
```

```
spark.sql("SHOW TABLES").show()
```

```
+--------+------------+-----------+
|database|   tableName|isTemporary|
+--------+------------+-----------+
|    mydb| author_bday|      false|
|    mydb|author_names|      false|
|    mydb|     authors|      false|
|    mydb|     titanic|      false|
|        |      author|       true|
|        |author_phone|       true|
|        |       sales|       true|
+--------+------------+-----------+
```

```
%%sparksql

CREATE TABLE spark_test (
    name string,
    age int)
```

```
spark.sql("DESC FORMATTED spark_test").show(20, False)
```

```
+----------------------------+--------------------------------------------------------------
+-------+
|col_name                    |data_type
|comment|
+----------------------------+--------------------------------------------------------------
+-------+
|name                        |string
|null   |
|age                         |int
|null   |
|                            |
|       |
|# Detailed Table Information|
|       |
|Database                    |mydb
|       |
|Table                       |spark_test
|       |
|Owner                       |student
|       |
|Created Time                |Wed Sep 22 22:42:59 KST 2021
|       |
|Last Access                 |UNKNOWN
|       |
|Created By                  |Spark 3.1.2
|       |
|Type                        |MANAGED
|       |
|Provider                    |hive
|       |
|Table Properties            |[transient_lastDdlTime=1632318179]
|       |
|Location                    |hdfs://localhost:9000/user/hive/warehouse/mydb.db/spark_tes
t|       |
|Serde Library               |org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
|       |
|InputFormat                 |org.apache.hadoop.mapred.TextInputFormat
|       |
|OutputFormat                |org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
|       |
|Storage Properties          |[serialization.format=1]
```

```
%%sparksql

INSERT INTO spark_test VALUES ("Henry", 18);
```

```
spark.sql(""" INSERT INTO spark_test VALUES ("Jason", 24) """)
```

```
DataFrame[]
```

```
%%sparksql

SELECT * FROM spark_test;
```

name	age
Henry	18
Jason	24

5.2 Use either spark.sql() or a sparksql-magic cell to execute these DML commands

5.2.1 Add another column: gender of type string

```
spark.sql(""" ALTER TABLE spark_test
              ADD COLUMNS (gender string)  """)
```

5.2.2 Test by adding a few more rows to spark_test, but now including the gender

```
spark.sql(""" ALTER TABLE spark_test
              ADD COLUMNS (gender string)  """)
spark.sql(""" INSERT INTO spark_test
              VALUES ("Shaun", 42, "Male") """)
```

5.2.3 Display the entire spark_test table to verify the results

```
%%sparksql
SELECT * FROM spark_test;
```

name	age	gender
Shaun	42	Male
Sharon	22	Female
Henry	18	null
Jason	24	null

6. Using the Catalog API

6.1 Use the SparkSesstion.catalog to access the Catalog API. Return list of databases, list of tables, list of columns. Change the current database.

6.1.1 Use spark.catalog.listDatabases() to return a list of databases. Use a for loop to print each of the databases

6.1.2 Use spark.sql("show databases").show() to display all the databases. Notice that show() was used to display the dataframe containing the databases.

6.1.3 Use spark.catalog.setCurrentDatabase(<database>) to change the current database. Change the database to mydb

6.1.4 Use spark.sql("use mydb") to change the database to mydb

6.1.5 Use spark.catalog.listTables() to list all the tables in the current database. Use a for loop to print all the tables.

6.1.6 Use spark.catalog.listColumns(<table>) to list all the columns in table <table>

```
for db in spark.catalog.listDatabases():
    print(db)
```

```
Database(name='default', description='Default Hive database', locationUri='hdfs://localhos
t:9000/user/hive/warehouse')
Database(name='mydb', description='', locationUri='hdfs://localhost:9000/user/hive/warehou
se/mydb.db')
```

```
spark.sql("SHOW DATABASES").show()
```

```
+---------+
|namespace|
+---------+
|  default|
|     mydb|
+---------+
```

```
spark.catalog.setCurrentDatabase("mydb")
for tb in spark.catalog.listTables():
    print(tb)
```

```
Table(name='author_bday', database='mydb', description=None, tableType='MANAGED', isTempor
ary=False)
Table(name='author_names', database='mydb', description=None, tableType='EXTERNAL', isTemp
orary=False)
Table(name='authors', database='mydb', description='Imported by sqoop on 2021/09/20 20:44:
32', tableType='MANAGED', isTemporary=False)
Table(name='spark_test', database='mydb', description=None, tableType='MANAGED', isTempora
ry=False)
Table(name='titanic', database='mydb', description=None, tableType='MANAGED', isTemporary=
False)
Table(name='author', database=None, description=None, tableType='TEMPORARY', isTemporary=T
rue)
Table(name='author_phone', database=None, description=None, tableType='TEMPORARY', isTempo
rary=True)
Table(name='sales', database=None, description=None, tableType='TEMPORARY', isTemporary=Tr
ue)
```

```
spark.sql("USE mydb")
spark.sql("SHOW TABLES").show()
```

```
+--------+------------+-----------+
|database|   tableName|isTemporary|
+--------+------------+-----------+
|    mydb| author_bday|      false|
|    mydb|author_names|      false|
|    mydb|     authors|      false|
|    mydb|  spark_test|      false|
|    mydb|     titanic|      false|
|        |      author|       true|
|        |author_phone|       true|
|        |       sales|       true|
+--------+------------+-----------+
```

```
spark.catalog.listColumns("authors")
```

```
[Column(name='id', description=None, dataType='int', nullable=True, isPartition=False, isB
ucket=False),
 Column(name='first_name', description=None, dataType='string', nullable=True, isPartition
=False, isBucket=False),
 Column(name='last_name', description=None, dataType='string', nullable=True, isPartition=
False, isBucket=False),
 Column(name='email', description=None, dataType='string', nullable=True, isPartition=Fals
e, isBucket=False),
 Column(name='birthdate', description=None, dataType='string', nullable=True, isPartition=
False, isBucket=False),
 Column(name='added', description=None, dataType='string', nullable=True, isPartition=Fals
e, isBucket=False)]
```

Lab 10: Transforming RDDs to DataFrames

In this chapter, we will begin with a semi structured data source, transform it to give it structure and convert it to a DataFrame for query operations.

1. Create a dataframe from a JSON data source

   1.1 Import json and create a function that will parse the phone name and set it to a friendly text, parsing multiple phones as necessary.

```
Import json and use the following function to set the phone name
import json

def setPhoneName(s):
    if s == "": return "Unknown"
    elif "," in s: return s.replace(",", " or")
    else: return s
```

   1.1.1 Create a RDD using wholeTextFiles() from /user/student/author_phone.json

   1.1.2 Use flatMap to create separate rows from parsing the json records. Use json.load() on the content of the file to parse the JSON records. Recall that wholeTextFiles creates a tuple of form (<path>,<content>). Pass <content> to json.load

   1.1.3 The json.load() creates a collection of type dictionary. flatMap places each dictionary record as a row item. Use the dictionary.get(<key>) to get the author_id and phone_model values. Create a tuple of form (author_id, phone_model)

   1.1.4 Transform the phone_model part of the key:value tuple by applying the setPhoneName function defined above to the value portion of the pair rdd

   1.2 Creating a Schema

   1.2.1 There are several ways to create a schema definition. A quick and dirty method is to create a string with the following format:
" <col name> <col type>, <col name> <col type>,<col name> <col type>,..."

   1.2.2 Create a string variable and name it schema. Enter the following value for the string:

   1.3 Create a dataframe with the schema

1.3.1 Use spark.createDataFrame(<rdd>, <schema>) to create phoneDF dataframe

1.3.2 Print the schema and show(5, truncate=False) to verify that the dataframe has been created as expected

```
import json

def setPhoneName(s):
    if s == "": return "Unknown"
    elif "," in s: return s.replace(",", " or")
    else: return s

src = "/user/student/author_phone.json"
phoneRDD = sc \
    .wholeTextFiles(src) \
    .flatMap(lambda tup: json.loads(tup[1])) \
    .map(lambda kv: (kv.get("author_id", None), kv.get("phone_model", None))) \
    .map(lambda kv: (kv[0], setPhoneName(kv[1])))
phoneRDD.take(5)
```

```
[(1, 'Samsung Galaxy s21 Ultra'),
 (2, 'Samsung Galaxy A52s'),
 (3, 'Samsung Galaxy Z Fold3'),
 (4, 'Apple iPhone 7'),
 (5, 'Apple iPhone 11')]
```

```
schema = "author_id int, phone_description string"
phoneDF = spark.createDataFrame(phoneRDD,schema)
phoneDF.printSchema()
phoneDF.show(5, truncate=False)
```

```
root
 |-- author_id: integer (nullable = true)
 |-- phone_description: string (nullable = true)

+---------+-----------------------+
|author_id|phone_description      |
+---------+-----------------------+
|1        |Samsung Galaxy s21 Ultra|
|2        |Samsung Galaxy A52s    |
|3        |Samsung Galaxy Z Fold3 |
|4        |Apple iPhone 7         |
|5        |Apple iPhone 11        |
+---------+-----------------------+
only showing top 5 rows
```

2. Create dataframe from XML data source

2.1 Import xml.etree.ElementTree and create 3 functions. getPosts() will parse XML strings and produce a collection of XML records. getPostID() will take an XML record and return the post_id as text. getPostLocataion() will read an XML record and return the location as text. The location will consist of the latitude and longitude string delimited by a comma (","). Use the following functions:

```
import xml.etree.ElementTree as ET
def getPosts(s):
    posts = ET.fromstring(s)
    return posts.iter("record")

def getPostID(elem):
    return elem.find("post_id").text

def getPostLocation(elem):
    return elem.find("location").text
```

2.2 Use wholeTextFiles() to read all the XML files in /user/student/post_records/ directory.

2.3 Use flatMap to create a row for each XML record. Pass the <content> portion of the (<path>, <content>) tuple resulting from wholeTextFiles to getPosts(<content>). Recall that getPosts returns a collection of XML records. flatMap() then breaks the collection up into individual items and creates a new row for each XML record.

2.4 Use the map() transformation to create a List of form [<post_id>, [<latitude>, <longitude>]]. Use getPostID for the <post_id>. Use getPostLocation to return a string delimited by a comma (","). Use the String.split(<delimiter>) function to parse the string into a 2 item List of latitude and longitude.

2.5 Use the map() transformation to break the nested List from above to a simple List consisting of [<post_id>, <latitude>, <longitude>]. Use the float() function to cast the string latitude and longitude to a float datatype.

2.6 Create a schema using StructType() and StructField().

```
from pyspark.sql.types import *
schema = StructType([
            StructField("post_id", StringType(), True),
            StructField("lat", DoubleType(), True),
            StructField("lon", DoubleType(), True)])
```

2.1 Create a dataframe with the schema

2.1.1 Use spark.createDataFrame(<rdd>, <schema>) to create latlonDF dataframe

2.1.2 Print the schema and show(5, truncate=False) to verify that the dataframe has been created as expected

2.2 Save latlonDF dataframe as a Hive managed table under mydb database. Name the table post_latlon

2.3 Use either the Catalog API or SparkSession.sql() to verify that the table has been created

```
import xml.etree.ElementTree as ET
def getPosts(s):
    posts = ET.fromstring(s)
    return posts.iter("record")

def getPostID(elem):
    return elem.find("post_id").text

def getPostLocation(elem):
    return elem.find("location").text
```

```
latlongRDD = sc \
    .wholeTextFiles("post_records/*") \
    .flatMap(lambda xmls: getPosts(xmls[1])) \
    .map(lambda element: [getPostID(element),
                          getPostLocation(element).split(",")]) \
    .map(lambda coll: [coll[0], float(coll[1][0]), float(coll[1][1])])
latlongRDD.take(5)
```

```
[['1', -78.67343, 146.15251],
 ['2', -24.8449, 71.73862],
 ['3', 33.15167, 90.2584],
 ['4', 78.53576, -113.2306],
 ['5', 37.09904, 141.78509]]
```

```
from pyspark.sql.types import *
schema = StructType([
            StructField("post_id", StringType(), True),
            StructField("lat", DoubleType(), True),
            StructField("lon", DoubleType(), True)])


latlonDF = spark.createDataFrame(latlongRDD, schema)
latlonDF.printSchema()
latlonDF.show(5, truncate=False)
```

```
root
 |-- post_id: string (nullable = true)
 |-- lat: double (nullable = true)
 |-- lon: double (nullable = true)

+-------+---------+---------+
|post_id|lat      |lon      |
+-------+---------+---------+
|1      |-78.67343|146.15251|
|2      |-24.8449 |71.73862 |
|3      |33.15167 |90.2584  |
|4      |78.53576 |-113.2306|
|5      |37.09904 |141.78509|
+-------+---------+---------+
only showing top 5 rows
```

```
latlonDF.write.saveAsTable("mydb.posts_latlon")
```

```
spark.sql(" USE mydb")
spark.sql(" SHOW TABLES").show()
spark.sql(" DESC posts_latlon").show()
```

```
+--------+------------+-----------+
|database|   tableName|isTemporary|
+--------+------------+-----------+
|    mydb| author_bday|      false|
|    mydb|author_names|      false|
|    mydb|     authors|      false|
|    mydb|posts_latlon|      false|
|    mydb|  spark_test|      false|
|    mydb|     titanic|      false|
+--------+------------+-----------+

+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
| post_id|   string|   null|
|     lat|   double|   null|
|     lon|   double|   null|
+--------+---------+-------+
```

Lab 11: Working with the DStream API

In this lab, we will capture unstructured streaming data using the DStream API.

1. Simulate a streaming data source

   1.1 Navigate to /home/student/Scripts

   1.2 Execute streams.sh shell script to begin simulating a streaming data source on the Linux socket at port 44444

```
$ /home/student/Scripts/stream_alice.sh
```

```
[student@localhost Scripts]$ /home/student/Scripts/stream.sh
Waiting for connection on localhost : 44444
```

2. On Jupyter, create a StreamingContext

   2.1 Start pyspark in local mode with at least two threads

```
$ pyspark --master 'local[2]'
```

   2.2 Create SparkStreamingContext:
   While the Spark Shell provides pre-instantiated SparkContext and SparkSession objects, the StreamingContext object must be manually instantiated in the shell

      2.2.1 Import the StreamingContext library

```
from pyspark.streaming import StreamingContext
```

      2.2.2 Instantiate a StreamingContext with a DStream duration of 5 seconds

```
ssc = StreamingContext(sc, 5)
```

      2.2.3 Check to make sure that ssc has been created properly

```
ssc
```

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 5)
```

```
ssc
```

```
<pyspark.streaming.context.StreamingContext at 0x7f3e88722e80>
```

3. Develop the transformation logic for each DStream

   3.1 Set the streaming data source parameters

      3.1.1 Create a variable named port and set value to 44444

      3.1.2 Create a variable named host and set the value to 'localhost'

```
host = 'localhost'
port = 44444
```

   3.2 Create a DStream with socketTextStream. Set the host and port to the variables that we created in the previous step

```
aliceDS = ssc.socketTextStream(host,port)
```

   3.3 Create the transformations to aliceDS in order to count the number of words for each DStream

      3.3.1 Split the string into an individual words. Hint: Use flatMap instead of map

      3.3.2 Create a key:value tuple. Each word will generate a (word, 1) tuple

      3.3.3 Use reduceByKey() to aggregate all the values with the same word

```
host = 'localhost'
port = 44444
```

```
aliceDS = ssc.socketTextStream(host,port)
```

```
wcDS = aliceDS \
    .flatMap(lambda line: line.split(' ')) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda v1,v2: v1+v1)
```

   3.4 Use the pprint() action to pretty print wcDS

```
wcDS.pprint()
```

3.5 Create a function to print the word count for each DStream

```
def printCount(t, r):
    print("Word Count at time", t)
    for cnt in r.take(10):
        print(cnt[0], ":", cnt[1])
```

3.6 Execute printCount() on each DStream

3.6.1 Use foreachRDD to execute printCount on each DStream. foreachRDD passes the timestamp and pointer to the underlying RDD in the current DStream.

```
wcDS.foreachRDD(lambda time, rdd: printCount(time, rdd))
```

3.7 Start the streaming engine and await termination

```
ssc.start()
ssc.awaitTermination()
```

4. Test the streaming application

4.1 As soon as you start the StreamingContext, the simulated streaming data source will engage and begin streaming to the designated socket

```
being drowned in my own tears! That _will_ be a queer thing, to be

sure! However, everything is queer to-day."


Just then she heard something splashing about in the pool a little way
```

4.2 You will see output similar to below

```
Word Count at time 2021-09-23 01:09:30
Project : 2
of : 134217728
Alice’s : 64
in : 67108864
Wonderland, : 1
Lewis : 4
Carroll : 4
 : 17179869184
is : 524288
use : 256
```

Lab 12: Working with Multi-Batch DStream API

In this lab, we will work with multiple DStream batches.

1. Modify program and create streaming application

   1.1 Review the source data

      1.1.1 Navigate to /home/student/Data/weblogs

      1.1.2 Open any of the weblogs and review the source data.

```
62.133.252.161 - 7645 [23/Sep/2021:06:20:05 +0900] "PUT
/apps/cart.jsp?appID=7565 HTTP/1.0" 200 4953
"https://www.wallace-briggs.com/app/wp-content/author.html"
"Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.2 (KHTML, like
Gecko) Chrome/37.0.899.0 Safari/534.2"
```

      1.1.3 The first element of the weblog is the IP address through which the user connected to the webserver

      1.1.4 There is a (-) followed by another number. This third element is the user_id of the person browsing the web. In our case, this user_id is the author_id in our authors table in Hive.

      1.1.5 The rest of the web is the timestamp, the GET/PUT operation, and other miscellaneous items that we are not interested in for now.

   1.2 Review the template code

      1.2.1 Navigate to /home/student/Labs/C6U3

      1.2.2 Copy multi_dstream.py.template to mult_dstream.py

      1.2.3 You have been provided a template to begin programming a multi-batch Spark Streaming application. The program expects to receive the hostname and port number where it will read the streaming data. In our case, we will be passing "localhost" and "44444" to it

```
import sys
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

if __name__ == "__main__":
    if len(sys.argv) != 3:
```

```
        print >> sys.stderr, "Usage: multi_dstream.py <hostname>
<port>"
        sys.exit(-1)

    # get hostname and port of data source from application
arguments
    hostname = sys.argv[1]
    port = int(sys.argv[2])

    # Create a new SparkContext
    sc = SparkContext()

    # Set log level to ERROR to avoid distracting extra output
    sc.setLogLevel("ERROR")

    # Create and configure a new Streaming Context
    # with a 1 second batch duration
    ssc = StreamingContext(sc,1)

    # Create a DStream of log data from the server and port
specified
    logs = ssc.socketTextStream(hostname,port)

    # Put you application logic here



    ssc.start()
    ssc.awaitTermination()
```

1.2.4 The housekeeping logic has been done for you. The SparkContext has been created. The StreamingContext with a DStream duration of 1 second has been created.

1.2.5 A streaming data source has been set to the socketTextStream at the provided hostname and port number.

1.3 Modify multi_dstream.py to add the application logic

1.3.1 Use an editor to modify multi_dstream.py. You have the option of using vim, or Kwrite as your editor in the current environment.

1.3.2 When we work with multiple batches of DStreams where state is required, a checkpoint directory must be setup. Add the following:

```
ssc.checkpoint("checkpoint_dir")
```

1.3.3 The streaming weblogs are saved to the variable `logs`. Count how many connections are coming in over a window of 5 seconds. Calculate this count every 2 seconds. Use the DStream.countByWindow(windowDuration, slideDuration) transformation for this

```
cnt = logs.countByWindow(5,2)
```

1.3.4 Use pprint() to print the result

```
cnt.pprint()
```

1.4 Simulate streaming weblogs

1.4.1 Navigate to /home/student/Labs/C6U3 in the local drive

1.4.2 Use the cat Linux command to review stream_web.sh

```
$ cat stream_web.sh
```

1.4.3 The bash script simply calls port_stream.py with the hostname, port number and the data source directory to stream the weblogs to the Linux socket.

1.4.4 Execute stream_web.sh

```
$ ./stream_web.sh
```

```
[student@localhost C6U3]$ ./stream_web.sh
Waiting for connection on localhost : 44444
```

The script will respond by letting you know that it is waiting for a connection at port 44444

1.5 Execute the multi_dstream.py

1.5.1 Before starting the program, we have to switch the Spark Shell to iPython mode.

1.5.2 Navigate to your local home directory at /home/student

1.5.3 Select .bashrc file and edit the file to enable ipython mode

```
# Uncheck the following two lines to run PySpark from ipython
export PYSPARK_DRIVER_PYTHON="ipython"
export PYSPARK_DRIVER_PYTHON_OPTS=""

# Uncheck the following two lines to run PySpark from jupyter
# export PYSPARK_DRIVER_PYTHON="jupyter"
# export PYSPARK_DRIVER_PYTHON_OPTS="notebook --port 3333"
```

The # in the beginning of the line indicates that the line is a comment. Remove the # comment from the two lines of setting for ipython mode. When the # is removed, KWrite will change the color of the text as shown above to indicate that the commands will be executed. Make sure that only the ipython settings are enabled. The Jupyter settings should remain commented with # in the beginning of the line

1.5.4 Navigate back to /home/student/C6U3

1.5.5 Use spark-submit with --master 'local[2]' to make sure that there are at least two threads

```
spark-submit --master 'local[2]' multi_dstream.py localhost 44444
```

1.5.6 The output will be similar to below:

```
-------------------------------------------
Time: 2021-09-23 03:36:48
-------------------------------------------
94

-------------------------------------------
Time: 2021-09-23 03:36:50
-------------------------------------------
95

-------------------------------------------
Time: 2021-09-23 03:36:52
-------------------------------------------
94

-------------------------------------------
Time: 2021-09-23 03:36:54
-------------------------------------------
95
```

2. Create a streaming application that keeps state

   2.1 Save the old program and prepare for new code

      2.1.1 Copy multi_stream.py to multi_stream.py.countbywindow

      2.1.2 Return to multi_dstream.py and edit the pyspark program

      2.1.3 Remove the following two lines of code

```
cnt = logs.countByWindow(5,2)
cnt.pprint()
```

   2.2 Count the number of occurrences of an author_id

      2.2.1 Use the wordcount logic that we have seen many times to count the number of occurrences of author_id

      2.2.2 Use map() with String.split(" ") to parse the weblogs

```
authorCnt = logs.map(lambda line: line.split(" "))
```

      2.2.3 Create a Pair RDD consisting of (author_id, 1)

```
.map(lambda coll: (coll[2], 1))
```

2.2.4 Use reduceByKey to add all the occurrences of an author_id

```
.reduceByKey(lambda v1, v2: v1+v1)
```

2.3 Use updateStateByKey() to update the author_id occurrences.

2.3.1 Create a function that will update the count. The function is expected to receive a current state, and a collection of new occurrences. If needs to check if the state is None (meaning it is the first time we are seeing this author_id), it should return a count of the new occurrences. If there is a state, it should return the state plus the count of new occurrences. The function would be similar to the following:

```
def updateAuthorCount(new_occurences, curr_state):
    if curr_state == None: return sum(new_occurences)
    else: return curr_state + sum(new_occurences)
```

2.3.2 Transform authorCnt with updateStateByKey using the updateAuthorCount function

```
totalAuthorCnt = authorCnt \
      .updateStateByKey(lambda new_occurences, curr_state:
updateAuthorCount(new_occurences, curr_state))
```

2.3.3 Use pprint() to print totalAuthorCnt

2.3.4 Use spark-submit with --master 'local[2]' to make sure that there are at least two threads

```
spark-submit --master 'local[2]' multi_dstream.py localhost 44444
```

2.3.5 The output will be similar to below:

```
Time: 2021-09-23 04:33:03
-------------------------------------------
('3838', 1)
('1098', 1)
('3297', 1)
('20', 1)
('1204', 1)
('1517', 1)
('2990', 1)
('6678', 1)
('4799', 1)
('772', 2)
...

-------------------------------------------
Time: 2021-09-23 04:33:04
-------------------------------------------
('3838', 1)
('1098', 1)
('3297', 2)
('20', 1)
('1204', 1)
('1517', 1)
('2990', 1)
```

2.4 Currently the application prints out authors who have accessed the website and the frequency of their contacts. However, there is no ordering or sorting in place. Modify the application so that authors with the most frequent accesses to the website are displayed first

2.4.1 Use the map() transformation to change the (<author_id>,<frequency>) to (<frequency >,<author_id >)

```
mostFreqAuthor = totalAuthorCnt \
      .map(lambda tup:(tup[1], tup[0])) \
```

2.4.2 Use transform() to execute sortByKey RDD transformation. Recall that transform is a wrapper transformation that allows RDD transformations to called on DStreams. Set the sortByKey parameter to False. This states that ascending is False, or descending is True.

```
.transform(lambda rdd: rdd.sortByKey(False)) \
```

2.4.3 Use map() to flip the pair rdd back to (<author_id>,<frequency>)

```
.map(lambda tup:(tup[1], tup[0]))
```

2.4.4 Use spark-submit with --master 'local[2]' to make sure that there are at least two threads

```
spark-submit --master 'local[2]' multi_dstream.py localhost 44444
```

2.4.5 The output will be similar to below:

```
-------------------------------------------
Time: 2021-09-23 04:43:23
-------------------------------------------
('772', 2)
('1279', 2)
('4714', 2)
('5802', 2)
('6881', 2)
('1824', 2)
('3447', 2)
('3838', 1)
('1098', 1)
('3297', 1)
```

```
import sys
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

def updateAuthorCount(new_occurences, curr_state):
    if curr_state == None: return sum(new_occurences)
    else: return curr_state + sum(new_occurences)

if __name__ == "__main__":
    if len(sys.argv) != 3:
```

```
        print >> sys.stderr, "Usage: multi_dstream.py <hostname>
<port>"
        sys.exit(-1)

    # get hostname and port of data source from
    # application arguments
    hostname = sys.argv[1]
    port = int(sys.argv[2])

    # Create a new SparkContext
    sc = SparkContext()

    # Set log level to ERROR to avoid distracting extra output
    sc.setLogLevel("ERROR")

    # Create and configure a new Streaming Context
    # with a 1 second batch duration
    ssc = StreamingContext(sc,1)

    # Create a DStream of log data from the server
    # and port specified
    logs = ssc.socketTextStream(hostname,port)

    # Put you application logic here
    ssc.checkpoint("checkpoint_dir")
    authorCnt = logs \
      .map(lambda line: line.split(" ")) \
      .map(lambda coll: (coll[2], 1)) \
      .reduceByKey(lambda v1, v2: v1+v1)

    totalAuthorCnt = authorCnt \
      .updateStateByKey(lambda new_occurences, curr_state:
updateAuthorCount(new_occurences, curr_state))
```

```
    mostFreqAuthor = totalAuthorCnt \
      .map(lambda tup:(tup[1], tup[0])) \
      .transform(lambda rdd: rdd.sortByKey(False)) \
      .map(lambda tup:(tup[1], tup[0]))

    mostFreqAuthor.pprint()

    ssc.start()
    ssc.awaitTermination()
```

Lab 13: Working with Structured Streaming API

In this lab, we will work with streaming data that is structured. The streaming source will be created as dataframes where isStreaming is true.

1. Setup a structured streaming source

   1.1 Copy the posts data from HDFS to local drive

      1.1.1 In an earlier lab, the posts table in MySQL was imported to HDFS under /user/student/posts directory. Verify that the data is there. If not, use Sqoop to import the data.

      1.1.2 Use hdfs dfs -get to copy the data from HDFS to the local disk. Use the following command:

```
$ cd /home/student/Data
$ hdfs dfs -get posts
```

      1.1.3 Verify that a new directory named posts had been created in your local disk

      1.1.4 Navigate to posts. There are several data partitions. There is also a _SUCCESS file. Remove this file.

```
$ cd ./posts
$ rm _SUCCESS
```

      1.1.5 Review one of the data partitions. The data is in CSV format but there is no header row. We will need to know the schema information in order to use this data.

   1.2 Review Posts table schema

      1.2.1 Open MariaDB as user student and use "student" as password

```
$ mysql -u student -p
#type student when prompted for password
```

      1.2.2 From MariaDB, navigate to labs database and use DESC to get the schema information for the posts table

```
MariaDB [(none)]>  USE labs;
```

```
MariaDB [(labs)]>  DESC posts;
```

1.2.3 The output will be similar to below:

```
MariaDB [labs]> desc posts;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| id          | int(11)      | NO   | PRI | NULL    | auto_increment |
| author_id   | int(11)      | NO   |     | NULL    |                |
| title       | varchar(255) | NO   |     | NULL    |                |
| description | varchar(500) | NO   |     | NULL    |                |
| content     | text         | NO   |     | NULL    |                |
| date        | date         | NO   |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)
```

1.3 Execute the script to create a file streaming source

1.3.1 Execute stream_posts.sh to stream the posts to the Linux socket at localhost via port number 44444

2. Create a streaming DataFrame

2.1 Reset PySpark to start from Jupyter by modifying .bashrc in the home directory

2.2 Start pyspark with at least two threads.

```
pyspark --master local[2]
```

2.3 Create a dataframe from a socket data source

```
postsDF = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", "44444") \
    .load()
```

2.4 Sockets read data with a fixed schema. All data is contained in a single column named "value." It is necessary to use column expressions to create the columns with the data types as observed from the description of the Posts table in MySQL

```
from pyspark.sql.functions import *

aPostDF = postsDF \
    .withColumn("id",
                split(postsDF.value, ",")[0].cast("integer")) \
    .withColumn("author_id",
                split(postsDF.value, ",")[1].cast("integer")) \
    .withColumn("title",
                split(postsDF.value, ",")[2]) \
    .withColumn("description",
                split(postsDF.value, ",")[3]) \
    .withColumn("content",
                split(postsDF.value, ",")[4]) \
    .withColumn("date",
                split(postsDF.value, ",")[5])
```

3. Modify and transform Streaming DataFrame

   3.1 Once the base streaming dataframe has been created, use dataframe transformations to format the data as desired

```
myOutDF = aPostDF \
    .select("author_id",
            aPostDF.title[0:10].alias("Post_Title"),
            "date")
```

4. Start the Structure Streaming Engine

   4.1 Use writeStream with console format to display the output on the console. Set the output mode to append and do not truncate the output. Set a trigger for each micro-batch of 2 seconds.

```
myStream = myOutDF.writeStream \
    .format("console") \
    .option("truncate","false") \
    .outputMode("append") \
    .trigger(processingTime="2 seconds") \
    .start()
myStream.awaitTermination()
```

   4.2 Output will be similar to below

```
-------------------------------------------
Batch: 3
-------------------------------------------
+---------+----------+----------+
|author_id|Post_Title|date      |
+---------+----------+----------+
|38       |Dolorum su|1989-05-09|
|40       |A ad est a|1981-07-24|
|42       |Iure ad at|2013-11-01|
|44       |Qui volupt|2008-06-28|
|46       |Ex accusan|1983-10-06|
|48       |Laudantium|1972-08-15|
|50       |Non quia e|1972-11-22|
|52       |Et volupta|1992-05-30|
|54       |Perspiciat|2003-04-30|
|56       |Nesciunt a|1993-11-10|
```

Lab 14: Create an Apache Spark Application

So far, the spark shell has been used to develop and test applications. In this lab, we will create a PySpark application and execute it using spark-submit. We will also practice setting various configurations.

1. Create a Spark Application

   1.1 Navigate to /home/student/Labs/C6U4

   1.2 Open and review CountRabbits.py.skeleton
   This is a skeleton file that needs to be modified to create the application.

   1.3 Copy CountRabbits.py.skeleton to CountRabbits.py

   1.4 Create the SparkConf object

      1.4.1 The SparkConf library has already been imported. Use the imported library to instantiate a new instance of SparkConf. Save it to variable sconf. Set the application name to "Bunch O' Rabbits" using the setAppName() method

```
sconf = SparkConf().setAppName("Bunch O' Rabbits")
```

   1.5 Create the SparkContext object

      1.5.1 The SparkContext library has already been imported. Use the imported library to instantiate a new instance of SparkContext. Save it to variable sc

```
sc = SparkContext(conf = sconf)
```

      1.5.2 Set the log level to ERROR

```
sc.setLogLevel("ERROR")
```

   1.6 Create the transformations to read "alice.txt" from the HDFS home directory and count the number of times the word "Rabbit" appears in the text. Make sure the logic is case-insensitive, i.e. capture both Rabbit and rabbit.

      1.6.1 Create a new RDD by reading the filename passed to the system. The filename is in sys.argv[1]

```
src = sys.argv[1]
```

1.6.2 Change all letters to uppercase.

1.6.3 Filter for lines that contain "RABBIT"

1.6.4 Count the number of rows and print it out.

1.6.5 Stop the Spark Context

1.7 Save the file as CountRabbits.py

2. Submitting a PySpark application

2.1 Change Apache Spark to iPython mode

2.1.1 Before submitting a Spark application, Jupyter mode must be disabled. Modify the .bashrc file to disable Jupyter and enable iPython.

```
export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH

# Uncheck the following two lines to run PySpark from ipython
export PYSPARK_DRIVER_PYTHON="ipython"
export PYSPARK_DRIVER_PYTHON_OPTS=""

# Uncheck the following two lines to run PySpark from jupyter
# export PYSPARK_DRIVER_PYTHON="jupyter"
# export PYSPARK_DRIVER_PYTHON_OPTS="notebook --port 3333"
```

2.1.2 Every time a bash terminal is opened, .bashrc is read and executed. Either close the current terminal and open a new one to execute .bashrc or use source ~/.bahsrc to manually execute it.

```
source ~/.bashrc
```

2.2 Use spark-submit to execute CountRabbits.py. Pass alice.txt as the text file to read.

```
spark-submit CountRabbits.py alice.txt
```

```
extHandler@10934f48{/metrics/json,null,AVAILABLE,@Spark}
There are 53 occurences of Rabbit in Alice In Wonderland
[student@localhost C6U4]$
```

2.3 View the Spark Web UI

2.3.1 When a Spark application is submitted, the Spark Web UI is started and available to view on the browser. The address depends on what master mode, the program is executing on. The default mode for spark-submit is local mode.

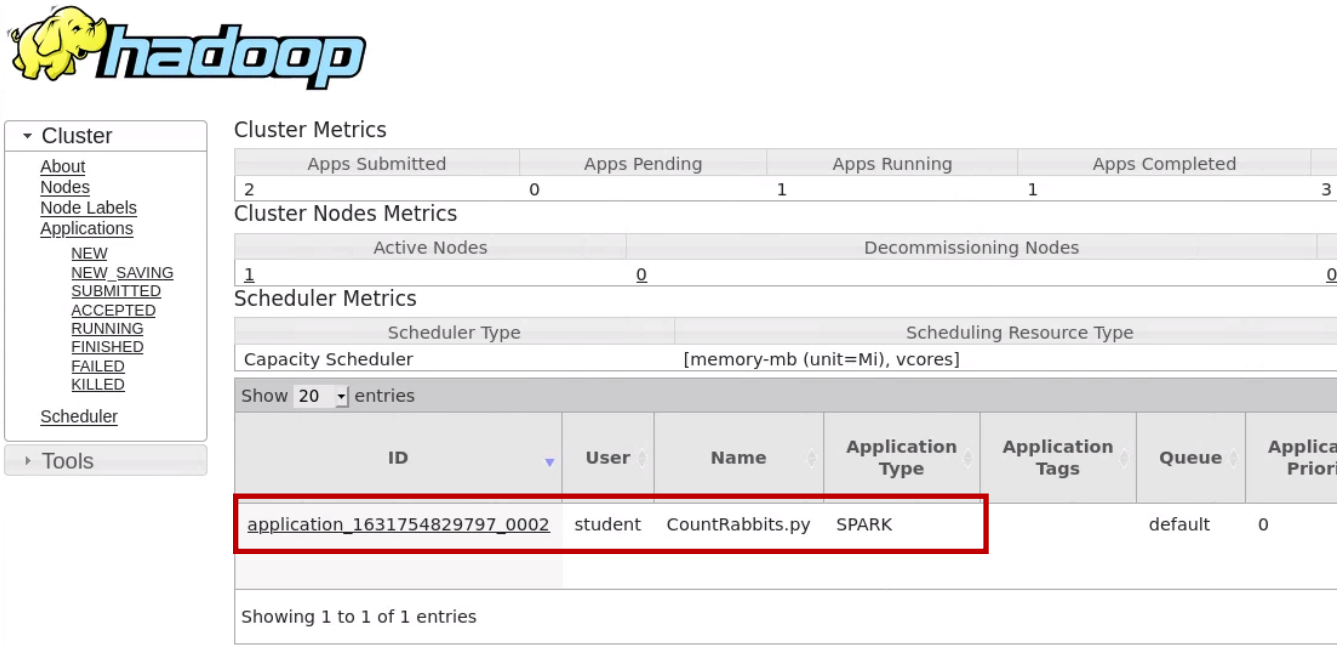
If no other SparkContext is active, Spark Web UI should start port 4040 and can be viewed by navigating to http://localhost:4040

2.3.2 Submit CountRabbits.py again. This time open a browser and navigate to the appropriate link. You should open the browser and be ready to navigate quickly since the program finishes fairly quickly. (It might be impossible to view the Spark Web UI in local mode since the application completes before the Jetty server for Spark Web UI is able to properly initiate. If you have trouble timing it, don't worry.)

2.3.3 This time submit the program in with YARN master and cluster deploy mode.

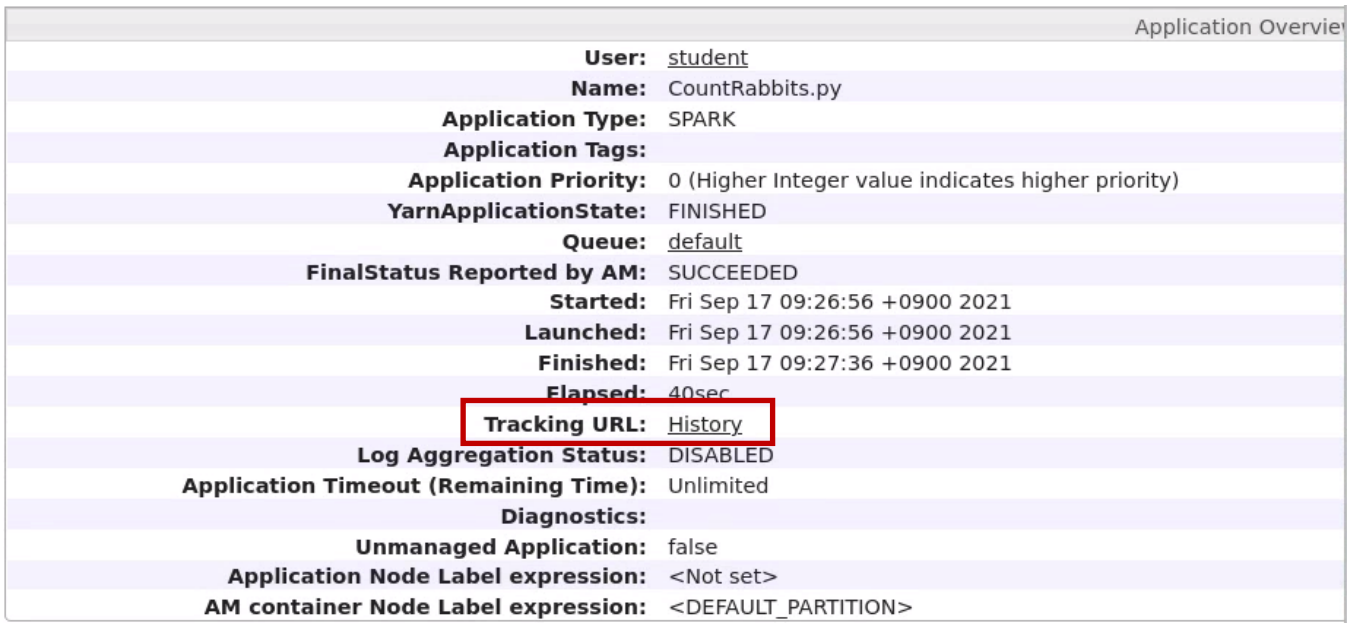
```
spark-submit --master yarn --deploy-mode cluster \
CountRabbits.py alice.txt
```

2.3.4 Open the Yarn Web UI and look for either a running job or finished job, depending on your timing



2.3.5 Click on the application id as shown above

2.3.6 If the application has already completed, the tracking URL will show the history server, otherwise, it will show the applications master.



3. Setting Spark Configurations

END OF LAB