Large-Scale Stream Processing Homework 1 Report

Chenye Yang cy2540

Code Explanation

Part 1

There exists a lot of domain name as “wpbfl2-45.gate.net” in the “epa-http.txt” file. According to the requirements, these domain names should be discarded. Therefore, a python regular expression [5] is used to judge whether an IP address is valid (141.243.1.172) or not (wpbfl2-45.gate.net).

```
import re
regex = r'(([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])\.){3}([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])'
re.search(regex, ip_address)
```

To creat a spark context object with specific configuration, we use [1]:

```
conf = pyspark.SparkConf().setAppName('hw1_part1_ChenyeYang').setMaster('local[*]')
sc = pyspark.SparkContext(conf=conf) # creat a spark context object
```

Use `log_lines = sc.textFile('../epa-http.txt')` to read text file line by line to create RDDs.

Use `filter()` to select the RDD with valid IP address. In http return code, 302 is a FOUND code, which means “Tells the client to look at (browse to) another URL” [10]. The request with 302 code and “-” served bytes cannot be counted as a http request because the client will get response from another server and that response will be counted as valid request. Thus, the RDDs with “-” served bytes should also be discarded. Then the `filter()` is:

```
log_lines_validip = log_lines.filter(lambda x: re.search(regex, x.split(' ')[0]) and x.split(' ')[-1] != '-')
```

After we get the RDDs with valid IP address and request, we can transform them into pair RDDs and only keep the information we need, which are IP address and bytes. Key: IP address. Value: Bytes served. In this code, use function `map()` rather than `flatMap()` because one input RDD is only corresponding to one output RDD.

```
log_ip_bytes_pairs = log_lines_validip.map(lambda x: (x.split(' ')[0], int(x.split(' ')[-1])))
```

Then count the total bytes served to same IP addresses, according to keys in ('ip',bytes) pairs.

```
log_ip_total_bytes = log_ip_bytes_pairs.reduceByKey(lambda x, y: x + y)
```

And sort the reduced results by IP in string ascending order so 100.1.2.3 comes before 99.2.3.5

```
log_sort_by_ip = log_ip_total_bytes.sortByKey(ascending=True)
```

Save RDD as CSV. In this way, the csv file will have a blank line at the end.

```
f = open('hw1_part1.csv', 'w', newline='')
f_csv = csv.writer(f)
f_csv.writerows(log_sort_by_ip.collect())
```

Some results:

```
The number of lines with valid IP address and HTTP request: 9223
The number of different IPs: 592
The total bytes served to first 20 IP: [('128.104.66.114', 4889), ('128.120.125.246', 11359),
```

Part 2

Apart from the part 1, the following code are added to make the program able to take command line parameters (configurable parameter):

```
import argparse
parser = argparse.ArgumentParser(description='Homework 1 Part 2 from Chenye Yang')
parser.add_argument("-K", default=10, help="K is an integer number, to get the top K IPs that were served the most number of bytes")
args = parser.parse_args()
K = int(args.K)
```

It takes two parameters which are '-K' and '-h'. '-K' needs an integer input which indicates "the top K IPs that were served the most number of bytes". In command line, the usage is:

`python hw1_part2.py -h` or `python hw1_part2.py -K 10`

The different thing between part 1 and part 2 is that part 1 is sorted by key while part 2 is sorted by value. Actually, we have two ways to get the K greatest number of bytes.

The first way is to use `sortBy()` and then `take()`:

```
# sort the pair RDDs by value, from large to small
log_sort_by_total_bytes = log_ip_total_bytes.sortBy(lambda x: x[1], ascending=False)
# after the sort done, we can directly use take() to "Take the first num elements of the RDD"
print(log_sort_by_total_bytes.take(K))
```

Because the IPs should be sorted from the most bytes to least bytes, the parameter in `sortBy()` must be set as `ascending=False`. Use the value in pair RDDs to do the sort. After sorting, we can directly use `take()` to get the required IP-bytes pairs.

The second way is to use `top()`:

```
# or we can ues the top() function to "Get the top N elements from an RDD",
# and choose the key function as the value in pairs.
# this method don't need to use sortBy() function to sort RDDs first
print(log_ip_total_bytes.top(K, lambda x: x[1]))
```

By using this function, the required IP-bytes pairs can be directly outputted. This function `top()` is like the combination of `sortBy()` and `take()`. Remember to specify the number `K` to get and the variable `lambda x: x[1]` to be sorted.

Some results:

```
First Method
[('139.121.98.45', 5041738), ('203.251.228.110', 3785626), ('155.84.92.3', 3353172),
Second Method
[('139.121.98.45', 5041738), ('203.251.228.110', 3785626), ('155.84.92.3', 3353172),
```

Part 3

Part 3 is quite different from the former parts, not only because it has to consider timestamps but also because it requires the use of window. Therefore, after the items with valid IP address and HTTP return code is selected out, the original RDDs should be mapped into pair RDDs with three attributes: IP, timestamp, bytes. datetime() is used to create a timestamp from the time string (year and month have to be set manually).

```
# use "datetime()" to change the time string to real time
# 141.243.1.172 [29:23:53:25] "GET /Software.html HTTP/1.0" 200 1497
log_pairs = log_lines_validip.map(lambda x:
                                  (x.split(' ')[0],
                                   datetime.strptime('20-01-{}{}'.format(x.split(' ')[1][1:3], x.split('
')[1][4:-1]), '%y-%m-%d%H:%M:%S'),
                                   int(x.split(' ')[-1])))
```

Window can be used in spark streaming processing. However, it cannot deal with the late data. Because in the real-world environment, there are a lot of late data and mis-ordered frames and people sometimes are more interested in the event time (not receiving time) of one data, a fault-tolerant tool should be used to process data with its event time [1]. As a reason of above, Spark Structured Streaming should be used in this part 3 to deal with the late data.

To use the Spark Structured Streaming process, we should first build a SparkSession, which is based on Spark SQL. Thus, some libraries should be imported first:

```
import pyspark
from pyspark.sql import *
from pyspark.sql.types import *
from datetime import datetime
from pyspark.sql.functions import *
```

Then we create a spark session and set the schema of data frame:

```
# create a spark session
spark = SparkSession \
    .builder \
    .appName('hw1_part3_ChenyeYang') \
    .getOrCreate()
# set the schema of data frame
schema = StructType([
    StructField('ip', StringType(), True),
    StructField('datetime', TimestampType(), True),
    StructField('bytes', IntegerType(), True)])
```

After that, a data frame can be created from RDDs and schema:

```
# creat data frame with RDDs and schema
# columns in log_dataframe are ['ip', 'datetime', 'bytes']
log_dataframe = spark.createDataFrame(log_pairs, schema)
```

In order to deal with the late data properly and not process too-late data, watermark should be used. The data which is later than 10 minutes will be discarded.

```
# with watermark, we can handle the late data properly. Discard very late data and keep not very late data.
# with window size = 60min slide = 60min, we group the log in one hour by "datetime"
# with groupBy, we group the DataFrame using the specified columns, so we can run aggregation on them.
# with agg, we aggregate the items in window and "ip", the result should be the sum of "bytes"
# with orderBy, we get new DataFrame sorted by the specified column(s) by timestamp ascending and total bytes descending
# with show, we print 20 sorted items without truncating strings longer than 20 chars
# CONTENT IN log_windowed:
# columns in log_windowed are ['window', 'ip', 'sum(bytes)']
# data types are [('window', 'struct<start:timestamp,end:timestamp>'), ('ip', 'string'), ('sum(bytes)', 'bigint')]
log_windowed = log_dataframe \
    .withWatermark('datetime', '10 minutes') \
    .groupBy(window(log_dataframe.datetime, '60 minutes', '60 minutes'), log_dataframe.ip) \
    .agg(sum('bytes')) \
    .orderBy(['window.start', 'sum(bytes)'], ascending=[True, False])
log_windowed.show(20, truncate=False)
```

As required, data in interval 30:00:00:00 to 30:00:59:59 need to be stored. Then a filter is set to the windowed dataframe to extract the specific data.

```
# window.start and window.end are timestamps and we can use the built in method to get year(), month(), day()
# use filter() and SQL like commands to select the specific data in one day between some hours
log_specific_time_interval = log_windowed \
    .filter('year(window.start) == 2020') \
    .filter('month(window.start) == 01') \
    .filter('day(window.start) == 30') \
    .filter('Hour(window.start) == 00 AND Hour(window.end) == 01')
```

Then the attribute/column ‘window’ is dropped because it is not in the output csv file.

```
# drop the 'window' column because it is not in output csv file
# sort by ip in string ascending order so 100.1.2.3 comes before 99.2.3.5.
log_csv = log_specific_time_interval.drop('window').sort('ip', ascending=True)
```

Some results:

```
All the windowed data
+------------------------------------------+--------------+----------+
|window                                    |ip            |sum(bytes)|
+------------------------------------------+--------------+----------+
|[2020-01-29 23:00:00, 2020-01-30 00:00:00]|131.215.67.47 |32988     |
|[2020-01-29 23:00:00, 2020-01-30 00:00:00]|140.112.68.165|7811      |
|[2020-01-29 23:00:00, 2020-01-30 00:00:00]|131.170.154.29|7513      |
|[2020-01-29 23:00:00, 2020-01-30 00:00:00]|141.243.1.172 |1497      |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|141.243.1.172 |177370    |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|149.159.22.10 |67951     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|161.122.12.78 |45957     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|138.25.148.25 |28591     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|204.62.245.32 |21181     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|204.188.47.212|21118     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|202.32.50.6   |16682     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|141.243.1.174 |10739     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|157.22.192.30 |7513      |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|131.170.154.29|5075      |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|203.1.203.222 |3717      |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|137.132.52.66 |2235      |
|[2020-01-30 01:00:00, 2020-01-30 02:00:00]|198.69.241.75 |34462     |
|[2020-01-30 01:00:00, 2020-01-30 02:00:00]|141.243.1.172 |20750     |
|[2020-01-30 01:00:00, 2020-01-30 02:00:00]|138.25.148.25 |16900     |
|[2020-01-30 01:00:00, 2020-01-30 02:00:00]|140.120.29.161|11516     |
+------------------------------------------+--------------+----------+
only showing top 20 rows

Data in specific time interval
+------------------------------------------+--------------+----------+
|window                                    |ip            |sum(bytes)|
+------------------------------------------+--------------+----------+
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|141.243.1.172 |177370    |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|149.159.22.10 |67951     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|161.122.12.78 |45957     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|138.25.148.25 |28591     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|204.62.245.32 |21181     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|204.188.47.212|21118     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|202.32.50.6   |16682     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|141.243.1.174 |10739     |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|157.22.192.30 |7513      |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|131.170.154.29|5075      |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|203.1.203.222 |3717      |
|[2020-01-30 00:00:00, 2020-01-30 01:00:00]|137.132.52.66 |2235      |
+------------------------------------------+--------------+----------+
```

Note: The above interval data is sorted by descending order of total bytes, which is different with the order in csv file.

Part 4

Part 4 is kind of similar to part 1 and part 2. Therefore, it can also be solved by the same method in part 1 and part 2. However, for convenience, I just use the same method and code in part 3. Because I'm not sure whether part 4 should consider timestamp and window and watermark or not, and using part 3 code is easy for future modification.

Besides the normal way to map the RDDs, a function should be defined to split the IP addresses to subnet. I'm a little bit confused about the requirements so I use the first 6 digitals as the subnet, just as stated in the latest requirements.

```
# find subnet in which the ip belongs to, use first 6 numbers as the subnet
def subnet(ip):
    point_location = [i for i in range(len(ip)) if ip.startswith('.', i)][1]
    sub = '{}.*.*'.format(ip[0:point_location])
    # sub = '{}.*'.format(ip[0:point_location])
    return sub
```

Then the map() function should be like:

```
# divide ips into different subnets
log_subnet = log_pairs.map(lambda x: (subnet(x[0]), x[1]))
```

Because this part doesn't need the timestamp, the schema is as follow:

```
# set the schema of data frame
schema = StructType([
    StructField('subnet', StringType(), True),
    StructField('bytes', IntegerType(), True)])
```

Then create the data frame and group it with subnet and sum the bytes.

```
# create data frame with RDDs and schema
# columns in log_dataframe are ['subnet', 'bytes']
log_dataframe = spark.createDataFrame(log_subnet, schema)
# group ips in one subnet and sum up the bytes in this subnet
# sort by subnet in string ascending order so 100.*.*.* comes before 99.*.*.*
log_csv = log_dataframe.groupBy('subnet').agg(sum('bytes')).sort('subnet', ascending=True)
```

Some results:

```
The total number of items: 387
+-----------+----------+
|subnet     |sum(bytes)|
+-----------+----------+
|128.104.*.*|4889      |
|128.120.*.*|553346    |
|128.138.*.*|387651    |
|128.144.*.*|521757    |
|128.148.*.*|3947      |
|128.150.*.*|35238     |
|128.158.*.*|73249     |
|128.159.*.*|1087358   |
|128.163.*.*|482503    |
|128.165.*.*|92188     |
|128.180.*.*|5388      |
|128.192.*.*|2624      |
|128.196.*.*|24398     |
|128.206.*.*|4233      |
|128.220.*.*|11746     |
|128.227.*.*|456719    |
|128.23.*.* |11747     |
|128.252.*.*|325001    |
|128.253.*.*|269703    |
|128.255.*.*|32988     |
+-----------+----------+
only showing top 20 rows
```

Resources about PySpark

[1] Tutorial

https://spark.apache.org/docs/latest/quick-start.html

[2] API Document

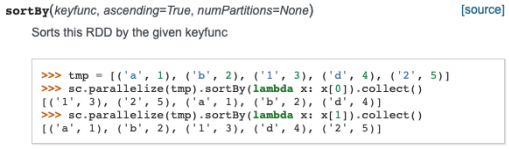
https://spark.apache.org/docs/latest/api/python/index.html

[3] First Steps With PySpark and Big Data Processing

https://realpython.com/pyspark-intro/#hello-world-in-pyspark

[4] How to sort by value efficiently in PySpark?

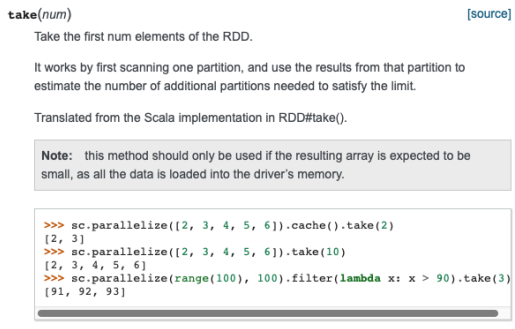
https://stackoverflow.com/questions/33706408/how-to-sort-by-value-efficiently-in-pyspark



sortBy(*keyfunc*, *ascending=True*, *numPartitions=None*) [source]

Sorts this RDD by the given keyfunc

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortBy(lambda x: x[0]).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> sc.parallelize(tmp).sortBy(lambda x: x[1]).collect()
[('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
```



take(*num*) [source]

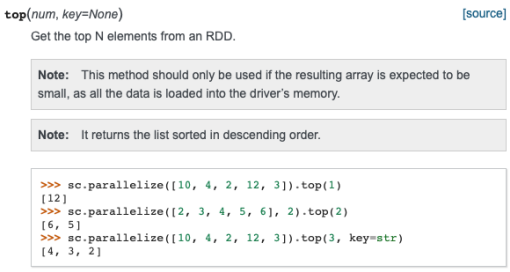
Take the first num elements of the RDD.

It works by first scanning one partition, and use the results from that partition to estimate the number of additional partitions needed to satisfy the limit.

Translated from the Scala implementation in RDD#take().

Note: this method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
>>> sc.parallelize([2, 3, 4, 5, 6]).cache().take(2)
[2, 3]
>>> sc.parallelize([2, 3, 4, 5, 6]).take(10)
[2, 3, 4, 5, 6]
>>> sc.parallelize(range(100), 100).filter(lambda x: x > 90).take(3)
[91, 92, 93]
```



top(*num*, *key=None*) [source]

Get the top N elements from an RDD.

Note: This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

Note: It returns the list sorted in descending order.

```
>>> sc.parallelize([10, 4, 2, 12, 3]).top(1)
[12]
>>> sc.parallelize([2, 3, 4, 5, 6], 2).top(2)
[6, 5]
>>> sc.parallelize([10, 4, 2, 12, 3]).top(3, key=str)
[4, 3, 2]
```


[5] Python program to validate an IP Address
https://www.geeksforgeeks.org/python-program-to-validate-an-ip-address/ (wrong regular expression)
https://www.runoob.com/note/38097 (right regular expression)

[6] Find weekly average in 2016
http://blog.madhukaraphatak.com/introduction-to-spark-two-part-5/

[7] How to delete columns in pyspark dataframe
https://intellipaat.com/community/7516/how-to-delete-columns-in-pyspark-dataframe

[8] Python | All occurrences of substring in string
https://www.geeksforgeeks.org/python-all-occurrences-of-substring-in-string/

[9] The Easy Guide to Python Command Line Arguments
https://levelup.gitconnected.com/the-easy-guide-to-python-command-line-arguments-96b4607baea1

[10] List of HTTP status codes
https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Errors & Solutions

Error:

py4j.protocol.Py4JJavaError: An error occurred while calling z:org.apache.spark.api.python.PythonRDD.collectAndServe.
: java.lang.IllegalArgumentException: Unsupported class file major version 57

Solution:

Spark use Java 8 while my Java is Java 13
https://stackoverflow.com/questions/53583199/pyspark-error-unsupported-class-file-major-version-55
JDK 13 Uninstall
https://explainjava.com/uninstall-java-macos/
JDK 8 Installation for OS X
https://docs.oracle.com/javase/8/docs/technotes/guides/install/mac_jdk.html#CHDBADCG