Big Data Infrastructures & Technology - Assignment 2

The second practicum will focus on practicing ETL (Extract, Transform and Load).

Loading the Data

In this assignment we will be working with the Apache web access log of the NASA website. This access log contains information about the HTTP requests that were made to the NASA website.

The access log has the following format:

```
$CLIENT_IP - - [$DATE] "$COMMAND $RESOURCE $PROTOCOL" $CODE  $SIZE
```

Here is a small sample from the access log:

```
gw1.cat.com - - [03/Jul/1995:13:46:41 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 200 786
pc13.ael2.ocps.k12.fl.us - - [03/Jul/1995:13:46:41 -0400] "GET /history/apollo/apollo-13/apollo-13.html HTTP/1.0" 200 18114
pc13.ael2.ocps.k12.fl.us - - [03/Jul/1995:13:46:42 -0400] "GET /history/apollo/apollo-13/apollo-13-patch-small.gif HTTP/1.0" 200 12859
pm2_11.digital.net - - [03/Jul/1995:13:46:42 -0400] "GET /history/apollo/apollo-13/images/70HC464.GIF HTTP/1.0" 200 85750
153.64.71.78 - - [03/Jul/1995:13:46:42 -0400] "GET /shuttle/countdown/lps/osr/osr.html HTTP/1.0" 200 1331
199.104.22.34 - - [03/Jul/1995:13:46:44 -0400] "GET /shuttle/missions/sts-71/movies/sts-71-launch.mpg HTTP/1.0" 200 57344
128.220.116.211 - - [03/Jul/1995:13:46:45 -0400] "GET /shuttle/missions/sts-49/mission-sts-49.html HTTP/1.0" 200 9379
shield.lightspeed.net - - [03/Jul/1995:13:46:45 -0400] "GET /facilities/spaceport.html HTTP/1.0" 200 6755
```

We will be loading the data, cleaning it up and performing basic aggregations on the loaded data. We will be using Spark to do this in a parallel and scalable manner.

In this exercise, we will restrict ourselves to the access log from the month July in 1995. This contains around 200MB of data in text format divided into 8 files. We start off by downloading the files, unzipping them and installing several packages

Setup [1]:

```
!wget https://github.com/Mytherin/NASAData/raw/master/nasa.zip
!unzip nasa.zip
!pip install pyspark plotly
```

We start off by initializing a Spark context.

In [1]:

```
from pyspark.sql import SparkSession, Row
spark = SparkSession.builder.master("local").config("spark.ui.enabled",
"false").getOrCreate()
sc = spark.sparkContext
```

We then load the data into Spark using the `textFile` command as we did in the previous assignment. In this case, however, we are loading data from multiple files. To load data from all files in the directory we use the wildcard character (`*`).

In [2]:
```
data = sc.textFile("nasa/*.txt")
```

The data can also be found here: https://github.com/Mytherin/NASAData

We can verify that the data was successfully loaded by printing a sample to the screen using the `take` or `collect` methods. This triggers the actual loading of the files from disk.

In [3]:
```
data.take(10)
```

Out [3]:
```
['cimar.me.ufl.edu - - [13/Jul/1995:12:17:41 -0400] "GET /images/MOSAIC-logosmall.gif
HTTP/1.0" 200 363',
 'cimar.me.ufl.edu - - [13/Jul/1995:12:17:41 -0400] "GET /images/USA-logosmall.gif
HTTP/1.0" 200 234',
 'seaborg5.nmu.edu - - [13/Jul/1995:12:17:42 -0400] "GET /shuttle/countdown/liftoff.html
HTTP/1.0" 200 3464']
```

Parsing the Data

After we have loaded the data into Spark we can use the map and reduce methods to extract useful data from the text. Currently, we only have a set of lines loaded into Spark. Instead, we want to extract the actual information from the log messages.

At a first glance, it might seem that we can easily parse this format with a Regex. We can quickly create the following Regex:

```
(.+) - - \[(\d+)/([a-zA-Z]+)/(\d+):(\d+):(\d+):(\d+) -\d+\] "([a-zA-Z]+) ([^
]+) ([^"]*)" (\d+) ([\d-]+)
```

When we run it on a small sample, this appears to work perfectly:

In [4]:
```
import re
regex = re.compile('(.+) - - \[(\d+)/([a-zA-Z]+)/(\d+):(\d+):(\d+):(\d+) -\d+\]
"([a-zA-Z]+) ([^ ]+)([^"]*)" (\d+) ([\d-]+)')
def extract_data(line):
    match = regex.match(line)
    return match.groups()
extracted_data = data.map(extract_data)
                    extracted_data.take(3)
```

Out [4]:
```
[('cimar.me.ufl.edu', '13', 'Jul', '1995', '12', '17', '41', 'GET',
'/images/MOSAIC-logosmall.gif', ' HTTP/1.0', '200', '363'),
```

```
 ('cimar.me.ufl.edu', '13', 'Jul', '1995', '12', '17', '41', 'GET',
'/images/USA-logosmall.gif', ' HTTP/1.0', '200', '234'),
 ('seaborg5.nmu.edu', '13', 'Jul', '1995', '12', '17', '42', 'GET',
'/shuttle/countdown/liftoff.html', ' HTTP/1.0', '200', '3464')]
```

However, when we actually run it on the entire dataset, we quickly find that there are certain lines that our Regex cannot parse. While our Regex will parse the majority of lines correctly, there are a number of edge cases that we do not handle correctly.

For example, the following lines are incorrectly parsed by our Regex:

```
[('jumbo.jet.uk - - [11/Jul/1995:08:06:29 -0400] "GET  HTTP/1.0" 302 -',),
 ('128.159.122.20 - - [20/Jul/1995:15:28:50 -0400] "k\x03tx\x04tG\x07t" 400 -',),
 ('128.159.122.20 - - [24/Jul/1995:13:52:50 -0400] "k\x03tx\x04tG\x07t" 400 -',)]
```

In the first case, an empty request is made. In the second case, strange unicode characters are used in the request. While it might be tempting to discard these lines (as there are only a few lines that are not correctly parsed by our Regex), we have to be careful that discarding these requests does not influence the results of our analysis.

While we would only discard a few requests, all of the requests that we discard are requests that result in **HTTP errors**. This means that the lines we discard are **biased**, since the majority of all requests are successful. If we would do an analysis looking at the amount of errors after discarding these requests, we might well be influencing the results of our analysis by discarding these lines.

Instead of using our own Regex, we will use the `apache_log_parser` library to parse these log files. This library is much more robust than our hand-rolled regex, and will correctly handle these edge cases. Like before, we need to install the library first.

Setup [2]:

```
!pip install apache_log_parser
```

In [5]:

```
import apache_log_parser
line_parser = apache_log_parser.make_parser("%h %l %u %t \"%r\" %>s %b")
```

We can then use the apache log parser to parse the log files. If we do so, we would get the following Python dictionary:

In [6]:

```
line_parser(data.take(1)[0])
```

Out [6]:

```
{'remote_host': 'cimar.me.ufl.edu',
 'remote_logname': '-',
 'remote_user': '-',
 'time_received': '[13/Jul/1995:12:17:41 -0400]',
 'time_received_datetimeobj': datetime.datetime(1995, 7, 13, 12, 17, 41),
 'time_received_isoformat': '1995-07-13T12:17:41',
```

```
 'time_received_tz_datetimeobj': datetime.datetime(1995, 7, 13, 12, 17, 41,
tzinfo='0400'),
 'time_received_tz_isoformat': '1995-07-13T12:17:41-04:00',
 'time_received_utc_datetimeobj': datetime.datetime(1995, 7, 13, 16, 17, 41,
tzinfo='0000'),
 'time_received_utc_isoformat': '1995-07-13T16:17:41+00:00',
 'request_first_line': 'GET /images/MOSAIC-logosmall.gif HTTP/1.0',
 'request_method': 'GET',
 'request_url': '/images/MOSAIC-logosmall.gif',
 'request_http_ver': '1.0',
 'request_url_scheme': '',
 'request_url_netloc': '',
 'request_url_path': '/images/MOSAIC-logosmall.gif',
 'request_url_query': '',
 'request_url_fragment': '',
 'request_url_username': None,
 'request_url_password': None,
 'request_url_hostname': None,
 'request_url_port': None,
 'request_url_query_dict': {},
 'request_url_query_list': [],
 'request_url_query_simple_dict': {},
 'status': '200',
 'response_bytes_clf': '363'}
```

We then create a new extract method using the `apache_log_parser` and map that over our files in Spark:

In [7]:

```
keys = ['remote_host', 'time_received_datetimeobj', 'request_first_line', 'status',
'request_url', 'response_bytes_clf']
def extract_data(line):
    try:
        result = line_parser(line)
        return tuple([result[key] if key in result else None for key in keys])
    except apache_log_parser.LineDoesntMatchException:
        return tuple()
extracted_data = data.map(extract_data)
filtered_data = extracted_data.filter(lambda x: len(x)>0)
filtered_data.take(3)
```

Out [7]:

```
[('cimar.me.ufl.edu', datetime.datetime(1995, 7, 13, 12, 17, 41), 'GET
/images/MOSAIC-logosmall.gif HTTP/1.0', '200', '/images/MOSAIC-logosmall.gif',
'363'),('cimar.me.ufl.edu', datetime.datetime(1995, 7, 13, 12, 17, 41), 'GET
/images/USA-logosmall.gif HTTP/1.0', '200', '/images/USA-logosmall.gif', '234'),
 ('seaborg5.nmu.edu', datetime.datetime(1995, 7, 13, 12, 17, 42), 'GET
/shuttle/countdown/liftoff.html HTTP/1.0', '200',
'/shuttle/countdown/liftoff.html', '3464')]
```

Creating a Spark DataFrame and writing to Parquet

Now we have successfully parsed the data from the individual lines, we will create a Spark DataFrame object. Whereas before we were working directly with text data, a Spark DataFrame is a structured representation of data (similar to a table in a database). This structured representation allows the data to be processed much faster than with the previous approaches.

A Spark DataFrame can be created from our RDD using the following syntax:

In [8]:

```
RowFormat = Row('remote_host', 'time_received', 'request_first_line', 'status',
'request_url', 'response_bytes')
formatted = filtered_data.map(lambda x: RowFormat(*x))
df = spark.createDataFrame(formatted)
df.show()
```

Out [8]:

```
+--------------------+------------------+------+--------------+
|         remote_host|     time_received|status|response_bytes|
+--------------------+------------------+------+--------------+
|    cimar.me.ufl.edu|1995-07-13 12:17:41|   200|           363|
|    cimar.me.ufl.edu|1995-07-13 12:17:41|   200|           234|
|    seaborg5.nmu.edu|1995-07-13 12:17:42|   200|          3464|
|pc-151.mrc.gla.ac.uk|1995-07-13 12:17:42|   200|         17083|
|    cimar.me.ufl.edu|1995-07-13 12:17:42|   200|           669|
|  jaxfl2-10.gate.net|1995-07-13 12:17:43|   200|        101950|
|     204.157.204.187|1995-07-13 12:17:44|   200|          3537|
|       165.103.5.123|1995-07-13 12:17:44|   304|             0|
|piweba3y.prodigy.com|1995-07-13 12:17:44|   200|          3537|
|        163.205.1.45|1995-07-13 12:17:44|   200|         62338|
|       165.103.5.123|1995-07-13 12:17:45|   304|             0|
|    jay.imagauto.com|1995-07-13 12:17:45|   200|          4247|
|frehra42.adpc.pur...|1995-07-13 12:17:45|   200|          4887|
|stratacom.strata.com|1995-07-13 12:17:46|   200|          3464|
| ariel.earth.nwu.edu|1995-07-13 12:17:46|   200|          1382|
|     128.252.178.183|1995-07-13 12:17:46|   200|          8678|
|      128.138.156.18|1995-07-13 12:17:46|   200|             0|
|piweba3y.prodigy.com|1995-07-13 12:17:47|   200|          1204|
| ariel.earth.nwu.edu|1995-07-13 12:17:47|   200|           509|
| ariel.earth.nwu.edu|1995-07-13 12:17:47|   200|           527|
+--------------------+------------------+------+--------------+
```

The Spark DataFrame can then be used to perform aggregations and data analysis. For now, let us first write the Spark DataFrame to disk in the efficient `parquet` format. This prevents us from having to perform the ETL steps every time we perform any kind of aggregation or analysis. We can write to a parquet file using the following command:

In [9]:

```
df.write.mode('overwrite').format("parquet").save("nasa.parquet")
```

We can then read from the parquet file again using the following command:

In [10]:

```
nasa = spark.read.parquet("nasa.parquet")
```

Now our data is ready for analysis.

Performing Analysis with Spark DataFrames

We can perform analysis on the resulting Spark DataFrames using the various dataset operations. In this section, we will perform two simple aggregations on the dataset. More examples of DataSet operations can be found in the Spark DataFrame guide:

https://spark.apache.org/docs/2.2.0/sql-programming-guide.html#untyped-dataset-operations-aka-dataframe-operations

Biggest File

We want to find out the biggest file on the NASA website. For this, we will first query the maximum response size using the `agg` function.

In [11]:

```
nasa.agg({"response_bytes": "max"}).show()
```

Out [11]:

```
+--------------------+
|max(response_bytes)|
+--------------------+
|               99981|
+--------------------+
```

Afterwards, we will use a `filter` and `selection` to find the matching `request_url`.

In [12]:

```
result = nasa.agg({"response_bytes": "max"}).collect()[0]
nasa.filter(nasa['response_bytes'] == result[0]).select('request_url').collect()
```

Out [12]:

```
[Row(request_url='/images/cdrom-1-95/img0007.jpg')]
```

We have discovered that the biggest file that was served by the NASA website is `/images/cdrom-1-95/img0007.jpg`.

Most Frequent Status Codes

For our second analysis, we will find out the most frequently occurring status codes. We will do this by using a `grouping` operator. We can group by the status, and count the amount of times that each specific status occurs in our data set using the `count` operator. We then order by the count that we have computed so we can figure out the frequency of each of the status codes.

In [13]:

```
nasa.groupby('status').agg({'status': 'count'}).orderBy('count(status)').show()
```

Out [13]:

```
+------+-------------+
|status|count(status)|
+------+-------------+
|   400|            5|
|   501|           14|
|   403|           54|
|   500|           62|
|   404|        10845|
|   302|        46573|
|   304|       132627|
|   200|      1701534|
+------+-------------+
```

Assignment

For the assignment, we want you to perform the following analyses on the NASA data set:

1. Find the `average` number of response bytes.
2. Find the percentage of successful requests (a successful request is one that has a status code of 200).
3. Find the `remote_host` that made the highest amount of requests. How many requests did he make?
4. Find the `remote_host` that made the highest amount of failed requests (a failed request is one that has a status code not equal to 200). How many failed requests did he make?
5. What are the top-5 most frequently requested `request_urls`?
6. What are the top-5 `request_urls` that have consumed the most bandwidth?

Bonus Assignment

We are interested in seeing the traffic that the NASA website receives every day. Create a plot that shows the total amount of requests that are made to the NASA website for each day in the month.

We are also interested in seeing the traffic for each hour of the day (i.e. how busy is it in the afternoon vs how busy it is at night). Make a plot showing the total amount of requests that are made to the NASA website aggregated per hour of the day.