Web Server Log Analysis with Apache Spark

This assignment will demonstrate how easy it is to perform web server log analysis with Apache Spark.

Server log analysis is an ideal use case for Spark. It's a very large, common data source and contains a rich set of information. Spark allows you to store your logs in files on disk cheaply, while still providing a quick and simple way to perform data analysis on them. This assignment will show you how to use Apache Spark on real-world text-based production logs and fully harness the power of that data. Log data comes from many sources, such as web, file, and compute servers, application logs, user-generated content, and can be used for monitoring servers, improving business and customer intelligence, building recommendation systems, fraud detection, and much more.

How to complete this assignment:

- *Part 1:* Creating a base RDD and pair RDDs
- *Part 2:* Counting with pair RDDs
- *Part 3:* Finding unique words and a mean value
- *Part 4:* Apply word count to a file

This assignment is broken up into sections with bite-sized examples for demonstrating Spark functionality for log processing. For each problem, you should start by thinking about the algorithm that you will use to *efficiently* process the log in a parallel, distributed manner. This means using the various RDD operations along with lambda functions that are applied at each worker.

Part 1: Apache Web Server Log File Format

The log files that we use for this assignment are in the Apache Common Log Format (CLF). The log file entries produced in CLF will look something like this:

```
127.0.0.1 - - [01/Aug/1995:00:00:01 -0400] "GET /images/launch-logo.gif HTTP/1.0" 200 1839
```

Each part of this log entry is described below.

- `127.0.0.1` This is the IP address (or host name, if available) of the client (remote host) which made the request to the server.
- – The "hyphen" in the output indicates that the requested piece of information (user identity from remote machine) is not available.
- – The "hyphen" in the output indicates that the requested piece of information (user identity from local logon) is not available.
- `[01/Aug/1995:00:00:01 -0400]` The time that the server finished processing the request. The format is: `[day/month/year:hour:minute:second timezone]`
  - day = 2 digits
  - month = 3 letters
  - year = 4 digits
  - hour = 2 digits
  - minute = 2 digits
  - second = 2 digits
  - zone = (+ | -) 4 digits

• `"GET /images/launch-logo.gif HTTP/1.0"` This is the first line of the request string from the client. It consists of a three components: the request method (e.g., `GET`, `POST`, etc.), the endpoint (a Uniform Resource Identifier), and the client protocol version.
• `200` This is the status code that the server sends back to the client. This information is very valuable, because it reveals whether the request resulted in a successful response (codes beginning in 2), a redirection (codes beginning in 3), an error caused by the client (codes beginning in 4), or an error in the server (codes beginning in 5). The full list of possible status codes can be found in the HTTP specification (RFC 2616 section 10).
• `1839` The last entry indicates the size of the object returned to the client, not including the response headers. If no content was returned to the client, this value will be "-" (or sometimes 0).

Note that log files contain information supplied directly by the client, without escaping. Therefore, it is possible for malicious clients to insert control-characters in the log files, *so care must be taken in dealing with raw logs*.

NASA-HTTP Web Server Log
For this assignment, we will use a data set from NASA Kennedy Space Center WWW server in Florida. The full data set is freely available (http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html) and contains two month's of all HTTP requests. We are using a subset that only contains several days worth of requests.

(1a) Parsing Each Log Line
Using the CLF as defined above, we create a regular expression pattern to extract the nine fields of the log line using the Python regular expression search function. The function returns a pair consisting of a Row object and 1. If the log line fails to match the regular expression, the function returns a pair consisting of the log line string and 0. A '-' value in the content size field is cleaned up by substituting it with 0. The function converts the log line's date string into a Python `datetime` object using the given `parse_apache_time` function.

```
>
import re
import datetime

from pyspark.sql import Row

month_map = {'Jan': 1, 'Feb': 2, 'Mar':3, 'Apr':4, 'May':5, 'Jun':6, 'Jul':7,
    'Aug':8,  'Sep': 9, 'Oct':10, 'Nov': 11, 'Dec': 12}

def parse_apache_time(s):
    """ Convert Apache time format into a Python datetime object
    Args:
        s (str): date and time in Apache time format
    Returns:
        datetime: datetime object (ignore timezone for now)
    """
    return datetime.datetime(int(s[7:11]),
                             month_map[s[3:6]],
                             int(s[0:2]),
                             int(s[12:14]),
                             int(s[15:17]),
                             int(s[18:20]))


def parseApacheLogLine(logline):
    """ Parse a line in the Apache Common Log format
    Args:
        logline (str): a line of text in the Apache Common Log format
    Returns:
        tuple: either a dictionary containing the parts of the Apache Access Log and 1,
               or the original invalid log line and 0
    """
    match = re.search(APACHE_ACCESS_LOG_PATTERN, logline)
    if match is None:
        return (logline, 0)
    size_field = match.group(9)
    if size_field == '-':
        size = long(0)
    else:
        size = long(match.group(9))
    return (Row(
        host          = match.group(1),
        client_identd = match.group(2),
        user_id       = match.group(3),
        date_time     = parse_apache_time(match.group(4)),
        method        = match.group(5),
        endpoint      = match.group(6),
        protocol      = match.group(7),
        response_code = int(match.group(8)),
        content_size  = size
    ), 1)

# A regular expression pattern to extract fields from the log line
APACHE_ACCESS_LOG_PATTERN = '^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+)\s*(\S*)" (\d{3}) (\S+)'
```

(1b) Configuration and Initial RDD Creation

We are ready to specify the input log file and create an RDD containing the parsed log file data. Download `NASAlog.txt` file and store it in your Spark directory.

To create the primary RDD that we'll use in the rest of this assignment, we first load the text file using sc.textfile(logFile) to convert each line of the file into an element in an RDD. Next, we use map(parseApacheLogLine) to apply the parse function to each element (that is, a line from the log file) in the RDD and turn each line into a pair Row object. Finally, we cache the RDD in memory since we'll use it throughout this assignment.

```
>
logFile = "NASAlog.txt"

def parseLogs():
    """ Read and parse log file """
    parsed_logs = (sc
                   .textFile(logFile)
                   .map(parseApacheLogLine)
                   .cache())

    access_logs = (parsed_logs
                   .filter(lambda s: s[1] == 1)
                   .map(lambda s: s[0])
                   .cache())

    failed_logs = (parsed_logs
                   .filter(lambda s: s[1] == 0)
                   .map(lambda s: s[0]))
    failed_logs_count = failed_logs.count()
    if failed_logs_count > 0:
        print 'Number of invalid logline: %d' % failed_logs.count()
        for line in failed_logs.take(20):
            print 'Invalid logline: %s' % line

    print 'Read %d lines, successfully parsed %d lines, failed to parse %d lines' % (parsed_logs.count(),
access_logs.count(), failed_logs.count())
    return parsed_logs, access_logs, failed_logs


parsed_logs, access_logs, failed_logs = parseLogs()
```

(1c) Data Cleaning
Notice that there are a large number of log lines that failed to parse. Examine the sample of invalid lines and compare them to the correctly parsed line, an example is included below. Based on your observations, you can alter the `APACHE_ACCESS_LOG_PATTERN` regular expression below so that the failed lines will correctly parse, and rerun `parseLogs()`.

```
wxs6-7.worldaccess.nl - - [02/Jul/1995:08:09:27 -0400] "GET / /   HTTP/1.0" 200 7074
```

If you are not familiar with Python regular expression search function, it would be good to check up on the documentation. One tip that might be useful when you test different regular expressions is to use an online tester like `http://pythex.org` or `http://www.python-regex.com`.

```
>
# This was originally '^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+)\s*(\S*)" (\d{3}) (\S+)'
APACHE_ACCESS_LOG_PATTERN = '^(\S+) (\S+) (\S+).*\[([\w:\/]+\s[+\-]\d{4})\] "(\S+) (\S*)( *\S+ *)*" (\d{3})
(\S+)'

parsed_logs, access_logs, failed_logs = parseLogs()

print failed_logs.count()
0
print parsed_logs.count()
314876
print access_logs.count()
314876
```

Part 2: Sample Analyses on the Web Server Log File

Now that we have an RDD containing the log file as a set of Row objects, we can perform various analyses.

(2a) Example: Content Size Statistics
Let's compute some statistics about the sizes of content being returned by the web server. In particular, we'd like to know what are the average, minimum, and maximum content sizes.

We can compute the statistics by applying a `map` to the `access_logs` RDD. The `lambda` function we want for the map is to extract the `content_size` field from the RDD. The map produces a new

RDD containing only the `content_sizes` (one element for each Row object in the `access_logs` RDD). To compute the minimum and maximum statistics, we can use min() and max() functions on the new RDD. We can compute the average statistic by using the reduce function with a `lambda` function that sums the two inputs, which represent two elements from the new RDD that are being reduced together. The result of the `reduce()` is the total content size from the log and it is to be divided by the number of requests as determined using the count() function on the new RDD.

```
>
# Calculate statistics based on the content size.
content_sizes = access_logs.map(lambda log: log.content_size).cache()
print 'Content Size Avg: %i, Min: %i, Max: %s' % (
    content_sizes.reduce(lambda a, b : a + b) / content_sizes.count(),
    content_sizes.min(),
    content_sizes.max())
```

(2b) Example: Response Code Analysis

Next, lets look at the response codes that appear in the log. As with the content size analysis, first we create a new RDD by using a `lambda` function to extract the `response_code` field from the `access_logs` RDD. The difference here is that we will use a pair tuple instead of just the field itself. Using a pair tuple consisting of the response code and 1 will let us count how many records have a particular response code. Using the new RDD, we perform a reduceByKey function. `reduceByKey` performs a reduce on a per-key basis by applying the `lambda` function to each element, pairwise with the same key. We use the simple `lambda` function of adding the two values. Then, we cache the resulting RDD and create a list by using the take function.

```
>
# Response Code to Count
responseCodeToCount = (access_logs
                       .map(lambda log: (log.response_code, 1))
                       .reduceByKey(lambda a, b : a + b)
                       .cache())
responseCodeToCountList = responseCodeToCount.take(100)

print 'Found %d response codes' % len(responseCodeToCountList)
print 'Response Code Counts: %s' % responseCodeToCountList
```

(2c) Example: Response Code Graphing with `matplotlib`

Let's visualize the results from the last example. We can visualize the results from the last example using matplotlib. First we need to extract the labels and fractions for the graph. We do this with two separate `map` functions with a `lambda` functions. The first `map` function extracts a list of of the response code values, and the second `map` function extracts a list of the per response code counts divided by the total size of the access logs. Next, we create a figure with `figure()` constructor and use the `pie()` method to create the pie plot.

```
>
labels = responseCodeToCount.map(lambda (x, y): x).collect()
print labels

count = access_logs.count()
fracs = responseCodeToCount.map(lambda (x, y): (float(y) / count)).collect()
print fracs

import matplotlib.pyplot as plt
```

```
def pie_pct_format(value):
    """ Determine the appropriate format string for the pie chart percentage label
    Args:
        value: value of the pie slice
    Returns:
        str: formated string label; if the slice is too small to fit, returns an empty string for label
    """
    return '' if value < 7 else '%.0f%%' % value


fig = plt.figure(figsize=(4.5, 4.5), facecolor='white', edgecolor='white')
colors = ['yellowgreen', 'lightskyblue', 'gold', 'purple', 'lightcoral', 'yellow', 'black']
explode = (0.05, 0.05, 0.1, 0, 0, 0, 0)
patches, texts, autotexts = plt.pie(fracs, labels=labels, colors=colors,
                                    explode=explode, autopct=pie_pct_format,
                                    shadow=False,  startangle=125)
for text, autotext in zip(texts, autotexts):
    if autotext.get_text() == '':
        text.set_text('')  # If the slice is small to fit, don't show a text label
plt.legend(labels, loc=(0.80, -0.1), shadow=True)
display(fig)
```

(2d) Example: Frequent Hosts
Let's look at hosts that have accessed the server multiple times (e.g., more than ten times). As with the response code analysis in (2b), first we create a new RDD by using a `lambda` function to extract the `host` field from the `access_logs` RDD using a pair tuple consisting of the host and 1 which will let us count how many records were created by a particular host's request. Using the new RDD, we perform a `reduceByKey` function with a `lambda` function that adds the two values. We then filter the result based on the count of accesses by each host (the second element of each pair) being greater than ten. Next, we extract the host name by performing a `map` with a `lambda` function that returns the first element of each pair. Finally, we extract 20 elements from the resulting RDD - *note that the choice of which elements are returned is not guaranteed to be deterministic*.

```
>
# Any hosts that has accessed the server more than 10 times.
hostCountPairTuple = access_logs.map(lambda log: (log.host, 1))

hostSum = hostCountPairTuple.reduceByKey(lambda a, b : a + b)

hostMoreThan10 = hostSum.filter(lambda s: s[1] > 10)

hostsPick20 = (hostMoreThan10
               .map(lambda s: s[0])
               .take(20))

print 'Any 20 hosts that have accessed more than 10 times: %s' % hostsPick20
```

(2e) Example: Visualizing Endpoints
Now, let's visualize the number of hits to endpoints (URIs) in the log. To perform this task, we first create a new RDD by using a `lambda` function to extract the `endpoint` field from the `access_logs` RDD using a pair tuple consisting of the endpoint and 1 which will let us count how many records

were created by a particular host's request. Using the new RDD, we perform a `reduceByKey` function with a `lambda` function that adds the two values. We then cache the results.

Next we visualize the results using `matplotlib`. We previously imported the `matplotlib.pyplot` library, so we do not need to import it again. We perform two separate `map` functions with `lambda` functions. The first `map` function extracts a list of endpoint values, and the second `map` function extracts a list of the visits per endpoint values. Next, we create a figure with `figure()` constructor, set various features of the plot (axis limits, grid lines, and labels), and use the `plot()` method to create the line plot.

```
>
endpoints = (access_logs
            .map(lambda log: (log.endpoint, 1))
            .reduceByKey(lambda a, b : a + b)
            .cache())
ends = endpoints.map(lambda (x, y): x).collect()
counts = endpoints.map(lambda (x, y): y).collect()

fig = plt.figure(figsize=(8,4.2), facecolor='white', edgecolor='white')
plt.axis([0, len(ends), 0, max(counts)])
plt.grid(b=True, which='major', axis='y')
plt.xlabel('Endpoints')
plt.ylabel('Number of Hits')
plt.plot(counts)
display(fig)
```

(2f) Example: Top Endpoints
For the final example, we'll look at the top endpoints (URIs) in the log. To determine them, we first create a new RDD by using a `lambda` function to extract the `endpoint` field from the `access_logs` RDD using a pair tuple consisting of the endpoint and 1 which will let us count how many records were created by a particular host's request. Using the new RDD, we perform a `reduceByKey` function with a `lambda` function that adds the two values. We then extract the top ten endpoints by performing a takeOrdered with a value of 10 and a `lambda` function that multiplies the count (the second element of each pair) by -1 to create a sorted list with the top endpoints at the bottom.

```
>
# Top Endpoints
endpointCounts = (access_logs
                .map(lambda log: (log.endpoint, 1))
                .reduceByKey(lambda a, b : a + b))

topEndpoints = endpointCounts.takeOrdered(10, lambda s: -1 * s[1])

print 'Top Ten Endpoints: %s' % topEndpoints
Top Ten Endpoints: [(u'/images/NASA-logosmall.gif', 19065), (u'/images/KSC-logosmall.gif', 16502), (u'/shuttle/countdown/count.gif', 12230), (u'/shuttle/countdown/', 11992), (u'/images/MOSAIC-logosmall.gif', 7912), (u'/images/ksclogo-medium.gif', 7902), (u'/images/USA-logosmall.gif', 7887), (u'/images/WORLD-logosmall.gif', 7729), (u'/shuttle/missions/sts-71/images/images.html', 6718), (u'/shuttle/missions/sts-71/sts-71-patch-small.gif', 6628)]
```

Part 3: Analyzing Web Server Log File

Now it is your turn to perform analyses on web server log files.

(3a) Exercise: Top Ten Error Endpoints
What are the top ten endpoints which did not have return code 200? Create a sorted list containing top ten endpoints and the number of times that they were accessed with non-200 return code.

Think about the steps that you need to perform to determine which endpoints did not have a 200 return code, how you will uniquely count those endpoints, and sort the list.

You might want to refer back to the previous assignment (Word Count) for insights.

```
>
# TODO: Replace <FILL IN> with appropriate code
# HINT: Each of these <FILL IN> below could be completed with a single transformation or action.
# You are welcome to structure your solution in a different way, so long as
# you ensure the variables used in the next Test section are defined (ie. endpointSum, topTenErrURLs).

not200 = access_logs.<FILL IN>

endpointCountPairTuple = not200.<FILL IN>

endpointSum = endpointCountPairTuple.<FILL IN>

topTenErrURLs = endpointSum.<FILL IN>

print endpointSum.count()
4690

print 'Top Ten failed URLs: %s' % topTenErrURLs
Top Ten failed URLs: [(u'/images/NASA-logosmall.gif', 2404), (u'/images/KSC-logosmall.gif', 1806), (u'/shuttle/countdown/', 944), (u'/images/MOSAIC-logosmall.gif', 845), (u'/images/USA-logosmall.gif', 830), (u'/images/WORLD-logosmall.gif', 811), (u'/shuttle/countdown/count.gif', 777), (u'/images/ksclogo-medium.gif', 751), (u'/shuttle/countdown/liftoff.html', 590), (u'/shuttle/missions/sts-71/sts-71-patch-small.gif', 543)]
```

(3b) Exercise: Number of Unique Hosts
How many unique hosts are there in the entire log? Think about the steps that you need to perform to count the number of different hosts in the log.

```
>
# TODO: Replace <FILL IN> with appropriate code
# HINT: Do you recall the tips from (3a)? Each of these <FILL IN> could be an transformation or action.

hosts = access_logs.<FILL IN>

uniqueHosts = hosts.<FILL IN>

uniqueHostCount = uniqueHosts.<FILL IN>

print 'Unique hosts: %d' % uniqueHostCount
22013
```

(3c) Exercise: Number of Unique Daily Hosts

For an advanced exercise, let's determine the number of unique hosts in the entire log on a day-by-day basis. This computation will give us counts of the number of unique daily hosts. We'd like a list sorted by increasing day of the month which includes the day of the month and the associated number of unique hosts for that day (Hint: use `set` of Python to create an unordered list with no duplicate items OR `distinct()` of PySpark). Depending on the implementation, your code structure may look different from the following code. But make sure that the result of print statement is the same as the following.

Think about the steps that you need to perform to count the number of different hosts that make requests each day. *Since the log only covers a single month, you can ignore the month.*

```
>
# TODO: Replace <FILL IN> with appropriate code

dayToHostPairTuple = access_logs.<FILL IN>

dayGroupedHosts = dayToHostPairTuple.<FILL IN>

dayHostCount = dayGroupedHosts.<FILL IN>

dailyHosts = (dayHostCount
          <FILL IN>)
dailyHostsList = dailyHosts.takeOrdered(4)

print 'Unique hosts per day: %s' % dailyHostsList
Unique hosts per day: [(2, 4859), (3, 7336), (4, 5524), (5, 7383)]

dailyHosts.cache()
```

(3d) Exercise: Visualizing the Number of Unique Daily Hosts
Using the results from the previous exercise, use `matplotlib` to plot a "Line" graph of the unique hosts requests by day. `daysWithHosts` should be a list of days and `hosts` should be a list of number of unique hosts for each corresponding day.

```
>
# TODO: Replace <FILL IN> with appropriate code

daysWithHosts = dailyHosts.<FILL IN>
hosts = dailyHosts.<FILL IN>

print daysWithHosts
[2, 3, 4, 5]

print hosts
[4859, 7336, 5524, 7383]

fig = plt.figure(figsize=(8,4.5), facecolor='white', edgecolor='white')
plt.axis([min(daysWithHosts), max(daysWithHosts), 0, max(hosts)+500])
plt.grid(b=True, which='major', axis='y')
plt.xlabel('Day')
plt.ylabel('Hosts')
plt.plot(daysWithHosts, hosts)
display(fig)
```

(3e) Exercise: Average Number of Daily Requests per Hosts
Next, let's determine the average number of requests on a day-by-day basis. We'd like a list by increasing day of the month and the associated average number of requests per host for that day. To compute the average number of requests per host, get the total number of request across all hosts and divide that by the number of unique hosts. (Hint: use `join()` of PySpark) *Since the log only covers a single month, you can skip checking for the month. Also to keep it simple, when calculating the approximate average use the integer value - you do not need to upcast to float.* Depending on the implementation, your code structure may look different from the following code. But make sure that the result of print statement is the same as the following.

```
>
# TODO: Replace <FILL IN> with appropriate code

dayAndHostTuple = access_logs.<FILL IN>

groupedByDay = dayAndHostTuple.<FILL IN>

sortedByDay = groupedByDay.<FILL IN>

avgDailyReqPerHost = (sortedByDay
                     <FILL IN>)
avgDailyReqPerHostList = avgDailyReqPerHost.take(4)

print 'Average number of daily requests per Hosts is %s' % avgDailyReqPerHostList
Average number of daily requests per Hosts is [(2, 12), (3, 12), (4, 12), (5, 12)]

avgDailyReqPerHost.cache()
```

(3f) Exercise: Visualizing the Average Daily Requests per Unique Host
Using the result `avgDailyReqPerHost` from the previous exercise, use `matplotlib` to plot a "Line" graph of the average daily requests per unique host by day. `daysWithAvg` should be a list of days and `avgs` should be a list of average daily requests per unique hosts for each corresponding day.

```
>
# TODO: Replace <FILL IN> with appropriate code

daysWithAvg = avgDailyReqPerHost.<FILL IN>
avgs = avgDailyReqPerHost.<FILL IN>

print daysWithAvg
[2, 3, 4, 5]

print avgs
[12, 12, 12, 12]

fig = plt.figure(figsize=(8,4.2), facecolor='white', edgecolor='white')
plt.axis([0, max(daysWithAvg), 0, max(avgs)+2])
plt.grid(b=True, which='major', axis='y')
plt.xlabel('Day')
plt.ylabel('Average')
plt.plot(daysWithAvg, avgs)
display(fig)
```

Part 4: Exploring 404 Response Codes

Let's drill down and explore the error 404 response code records. 404 errors are returned when an endpoint is not found by the server (i.e., a missing page or object).

(4a) Exercise: Counting 404 Response Codes
Create a RDD containing only log records with a 404 response code. How many 404 records are in the log?

```
>
# TODO: Replace <FILL IN> with appropriate code

badRecords = (access_logs
            <FILL IN>)

print 'Found %d 404 URLs' % badRecords.count()
Found 1621 404 URLs

badRecords.cache()
```

(4b) Exercise: Listing 404 Response Code Records
Using the RDD containing only log records with a 404 response code that you cached in part (4a), print out a list up to 40 *distinct* endpoints that generate 404 errors - *no endpoint should appear more than once in your list*.

```
>
# TODO: Replace <FILL IN> with appropriate code

badEndpoints = badRecords.<FILL IN>

badUniqueEndpoints = badEndpoints.<FILL IN>

badUniqueEndpointsPick40 = badUniqueEndpoints.takeOrdered(40)

print '404 URLS: %s' % badUniqueEndpointsPick40
404 URLS: [u'/%3A//spacelink.msfc.nasa.gov', u'/%7Eadverts/ibm/ad1/banner.gif', u'//history/apollo/apollo-13/
apollo-13html', u'//spacelink.msfc.nasa.gov/', u'//spacelink.msfc.nasa.gov:70/00/About.Spacelink/
File.Transfer.Protocols', u'/11/history/apollo/images/', u'/://spacelink.msfc.nasa.gov', u'/:/
spacelink.msfc.nasa.gov', u'/DataSources/MetData.html', u'/Government/Research_Labs/NASA/
Kennedy_Space_Center/', u'/HISTOTY/APOLLO/APOLLO-13/', u'/KSC.', u'/KSC.HTML', u'/KSC.HTML/', u'/
LDAR/LDARhp.html', u'/NASA_TV/NASA_TV.html', u'/SDG/Experimental/demoweb/return.gif', u'/SHUTTLE', u'/
SHUTTLE/COUNTDOWN/', u'/SHUTTLE/MISSION/STS-71/IMAGES/IMAGES.HTML', u'/SHUTTLE/MISSIONS/
STS-71/IMAGES/IMAGES.HTML', u'/Science/Space/Missions/Magellan_Mission_to_Venus/', u'/Tools/Bars/
Front/main_bar.gif', u'/Tools/Icons/Help/hw.embossed.grey.gif', u'/apollo-13.html', u'/att.net/dir800/', u'/
bann04.gif', u'/base-ops/procurement/kscbus.htm', u'/bigspot.com', u'/cleve.net/cnnavbar.gif', u'/daily/ecs/
s71e0003.jpg', u'/daily/ecs/s71e0004.jpg', u'/daily/esc/', u'/daily/esc/s71e0001.jpg', u'/de/systems.html', u'/demo/',
u'/demos/', u'/elv/DELTA/uncons.htm', u'/elv/DOCS/LLV_DATE.HTM', u'/elv/SCOUT/elvhead2.gif']

print len(set(badUniqueEndpointsPick40))
40
```

(4c) Exercise: Listing the Top Twenty 404 Response Code Endpoints
Using the RDD containing only log records with a 404 response code that you cached in part (4a), print out a list of the top twenty endpoints that generate the most 404 errors. *Remember, top endpoints should be in sorted order*.

```
>
# TODO: Replace <FILL IN> with appropriate code

badEndpointsCountPairTuple = badRecords.<FILL IN>

badEndpointsSum = badEndpointsCountPairTuple.<FILL IN>

badEndpointsTop20 = badEndpointsSum.<FILL IN>

print 'Top Twenty 404 URLs: %s' % badEndpointsTop20
```

```
Top Twenty 404 URLs: [(u'/pub/winvn/readme.txt', 84), (u'/pub/winvn/release.txt', 82), (u'/history/apollo/
publications/sp-350/sp-350.txt~', 67), (u'/shuttle/resources/orbiters/atlantis.gif', 65), (u'/shuttle/missions/sts-71/
images/KSC-95EC-0916.txt', 55), (u'/://spacelink.msfc.nasa.gov', 47), (u'/history/apollo/a-001/a-001-patch-
small.gif', 46), (u'/history/apollo/pad-abort-test-1/pad-abort-test-1-patch-small.gif', 44), (u'/history/apollo/sa-1/
sa-1-patch-small.gif', 42), (u'/history/apollo/apollo-13.html', 36), (u'/images/crawlerway-logo.gif', 31), (u'/shuttle/
missions/sts-68/ksc-upclose.gif', 29), (u'/histroy/apollo-13/apollo-13.html', 20), (u'/history/apollo/sa-9/sa-9-patch-
small.gif', 19), (u'/shuttle/missions/technology/sts-newsref/stsref-toc.html', 19), (u'/shuttle/resources/orbiters/
challenger.gif', 18), (u'/shuttle/technology/images/tps_mods-small.gif', 14), (u'/history/apollo/images/little-joe.jpg',
13), (u'/persons/astronauts/a-to-d/conradCC.txt', 12), (u'/pub', 12)]
```

(4d) Exercise: Listing the Top Twenty-Five 404 Response Code Hosts
Instead of looking at the endpoints that generated 404 errors, let's look at the hosts that encountered 404 errors. Using the RDD containing only log records with a 404 response code that you cached in part (4a), print out a list of the top twenty-five hosts that generate the most 404 errors.

```
>
# TODO: Replace <FILL IN> with appropriate code

errHostsCountPairTuple = badRecords.<FILL IN>

errHostsSum = errHostsCountPairTuple.<FILL IN>

errHostsTop25 = errHostsSum.<FILL IN>

print 'Top 25 hosts that generated errors: %s' % errHostsTop25
```

```
Top 25 hosts that generated errors: [(u'piweba3y.prodigy.com', 29), (u'128.158.48.26', 26),
(u'espresso.sd.inri.com', 25), (u'www-b6.proxy.aol.com', 22), (u'ch025.chance.berkeley.edu', 17), (u'www-
d4.proxy.aol.com', 15), (u'ffong-sl.cc.emory.edu', 15), (u'alyssa.prodigy.com', 14), (u'cpcug.org', 14),
(u'advantis.vnet.ibm.com', 13), (u'128.158.32.107', 12), (u'sta07.oit.unc.edu', 12), (u'dialup-2-89.gw.umn.edu',
11), (u'www-b5.proxy.aol.com', 11), (u'www-d1.proxy.aol.com', 10), (u'ten-nash.ten.k12.tn.us', 10),
(u'dd13-025.compuserve.com', 10), (u'w20-575-4.mit.edu', 10), (u'disc.dna.affrc.go.jp', 9), (u'ix-
bal2-14.ix.netcom.com', 9), (u'wstabnow.gsfc.nasa.gov', 9), (u'www-b1.proxy.aol.com', 8),
(u'dawn14.cs.berkeley.edu', 8), (u'gatekeeper.mitre.org', 8), (u'204.64.145.73', 8)]
```

(4e) Exercise: Listing 404 Response Codes per Day

Let's explore the 404 records temporally. Break down the 404 requests by day and get the daily counts sorted by day as a list. *Since the log only covers a single month, you can ignore the month in your checks*.

```
>
# TODO: Replace <FILL IN> with appropriate code

errDateCountPairTuple = badRecords.<FILL IN>

errDateSum = errDateCountPairTuple.<FILL IN>

errDateSorted = (errDateSum
             <FILL IN>)

errByDate = errDateSorted.<FILL IN>

print '404 Errors by day: %s' % errByDate
404 Errors by day: [(2, 291), (3, 474), (4, 359), (5, 497)]

errDateSorted.cache()
```

(4f) Exercise: Visualizing the 404 Response Codes by Day
Using the results from the previous exercise, use `matplotlib` to plot a “Line" graph of the 404 response codes by day.

```
>
# TODO: Replace <FILL IN> with appropriate code

daysWithErrors404 = errDateSorted.<FILL IN>
errors404ByDay = errDateSorted.<FILL IN>

fig = plt.figure(figsize=(8,4.2), facecolor='white', edgecolor='white')
plt.axis([0, max(daysWithErrors404), 0, max(errors404ByDay)])
plt.grid(b=True, which='major', axis='y')
plt.xlabel('Day')
plt.ylabel('404 Errors')
plt.plot(daysWithErrors404, errors404ByDay)
display(fig)
```

(4g) Exercise: Hourly 404 Response Codes
Using the RDD `badRecords` you cached in the part (4a) and by hour of the day and in increasing order, create an RDD containing how many requests had a 404 return code for each hour of the day (midnight starts at 0). Cache the resulting RDD `hourRecordsSorted` and print that as a list.

```
>
# TODO: Replace <FILL IN> with appropriate code

hourCountPairTuple = badRecords.<FILL IN>

hourRecordsSum = hourCountPairTuple.<FILL IN>

hourRecordsSorted = (hourRecordsSum
               <FILL IN>)
```

```
errHourList = hourRecordsSorted.<FILL IN>

print '404 Errors by hour: %s' % errHourList
404 Errors by hour: [(0, 72), (1, 77), (2, 37), (3, 30), (4, 13), (5, 22), (6, 26), (7, 42), (8, 72), (9, 104), (10, 64), (11,
68), (12, 69), (13, 52), (14, 98), (15, 99), (16, 121), (17, 76), (18, 101), (19, 54), (20, 58), (21, 52), (22, 73), (23,
141)]

hourRecordsSorted.cache()
```

(4h) Exercise: Visualizing the 404 Response Codes by Hour
Using the results from the previous exercise, use `matplotlib` to plot a “Line" graph of the 404 response codes by hour.

```
>
# TODO: Replace <FILL IN> with appropriate code

hoursWithErrors404 = hourRecordsSorted.<FILL IN>
errors404ByHours = hourRecordsSorted.<FILL IN>

fig = plt.figure(figsize=(8,4.2), facecolor='white', edgecolor='white')
plt.axis([0, max(hoursWithErrors404), 0, max(errors404ByHours)])
plt.grid(b=True, which='major', axis='y')
plt.xlabel('Hour')
plt.ylabel('404 Errors')
plt.plot(hoursWithErrors404, errors404ByHours)
display(fig)
```