**Web Server Log Analysis with Apache Spark**

It consists of 5 parts:

* *Part 1:* Introduction and Imports
* *Part 2:* Exploratory Data Analysis
* *Part 3*: Analysis Walk-Through on the Web Server Log File
* *Part 4*: Analyzing Web Server Log File
* *Part 5*: Exploring 404 Response Codes

Let's import some of the libraries we'll need:

* re: The regular expression library
* datetime: Date and time functions
* Test: Our Databricks test helper library

**import re**

**import datetime**

**from databricks\_test\_helper import Test**

# Quick test of the datetime library

**print 'This was last run on: {0}'.format(datetime.datetime.now())**

You can use Python's [dir()](https://docs.python.org/2/library/functions.html?highlight=dir" \l "dir) function to get a list of all the attributes (including methods) accessible through the sqlContext object.

# List sqlContext's attributes

**dir(sqlContext)**

Let's begin looking at our data. For this lab, we will use a data set from NASA Kennedy Space Center web server in Florida. The full data set is freely available at<http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html>, and it contains all HTTP requests for two months. We are using a subset that only contains several days' worth of requests. The log file has already been downloaded for you.

# Specify path to downloaded log file

**import sys**

**import os**

**log\_file\_path = 'dbfs:/' + os.path.join('databricks-datasets', 'cs100', 'lab2', 'data-001', 'apache.access.log.PROJECT')**

Now that we have the path to the file, let's load it into a DataFrame. We'll do this in steps. First, we'll use sqlContext.read.text() to read the text file. This will produce a DataFrame with a single string column called value.

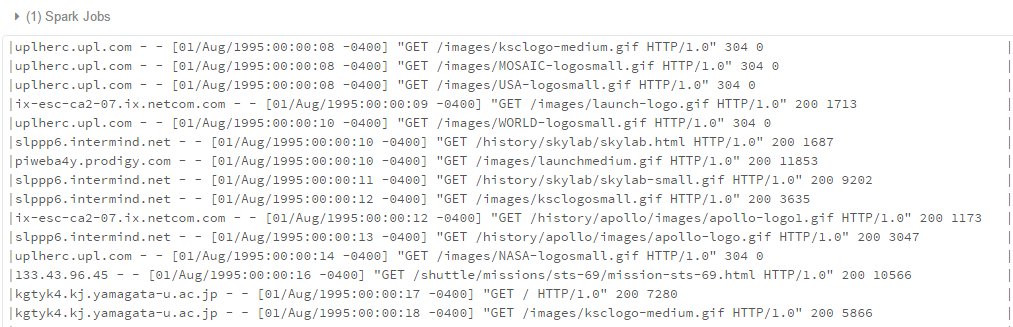
**base\_df = sqlContext.read.text(log\_file\_path)**

**# Let's look at the schema**

**base\_df.printSchema()**

Let's take a look at some of the data.

**base\_df.show(truncate=False)**



f you're familiar with web servers at all, you'll recognize that this is in [Common Log Format](https://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format). The fields are:

*remotehost rfc931 authuser [date] "request" status bytes*

| **field** | **meaning** |
| --- | --- |
| *remotehost* | Remote hostname (or IP number if DNS hostname is not available). |
| *rfc931* | The remote logname of the user. We don't really care about this field. |
| *authuser* | The username of the remote user, as authenticated by the HTTP server. |
| *[date]* | The date and time of the request. |
| *"request"* | The request, exactly as it came from the browser or client. |
| *status* | The HTTP status code the server sent back to the client. |
| *bytes* | The number of bytes (Content-Length) transferred to the client. |

Next, we have to parse it into individual columns. We'll use the special built-in [regexp\_extract()](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html" \l "pyspark.sql.functions.regexp_extract) function to do the parsing. This function matches a column against a regular expression with one or more [capture groups](http://regexone.com/lesson/capturing_groups) and allows you to extract one of the matched groups. We'll use one regular expression for each field we wish to extract.

If you can't read these regular expressions, don't worry. Trust us: They work. If you find regular expressions confusing (and they certainly *can* be), and you want to learn more about them, start with the [RegexOne web site](http://regexone.com/). You might also find [*Regular Expressions Cookbook*](http://shop.oreilly.com/product/0636920023630.do), by Jan Goyvaerts and Steven Levithan, to be helpful.

*Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.* (attributed to Jamie Zawinski)

**from pyspark.sql.functions import split, regexp\_extract**

**split\_df = base\_df.select(regexp\_extract('value', r'^([^\s]+\s)', 1).alias('host'),**

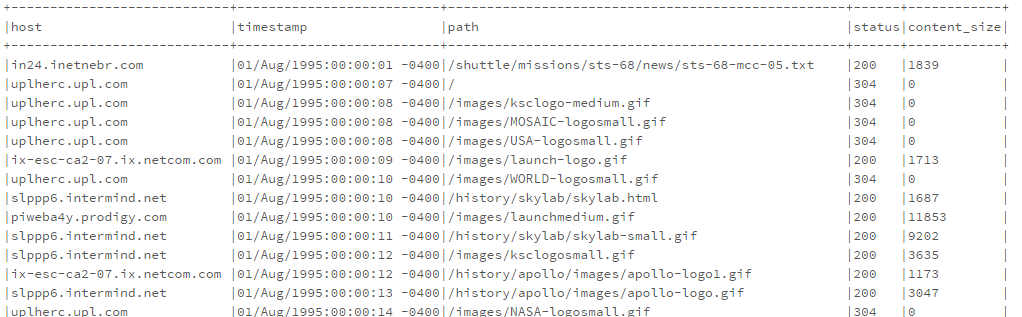
**regexp\_extract('value', r'^.\*\[(\d\d/\w{3}/\d{4}:\d{2}:\d{2}:\d{2} -\d{4})]', 1).alias('timestamp'),**

**regexp\_extract('value', r'^.\*"\w+\s+([^\s]+)\s+HTTP.\*"', 1).alias('path'),**

**regexp\_extract('value', r'^.\*"\s+([^\s]+)', 1).cast('integer').alias('status'),**

**regexp\_extract('value', r'^.\*\s+(\d+)$', 1).cast('integer').alias('content\_size'))**

**split\_df.show(truncate=False)**



Let's see how well our parsing logic worked. First, let's verify that there are no null rows in the original data set.

**base\_df.filter(base\_df['value'].isNull()).count()**



**bad\_rows\_df = split\_df.filter(split\_df['host'].isNull() |**

**split\_df['timestamp'].isNull() |**

**split\_df['path'].isNull() |**

**split\_df['status'].isNull() |**

**split\_df['content\_size'].isNull())**

**bad\_rows\_df.count()**

Not good. We have some null values. Something went wrong. Which columns are affected?

(Note: This approach is adapted from an [excellent answer](http://stackoverflow.com/a/33901312) on StackOverflow.)

**from pyspark.sql.functions import col, sum**

**def count\_null(col\_name):**

**return sum(col(col\_name).isNull().cast('integer')).alias(col\_name)**

**# Build up a list of column expressions, one per column.**

**# This could be done in one line with a Python list comprehension, but we're keeping**

**# it simple for those who don't know Python very well.**

**exprs = []**

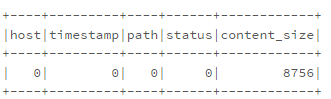
**for col\_name in split\_df.columns:**

**exprs.append(count\_null(col\_name))**

**# Run the aggregation. The \*exprs converts the list of expressions into**

**# variable function arguments.**

**split\_df.agg(\*exprs).show()**



Okay, they're all in the content\_size column. Let's see if we can figure out what's wrong. Our original parsing regular expression for that column was:

regexp\_extract('value', r'^.\*\s+(\d+)$', 1).cast('integer').alias('content\_size')

The \d+ selects one or more digits at the end of the input line. Is it possible there are lines without a valid content size? Or is there something wrong with our regular expression? Let's see if there are any lines that do not end with one or more digits.

**Note**: In the expression below, ~ means "not"

**bad\_content\_size\_df = base\_df.filter(~ base\_df['value'].rlike(r'\d+$'))**

**bad\_content\_size\_df.count()**

That's it! The count matches the number of rows in bad\_rows\_df exactly.

Let's take a look at some of the bad column values. Since it's possible that the rows end in extra white space, we'll tack a marker character onto the end of each line, to make it easier to see trailing white space

**from pyspark.sql.functions import lit, concat**

**bad\_content\_size\_df.select(concat(bad\_content\_size\_df['value'], lit('\*'))).show(truncate=False)**



Ah. The bad rows correspond to error results, where no content was sent back and the server emitted a "-" for the content\_size field. Since we don't want to discard those rows from our analysis, let's map them to 0.

The easiest solution is to replace the null values in split\_df with 0. The DataFrame API provides a set of functions and fields specifically designed for working with null values, among them:

* [fillna()](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.fillna), which fills null values with specified non-null values.
* [na](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.na), which returns a [DataFrameNaFunctions](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html" \l "pyspark.sql.DataFrameNaFunctions) object with many functions for operating on null columns.

We'll use fillna(), because it's simple. There are several ways to invoke this function. The easiest is just to replace *all* null columns with known values. But, for safety, it's better to pass a Python dictionary containing (column\_name, value) mappings. That's what we'll do.

**# Replace all null content\_size values with 0.**

**cleaned\_df = split\_df.na.fill({'content\_size': 0})**

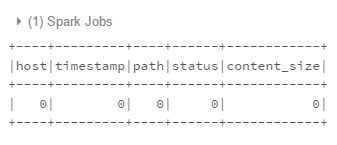
**# Ensure that there are no nulls left.**

**exprs = []**

**for col\_name in cleaned\_df.columns:**

**exprs.append(count\_null(col\_name))**

**cleaned\_df.agg(\*exprs).show()**



Okay, now that we have a clean, parsed DataFrame, we have to parse the timestamp field into an actual timestamp. The Common Log Format time is somewhat non-standard. A User-Defined Function (UDF) is the most straightforward way to parse it.

**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\_clf\_time(s):**

**""" Convert Common Log time format into a Python datetime object**

**Args:**

**s (str): date and time in Apache time format [dd/mmm/yyyy:hh:mm:ss (+/-)zzzz]**

**Returns:**

**a string suitable for passing to CAST('timestamp')**

**"""**

**# NOTE: We're ignoring time zone here. In a production application, you'd want to handle that.**

**return "{0:04d}-{1:02d}-{2:02d} {3:02d}:{4:02d}:{5:02d}".format(**

**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])**

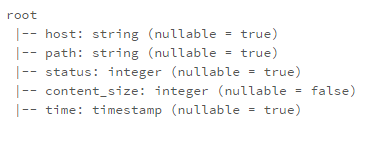
**)**

**u\_parse\_time = udf(parse\_clf\_time)**

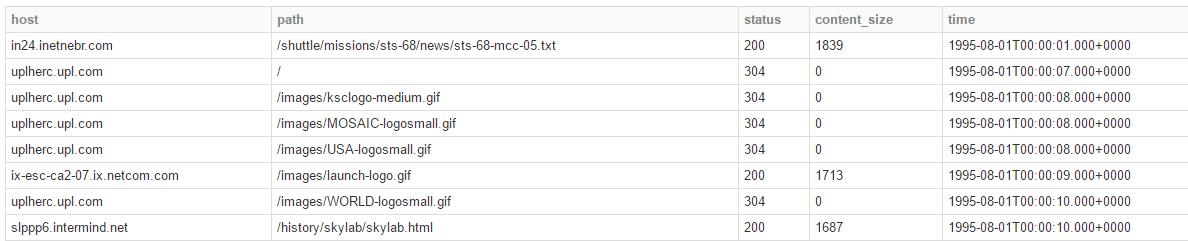
**logs\_df = cleaned\_df.select('\*', u\_parse\_time(cleaned\_df['timestamp']).cast('timestamp').alias('time')).drop('timestamp')**

**total\_log\_entries = logs\_df.count()**

**logs\_df.printSchema()**



**display(logs\_df)**



Let's cache logs\_df. We're going to be using it quite a bit from here forward.

**logs\_df.cache()**

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

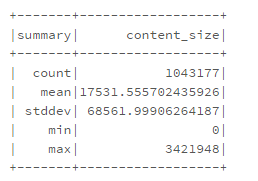
Let's compute some statistics about the sizes of content being returned by the web server. In particular, we'd like to know what are the average, minimum, and maximum content sizes.

We can compute the statistics by calling .describe() on the content\_size column of logs\_df. The .describe() function returns the count, mean, stddev, min, and max of a given column.

**# Calculate statistics based on the content size.**

**content\_size\_summary\_df = logs\_df.describe(['content\_size'])**

**content\_size\_summary\_df.show()**



Alternatively, we can use SQL to directly calculate these statistics. You can explore the many useful functions within the pyspark.sql.functions module in the [documentation](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions).

After we apply the .agg() function, we call .first() to extract the first value, which is equivalent to .take(1)[0].

**from pyspark.sql import functions as sqlFunctions**

**content\_size\_stats = (logs\_df**

**.agg(sqlFunctions.min(logs\_df['content\_size']),**

**sqlFunctions.avg(logs\_df['content\_size']),**

**sqlFunctions.max(logs\_df['content\_size']))**

**.first())**

**print 'Using SQL functions:'**

**print 'Content Size Avg: {1:,.2f}; Min: {0:.2f}; Max: {2:,.0f}'.format(\*content\_size\_stats)**

Using SQL functions: Content Size Avg: 17,531.56; Min: 0.00; Max: 3,421,948

Next, let's look at the status values that appear in the log. We want to know which status values appear in the data and how many times. We again start with logs\_df, then group by the status column, apply the .count() aggregation function, and sort by the status column.

**status\_to\_count\_df =(logs\_df**

**.groupBy('status')**

**.count()**

**.sort('status')**

**.cache())**

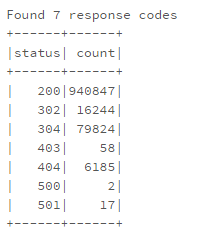
**status\_to\_count\_length = status\_to\_count\_df.count()**

**print 'Found %d response codes' % status\_to\_count\_length**

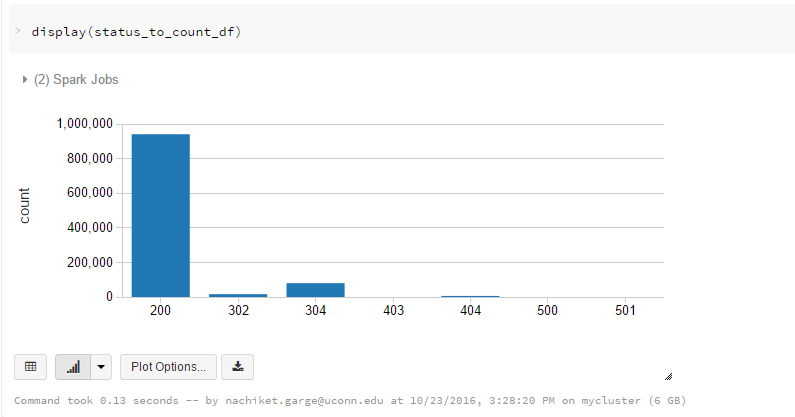
**status\_to\_count\_df.show()**

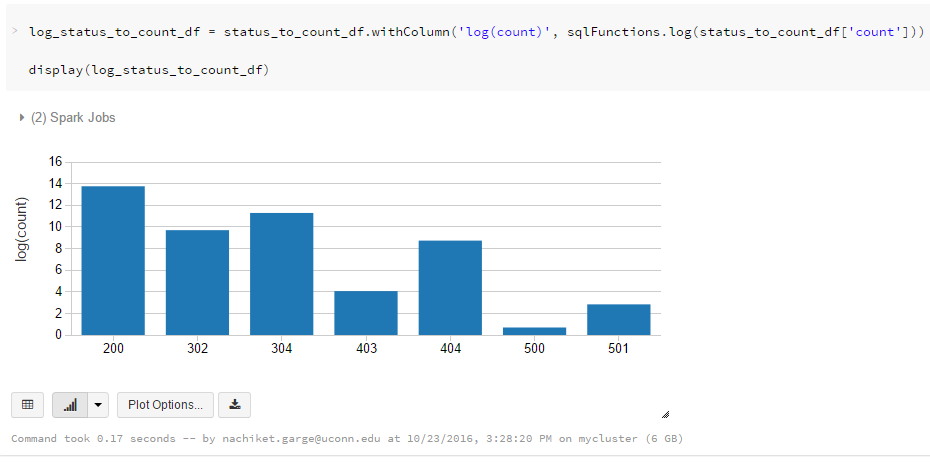
**assert status\_to\_count\_length == 7**

**assert status\_to\_count\_df.take(100) == [(200, 940847), (302, 16244), (304, 79824), (403, 58), (404, 6185), (500, 2), (501, 17)]**



Now, let's visualize the results from the last example. We can use the built-in display() function to show a bar chart of the count for each response code. After running this cell, select the bar graph option, and then use "Plot Options..." and drag status to the key entry field and drag count to the value entry field. See the diagram, below, for an example.





Let's look at hosts that have accessed the server frequently (e.g., more than ten times). As with the response code analysis in (3b), we create a new DataFrame by groupingsuccessLogsDF by the 'host' column and aggregating by count.

We then filter the result based on the count of accesses by each host being greater than ten. Then, we select the 'host' column and show 20 elements from the result.

**# Any hosts that has accessed the server more than 10 times.**

**host\_sum\_df =(logs\_df**

**.groupBy('host')**

**.count())**

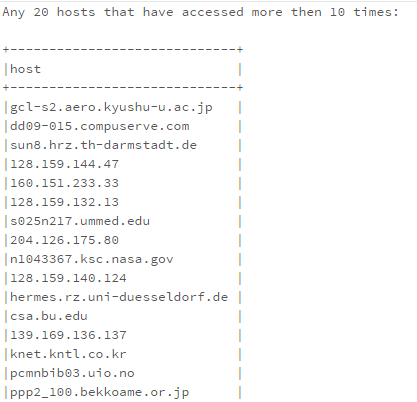
**host\_more\_than\_10\_df = (host\_sum\_df**

**.filter(host\_sum\_df['count'] > 10)**

**.select(host\_sum\_df['host']))**

**print 'Any 20 hosts that have accessed more then 10 times:\n'**

**host\_more\_than\_10\_df.show(truncate=False)**



For the final example, we'll find the top paths (URIs) in the log. Because we sorted paths\_df for plotting, all we need to do is call .show() and pass in n=10 and truncate=Falseas the parameters to show the top ten paths without truncating.

**# Top Paths**

**print 'Top Ten Paths:'**

**paths\_df.show(n=10, truncate=False)**

**expected = [**

**(u'/images/NASA-logosmall.gif', 59666),**

**(u'/images/KSC-logosmall.gif', 50420),**

**(u'/images/MOSAIC-logosmall.gif', 43831),**

**(u'/images/USA-logosmall.gif', 43604),**

**(u'/images/WORLD-logosmall.gif', 43217),**

**(u'/images/ksclogo-medium.gif', 41267),**

**(u'/ksc.html', 28536),**

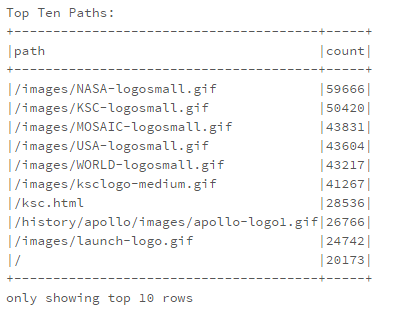
**(u'/history/apollo/images/apollo-logo1.gif', 26766),**

**(u'/images/launch-logo.gif', 24742),**

**(u'/', 20173)**

**]**

**assert paths\_df.take(10) == expected, 'incorrect Top Ten Paths'**



What are the top ten paths which did not have return code 200? Create a sorted list containing the paths and the number of times that they were accessed with a non-200 return code and show the top ten.

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

**# 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**

**# DataFrame containing all accesses that did not return a code 200**

**from pyspark.sql.functions import desc**

**not200DF = logs\_df.filter(logs\_df['status'] != '200')**

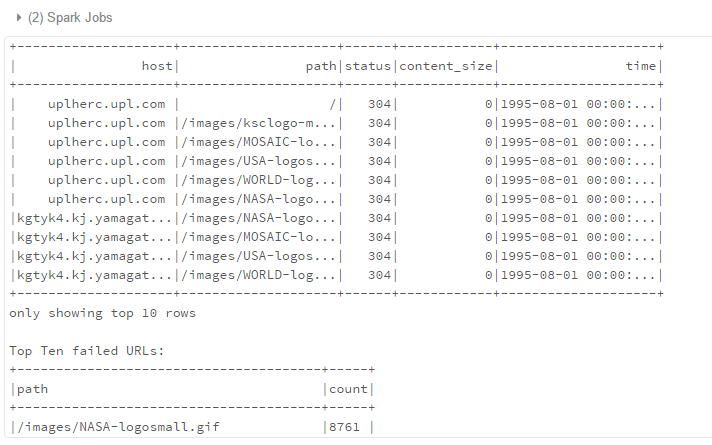
**not200DF.show(10)**

**# Sorted DataFrame containing all paths and the number of times they were accessed with non-200 return code**

**logs\_sum\_df = not200DF.groupBy('path').count().sort('count', ascending=False)**

**print 'Top Ten failed URLs:'**

**logs\_sum\_df.show(10, False)**



How many unique hosts are there in the entire log?

There are multiple ways to find this. Try to find a more optimal way than grouping by 'host'.

**unique\_host\_count = logs\_df.select(logs\_df['host']).distinct().count()**

**print 'Unique hosts: {0}'.format(unique\_host\_count)**

(1) Spark Jobs

Unique hosts: 54507

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 DataFrame sorted by increasing day of the month which includes the day of the month and the associated number of unique hosts for that day. Make sure you cache the resulting DataFrame daily\_hosts\_df so that we can reuse it in the next exercise.

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.* You may want to use the [dayofmonth function](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html" \l "pyspark.sql.functions.dayofmonth) in the pyspark.sql.functions module.

**Description of each variable**

**day\_to\_host\_pair\_df**

A DataFrame with two columns

| **column** | **explanation** |
| --- | --- |
| host | the host name |
| day | the day of the month |

There will be one row in this DataFrame for each row in logs\_df. Essentially, you're just trimming and transforming each row of logs\_df. For example, for this row in logs\_df:

gw1.att.com - - [23/Aug/1995:00:03:53 -0400] "GET /shuttle/missions/sts-73/news HTTP/1.0" 302 -

your day\_to\_host\_pair\_df should have:

gw1.att.com 23

**day\_group\_hosts\_df**

This DataFrame has the same columns as day\_to\_host\_pair\_df, but with duplicate (day, host) rows removed.

**daily\_hosts\_df**

A DataFrame with two columns:

| **column** | **explanation** |
| --- | --- |
| day | the day of the month |
| count | the number of unique requesting hosts for that day |

**from pyspark.sql.functions import dayofmonth**

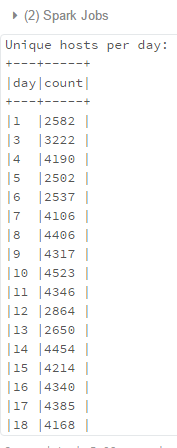
**day\_to\_host\_pair\_df = logs\_df.select(logs\_df['host'], dayofmonth(logs\_df['time']).alias('day'))**

**day\_group\_hosts\_df = day\_to\_host\_pair\_df.distinct()**

**daily\_hosts\_df = day\_group\_hosts\_df.select(day\_group\_hosts\_df['day']).groupBy(day\_group\_hosts\_df['day']).count().cache()**

**print 'Unique hosts per day:'**

**daily\_hosts\_df.show(30, False)**



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. Make sure you cache the resulting DataFrame avg\_daily\_req\_per\_host\_df so that we can reuse it in the next exercise.

To compute the average number of requests per host, find the total number of requests per day (across all hosts) and divide that by the number of unique hosts per day (which we found in part 4c and cached as daily\_hosts\_df).

**total\_req\_per\_day\_df = logs\_df.select(dayofmonth('time').alias('day')).groupBy('day').count()**

**avg\_daily\_req\_per\_host\_df = (**

**total\_req\_per\_day\_df.join(**

**daily\_hosts\_df, daily\_hosts\_df['day']==total\_req\_per\_day\_df['day']**

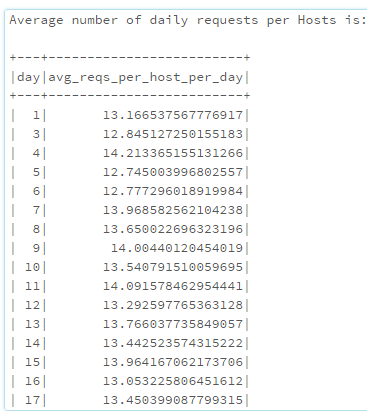
**).drop(daily\_hosts\_df['day']).select(**

**total\_req\_per\_day\_df['day'], (total\_req\_per\_day\_df['count'] / daily\_hosts\_df['count']).alias('avg\_reqs\_per\_host\_per\_day')**

**)).cache()**

**print 'Average number of daily requests per Hosts is:\n'**

**avg\_daily\_req\_per\_host\_df.show()**



Let's drill down and explore the error 404 status records. We've all seen those "404 Not Found" web pages. 404 errors are returned when the server cannot find the resource (page or object) the browser or client requested.

Create a DataFrame containing only log records with a 404 status code. Make sure you cache() not\_found\_df as we will use it in the rest of this exercise.

How many 404 records are in the log?

**not\_found\_df = logs\_df.filter(logs\_df['status'] == '404').cache()**

**print('Found {0} 404 URLs').format(not\_found\_df.count())**

(1) Spark Jobs

Found 6185 404 URLs

Using the DataFrame containing only log records with a 404 status code that you cached in part (5a), print out a list up to 40 distinct paths that generate 404 errors.

**No path should appear more than once in your list.**

**not\_found\_paths\_df = not\_found\_df.select('path').cache()**

**unique\_not\_found\_paths\_df = not\_found\_paths\_df.distinct()**

**print '404 URLS:\n'**

**unique\_not\_found\_paths\_df.show(n=40, truncate=False)**



Using the DataFrame containing only log records with a 404 response code that you cached in part (5a), print out a list of the top twenty paths that generate the most 404 errors.

Remember, top paths should be in sorted order

**top\_20\_not\_found\_df = (not\_found\_paths\_df.groupBy('path')**

**.count()**

**.sort(desc('count'))**

**.cache())**

**print 'Top Twenty 404 URLs:\n'**

**top\_20\_not\_found\_df.show(n=20, truncate=False)**



Instead of looking at the paths that generated 404 errors, let's look at the hosts that encountered 404 errors. Using the DataFrame containing only log records with a 404 status codes that you cached in part (5a), print out a list of the top twenty-five hosts that generate the most 404 errors

**hosts\_404\_count\_df = (not\_found\_df.groupBy('host')**

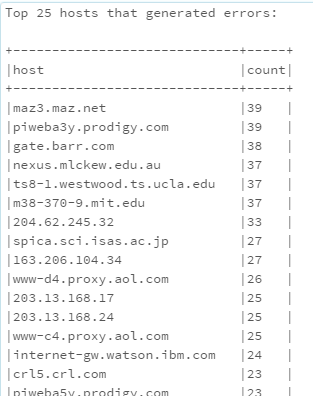
**.count()**

**.sort(desc('count'))**

**.cache())**

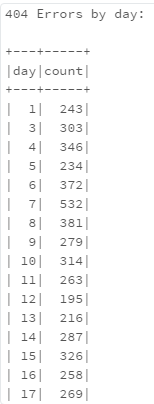
**print 'Top 25 hosts that generated errors:\n'**

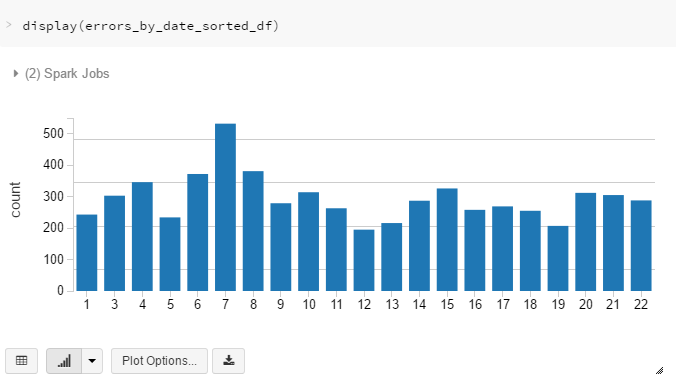
**hosts\_404\_count\_df.show(n=25, truncate=False)**



Let's explore the 404 records temporally. Break down the 404 requests by day (cache the errors\_by\_date\_sorted\_df DataFrame) and get the daily counts sorted by day inerrors\_by\_date\_sorted\_df.

Since the log only covers a single month, you can ignore the month in your checks.





**top\_err\_date\_df = errors\_by\_date\_sorted\_df.sort(desc('count'))**

**print 'Top Five Dates for 404 Requests:\n'**

**top\_err\_date\_df.show(5)**

