Log Analysis Example

How-To Guide

databricks™

Analyzing Apache Access Logs with Databricks

Databricks provides a powerful platform to process, analyze, and visualize small and big data in one place. In this example, we will illustrate how to analyze Apache® HTTP web server access logs using Notebooks. Notebooks allow users to write and run arbitrary Apache® Spark™ code and interactively visualize the results. Currently, notebooks support three languages: Scala, Python, and SQL. In this example, we will be using Python for illustration.

The analysis presented in this example is available in Databricks as part of the Databricks Guide. Find this notebook in your Databricks workspace at **"databricks_guide/Sample Applications/Log Analysis/Log Analysis in Python"** – it will also show you how to create a data frame of access logs with Python using the new Spark SQL 1.3 API. Additionally, there are also Scala & SQL notebooks in the same folder with similar analysis available.

Getting Started

First we need to locate the log file. In this example, we are using a synthetically generated log which is stored in the **"/dbguide/sample_log"** file. The command below (typed in the notebook) assigns the log file pathname to the **DBFS_SAMPLE_LOGS_FOLDER** variable, which will be used throughout the rest of this analysis.

```
> DBFS_SAMPLE_LOGS_FOLDER = "/dbguide/sample_logs"

  Command took 0.07s
```

Figure 1: Location of the synthetically generated logs in your instance of Databricks

Parsing the Log File

Each line in the log file corresponds to an Apache web server access request. To parse the log file, we define **parse_apache_log_line()**, a function that takes a log line as an argument and returns the main fields of the log line. The return type of this function is a PySpark SQL Row object which models the web log access request. For this we use the "re" module which implements regular expression operations. The **APACHE_ACCESS_LOG_PATTERN** variable contains the regular expression used to match an access log line. In particular, **APACHE_ACCESS_LOG_PATTERN** matches client IP address (**ipAddress**) and identity (**clientIdentd**), user name as defined by HTTP authentication (**userId**), time when the server has finished processing the request (**dateTime**), the HTTP command issued by the client, e.g., GET (**method**), protocol, e.g., HTTP/1.0 (**protocol**), response code (**responseCode**), and the size of the response in bytes (**contentSize**).

```
> import re
  from pyspark.sql import Row

  APACHE_ACCESS_LOG_PATTERN = '^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+) (\S+)" (\d{3}) (\d+)'

  # Returns a dictionary containing the parts of the Apache Access Log.
  def parse_apache_log_line(logline):
      match = re.search(APACHE_ACCESS_LOG_PATTERN, logline)
      if match is None:
          # Optionally, you can change this to just ignore if each line of data is not critical.
          #   Corrupt data is common when writing to files.
          raise Error("Invalid logline: %s" % logline)
      return Row(
          ipAddress    = match.group(1),
          clientIdentd = match.group(2),
          userId       = match.group(3),
          dateTime     = match.group(4),
          method       = match.group(5),
          endpoint     = match.group(6),
          protocol     = match.group(7),
          responseCode = int(match.group(8)),
          contentSize  = long(match.group(9)))

  Command took 0.04s
```

Figure 2: Example function to parse the log file in a Databricks notebook

Loading the Log File

Now we are ready to load the logs into a Resilient Distributed Dataset (RDD). RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel and is the primary data abstraction in Spark. Once the data is stored in an RDD, we can easily analyze and process it in parallel. To do so, we launch a Spark job that reads and parses each line in the log file using the **parse_apache_log_line()** function defined earlier, and then creates the **access_logs** RDD. Each tuple in access_logs contains the fields of a corresponding line (request) in the log file, **DBFS_SAMPLE_LOGS_FOLDER**. Note that once we create the **access_logs** RDD, we cache it into memory, by invoking the **cache()** method. This will dramatically speed up subsequent operations we will perform on **access_logs**.

```
> access_logs = (sc.textFile(DBFS_SAMPLE_LOGS_FOLDER)
                 # Call the parse_apace_log_line function on each line.
                 .map(parse_apache_log_line)
                 # Caches the objects in memory since they will be queried multiple times.
                 .cache())
# An action must be called on the RDD to actually populate the cache.
access_logs.count()

Out[3]: 100000

Command took 2.82s
```

Figure 3: Example code to load the log file in Databricks notebook

At the end of the above code snippet, notice that we count the number of tuples in **access_logs** (which returns 100,000 as a result).

Loading the Log File

Now we are ready to analyze the logs stored in the **access_logs** RDD. Below we give two simple examples:

1. Computing the average content size

2. Computing and plotting the frequency of each response code

1. Average Content Size

We compute the average content size in two steps. First, we create another RDD, **content_sizes**, that contains only the **"contentSize"** field from **access_logs**, and cache this RDD:

```
> # Create the content sizes rdd.
  content_sizes = (access_logs
                   .map(lambda row: row.contentSize)
                   .cache()) # Cache this as well since it will be queried many times.

  Command took 0.07s
```

Figure 4: Create the content size RDD in Databricks notebook

Second, we use the **reduce()** operator to compute the sum of all content sizes and then divide it into the total number of tuples to obtain the average:

```
> average_content_size = content_sizes.reduce(lambda x, y: x + y) / content_sizes.count()
average_content_size
Out[5]: 249L
```

*Figure 5: Computing the average content size with the **reduce()** operator*

The result is 249 bytes. Similarly we can easily compute the min and max, as well as other statistics of the content size distribution.

An important point to note is that both commands above run in parallel. Each RDD is partitioned across a set of workers, and each operation invoked on an RDD is shipped and executed in parallel at each worker on the corresponding RDD partition. For example the lambda function passed as the argument of **reduce()** will be executed in parallel at workers on each partition of the **content_sizes** RDD. This will result in computing the partial sums for each partition. Next, these partial sums are aggregated at the driver to obtain the total sum. The ability to cache RDDs and process them in parallel are two of the main features of Spark that allows us to perform large scale, sophisticated analysis.

2. Computing and Plotting the Frequency of Each Response Code

We compute these counts using a map-reduce pattern. In particular, the code snippet returns an RDD (**response_code_to_count_pair_rdd**) of tuples, where each tuple associates a response code with its count.

```
> # First, calculate the response code to count pairs.
  response_code_to_count_pair_rdd = (access_logs
                                     .map(lambda row: (row.responseCode, 1))
                                     .reduceByKey(lambda x, y: x + y))
  response_code_to_count_pair_rdd
  Out[9]: PythonRDD[430] at RDD at PythonRDD.scala:43
  Command took 0.07s
```

Figure 6: Counting the response codes using a map-reduce pattern

Next, we take the first 100 tuples from **response_code_to_count_pair_rdd** to filter out possible bad data, and store the result in another RDD, **response_code_to_count_array**.

```
> # View the responseCodeToCount by calling take on the RDD - which outputs an array of tuples.
  # Notice the use of take(100) - just in case bad data may have slipped in and there are too many response codes.
  response_code_to_count_array = response_code_to_count_pair_rdd.take(100)
  response_code_to_count_array
  Out[10]: [(200, 71322), (500, 14355), (401, 14323)]
  Command took 0.53s
```

Figure 7: Filter out possible bad data with take()

To plot data we convert the **response_code_to_count_array** RDD into a DataFrame. A DataFrame is conceptually equivalent to a table, and it is very similar to the DataFrame abstraction in the popular Python's pandas package. The resulting DataFrame (**response_code_to_count_data_frame**) has two columns "response code" and "count".

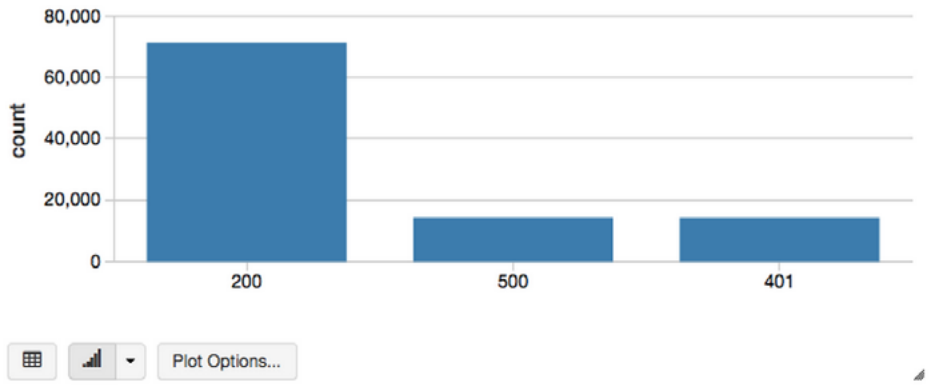
```
> # To call display(), the RDD of tuples must be converted to a DataFrame.
  # A simple map can accomplish that.
  response_code_to_count_row_rdd = response_code_to_count_pair_rdd.map(lambda (x, y): Row(response_code=x, count=y))
  response_code_to_count_data_frame = sqlContext.createDataFrame(response_code_to_count_row_rdd)

  Command took 0.27s
```

Figure 8: Converting RDD to DataFrame for easy data manipulation and visualization

Now we can plot the count of response codes by simply invoking **display()** on our data frame.

```
> # Now, display can be called on the resulting DataFrame.
  #   For this display, a Pie Chart is chosen by selecting that icon.
  #   Then, "Plot Options..." was used to make the response_code the key and count as the value.
  display(response_code_to_count_data_frame)
```




```
  Command took 0.72s
```

Figure 9: Visualizing response codes with display()

If you want to change the chart type, you can do so interactively by just clicking on the down arrow below the chart, and select another chart type. To illustrate this capability, below we show the same data using a pie-chart.

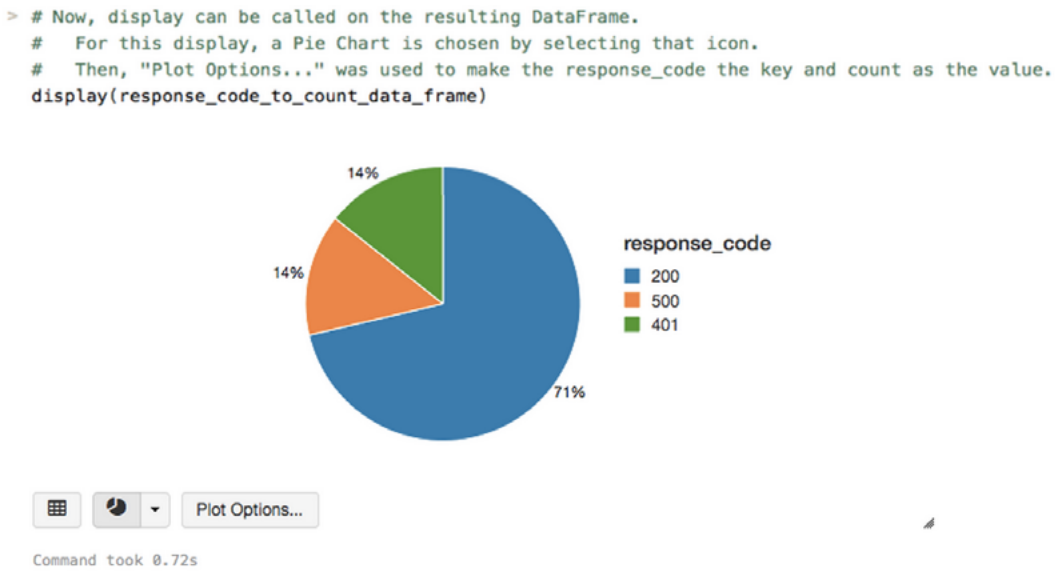


Figure 10: Changing the visualization of response codes to a pie chart

Additional Resoures

If you'd like to analyze your Apache access logs with Databricks, you can evaluate Databricks with a trial account now. You can also find the source code on Github.

Other Databricks how-tos can be found at:
The Easiest Way to Run Apache Spark Jobs

Evaluate Databricks with a trial account now:

databricks.com/try-databricks