Second Homework Assignment due 11 March

Use MapReduce to process some Apache Server logs, identifying the *maximum* average length of time between requests from a client. This can be done in two steps:

1. Let's identify a "client" with an IP address (see below how each request to the server is labelled with the client's IP address and a timestamp). Now, for each request, have the mapper send a pair `(client,timestamp)` to the reducer.

2. The reducer will get all the requests for a particular IP address aggregated together. More precisely, the reducer will get a pair `(client,timevector)` where `client` is the IP address for one client, and `timevector` is an iterator. By converting this iterator to a list, then sorting it, it's not hard to calculate the times between requests from that client. For instance, the timestamps could be numbers like `105, 240, 390`. That would make the times between requests equal to `240-105, 390-240` (that is, `135, 150`). Then the average for this simple example would be 142.5. **Note:** Of course, for the actual log data, each timestamp has to be converted into seconds (use Python modules like `datetime` and `time` to do the work) — best done in the mapper phase. The reducer should then output `(client,avetime)`. **Second Note:** what if a client has only one request? Then the average is not well defined, so please ignore the case where an iterator has only one request.

3. The problem isn't solved yet: now we need to find the *maximum* of all these average times, taken over all clients. So, this is the *second* MapReduce job. Input for this second job is just the `(client,avetime)` pairs from the first phase. What will a mapper do in this second phase? One easy answer is just to have the mapper output `(avetime,client)` for each input.

4. The reducer part of the second phase exploits some additional features of `mrjob`: specifying that there only be *one* reducer, and using the `final` method of the reducers. To specify the number of reducers, we would need to go deeper than what `mrjob` provides; since `mrjob` actually uses the streaming jar, then documentation for Hadoop

   `http://hadoop.apache.org/docs/r1.2.1/streaming.html`
   `http://pythonhosted.org/mrjob/guides/configs-hadoopy-runners.html`

   would be appropriate. The first link documents options to the streaming API of Hadoop. The second link describes how `mrjob` can pass customizing parameters to the streaming API. **Note:** for this assignment, you can just ignore this Hadoop customization on the test machine – it will use a single reducer. The `final` method is specified with another option, see

   `http://pythonhosted.org/mrjob/guides/writing-mrjobs.html`

   which describes how to have a `reducer_final()` method, called after all tuples have been fed to the reducer. When this final method is called, the previous calls to the

reducer can have saved the maximum from all the tuples, so it will be simple to output that value.

Multi-stage MapReduce . The intent of this assignment is for you to use more than just one iteration of MapReduce. Study the `steps` method in the `degreecount2.py` program located in `/opt/hadoop/mrjob/DegreeCount` to see how the output from one MapReduce can easily be the input to a second phase, also MapReduce, from which the final output is generated.

The Apache Log Format

The input for the assignment comes from HTTPD logs (Apache Webserver access logs), which record a line for each request sent to a webserver. Though the logs can be customized, most sites use a default format described in:

```
http://httpd.apache.org/docs/1.3/logs.html#combined
```

What the documentation shows is that each log record is actually the result of a formatting template, much like what `C`'s `printf` function or Python's `format` (or `%` operator) use to format a message with substitutable fields. The "combined" log record uses this format:

```
"%h %l %u %t \"%r\" %s %b \"%{Referer}i\" \"%{User-agent}i\""
```

To decipher this, we have to know what are the variables and their meanings for Apache when it generates a log record. Recall that any HTTP server has to parse an incoming request, which is a TCP connection (with IP addresses and ports for both ends of the connection), then needs to extract a URI (URL) for the request, plus go through HTTP keywords and their values in the request. Then the HTTP server will generate a response, encoded in HTTP, sending it back to the client. The client can be a browser or a bot (search engines use bots, but anybody can send a request to a server using a command-line tool like `wget` or `curl` for example). As a byproduct of doing all of this, Apache builds up some information about the request and response, then showing that information in the log record. Here are fields shown above:

`%h` is the remote host (ie the client IP)

`%l` is the identity of the user determined by identd (not usually used since not reliable) – shown as a hyphen if unknown.

`%u` is the user name determined by HTTP authentication – shown as a hyphen if unknown.

`%t` is the time the server finished processing the request.

`%r` is the request line from the client. ("GET / HTTP/1.0")

`%s` is the status code sent from the server to the client (200, 404 etc.)

`%b` is the size of the response to the client (in bytes).

`Referer` is the page that linked to this URL.

`User-agent` is the browser identification string.

Also, from the format statement above, these fields are separated by a blank (whitespace) in the log. That makes parsing convenient in later processing, since the format is deterministic and simple tokenizing (respecting the fact the some fields, like the time and the browser identification can embed some blanks themselves).

Examples

Here area two example log records and a breakdown by field. Because the log records are too long to show as a single line in this document, they are wrapped.

Get/CGI Request

```
120.43.29.214 - - [16/Jan/2014:10:34:06 -0600]
"GET /22c016f10/UserPreferences?action=login HTTP/1.1"
200 9535 "http://weblog.cs.uiowa.edu/22c016f10/UserPreferences"
"Mozilla/5.0 (Windows NT 6.2; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0"
```

Breakdown by fields:

- `120.43.29.214`
- `-`
- `-`
- `[16/Jan/2014:10:34:06 -0600]`
- `"GET /22c016f10/UserPreferences?action=login HTTP/1.1"`
- `200`
- `9535`
- `"http://weblog.cs.uiowa.edu/22c016f10/UserPreferences"`
- `"Mozilla/5.0 (Windows NT 6.2; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0"`

Post Request

```
120.43.29.214 - - [16/Jan/2014:10:34:07 -0600]
"POST /22c016f10/UserPreferences HTTP/1.1"
200 13145 "http://weblog.cs.uiowa.edu/22c016f10/UserPreferences?action=login"
"Mozilla/5.0 (Windows NT 6.2; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0"
```

Breakdown by fields:

- `120.43.29.214`
- `-`
- `-`
- `[16/Jan/2014:10:34:07 -0600]`
- `"POST /22c016f10/UserPreferences HTTP/1.1"`
- `200`
- `13145`
- `"http://weblog.cs.uiowa.edu/22c016f10/UserPreferences?action=login"`
- `"Mozilla/5.0 (Windows NT 6.2; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0"`

The Input

On the test machine, there is a new directory `/opt/hadoop/mrjob/apachelogs` containing the log files in compressed format. The names are `access_log.0.gz`, `access_log.1.gz`, ..., `access_log.9.gz`. Suppose your `mrjob` program is `apache.py`, then the command

```
$ python apache.py *.gz
```

will us all the "gz" files as input to mapper tasks. So, to save the output from the reducers, a command could be:

```
$ python apache.py *.gz > logoutput.txt
```