**Owen Galvin, CSCI E-88 Homework03**

**NOTE: solutions are in Python**

**Problem 1: (Points: 30)**

**Write an application that calculates answers to the specified below queries using the Hadoop MapReduce APIs:**

* **As input, use a few (2-5) files; each file has multiple events (one line is one event == one user click on a URL); you can generate these files using your app developed in HW2 Problem3, or you can create them manually; input files should be located in HDFS**

The four files I had generated for HW2 are in submitted folder = Problem1\_input. The 240 events, spanning a single day in time, had been randomly partitioned into 4 files. Each file has different line count but in the end add up to 240 lines. (HW config of 4 user ids \* 3 urls \* 20 events per-user per url produces 240 total events).

Screenshot etc. showing them being copied into HDFS occur later, after I’ve connected to the EMR Master.

* **create a separate MR job for each Query:**
  + **Query 1: get count of unique URLs**
  + **Query 2: get count of unique visitors per URL**
  + **Query 3: get count of unique (by userId) clicks per URL**

As will become more evident later on, I have a single shell file that I modify as appropriate to run three separate MR jobs, one for each query. Command line arguments determine which query is run – this seemed much more efficient, and safer, than having multiple reducer files.

First I’ll go over the mapper and reducer .py files I had created.

**p1\_mapper.py**

*#!/usr/bin/env python***import** sys  
**import** collections  
  
single\_map = collections.defaultdict(dict)  
  
**for** line **in** sys.stdin:  
  
 timestamp, url, user\_id = line.strip().split(**'\t'**)  
 **if** url **in** single\_map **and** user\_id **in** single\_map[url]:  
 *# this url already registered a hit for this uid, increment the count* single\_map[url][user\_id] = single\_map[url][user\_id] + 1  
 **else**:  
 *# either a brand new url or a brand new user to the existing url, register as a new hit* single\_map[url].update({user\_id: 1})  
  
**for** k, v **in** single\_map.items():  
 print(**'{}\t{}'**.format(k, v))

Create a parent defaultdict object with default object type = dict, to make assigning new keys and values very simple. As per the general format of Hadoop Streaming API, and specifically in terms of Python, the mapper is designed to read from stdin. Each line is in the format of below, tab separated, so by splitting on tab character the timestamp/url/userid variables are each populated with applicable value.

2017-09-12 00:27:07 yahoo.com u1  
2017-09-12 00:30:44 yahoo.com u1  
2017-09-12 00:40:12 yahoo.com u4  
2017-09-12 00:47:02 harvard.edu u4

Then as in HW2, if a given url/userId combo is already in the single\_map defaultdict, update the existing count value by 1. Otherwise, add the url/userid combo to the parent single\_map, by using defaultdict that can be done in a one liner, adding the url dictionary at the same time.

Once the single\_map object has been populated, iterate through the now-aggregated urls dicts and emit the results to stdout via simple print calls.

To see raw output I can feed in a single log file from local file system:

cat input/1\_input.log | python p1\_mapper.py

OUTPUT:

harvard.edu {'u1': 4, 'u3': 3, 'u2': 2, 'u4': 7}

google.com {'u1': 6, 'u3': 5, 'u2': 4, 'u4': 1}

yahoo.com {'u1': 5, 'u3': 5, 'u2': 1, 'u4': 4}

In Problem 2 I iterate down to the base level, i.e. print one line for every item in each of the nested dictionaries but I chose the dictionary-as-string approach above without really thinking about it. After going that route, I consider this way to be more fragile and if I were re-doing it would pursue the one-line-per-item solution.

**p1\_reducer.py**

*#!/usr/bin/env python***import** sys  
**import** collections  
**import** argparse  
  
**def** reduce(reduced\_map, input\_line):  
 input\_line = input\_line.strip()  
 url, dict\_string = input\_line.split(**"\t"**)  
  
 user\_dict = eval(dict\_string)  
  
 **if** url **in** reduced\_map:  
 *# in this toy example all the urls in additional maps will be in that reduced\_map, this will be expected path* **for** uid, count **in** user\_dict.items():  
 *# iterate through the user\_id: count map for the current url in "new" map* **if** uid **in** reduced\_map[url]:  
 *# this uid already appeared for this url, increment count* reduced\_map[url][uid] = reduced\_map[url][uid] + count  
 **else**:  
 *# new uid for current url, transfer the count from current map to reduced\_map* reduced\_map[url][uid] = count  
 **else**:  
 *# should a new url be encountered, add it to the base map* reduced\_map[url] = user\_dict  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 parser = argparse.ArgumentParser(description=**'Return results of Hadoop streaming map reduce queries'**)  
 parser.add\_argument(**'-q'**, **'--query-number'**, type=int, default=3,  
 help=**'Number (1, 2, or 3) of Query for which results should be returned'**)  
 parser.add\_argument(**"-d"**, **"--debug"**, type=bool, default=**False**)  
 args = parser.parse\_args()  
  
 reduced\_map = collections.defaultdict(dict)  
 **for** input\_line **in** sys.stdin:  
 reduce(reduced\_map, input\_line)  
  
 **if** args.debug:  
 print(**'-'**\*20, **'BEGIN DEBUG'**, **'-'**\*20)  
 **for** k, v **in** reduced\_map.items():  
 print(**'{}\t{}'**.format(k, v))  
 print(**'-'**\*20, **'END DEBUG'**, **'-'**\*20)  
  
 **if** args.query\_number == 1:  
 print(**'Query 1: get count of unique URLs: {}'**.format(len(reduced\_map)))  
 **elif** args.query\_number == 2:  
 query\_2 = [**'{}: {}'**.format(url, len(user)) **for** url, user **in** reduced\_map.items()]  
 print(**'Query 2: get count of unique visitors per URL: \n - {}'**.format(**'\n - '**.join(query\_2)))  
 **else**:  
 print( **'Query 3: get count of unique (by userId) clicks per URL: '**)  
 **for** url,uids **in** reduced\_map.items():  
 print(**' - {}'**.format(url))  
 **for** uid, count **in** uids.items():  
 print(**' - {}: {}'**.format(uid, count))

Above is for the other side of the Hadoop Streaming API, reading from stdin. To begin with though there are two possible arguments, first for the number of the query that should be performed, defaulting to 3 since it contains the most information, and second a debug argument that can be used to print out additional intermediate information. The main reducing action takes place by calling the internal reduce() function, passing in each line that stdin receives. One in there the line is split a single time since the incoming data will be in format of

url {'u1': #, 'u3': #, 'u2': #, 'u4': #}

The first half will be the full url string while the second half will be the uid dict, encoded as a simple text string. To get that string converted into an actual dict object, call eval() on it (as noted earlier, I somewhat regret this approach) The inline comments then discuss what happens in terms of reducing the multiple dict of dictionaries into a final reduced\_map object.

After the dict-of-dicts is fully populated, the applicable query is performed and results printed out.

Again, using local files only as an example, below would be typical output on a single file:

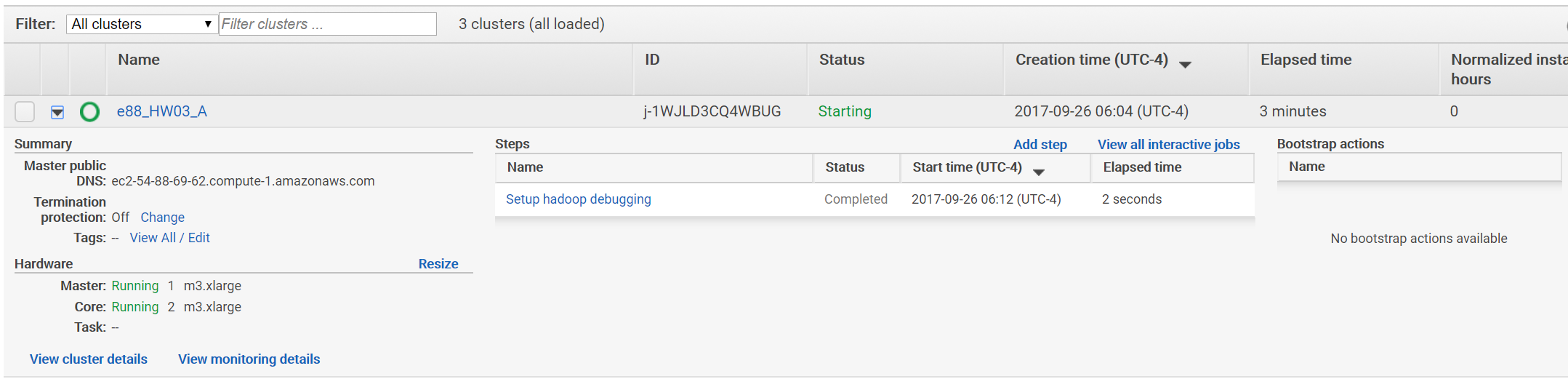
cat input/1\_input.log | python p1\_mapper.py | sort -k1,1 | python p1\_reducer.py -d True

|  |
| --- |
| -------------------- BEGIN DEBUG --------------------  yahoo.com {'u3': 5, 'u4': 4, 'u2': 1, 'u1': 5}  harvard.edu {'u3': 3, 'u4': 7, 'u2': 2, 'u1': 4}  google.com {'u3': 5, 'u4': 1, 'u2': 4, 'u1': 6}  -------------------- END DEBUG --------------------  Query 3: get count of unique (by userId) clicks per URL:  - yahoo.com  - u3: 5  - u4: 4  - u2: 1  - u1: 5  - harvard.edu  - u3: 3  - u4: 7  - u2: 2  - u1: 4  - google.com  - u3: 5  - u4: 1  - u2: 4  - u1: 6 |

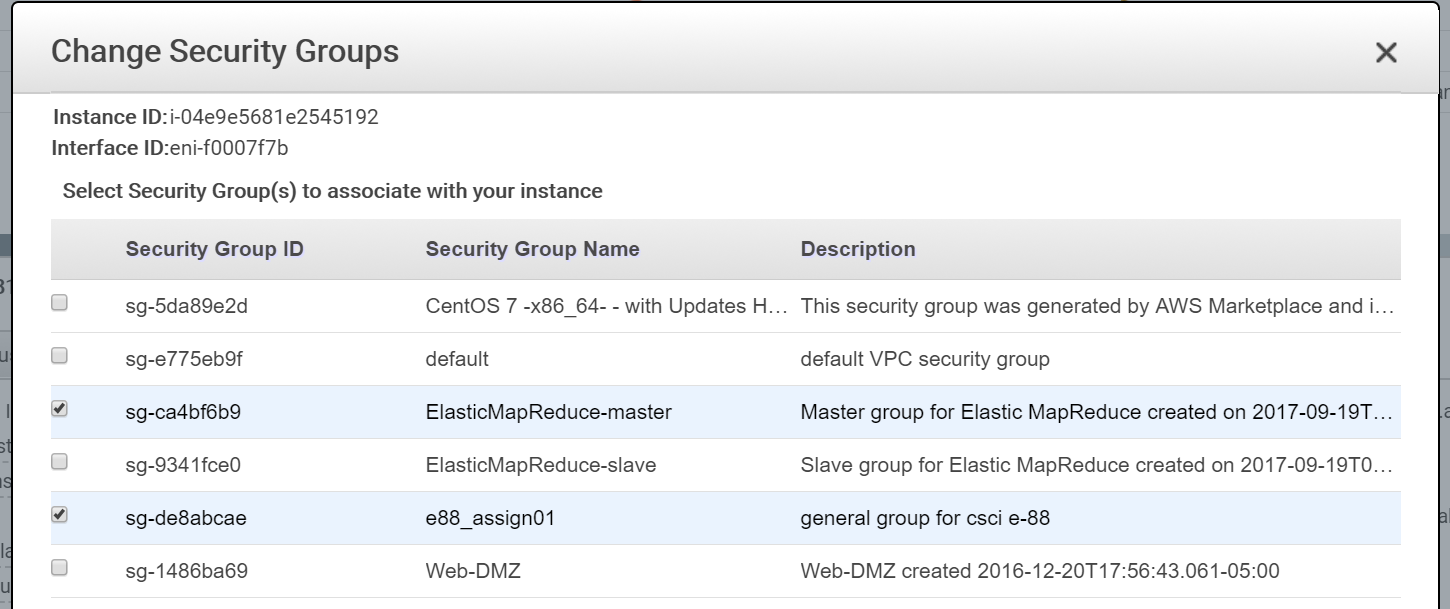
* **run your jobs on an EMR cluster**

Create the EMR cluster

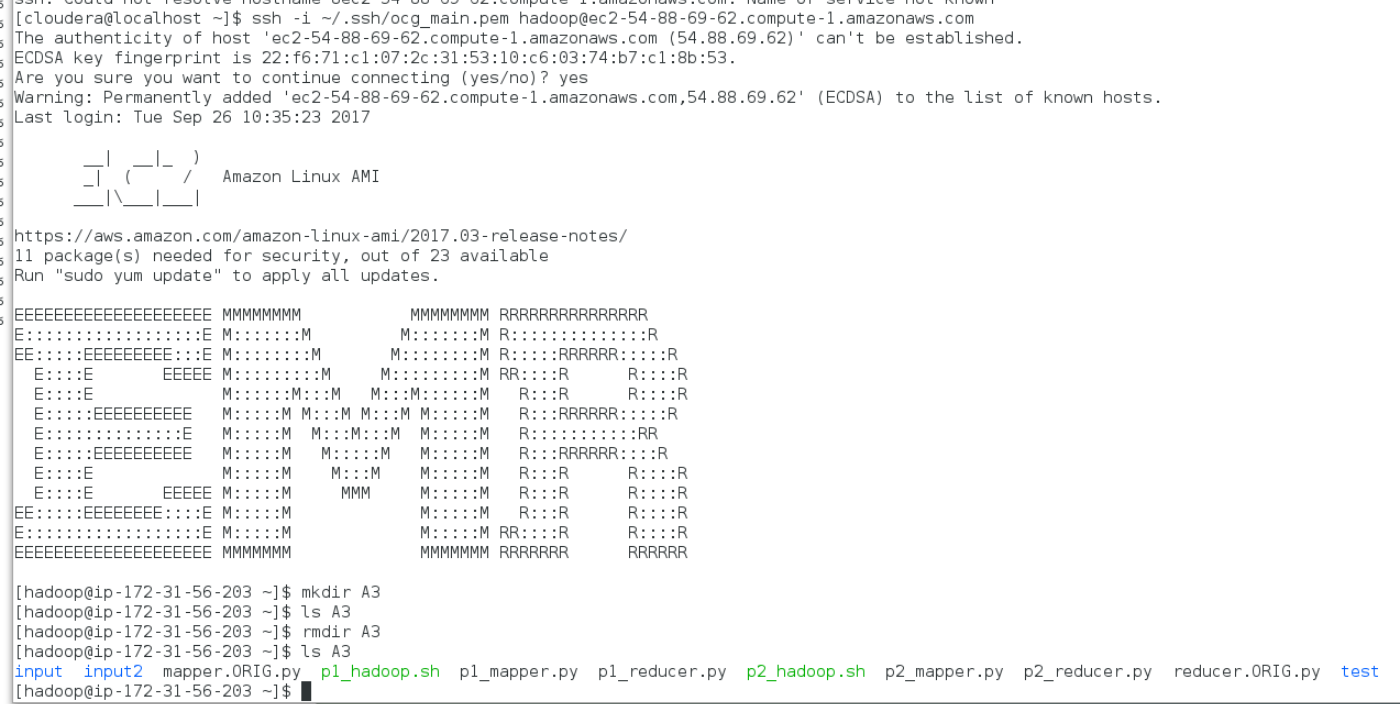
|  |
| --- |
|  |



Add master node to my e88 group that allows ssh



ssh to master (and the -ls after the scp from a different terminal window)



Copy all source files from my Linux test environment to ec2 master via scp

scp -i ~/.ssh/ocg\_main.pem -r ./A3 [hadoop@ec2-54-88-69-62.compute-1.amazonaws.com:~/A3](mailto:hadoop@ec2-54-88-69-62.compute-1.amazonaws.com:~/A3)

|  |
| --- |
|  |

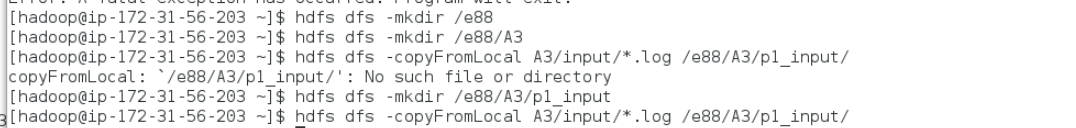
I watched the lab and begun copying files to S3 but the assignment had stated the input files should be in HDFS, so that is the route I went. Copy files to hdfs with -copyFromLocal, after making target directories

hdfs dfs -mkdir /e88

hdfs dfs -mkdir /e88/A3

hdfs dfs -mkdir /e88/A3/p1\_input

hdfs dfs -copyFromLocal A3/input/\*.log /e88/A3/p1\_input/



I had created a p1\_hadoop.sh for running the MR jobs, following general syntax from the lab, but using separate -file args for the two files. For this and all the remaining hadoop.sh files, the one of the first actions is to delete the output directory from HDFS if it does not already exist. STREAM\_JAR is separately defined as I was using a different path when developing on my local Linux VM.

|  |
| --- |
| input\_dir=/e88/A3/p1\_input  output\_dir=/e88/A3/p1\_output  hdfs dfs -rmr $output\_dir  STREAM\_JAR=hadoop-streaming  $STREAM\_JAR \  -mapper "$PWD/p1\_mapper.py" \  -reducer "$PWD/p1\_reducer.py -q 3 -d True" \  -input "$input\_dir/" \  -output "$output\_dir" \  -file "$PWD/p1\_mapper.py" \  -file "$PWD/p1\_reducer.py" |

run the p1\_hadoop.sh on the EMR master, with the arguments = -q 3 -d True

./p1\_hadoop.sh

|  |
| --- |
| … |

(screenshots above, and most of the remaining related to MR jobs, are truncatd to show begin & end of the runs, + perhaps a few lines of bash commands after that, if applicable)

List contents of output directory to confirm success. The lab mentioned a “combiner” that I guess would have resulted in a single part file, but I could find little helpful information on the internet with details. Either way the job did work. Output the first file and then cat contents of all part-\* files.

hdfs dfs -ls /e88/A3/p1\_output

hdfs dfs -cat /e88/A3/p1\_output/part-00000

hdfs dfs -cat /e88/A3/p1\_output/\*

|  |
| --- |
|  |

The above had worked as expected on my local Linux VM, wound up with only a single part file in HDFS. Of course that was a single node system.

Ok, that first run was set for query 3 results + debug mode, modify my .sh file to do only query2 w/o debug

|  |
| --- |
|  |

Results of “Query 1”:

|  |
| --- |
| … |

Multiple files again, some of which are empty, do a cat again on all files, which delivers all of the expected data… but spread across multiple files.

|  |
| --- |
|  |

**\*\* UPDATED \*\*** run again later, on the 2nd EMR I spun up for p2/p3 - after using the reducer=1 trick introduced in several lines. With that setting, see that we get the one file with total sum

|  |
| --- |
|  |

Finally run the remaining query = query 2. First I’m going to hack the p1\_hadoop.sh guessing that the map.reduce.tasks arg I found on the internet will result in everything being reduced into a single file:

Added:

-D mapred.reduce.tasks=1

|  |
| --- |
|  |

|  |
| --- |
|  |

And that seems to have worked.

hdfs dfs -ls /e88/A3/p1\_output

hdfs dfs -cat /e88/A3/p1\_output/part-00000

|  |
| --- |
|  |

The opposite of big data since everything is being forced down to a single reducer but for the small files in this assignment, this will hopefully be good enough.

\* on Piazza the professor said it was good enough for homework’s purposes

Manage to get another run in for query 3 with the map.reduce.tasks=1, and debug turned off, before the next hour of charges begin. -ls and then cat the single file, all looks good.

|  |
| --- |
|  |

**Problem 2: (Points: 30)**

1. **Modify data generation script to generate events with timestamps that span 3-4 hours in time; …**

Over 95% of my p2\_generate\_events.py was create as part of the last assignment so I won’t go into much detail, discussing only the changes I made. There is extraneous code included, such as control of threading, but it was the safest course to keep the pre-existing code as intact as possible.

**p2\_generate\_events.py**

**import** pprint  
**import** argparse  
**from** threading **import** Thread  
**from** random **import** randint  
**import** collections  
**import** os, shutil  
  
**def** prep\_input\_files(master\_log\_file, input\_folder, is\_debug):  
 num\_new\_files = 4  
 inputs = collections.defaultdict(list)  
 **with** open(master\_log\_file, **'r'**) **as** f:  
 **for** line **in** f.readlines():  
 *# each line will be randomly assigned to one of the files that will be written  
 # each file will wind up with similar number of lines but rarely the same number (unless source file has many lines)* file\_num = randint(1, num\_new\_files )  
 inputs[file\_num].append(line)  
 **if** is\_debug:  
 pprint.pprint(inputs)  
  
 *# for each group of clicks, write them to a new file* **for** k,v **in** inputs.items():  
 **with** open(**'{}/{}\_input.log'**.format(input\_folder, k), **'w'**) **as** f:  
 f.writelines(v)  
  
 **return** os.listdir(input\_folder)  
  
**def** prepare\_output\_dir(output\_dir):  
 **if** os.path.exists(output\_dir):  
 **if** os.path.isfile(output\_dir):  
 os.remove(output\_dir)  
 **else**:  
 shutil.rmtree(output\_dir)  
 wait\_seconds = 10  
 waited = 0  
 **while** os.path.exists(output\_dir) **and** waited < wait\_seconds:  
 **import** time  
 time.sleep(1)  
 waited += 1  
 os.mkdir(output\_dir)  
  
*# make a deque with all the (random over a static day) timestamps that will be needed for given set of inputs***def** get\_all\_timestamps(count):  
 *#'YYYY-MM-DD HH:MM:SS'  
 # for Assignment 3, modify to be within hours of 7am (inclusive) to 11am (exclusive)* **return** collections.deque(**'2017-09-12 {hour:02}:{minute:02}:{second:02}'**.format(  
 hour=randint(6,9), minute=randint(0,59), second=randint(0,59)) **for** i **in** range(count))  
  
*# generate a list of user-ids in format of u##, zero-filled from the left as appropriate to total count***def** get\_userids(count):  
 **return** [**'u'** + str.zfill(**'{}'**.format(i), len(str(count))) **for** i **in** range(1, count+1)]  
  
**def** generate\_log(thread\_number, output\_dir, data, is\_debug=**False**):  
 log\_path = os.path.join(output\_dir, **'{}\_events.txt'**.format(thread\_number))  
 **with** open(log\_path, **'w'**) **as** f:  
 lines = **'\n'**.join(data).strip()  
 f.write(lines)  
 **if** is\_debug:  
 print(**'-- wrote: {}'**.format(log\_path), flush=**True**)  
  
**def** do\_threads(num\_userids, urls, event\_count, thread\_count, total\_num\_events, output\_dir\_path, is\_debug):  
 prepare\_output\_dir(output\_dir\_path)  
  
 ts = get\_all\_timestamps(total\_num\_events)  
 uids = get\_userids(num\_userids)  
 visits = []  
 **for** uid **in** uids:  
 **for** url **in** urls:  
 **for** i **in** range(event\_count):  
 visits.append(**'{}\t{}\t{}'**.format(ts.pop(),url, uid,))  
  
 *#sort by the random timestamp, for better simulation* visits = sorted(visits) *#, key=lambda line: line.split()[-1])* **if** is\_debug:  
 **for** visit **in** visits:  
 print(visit)  
  
 log\_threads = []  
 **for** i **in** range(1, thread\_count + 1):  
 new\_thread = Thread(target=generate\_log, args=(i, output\_dir\_path, visits, is\_debug))  
 print(**'thread {} starting'**.format(i), flush=**True**)  
 log\_threads.append(new\_thread)  
 new\_thread.start()  
  
 *#don't return until all the threads have completed* [t.join() **for** t **in** log\_threads]  
  
 **return** output\_dir\_path  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 parser = argparse.ArgumentParser(description=**'Run selected number of threads'**)  
 parser.add\_argument(**'-n'**, **'--num-userids'**, type=int, default=5)  
 parser.add\_argument(**'-u'**, **'--urls'**, type=str, default=**''**, help=**'comma separated string of urls'**)  
 parser.add\_argument(**'-e'**, **'--event-count'**, type=int, default=5)  
 parser.add\_argument(**'-t'**, **'--thread-count'**, type=int, default=1)  
 parser.add\_argument(**'-d'**, **'--debug'**, type=bool, default=**False**)  
 args = parser.parse\_args()  
 urls = [u **for** u **in** args.urls.split(**','**) **if** u]  
 url\_count = len(urls)  
 print(**'BEGIN TEST, userid count: {}, number of urls: {}, number of events: {}, number of threads: {}, debug: {}'**.format(  
 args.num\_userids, url\_count, args.event\_count, args.thread\_count, args.debug))  
 total\_event\_count = args.num\_userids \* url\_count \* args.event\_count  
 print(**'TOTAL EVENT COUNT: {}'**.format(total\_event\_count))  
  
 output\_dir = **'output'** output\_path = do\_threads(args.num\_userids, urls, args.event\_count, args.thread\_count, total\_event\_count, output\_dir, args.debug)  
  
 print(**'\nDONE, contents of {}\n - {}'**.format(output\_path, **'\n - '**.join(os.listdir(output\_path))))  
  
 master\_log\_file = os.path.join(output\_dir, os.listdir(output\_path)[0])  
 input\_folder = **'output2'** prep\_input\_files(master\_log\_file, input\_folder, args.debug)

In fact the only change I can think of right now is in below – luckily all I needed to do was change the hour generation to be between 6 and 9 instead of 0 and 23 as for last assignment. For those earlier problems I had generated random timestamps within the span of a single hard-coded day.

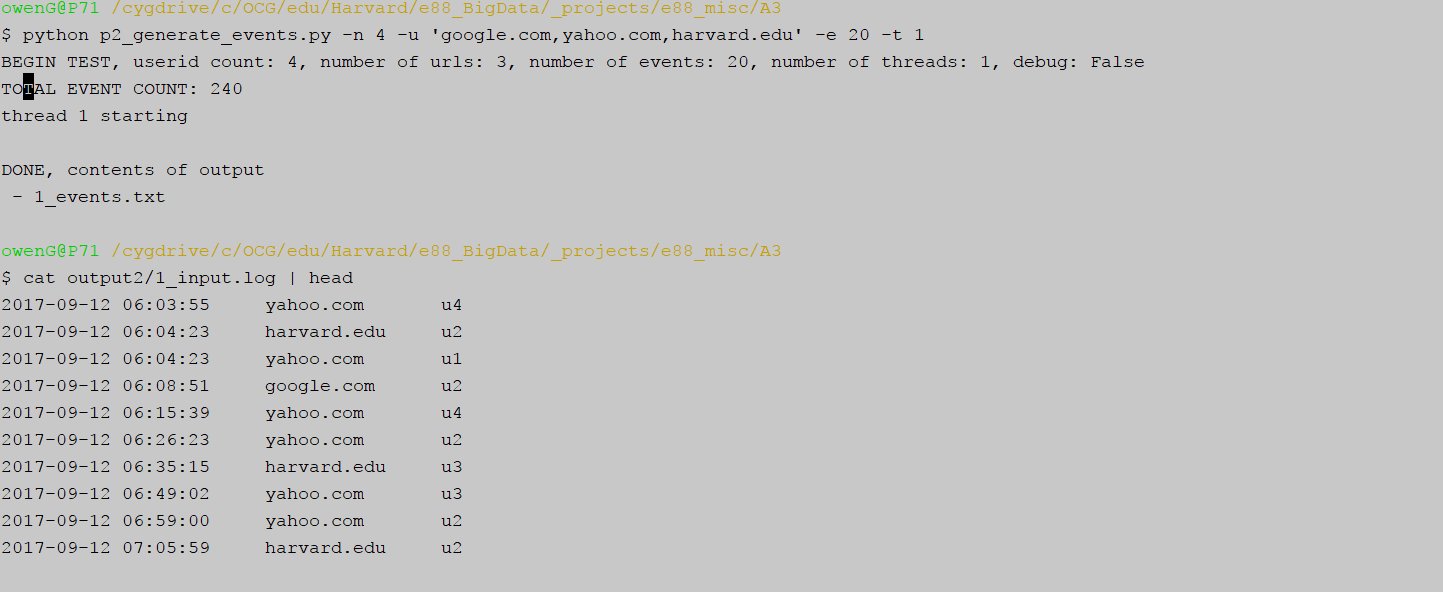
**def** get\_all\_timestamps(count):  
 *#'YYYY-MM-DD HH:MM:SS'  
 # for Assignment 3, modify to be within hours of 7am (inclusive) to 11am (exclusive)* **return** collections.deque(**'2017-09-12 {hour:02}:{minute:02}:{second:02}'**.format(  
 hour=randint(6,9), minute=randint(0,59), second=randint(0,59)) **for** i **in** range(count))

Other than that, I combine pieces from the last assignment, so it did both the generation of the full file and the randomized split into 4 separate .log files.

Here is a quick re-run of the job on my local Windows system, for illustrative purposes. Originally generated files will be included in Problem2\_input directory of zipped homework submission.

python p2\_generate\_events.py -n 4 -u 'google.com,yahoo.com,harvard.edu' -e 20 -t 1

cat output2/1\_input\_log | head

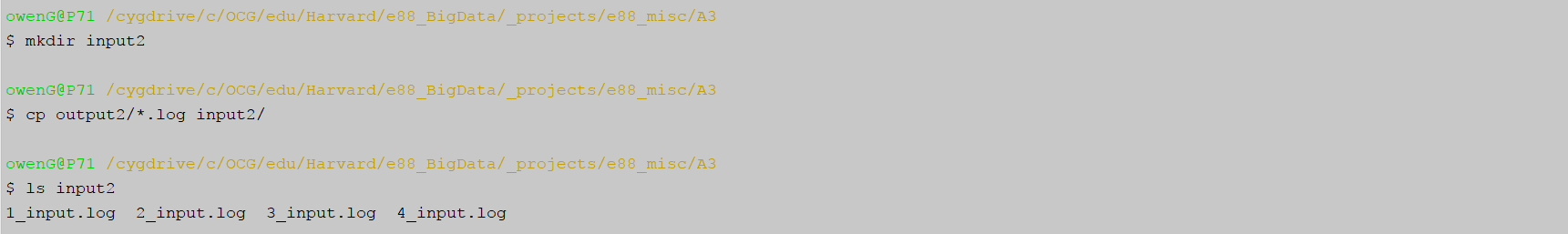


In my original workflow, copy those into a new input2 folder – use that as source while writing mapper/reducer for problem 2, before then putting in HDFS. Pointing this out only to cover naming convention on the input/output folders.

mkdir input2

cp output2/\*.log input2/

ls input2



… **place your generated data files into HDFS**

New EMR Cluster, cloned from the one used for P1, ssh in there

ssh -i ~/.ssh/ocg\_main.pem hadoop@ec2-184-72-192-30.compute-1.amazonaws.com

scp the files up there again as before.

Once on “new” EMR master, get HDFS folder structure in place again and copy my limited-hour log files into HDFS.

hdfs -mkdir /e88

hdfs dfs -mkdir /e88

hdfs dfs -mkdir /e88/A3

hdfs dfs -mkdir /e88/A3/p2\_input

hdfs dfs -copyFromLocal A3/input2/\*.log /e88/A3/p2\_input/

hdfs dfs -ls /e88/A3/p2\_input/

|  |
| --- |
|  |

1. **write a new set of jobs, that process these generated files and answer the same 3 queries, but with added "by hour" aggregation. Modified queries will look as following:**

**Query 1: get count of unique URLs by hour**

**Query 2: get count of unique visitors per URL by hour**

**Query 3: get count of unique (by userId) clicks per URL by hour**

Begin with the mapper, which resembles that from problem 1, with the addition of a parent “hour” map/dictionary. Inline comments discuss details. One notable bit is the line below, where I’ve used a defaultdict in multiple other previous assignments/problems but here with an additional level of nesting there needs to actually be a a defaultdict of type = default dict.

single\_map = collections.defaultdict(**lambda**: collections.defaultdict(dict)

That allows for direct assignment into the nested dictionary even when intermediate keys do not exist, e.g. the below adds an item to nested dicts even when the entry is “brand new”:

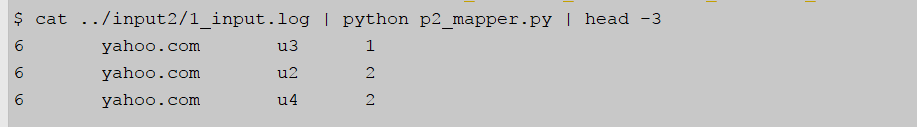
single\_map[hour][url].update({user\_id: 1})

**p2\_mapper.py**

**import** sys  
**import** collections  
**import** datetime  
  
*# some googling and playing around to get a defaultdict of type = defaultdict, suffice to say it works*single\_map = collections.defaultdict(**lambda**: collections.defaultdict(dict))  
  
**for** line **in** sys.stdin:  
  
 timestamp, url, user\_id = line.strip().split(**'\t'**)  
 *# string in format of "2017-09-12 06:31:55", turn that into a datetime object and pull out the hour* hour = datetime.datetime.strptime(timestamp, **'%Y-%m-%d %H:%M:%S'**).hour  
  
 *# same as with p1 mapper, only now there is a parent key=hour dict holding everything* **if** hour **in** single\_map **and** url **in** single\_map[hour]:  
 **if** user\_id **in** single\_map[hour][url]:  
 *# increase existing count by one* single\_map[hour][url][user\_id] = single\_map[hour][url][user\_id] + 1  
 **else**:  
 *# new user\_id for this hour/url, add with count = 1* single\_map[hour][url].update({user\_id: 1})  
 **else**: *# by definition, this url ain't in here* single\_map[hour][url].update({user\_id: 1})  
  
*# iterate through all levels of dictionaries, though I've only sorted the hour one, helps with troubleshooting***for** hour, url\_map **in** sorted(single\_map.items(), key=**lambda** item: item[0] ):  
 **for** url, uid\_map **in** url\_map.items():  
 **for** uid, count **in** uid\_map.items():  
 *# this time doing one single tab separate line for each item in the nested dicts* print(**'{}\t{}\t{}\t{}'**.format(hour, url, uid, count))

Quick sample of expected output for illustrative purposes:

cat ../input2/1\_input.log | python p2\_mapper.py | head -3



The reducer is also quite similar to the one from problem 1, inline comments discuss some of the logic. The main command line argument = “-q” controls which query gets run. And if “-d/-debug” is set then each of the queries are run and included in output.

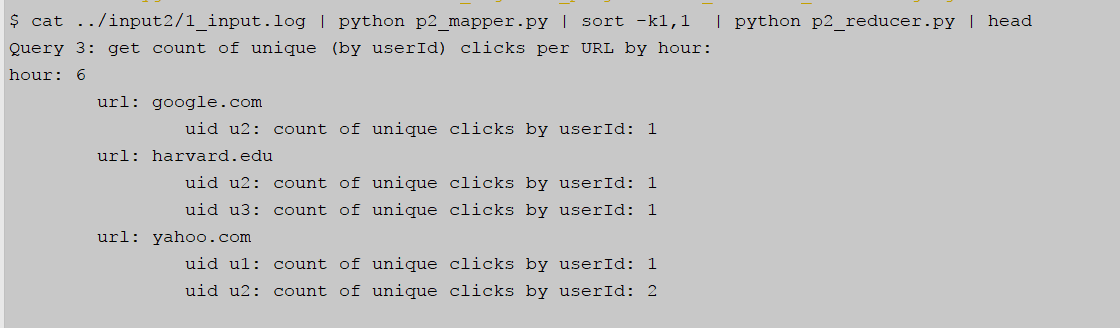
An empty reduced\_map object (defaultidct of defaultdict discussed previously) is fed to the reduce function that does the actual work. The populated reduced\_map is then fed into the various “queries”, which iterate through the contents as appropriate to each query. The dicts at each level are sorted inline so that the results are aesthetically what a viewer might expect, i.e. hour 1 then hour 2 etc.

**p2\_reducer.py**

*#!/usr/bin/env python***import** sys  
**import** collections  
**import** argparse  
  
  
**def** reduce(reduced\_map, input\_line):  
 input\_line = input\_line.strip()  
 hour, url, uid, count = input\_line.split(**"\t"**)  
  
 *# if we don't do this the numbers will be appended like strings '1'+'2' = '12', as I found out* count = int(count)  
 updated = **False  
 if** hour **in** reduced\_map:  
 **if** url **in** reduced\_map[hour]:  
 **if** uid **in** reduced\_map[hour][url]:  
 *#get the exisitng count and increase by the value of incoming count* reduced\_map[hour][url][uid] = reduced\_map[hour][url][uid] + count  
 updated = **True** *# brand new entry, add with the incoming count* **if not** updated:  
 reduced\_map[hour][url][uid] = count  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 parser = argparse.ArgumentParser(description=**'Return results of Hadoop streaming map reduce queries'**)  
 parser.add\_argument(**'-q'**, **'--query-number'**, type=int, default=3,  
 help=**'Number (1, 2, or 3) of Query for which results should be returned'**)  
 parser.add\_argument(**"-d"**, **"--debug"**, type=bool, default=**False**)  
 args = parser.parse\_args()  
  
 reduced\_map = collections.defaultdict(**lambda**: collections.defaultdict(dict))  
 **for** input\_line **in** sys.stdin:  
 reduce(reduced\_map, input\_line)  
  
 *# print out everything, each dictionary soreted on its keys* **if** args.debug:  
 print(**'-'**\*20, **'BEGIN DEBUG'**, **'-'**\*20)  
 **for** hour, url\_map **in** sorted(reduced\_map.items(), key=**lambda** item: item[0]):  
 **for** url, uid\_map **in** sorted(url\_map.items(), key=**lambda** item: item[0]):  
 **for** uid, count **in** sorted(uid\_map.items(), key=**lambda** item: item[0]):  
 print(**'{}\t{}\t{}\t{}'**.format(hour, url, uid, count))  
 print(**'-'**\*20, **'END DEBUG'**, **'-'**\*20)  
  
  
 *# if debug is set, run/print every query  
 # uery 1: get count of unique URLs by hour* **if** args.query\_number == 1 **or** args.debug:  
 query\_1 = [**'{}: {}'**.format(hour, len(url)) **for** hour, url **in** reduced\_map.items()]  
 print(**'Query 1: get count of unique URLs by hour: {}'**.format(query\_1))  
 print()  
 *# Query 2: get count of unique visitors per URL by hour* **if** args.query\_number == 2 **or** args.debug:  
 *#query\_2 = ['{}: {}'.format(url, len(user)) for url, user in reduced\_map.items()]* print(**'Query 2: get count of unique visitors per URL by hour:'**)  
 **for** hour, url **in** sorted(reduced\_map.items(), key=**lambda** item: item[0]):  
 print(**'hour: {}'**.format(hour))  
 **for** url, uid\_map **in** url.items():  
 print(**'\turl {}: count of unique visitors: {}'**.format(url, len(uid\_map)))  
 print()  
 *# Query 3: get count of unique (by userId) clicks per URL by hour* **if** args.query\_number == 3 **or** args.debug:  
 print( **'Query 3: get count of unique (by userId) clicks per URL by hour: '**)  
 **for** hour, url\_map **in** sorted(reduced\_map.items(), key=**lambda** item: item[0]):  
 print(**'hour: {}'**.format(hour))  
 **for** url, uid\_map **in** sorted(url\_map.items(), key=**lambda** item: item[0]):  
 print(**'\turl: {}'**.format(url))  
 **for** uid, count **in** sorted(uid\_map.items(), key=**lambda** item: item[0]):  
 print(**'\t\tuid {}: count of unique clicks by userId: {}'**.format(uid, count))

Another quick and dirty illustration of expected output, in this case for query 3:

cat ../input2/1\_input.log | python p2\_mapper.py | sort -k1,1 | python p2\_reducer.py | head



1. **run your jobs on an EMR cluster**

Here is the p2\_hadoop.sh that got run for first MR job:

|  |
| --- |
| input\_dir=/e88/A3/p2\_input  output\_dir=/e88/A3/p2\_output  hdfs dfs -rmr $output\_dir  STREAM\_JAR=hadoop-streaming  $STREAM\_JAR \  -D mapred.reduce.tasks=1 \  -mapper "$PWD/p2\_mapper.py" \  -reducer "$PWD/p2\_reducer.py -q 1 -d True" \  -input "$input\_dir/" \  -output "$output\_dir" \  -file "$PWD/p2\_mapper.py" \  -file "$PWD/p2\_reducer.py" |

Run ./p2\_hadoop.sh

./p2\_hadoop.sh

|  |
| --- |
|  |

check the output file, which is a tad excessive since debug was turned on, printing out extra info + running all 3 queries

|  |
| --- |
| … |

Update p2\_hadoop.sh to do a basic Query = 1,

-reducer "$PWD/p2\_reducer.py -q 1 -d True" \

to

-reducer "$PWD/p2\_reducer.py -q 1" \

Run and take another look at the output file, now we have just what we want, output in format of:

hour: unique URL count

|  |
| --- |
|  |

OUTPUT:

Query 1: get count of unique URLs by hour: ['9: 3', '8: 3', '7: 3', '6: 3']

Update p2\_hadoop.sh, now to pass in “-q 2” in order to run Query 2

|  |
| --- |
| …. |

OUTPUT:

Query 2: get count of unique visitors per URL by hour:

hour: 6

url yahoo.com: count of unique visitors: 4

url harvard.edu: count of unique visitors: 4

url google.com: count of unique visitors: 4

hour: 7

url yahoo.com: count of unique visitors: 4

url harvard.edu: count of unique visitors: 4

url google.com: count of unique visitors: 4

hour: 8

url yahoo.com: count of unique visitors: 4

url harvard.edu: count of unique visitors: 4

url google.com: count of unique visitors: 4

hour: 9

url yahoo.com: count of unique visitors: 4

url harvard.edu: count of unique visitors: 4

url google.com: count of unique visitors: 4

Some days after after development and running on EMR, I saw a note on Piazza indicating date should be included in the output but that isn’t really a concern for me, as all of the timestamps are from only one day.

One last run, for Query 3, “-q 3”:

|  |
| --- |
| … |

OUTPUT:

Query 3: get count of unique (by userId) clicks per URL by hour:

hour: 6

url: google.com

uid u1: count of unique clicks by userId: 5

uid u2: count of unique clicks by userId: 5

uid u3: count of unique clicks by userId: 4

uid u4: count of unique clicks by userId: 4

url: harvard.edu

uid u1: count of unique clicks by userId: 4

uid u2: count of unique clicks by userId: 3

uid u3: count of unique clicks by userId: 3

uid u4: count of unique clicks by userId: 7

url: yahoo.com

uid u1: count of unique clicks by userId: 7

uid u2: count of unique clicks by userId: 5

uid u3: count of unique clicks by userId: 6

uid u4: count of unique clicks by userId: 3

hour: 7

url: google.com

uid u1: count of unique clicks by userId: 3

uid u2: count of unique clicks by userId: 5

uid u3: count of unique clicks by userId: 2

uid u4: count of unique clicks by userId: 8

url: harvard.edu

uid u1: count of unique clicks by userId: 6

uid u2: count of unique clicks by userId: 9

uid u3: count of unique clicks by userId: 1

uid u4: count of unique clicks by userId: 3

url: yahoo.com

uid u1: count of unique clicks by userId: 6

uid u2: count of unique clicks by userId: 6

uid u3: count of unique clicks by userId: 4

uid u4: count of unique clicks by userId: 5

hour: 8

url: google.com

uid u1: count of unique clicks by userId: 7

uid u2: count of unique clicks by userId: 7

uid u3: count of unique clicks by userId: 2

uid u4: count of unique clicks by userId: 4

url: harvard.edu

uid u1: count of unique clicks by userId: 6

uid u2: count of unique clicks by userId: 5

uid u3: count of unique clicks by userId: 10

uid u4: count of unique clicks by userId: 5

url: yahoo.com

uid u1: count of unique clicks by userId: 2

uid u2: count of unique clicks by userId: 2

uid u3: count of unique clicks by userId: 3

uid u4: count of unique clicks by userId: 7

hour: 9

url: google.com

uid u1: count of unique clicks by userId: 5

uid u2: count of unique clicks by userId: 3

uid u3: count of unique clicks by userId: 12

uid u4: count of unique clicks by userId: 4

url: harvard.edu

uid u1: count of unique clicks by userId: 4

uid u2: count of unique clicks by userId: 3

uid u3: count of unique clicks by userId: 6

uid u4: count of unique clicks by userId: 5

url: yahoo.com

uid u1: count of unique clicks by userId: 5

uid u2: count of unique clicks by userId: 7

uid u3: count of unique clicks by userId: 7

uid u4: count of unique clicks by userId: 5

**Problem 3A: (Points: 40)**

* **define an Avro schema for your data; explain your schema design**

Below is the schema I’ve come up with, for matching against the reducer data. It seemed apparent we would be dealing with nested “Complex Type” objects of type = map but there wasn’t much documentation I could find that made clear details on the structure.

|  |
| --- |
| { **"name"**: **"A3\_Problem3"**,  **"namespace"**: **"owenG.net.e88"**,  **"type"**: **"map"**, **"values"**: {  **"type"**: **"map"**, **"values"**: {  **"type"**: **"map"**,  **"values"**: **"long"** }  } } |

Here is a broken-out version of same schema, with my notes as to the specific items

|  |
| --- |
| * **base name for the object**   { "name": "A3\_Problem3",   * a unique namespace value   "namespace": "owenG.net.e88",   * **parent dictionary with keys = hours, values are the url dictionaries**   "type": "map", "values": {   * **the dictionary holding urls, values are dictionary**   "type": "map", "values": {   * **the final dictionary with keys = userIds and values = aggregated counts**   "type": "map",   * **looks like long/int are aliases so either would be fine here**   "values": "long"  }  }  } |

….

I’m doing a final review of homework before submission and I’m fairly certain my schema answers the question that was asked, i.e. defining a schema that holds the fullest amount of information, as present in the reduced\_map nested dictionary object. On the off chance it relates to the raw data coming from the event/log files:

2017-09-12 06:01:55 yahoo.com u3

maybe something like this:

|  |
| --- |
| {  "type": "record",  "name": "event\_click",  "fields" : [  {"name": "timestamp", "type": "string"}, // keep timestamp as string  {"name": "url", "type": “string”}  {"name": "userid", "type": “string”}  ]  } |

Apparently there is a timestamp “logical” type introduced in Avro 1.8.x and perhaps that could be used instead of string… but I don’t know enough about cost/benefit to make a real choice so I’m going with simplicity and keeping string.

{name: timestamp, type: {type: 'long', logicalType: 'timestamp-millis'}}

**….**

* **write a MR job that reads your generated in Problem 2.1 input files (as text) and writes them out in Avro format**

The mapper/reducer code for this problem is below. I originally constructed similar to problem 2, with the idea of reading the source files and running queries. But a comment in Piazza from the professor indicated we weren’t doing any transformations on the data, so instead each piece serves as the equivalent of a simple piping operation, no transformation whatsoever.

**p3\_mapper.py**

*#!/usr/bin/env python***import** sys  
  
*# HW instructions indicate the map/reduce doesn't actually do anything, end-goal is to write as avro format  
# Professor comment in Piazza: "Problem 3 does not run any queries - it only converts your plain text input data into Avro output data:"  
# so we simply consume the files that are fed in and print them out***for** line **in** sys.stdin:  
  
 print(line)

**p3\_reducer.py**

*#!/usr/bin/env python***import** sys  
  
*# HW instructions indicate the map/reduce doesn't actually do anything, end-goal is to write as avro format  
# so we simply consume the files that are fed in and print them out***for** line **in** sys.stdin:  
  
 print(line)

* **run the job on your EMR cluster - demonstrate the results**

Continuing on the same EMR cluster from P2, so all the files are in place and ready to go.

The needed avro jars may be on the master node of the EMR somewhere, at least if I specified the correct “utilities” to be installed at cluster startup, but I had already done a dry run on my local Linux VM using the latest jars. (And I want to use the “tools” jar also.)

Prepare to get all the jars…

wget http:*//apache.cs.utah.edu/avro/avro-1.8.2/java/avro-1.8.2.jar*wget http:*//apache.cs.utah.edu/avro/avro-1.8.2/java/avro-mapred-1.8.2-hadoop1.jar*wget http:*//apache.cs.utah.edu/avro/avro-1.8.2/java/avro-tools-1.8.2.jar*

Doesn’t work out, presumably because my EMR’s security group is not open to port 80, but handily enough when I scp’d up my A3 directory it also included a copy of the jars I had downloaded for testing on my Linux VM.

|  |
| --- |
|  |

The main additions in below .sh are the two avro jars that are necessary, passed in to both -files and -libjars arguments, + an outputformat argument.

|  |  |  |
| --- | --- | --- |
| -files | Optional | Specify comma-separated files to be copied to the Map/Reduce cluster |
| -libjars | Optional | Specify comma-separated jar files to include in the classpath |

That AvroTextOutputFormat value is defined as “equivalent of [TextOutputFormat](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapred/TextOutputFormat.html?is-external=true) for writing to Avro Data Files with a "bytes" schema”. My main concern was that it worked, and apparently it did.

**p3\_hadoop.sh**

|  |
| --- |
| input\_dir=/e88/A3/p2\_input  output\_dir=/e88/A3/p3\_output  hdfs dfs -rmr $output\_dir  jars="avro-1.8.2.jar,avro-mapred-1.8.2-hadoop1.jar"  STREAM\_JAR=hadoop-streaming  $STREAM\_JAR \  -D mapred.reduce.tasks=1 \  -files $jars \  -libjars $jars \  -mapper "$PWD/p3\_mapper.py" \  -reducer "$PWD/p3\_reducer.py" \  -input "$input\_dir/1\_input.log" \  -output "$output\_dir" \  -file "$PWD/p3\_mapper.py" \  -file "$PWD/p3\_reducer.py" \  -outputformat org.apache.avro.mapred.AvroTextOutputFormat |

Run ./hadoop\_p3.sh

|  |
| --- |
| … |

Do an ls of the output dir in hdfs and it does appear to have worked, or at least it wrote a file with an .avro file extension. Peeking at it in “raw” format seems to indicate reasonable results also

hdfs dfs -ls /e88/A3/p3\_output

hdfs dfs -cat /e88/A3/p3\_output/part-00000.avro | tail

|  |
| --- |
|  |

I had come across an example of using avro tools jar to see contents more clearly and that worked on my Linux vm, want to try it here also. Pull the file from hdfs to local drive, decode using the avro-tools jar along with a tojson argument and tail that output.

hdfs dfs -copyToLocal /e88/A3/p3\_output/part-00000.avro

java -jar avro-tools-1.8.2 tojson part-00000.avro

|  |
| --- |
|  |