**Owen Galvin, CSCI E-88 Homework 6**

NOTE: I'm generally trying to cut down on screenshots in favor of raw text output in order to reduce size of final Word document, but sometimes screenshots tell a better story. The general formatting pattern is to use text like below for text that I enter into a console by itself:

echo $SPARK\_HOME

And then below for lines that includes both commands I entered and also the output generated by those commands, with the text I entered in bold:

**(v34\_e88) [cloudera@localhost A6]$ spark-submit --master local[2] a6\_p1\_streaming.py**

17/11/04 11:46:33 WARN Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.2.12 instead (on interface enp0s3)

17/11/04 11:46:33 WARN Utils: Set SPARK\_LOCAL\_IP if you need to bind to another address

**NOTE: for Problems 1-3 - you can also use netcat as the streaming data source, and modify your data generation scripts to push lines into your locally running netcat server**

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

1. **Configure your Spark Streaming job to read data from an input directory using the textFileStream source, with the batch window of 1 sec**
2. **You can use the same files with input data in the format <timestamp> <URL> <userID> that you've generated for Homework5. Make sure you have at least 20 unique URLs**

There isn't too much to discuss on the event generation script as it is only the latest version of similar, previous scripts, though of course the file itself will be included in the homework submission. One note, at least for this problem and likely for the following ones, is that I've gone with a demonstration that involves netcat. More specifically it does a tail of a file into netcat, so the purpose of the below generate events script is to add a batch of events to single file, which is then fed into netcat.

**a6\_generate\_events.py (snippets)**

|  |
| --- |
| **...**  log\_file\_path = **'p1\_events/events.log' while True**:  generate\_log(log\_file\_path, visits, is\_debug)  print(**'batch written, number of events: {}'**.format(total\_num\_events))  time.sleep(1.5)  **...**  **if** \_\_name\_\_ == **'\_\_main\_\_'**:  parser = argparse.ArgumentParser(description=**'Write/append events to log'**)  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(**'-d'**, **'--debug'**, type=bool, default=**False**) |

The first snippet demonstrates who a set of events are written to events.log file every 1.5 seconds while the script is running.

After that the arguments are displayed. For this problem that will mean arguments for creating 20 userIds across three urls. The number of events will be 5, so each batch appended to the file being monitored = 20 \* 3 \* 5 = 300.

1. **Implement a job that :**
   1. **counts the number of clicks per URL per batch and prints the results after each batch**
   2. **counts the running total of clicks per URL and prints the results after each batch as well**

**a6\_p1\_streaming.py**

|  |
| --- |
| *# based off of https://hortonworks.com/hadoop-tutorial/introduction-spark-streaming/ example* **from** pyspark **import** SparkContext **from** pyspark.streaming **import** StreamingContext  *# create local spark context with two threads* sc = SparkContext(**"local[2]"**, **"A6 Problem 1"**) *# StreamingContext with batch inverval of 1 second* ssc = StreamingContext(sc, 1)  *# add checkpoining directory, necessary for stateful transformations* ssc.checkpoint(**"checkpoint"**)  *# Create a DStream* events = ssc.socketTextStream(**"localhost"**, 7777)  *# events in format of tab separated: 2017-08-02 07:31:32 google.com u05 # split on tab and then map each url as key, value = int = 1* urls = events.map(**lambda** x: x.split(**'\t'**)).map(**lambda** row: (row[1], 1)) *# use reduceByKey to sum up the per-key 1 values, to get a total per key/url* url\_counts = urls.reduceByKey(**lambda** x, y: x + y)  *# print number of clicks per url, current batch* url\_counts.pprint()  *# define function to be called by updateStateByKey in order to do stateful calculations* update\_state = **lambda** batch\_count, running\_count: sum(batch\_count) + (running\_count **or** 0)  running\_counts = urls.reduceByKey(**lambda** x, y: x + y).updateStateByKey(update\_state) *# print running totals* running\_counts.pprint()  ssc.start() ssc.awaitTermination() |

There are several inline comments but one thing to point out is that I was unable to find an elegant, or even somwhat elegant, way of printing out some kind of header info to distinguish the per-batch counts from the running total counts. It is obvious enough from the results but it would have been nice to get that in also.

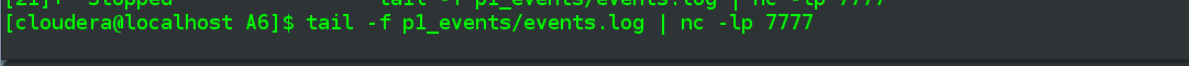
The port number = 7777 matches what netcat will be listening on.

1. **Keep adding a new data file (with a few hundred events) into the Spark job's input directory every 1-2 seconds or so - we are simulating a "stream" of events**
2. **observe your job execution and results - you should have at least 5-10 batches completed**

There are three parts to this environment:

1. set up the netcat to tail a p1\_events/events.log file
2. run a6\_p1\_streaming.py via spark-submit
3. run a6\_generate\_events.py, which will append 300 events to events.log, every ~1.5 seconds

Step 1 involves a terminal window that starts the tail, feeding in to netcat over port 7777; nothing else is returned in this terminal while it is running.



**NOTE**: between any new runs I truncate the contents of the events file via below command

> p1\_events/events.log

Step 2 is running the python script via spark-submit

**(v34\_e88) [cloudera@localhost A6]$ spark-submit --master local[2] a6\_p1\_streaming.py**

17/11/04 11:46:33 WARN Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.2.12 instead (on interface enp0s3)

17/11/04 11:46:33 WARN Utils: Set SPARK\_LOCAL\_IP if you need to bind to another address

17/11/04 11:46:34 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

-------------------------------------------

Time: 2017-11-04 11:46:39

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 11:46:39

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 11:46:40

-------------------------------------------

...

And in the 3rd step the generate events script is run with a setting to generate 300 events every 1.5 seconds, as previously discussed.

**(v34\_e88) [cloudera@localhost A6]$ python a6\_generate\_events.py -n 20 -u 'google.com,yahoo.com,harvard.edu' -e 5**

BEGIN, userid count: 20, number of urls: 3, number of events: 5, debug: False

batch written, number of events: 300

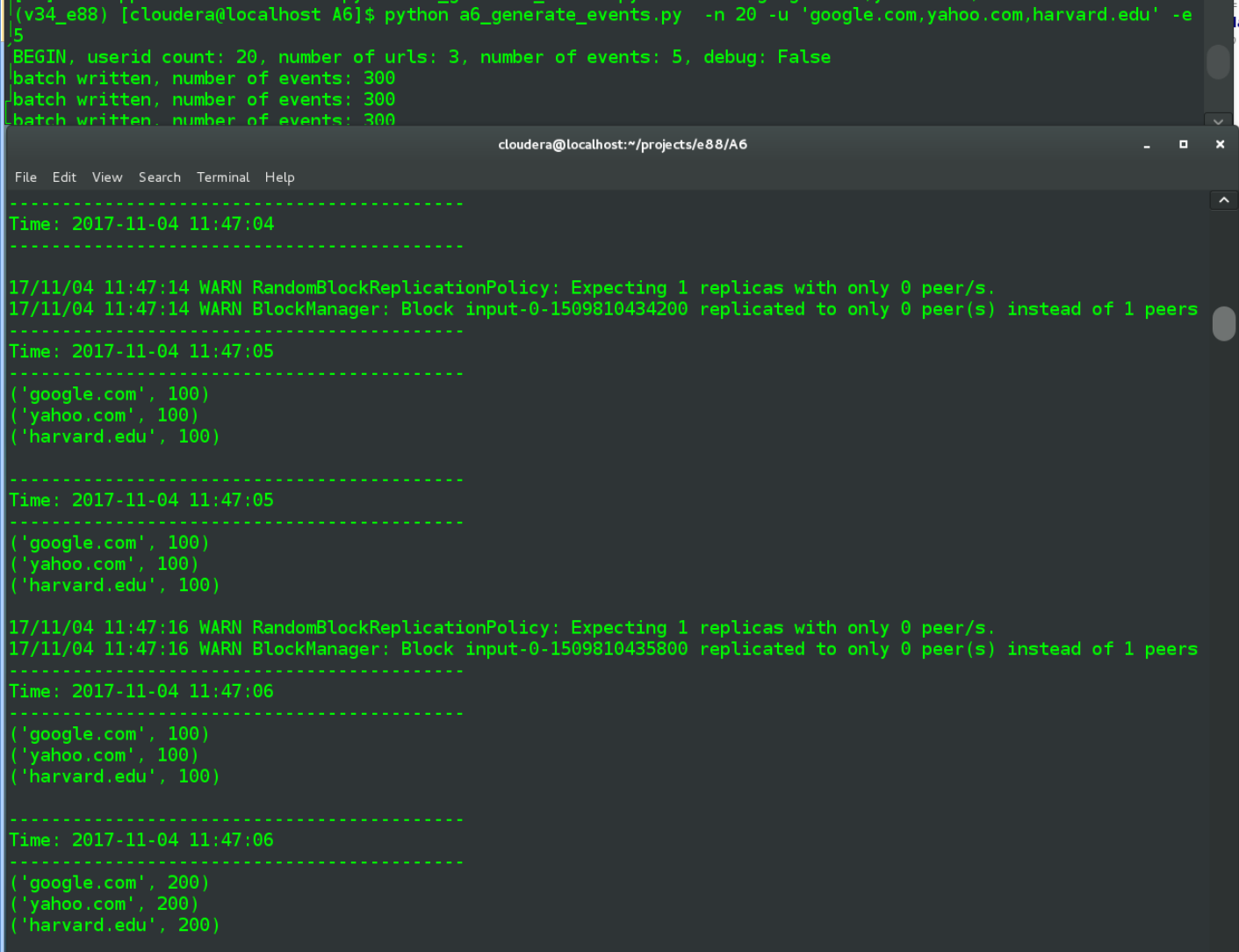
batch written, number of events: 300

batch written, number of events: 300

batch written, number of events: 300

batch written, number of events: 300

In following screenshot, step 3 is shown in top terminal, along with the beginning of batch counts followed by runnint totals



The first two sets of printouts at 11:47:05 represent the batch and running totals respectively. With the second set at 11:47:06 we can see another 100,100, 100 being sent in new batch, while the running total now shows 200, 200, 200, i.e. 100+100, 100+100, 100+1oo.

That pattern continues for several more batches. Note that the batches are being fed in every 1.5 seconds, so for approximately every third 1-second window, the per-window batch value will be empty - during that 1 second range no events were written to the log. But the running total will be output, with no further aggregation since the previous running total.

e.g. at Time: 2017-11-04 11:47:13 below there was no new batch. The running total from the previous second of 11:47:12 was 600 and that value is again displayed for the running total portion of 1 second interval represented by 11:47:13

-------------------------------------------

Time: 2017-11-04 11:47:08

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 11:47:08

-------------------------------------------

('google.com', 300)

('yahoo.com', 300)

('harvard.edu', 300)

-------------------------------------------

Time: 2017-11-04 11:47:09

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 11:47:09

-------------------------------------------

('google.com', 400)

('yahoo.com', 400)

('harvard.edu', 400)

-------------------------------------------

Time: 2017-11-04 11:47:10

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 11:47:10

-------------------------------------------

('google.com', 400)

('yahoo.com', 400)

('harvard.edu', 400)

-------------------------------------------

Time: 2017-11-04 11:47:11

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 11:47:11

-------------------------------------------

('google.com', 500)

('yahoo.com', 500)

('harvard.edu', 500)

-------------------------------------------

Time: 2017-11-04 11:47:12

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 11:47:12

-------------------------------------------

('google.com', 600)

('yahoo.com', 600)

('harvard.edu', 600)

-------------------------------------------

Time: 2017-11-04 11:47:13

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 11:47:13

-------------------------------------------

('google.com', 600)

('yahoo.com', 600)

('harvard.edu', 600)

-------------------------------------------

Time: 2017-11-04 11:47:14

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 11:47:14

-------------------------------------------

('google.com', 700)

('yahoo.com', 700)

('harvard.edu', 700)

-------------------------------------------

Time: 2017-11-04 11:47:15

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 11:47:15

-------------------------------------------

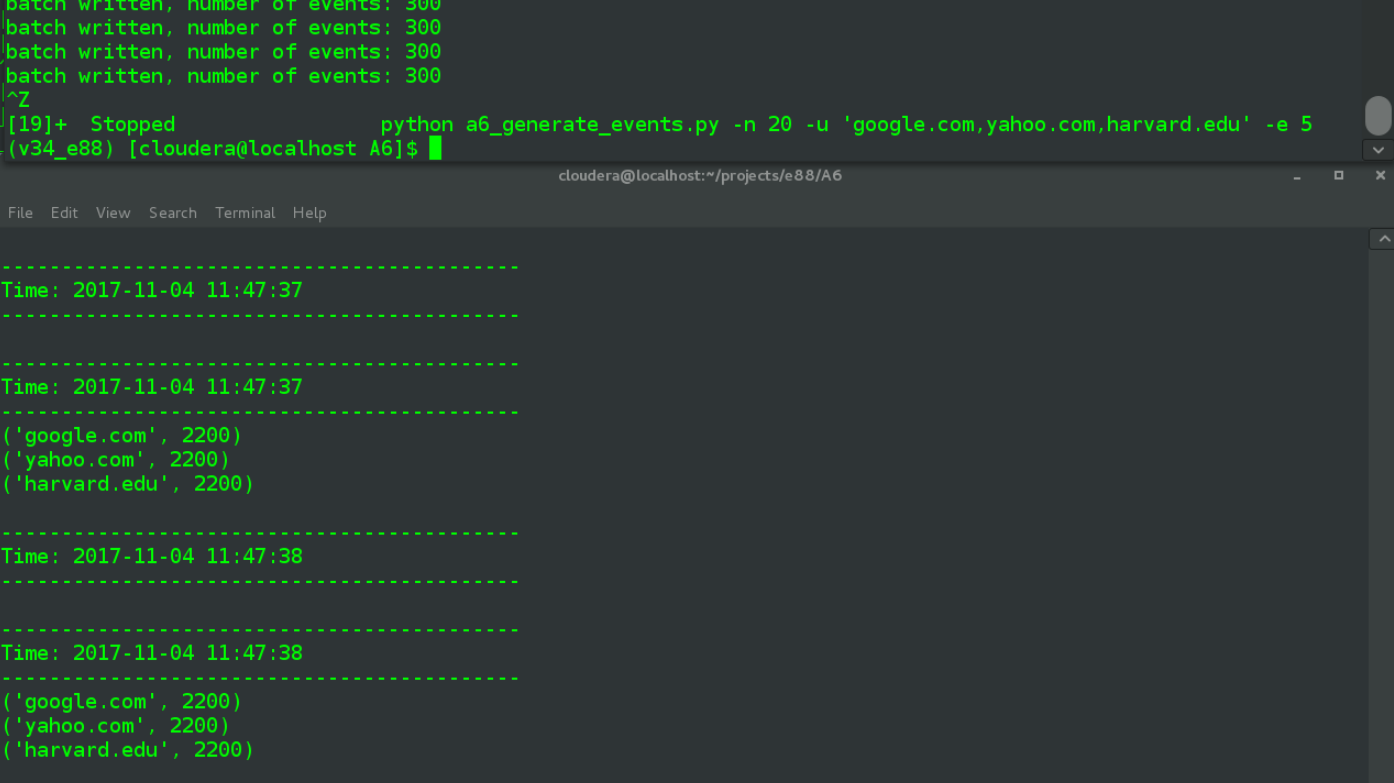
('google.com', 800)

('yahoo.com', 800)

('harvard.edu', 800)

Until I kill the generate events script:

* no new batch counts appear
* totals stop accumulating, staying at 2200 each (because 22 total batches had been written)



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

1. **Modify the job from Problem 1 to do a "windowed" count of events per URL - per 5 second windows, with the same 1 second batch duration**
2. **Use tumbling windowing, not sliding windows**

I'm understanding the above to mean that the per-1-second batch values can continue to be part of the regular output. I didn't do that initially but found displaying those values in context helped explain how everything hung together, so I went with that.

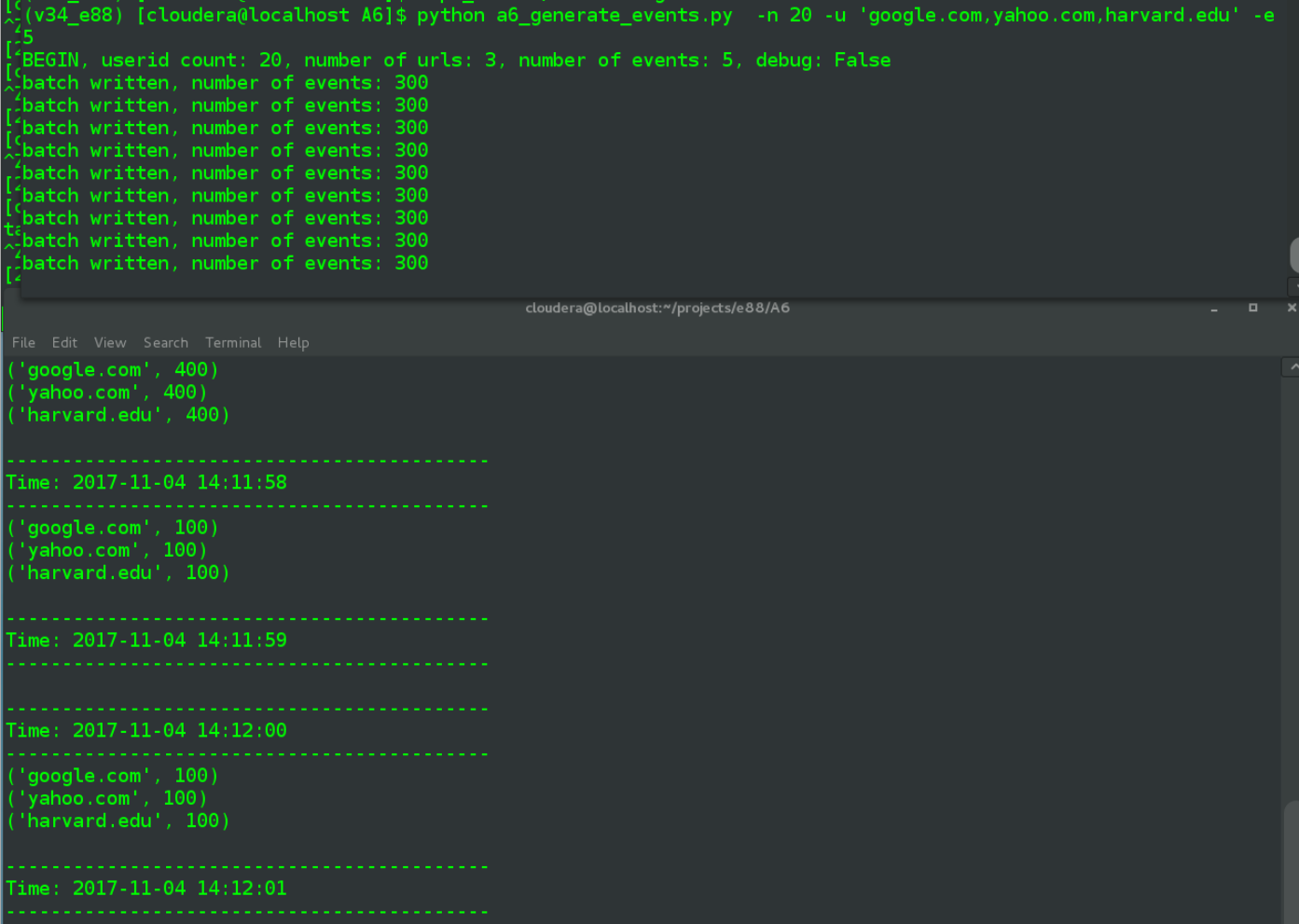
**a6\_p2\_streaming\_window.py**

|  |
| --- |
| *# 1. Modify the job from Problem 1 to do a (tumbling) "windowed" count of events per URL - per 5 second windows, with the same 1 second batch duration # drop update\_state function, call reduceByKeyAndWindow directly on the urls RDD # first arg does the adding of counts in window and the 2nd arg is an inverse of that first one, subtracting counts that have left the window # the first 5 is the window duration, i.e. 5 seconds, the final "5" arg is for slide duration # by making these last args the same value, the type of window is a non-overlapping tumbling window* running\_counts = urls.reduceByKeyAndWindow(**lambda** x, y: x + y, **lambda** x, y: x - y, 5, 5) |

The key change to the code from Problem 1 was to remove the .reduceByKey()/update\_state function, replacing them with the above .reduceByKeyAndWindow. The inline comments discuss the four arguments that are passed here, the new transformation basically does everything that is needed.

1. **Keep feeding new input files into the job's input directory every 1-2 seconds - show the results of your job - have at least 5-10 windows completed**

The three steps remain the same as from Problem 1, though I'll only show a single initial screenshot, followed by text output.



Spark-submit output. The counts related to the 5 second window aren't too diffcult to locate as they occur every 5 seconds beginning with the 57th second, i.e. 2nd, 7th, 12th ..., and will always be the second set of counts for every second-range in which there are two counts. But it still would have been nice to display some header info and instead I've manually bolded each of those per-window counts in below.

**(v34\_e88) [cloudera@localhost A6]$ spark-submit --master local[2] a6\_p2\_streaming\_window.py**

-------------------------------------------

Time: 2017-11-04 14:11:53

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:11:54

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:11:55

-------------------------------------------

('google.com', 300)

('yahoo.com', 300)

('harvard.edu', 300)

-------------------------------------------

Time: 2017-11-04 14:11:56

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:11:57

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

**-------------------------------------------**

**Time: 2017-11-04 14:11:57**

**-------------------------------------------**

**('google.com', 400)**

**('yahoo.com', 400)**

**('harvard.edu', 400)**

-------------------------------------------

Time: 2017-11-04 14:11:58

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:11:59

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:12:00

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:12:01

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:12:02

-------------------------------------------

**-------------------------------------------**

**Time: 2017-11-04 14:12:02**

**-------------------------------------------**

**('google.com', 300)**

**('yahoo.com', 300)**

**('harvard.edu', 300)**

-------------------------------------------

Time: 2017-11-04 14:12:03

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:12:04

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:12:05

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:12:06

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:12:07

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

**-------------------------------------------**

**Time: 2017-11-04 14:12:07**

**-------------------------------------------**

**('google.com', 400)**

**('yahoo.com', 400)**

**('harvard.edu', 400)**

-------------------------------------------

Time: 2017-11-04 14:12:08

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:12:09

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:12:10

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:12:11

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:12:12

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

**-------------------------------------------**

**Time: 2017-11-04 14:12:12**

**-------------------------------------------**

**('google.com', 300)**

**('yahoo.com', 300)**

**('harvard.edu', 300)**

...

-------------------------------------------

Time: 2017-11-04 14:13:18

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:13:19

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:13:20

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:13:21

-------------------------------------------

('google.com', 100)

('yahoo.com', 100)

('harvard.edu', 100)

-------------------------------------------

Time: 2017-11-04 14:13:22

-------------------------------------------

**-------------------------------------------**

**Time: 2017-11-04 14:13:22**

**-------------------------------------------**

**('google.com', 300)**

**('yahoo.com', 300)**

**('harvard.edu', 300)**

-------------------------------------------

Time: 2017-11-04 14:13:23

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:13:24

-------------------------------------------

-------------------------------------------

Time: 2017-11-04 14:13:25

-------------------------------------------

The above illustrates the effects of having batches written every 1.5 seconds. The pattern I observed was two windows with per-url totals = 300, followed by one window with per-url totals = 400; once the batches were regularly being written to file this pattern occurred over and over.

The inclusion of the per-second batch counts shows the makeup of each total - for a given 5 second window sometimes there were three batches written, sometimes four. For example, the crimson lines in above output can be described as:

* for window ending at 12:02 there were batches written during second-range-ending 11:58, 12:00, 12:01, making a 5-second window total of 300 per url at 12:02.
* following that were batch-writes at 12:03, 12:04, 12:06, 12:07, which all got caught up in the 5-second window ending at 12:07, with 400 per-url counts having been calculated.

**Problem 3: (Points: 40)**

1. **Use the same input data source (files fed into the input directory)**
2. **Calculate the number of unique users per 10 sec tumbling windows using two approaches:**

* **Using regular Spark aggregation methods**
* **Using HyperLogLog algorithm**

I initially created two separate scripts but for maintenance reasons I later merged into the one below

**a6\_p3\_streaming.py**

|  |
| --- |
| **import** argparse  **from** pyspark **import** SparkContext **from** pyspark.streaming **import** StreamingContext **import** datetime  **if** \_\_name\_\_ == **'\_\_main\_\_'**:  parser = argparse.ArgumentParser(description=**'Unique '**)  parser.add\_argument(**'-w'**, **'--window-duration'**, type=int, default=10)  parser.add\_argument(**'-a'**, **'--aggregation\_type'**, type=str, default=**'spark'**, help=**'"spark" or "hll "'**)  parser.add\_argument(**'-r'**, **'--relative-sd'**, type=float, default=0.05)  parser.add\_argument(**'-d'**, **'--debug'**, type=bool, default=**False**)  args = parser.parse\_args()   **if** args.aggregation\_type.lower() **not in** (**'spark'**, **'hll'**):  agg = **'spark'  else**:  agg = args.aggregation\_type.lower()   *# create local spark context with two threads* sc = SparkContext(**'local[2]'**, **'A6 Problem 3 {}'**.format(agg **if** agg == **'spark' else 'hll @ {}'**.format(args.relative\_sd)))  *# StreamingContext with batch inverval of 1 second* ssc = StreamingContext(sc, 1)  *# add checkpoining directory, necessary for stateful transformations* ssc.checkpoint(**"/tmp/checkpoint"**)   *# Create a DStream* events = ssc.socketTextStream(**"localhost"**, 7777)   *# events in format of tab separated: 2017-08-02 07:31:32 google.com u05  # split on tab and then map each userID as key* userids = events.map(**lambda** x: x.split(**'\t'**)).map(**lambda** row: row[2])   **if** args.debug:  userids.pprint()   *# Calculate the number of unique users per N sec tumbling windows, spark aggregation* userid\_counts = userids.countByValueAndWindow(args.window\_duration, args.window\_duration)  **if** args.debug:  *# with 20 userids the length of this rdd should never be > 20, print in full* userid\_counts.pprint(30)   **if** agg == **'spark'**:  *# the count of distinct userids in current window* count\_distinct = userid\_counts.count().map(**lambda** c: **'unique userid count, spark aggregation: {}'**.format(c))  count\_distinct.pprint()  **else**:  *#userid\_counts.count().map(lambda x: '').pprint()  # create the function that takes an RDD and performs countApproxDistinct on it* **def** hll(rdd):  stamp = datetime.datetime.now()  *# default relativeSD = 0.05, must be > 0.000017* **if** args.debug:  *# do a spread of relative standard deviation values, + whatever was passed in* relative\_sds = sorted(set([0.001, 0.05, 0.25] + [args.relative\_sd]))  **else**:  relative\_sds = [args.relative\_sd]  **for** sd **in** relative\_sds:  *# add timestamp value to output becausse "streaming window timestamp" does not print for HLL calcs  # my gues is it directly relates to use of foreeachRDD()* print(**'{} countApproxDistinct w/relative SD of {}: '**.format(stamp, sd), rdd.countApproxDistinct(sd), flush=**True**)  *# window duration and slide duration of equal values results in tumbling window* userids.window(args.window\_duration, args.window\_duration).foreachRDD(hll)   ssc.start()  ssc.awaitTermination() |

Above is my combined code for doing either basic spark aggregation or HLL (via .countApproxDistinct(). I'll note that I spent around 3 hours trying to get transform(), the function pointed out in the lab, to work in a Python environment. In the end it would only return a number and I wasn't able to include that number in the output. My conclusion is that it works differently in Scala vs. Python, Either way, integrating .foreachRDD() instead wound up being relatively straightforward.

The commmand line arguments include:

* -w for the window duration, which will always be a tumbling window
* -a aggregation type, either "spark" or "hll"
* -r for relative SD, which will be ignores if -a = "spark"
* -d for debug output, which will include .pprint() on lower level DStreams

The "standard" spark aggregation is done using countByValueAndWindow() followed by a .count directly on those results, seems appending text header at this point is pretty simple. The HLL aggregation is done using countApproxDistinct() as discussed above.

First I'll do a quick debug printout of a section that includes the window total, to show how the various elements look in their raw state, including the intermediate userid\_counts DStream that itself is count()'d to get the final displayed value of unique user ids in this window.

**(v34\_e88) [cloudera@localhost A6]$ spark-submit a6\_p3\_streaming.py -w 10 -a 'spark' -d True**

-------------------------------------------

Time: 2017-11-06 20:48:21

-------------------------------------------

...

-------------------------------------------

Time: 2017-11-06 20:48:30

-------------------------------------------

u05

u05

u20

u19

u07

u17

u13

u13

u04

u11

...

-------------------------------------------

Time: 2017-11-06 20:48:30

-------------------------------------------

('u15', 90)

('u05', 90)

('u06', 90)

('u10', 90)

('u16', 90)

('u19', 90)

('u01', 90)

('u12', 90)

('u07', 90)

('u20', 90)

('u04', 90)

('u17', 90)

('u14', 90)

('u13', 90)

('u02', 90)

('u11', 90)

('u18', 90)

('u08', 90)

('u09', 90)

('u03', 90)

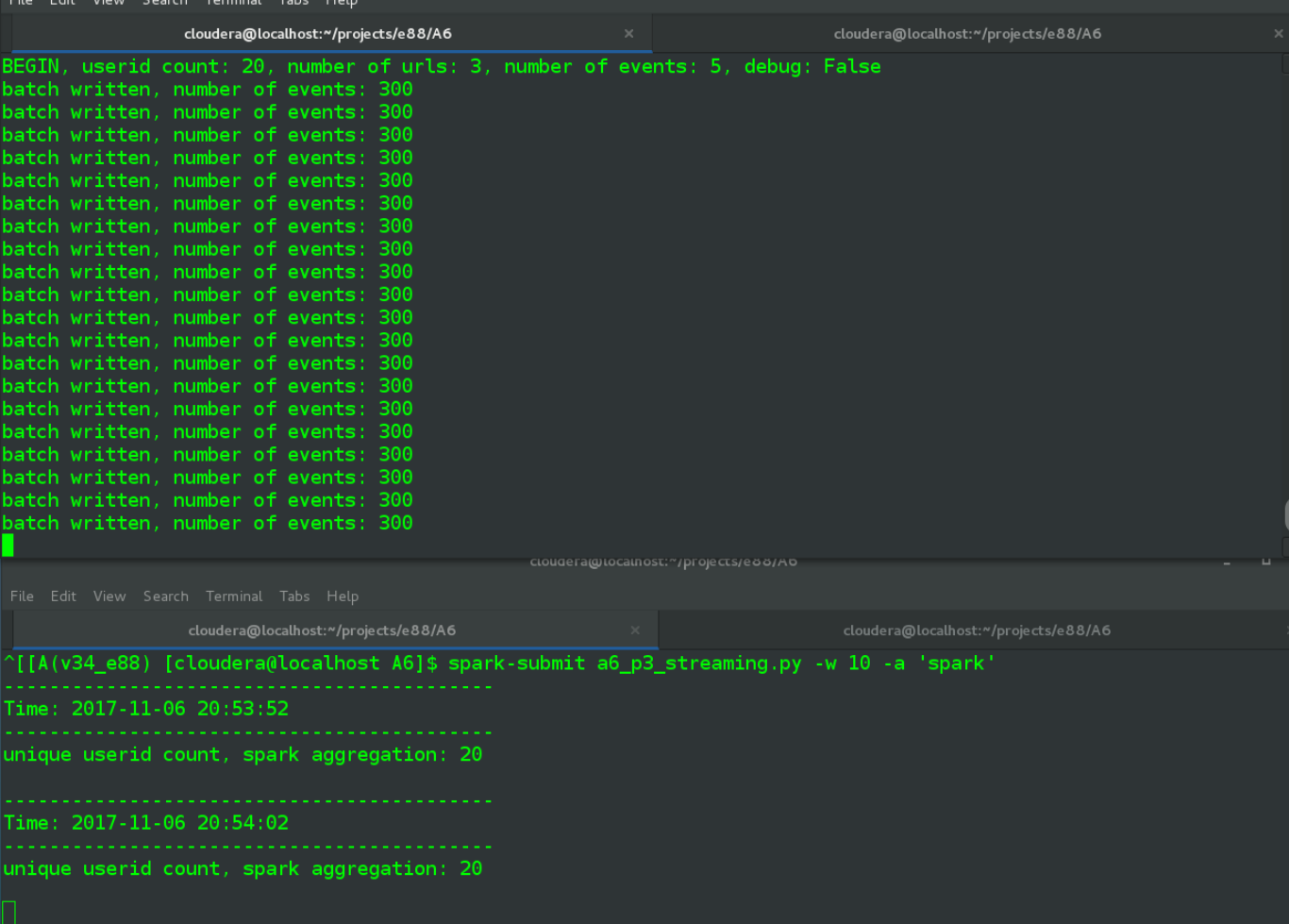
-------------------------------------------

Time: 2017-11-06 20:48:30

-------------------------------------------

unique userid count, spark aggregation: 20

Next is a straight "spark" aggregation count without debug, first with a screenshot:



Not very exciting but here is the text version, which goes for a few more windows. In each case 20 userids are counted, which matches the total number that was passed into the log generation script. That script generates userids using a call to range(), so it isn't like they are being randomly generated, they go consecutively from u01 to u20. One would be very surprised if the total count here were not = 20.

**(v34\_e88) [cloudera@localhost A6]$ spark-submit a6\_p3\_streaming.py -w 10 -a 'spark'**

-------------------------------------------

Time: 2017-11-06 20:53:52

-------------------------------------------

unique userid count, spark aggregation: 20

-------------------------------------------

Time: 2017-11-06 20:54:02

-------------------------------------------

unique userid count, spark aggregation: 20

-------------------------------------------

Time: 2017-11-06 20:54:12

-------------------------------------------

unique userid count, spark aggregation: 20

-------------------------------------------

Time: 2017-11-06 20:54:22

-------------------------------------------

unique userid count, spark aggregation: 20

-------------------------------------------

Time: 2017-11-06 20:54:32

-------------------------------------------

unique userid count, spark aggregation: 20

-------------------------------------------

Time: 2017-11-06 20:54:42

-------------------------------------------

unique userid count, spark aggregation: 20

-------------------------------------------

Time: 2017-11-06 20:54:52

-------------------------------------------

unique userid count, spark aggregation: 20

* **Try  2-3 different accuracy settings for HLL, and have at least 5-10 windows completed**

Running a debug session first, as a preview for the later singular tests. (Only after running this sample did I add a "proactive" timestamp to the standard HLL output). When debug argument is set the HLL values are hard-coded to cycle through SD values = 0.001, 0.05, 0.25, plus whatever SD value was passed as an argument, assuming it is not already covered in that list.

**(v34\_e88) [cloudera@localhost A6]$ spark-submit a6\_p3\_streaming.py -w 10 -a 'hll' -r 0.05 -d True**

-------------------------------------------

Time: 2017-11-06 21:05:21

-------------------------------------------

-------------------------------------------

Time: 2017-11-06 21:05:22

-------------------------------------------

u08

u09

u03

u18

u11

u07

u18

u01

u19

u01

...

-------------------------------------------

Time: 2017-11-06 21:05:29

-------------------------------------------

u08

u09

u03

u18

u11

u07

u18

u01

u19

u01

...

-------------------------------------------

Time: 2017-11-06 21:05:30

-------------------------------------------

-------------------------------------------

Time: 2017-11-06 21:05:30

-------------------------------------------

('u15', 105)

('u05', 105)

('u06', 105)

('u10', 105)

('u07', 105)

('u19', 105)

('u01', 105)

('u12', 105)

('u16', 105)

('u14', 105)

('u20', 105)

('u03', 105)

('u17', 105)

('u04', 105)

('u13', 105)

('u02', 105)

('u11', 105)

('u18', 105)

('u08', 105)

('u09', 105)

countApproxDistinct w/relative SD of 0.001: 20

countApproxDistinct w/relative SD of 0.05: 20

countApproxDistinct w/relative SD of 0.25: 22

-------------------------------------------

* **Compare the results of HLL with different accuracy settings and the regular aggregation results**

The earlier test of "standard" aggregation unsurprising return the number 20, for each of the several windows that completed.

unique userid count, spark aggregation: 20

Now, run the HLL version three times, each with a different Relative Standard Deviation value:

* + 0.001
  + 0.05
  + 0.25

0.001

**(v34\_e88) [cloudera@localhost A6]$ spark-submit a6\_p3\_streaming.py -w 10 -a 'hll' -r 0.001**

2017-11-06 21:31:22.273865 countApproxDistinct w/relative SD of 0.001: 20

2017-11-06 21:31:32.222947 countApproxDistinct w/relative SD of 0.001: 20

2017-11-06 21:31:42.145685 countApproxDistinct w/relative SD of 0.001: 20

2017-11-06 21:31:52.122374 countApproxDistinct w/relative SD of 0.001: 20

2017-11-06 21:32:02.211081 countApproxDistinct w/relative SD of 0.001: 20

2017-11-06 21:32:12.106879 countApproxDistinct w/relative SD of 0.001: 20

2017-11-06 21:32:22.115711 countApproxDistinct w/relative SD of 0.001: 20

0.05

**(v34\_e88) [cloudera@localhost A6]$ spark-submit a6\_p3\_streaming.py -w 10 -a 'hll' -r 0.05**

2017-11-06 21:33:51.336629 countApproxDistinct w/relative SD of 0.05: 20

2017-11-06 21:34:01.137702 countApproxDistinct w/relative SD of 0.05: 20

2017-11-06 21:34:11.114707 countApproxDistinct w/relative SD of 0.05: 20

2017-11-06 21:34:21.145729 countApproxDistinct w/relative SD of 0.05: 20

2017-11-06 21:34:31.117691 countApproxDistinct w/relative SD of 0.05: 20

2017-11-06 21:34:41.164129 countApproxDistinct w/relative SD of 0.05: 20

2017-11-06 21:34:51.150776 countApproxDistinct w/relative SD of 0.05: 20

0.25

**(v34\_e88) [cloudera@localhost A6]$ spark-submit a6\_p3\_streaming.py -w 10 -a 'hll' -r 0.25**

2017-11-06 21:36:09.359531 countApproxDistinct w/relative SD of 0.25: 22

2017-11-06 21:36:19.219332 countApproxDistinct w/relative SD of 0.25: 22

2017-11-06 21:36:29.123575 countApproxDistinct w/relative SD of 0.25: 22

2017-11-06 21:36:39.162523 countApproxDistinct w/relative SD of 0.25: 22

2017-11-06 21:36:49.134552 countApproxDistinct w/relative SD of 0.25: 22

2017-11-06 21:36:59.103544 countApproxDistinct w/relative SD of 0.25: 22

2017-11-06 21:37:09.125008 countApproxDistinct w/relative SD of 0.25: 22

With the two lower relativeSD values the count remained accurate at 20. Bumping it up to 0.25 though resulted in decreased accuracy, where it was now registering 22 unique userids.

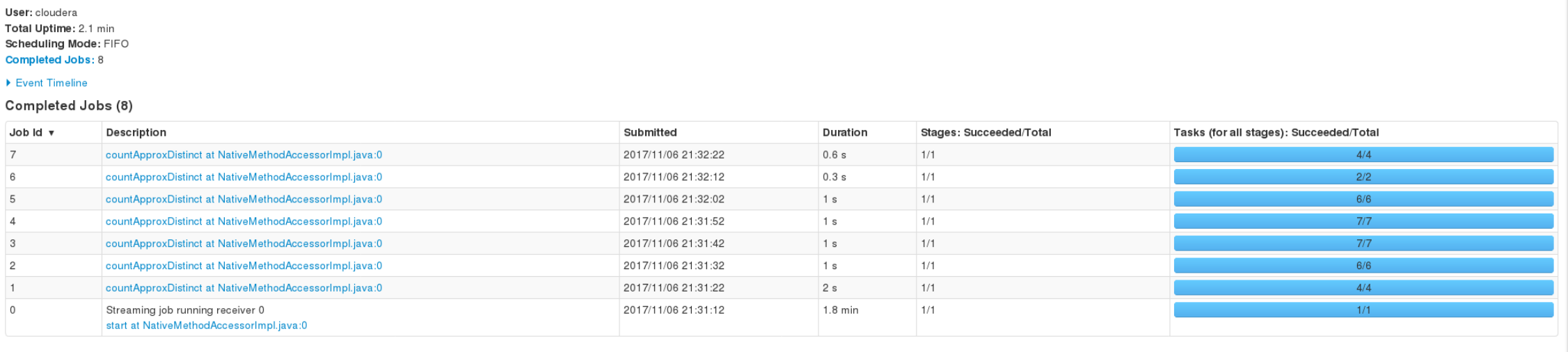
* **Compare the execution flow (number of jobs) for HLL with different accuracy settings - explain the results**

History server showing the three spark-submit runs.

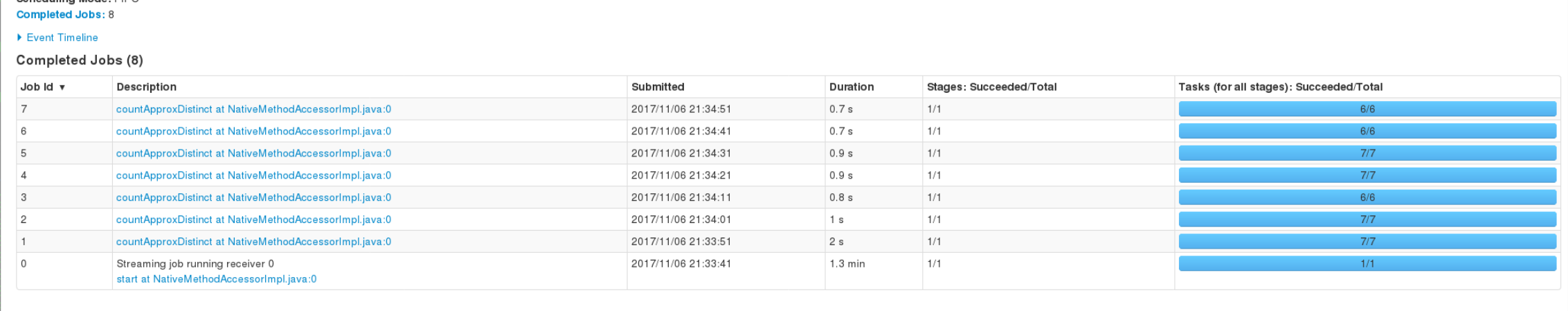


And... no difference in the job numbers, beyond the count tracking the number of windows, i.e. 5 windows results in 5 jobs while 8 windows completed results in 8 jobs. My a6\_p3\_streaming.py created app names that included the relativeSD values, so I'm certain the screenshots are accurate.

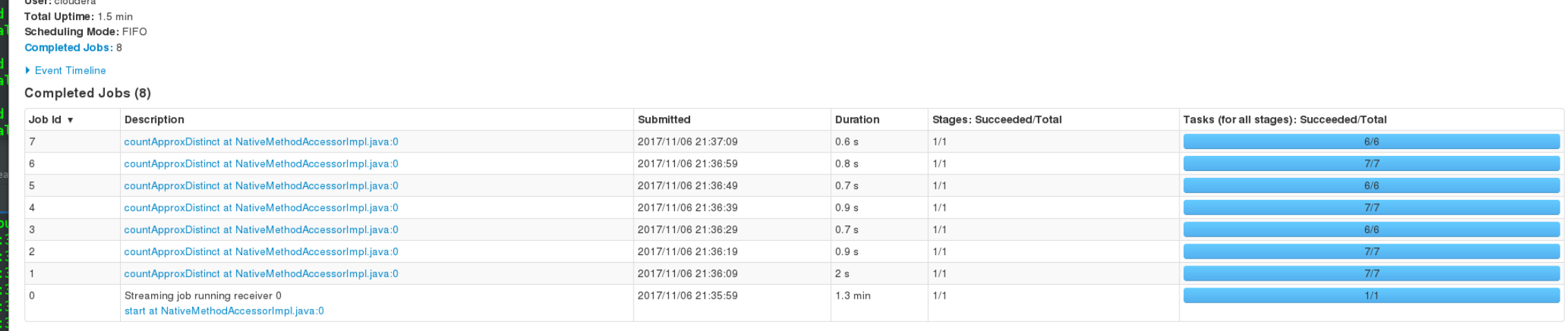
0.001 = 8 completed jobs



0.05, 8 completed jobs



0.25, 8 jobs



I did an additional run at 0.75 SD and again so no difference. The wording of this bullet of course implies that differing accuracy values would lead to differing numbers of jobs.The lecture notes speak of "m" independent substreams being spawned by the HyperLogLog algorithm and my best guess is that the raw number of events is not large enough to cause spark to launch new executors to correspond to the substreams, the resource needs just aren't there. There is also a strong possibility that implementation of the transform function would have resulted in the expected job-number differences.

**Problem 4: (Points: 40)**

1. **Modify your Problem 3 job to take input from Kafka , instead of an input directory; use only HLL method of counting unique users - you are still counting unique users per 10 sec tumbling windows**

Here we have the rewritten file

**a6\_p4\_kafka.py**

|  |
| --- |
| **import** sys  **from** pyspark **import** SparkContext **from** pyspark.streaming **import** StreamingContext **from** pyspark.streaming.kafka **import** KafkaUtils **import** datetime  **if** \_\_name\_\_ == **'\_\_main\_\_'**:  **if** len(sys.argv) != 3:  exit(**'Usage: a6\_p4\_kafka.py <topic>'**)   sc = SparkContext(appName=**'Problem 4'**)  *# 1 second duratoin* ssc = StreamingContext(sc, 1)   *# add checkpoining directory, necessary for stateful transformations* ssc.checkpoint(**"/tmp/checkpoint"**)   topic = sys.argv[1]  window\_duration = int(sys.argv[2])  group\_id = **'p4\_group'** *# KafkaUtils.createStream(streamingContext, [ZK quorum], [consumer group id], [per-topic number of Kafka partitions to consume])* kvs = KafkaUtils.createStream(ssc, **'localhost:2181'**, group\_id, {topic: 1})   *# events in format of tab separated: 2017-08-02 07:31:32 google.com u05  # use map to pull out value and discard key (index=1), then split on tab and then map each userID as key* userids = kvs.map(**lambda** x: x[1]).map(**lambda** x: x.split(**'\t'**)).map(**lambda** row: (row[2]))   **def** hll(rdd):  stamp = datetime.datetime.now()  *# add timestamp value to output becausse "streaming window timestamp" does not print for HLL calcs  # my gues is it directly relates to use of foreeachRDD()* sd = 0.05  print(**'{} countApproxDistinct w/relative SD of {}: '**.format(stamp, sd), rdd.countApproxDistinct(sd), flush=**True**)   *# window duration and slide duration of equal values results in tumbling window* userids.window(window\_duration, window\_duration).foreachRDD(hll)   ssc.start()  ssc.awaitTermination() |

There are two arguments, both required:

1. topic name
2. window duration (ability to pass in shorter window helped during development)

The countApproxDistinct() is handled same as in earlier problem, though it is hardcoded with relativeSD = 0.05, which is the default value anyway.

1. **Setup Kafka (either locally or remote) and create a topic "hw6problem4" with 4 partitions for that**

Get kafka up and running, leave the two terminals alone once they are started:

* in one terminal start zookeeper

**[cloudera@localhost ~]$ zookeeper-server-start.sh $KAFKA\_HOME/config/zookeeper.properties**

[2017-11-05 13:24:05,329] INFO Reading configuration from: /opt/kafka/kafka\_2.11-0.11.0.1/config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)

[2017-11-05 13:24:05,335] INFO autopurge.snapRetainCount set to 3 (org.apache.zookeeper.server.DatadirCleanupManager)

[2017-11-05 13:24:05,335] INFO autopurge.purgeInterval set to 0 (org.apache.zookeeper.server.DatadirCleanupManager)

[2017-11-05 13:24:05,335] INFO Purge task is not scheduled. (org.apache.zookeeper.server.DatadirCleanupManager)

[2017-11-05 13:24:05,336] WARN Either no config or no quorum defined in config, running in standalone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)

..

* and another terminal start the kafka server

**[cloudera@localhost ~]$ kafka-server-start.sh $KAFKA\_HOME/config/server.properties**

[2017-11-05 13:24:25,996] INFO KafkaConfig values:

advertised.host.name = null

advertised.listeners = null

advertised.port = null

alter.config.policy.class.name = null

authorizer.class.name =

...

Create the hw6problem4 topic, passing in "--partitions 4" to configure with requested 4 partitions

**[cloudera@localhost ~]$ kafka-topics.sh --create topic: kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 4 --topic hw6problem4**

Created topic "hw6problem4".

1. **Test the results first by manually sending events (strings in the specified format) via the Kafka console producer**

Start the console producer:

[cloudera@localhost kafka\_2.11-0.11.0.1]$ kafka-console-producer.sh --broker-list localhost:9092 --topic hw6problem4

spark-submit the a6\_p4\_kafka.py, passing in the hw6problem4 topic along with a window duration = 10 seconds

**(v34\_e88) [cloudera@localhost A6]$ spark-submit --packages org.apache.spark:spark-streaming-kafka-0-8\_2.11:2.1.0 a6\_p4\_kafka.py hw6problem4 10**

Ivy Default Cache set to: /home/cloudera/.ivy2/cache

The jars for the packages stored in: /home/cloudera/.ivy2/jars

:: loading settings :: url = jar:file:/opt/spark/spark-2.1.1-bin-hadoop2.7/jars/ivy-2.4.0.jar!/org/apache/ivy/core/settings/ivysettings.xml

org.apache.spark#spark-streaming-kafka-0-8\_2.11 added as a dependency

:: resolving dependencies :: org.apache.spark#spark-submit-parent;1.0

confs: [default]

...

Now back to console and paste a line, followed a few seconds later by pasting several lines, and a again another batch of a few lines:

>2017-06-01 09:07:55 google.com u13

>2017-06-02 08:58:25 yahoo.com u14

2017-06-02 10:15:17 yahoo.com u19

2017-06-02 10:37:29 google.com u01

2017-06-02 11:37:23 harvard.edu u19

2017-06-02 11:38:56 harvard.edu u16

2017-06-02 12:27:15 yahoo.com u09

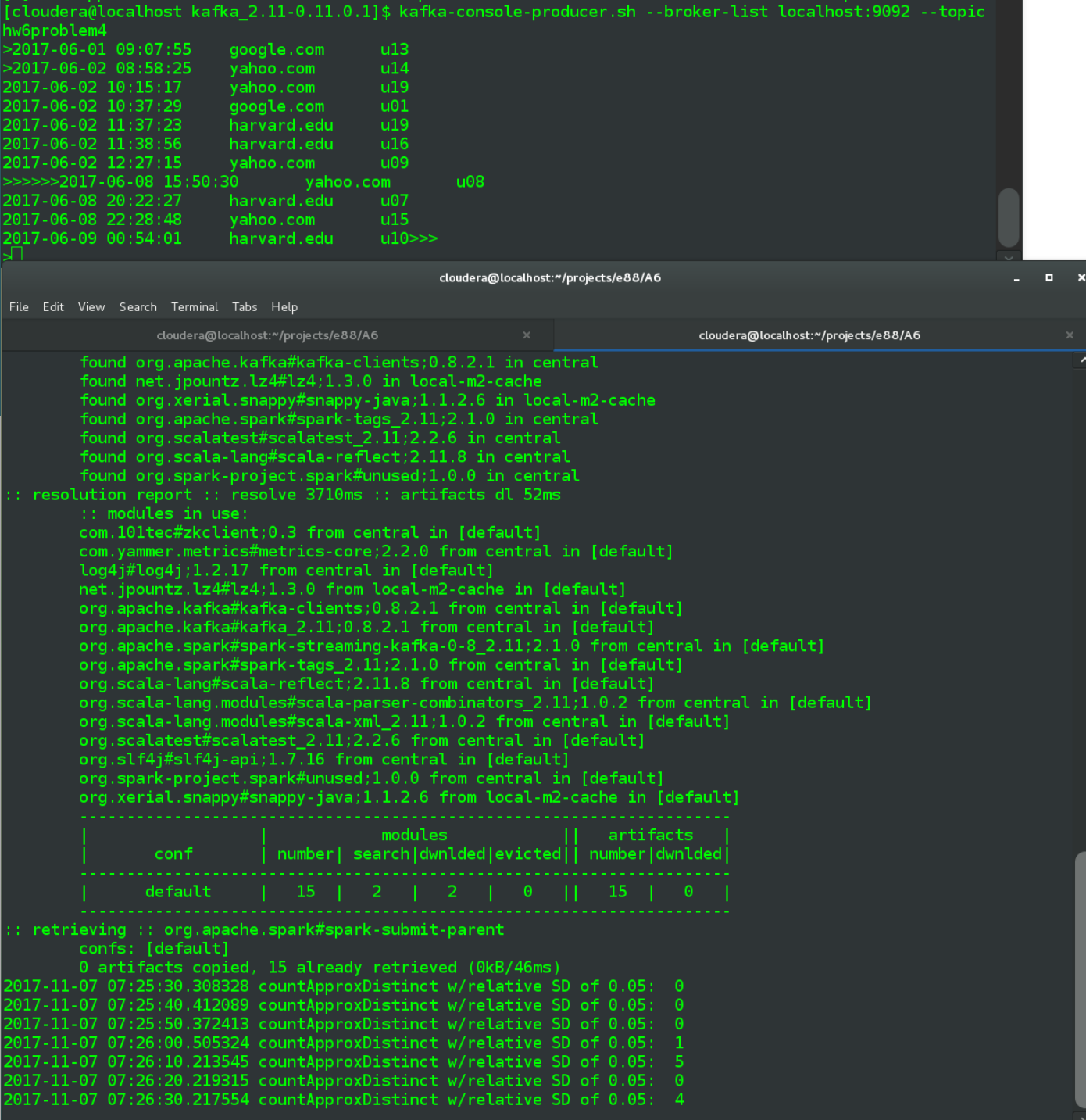
>>>>>>2017-06-08 15:50:30 yahoo.com u08

2017-06-08 20:22:27 harvard.edu u07

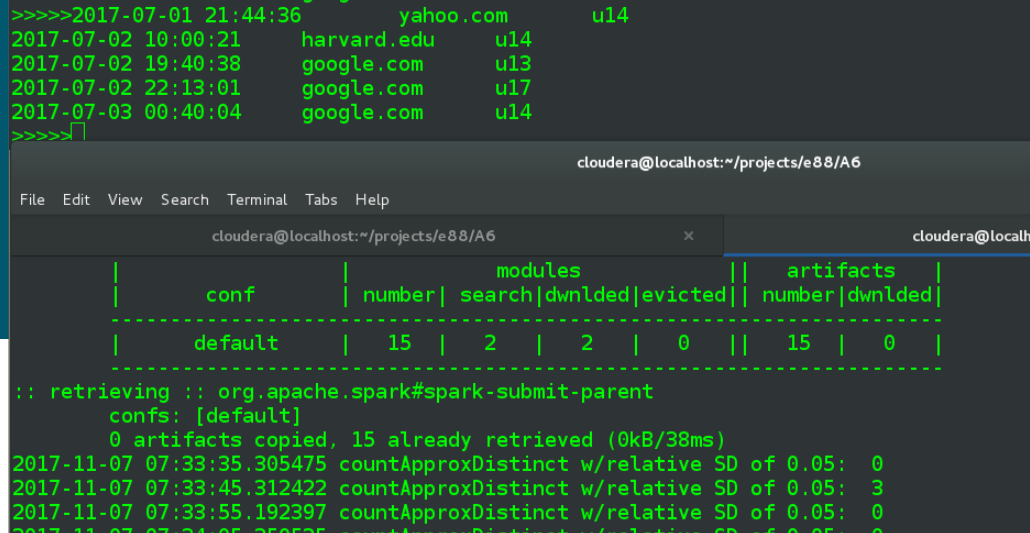
2017-06-08 22:28:48 yahoo.com u15

2017-06-09 00:54:01 harvard.edu u10>>>

Below shows the overal execution, where the spark job spun for several seconds, until I pasted the first line



And one more for good measure because the above didn't have multiple userids within the same batch. Below we see the three u14 values get counted as 1 distinct userid (+u13+u17 = 3 distinct)



1. **Next, use your previously developed KafkaProducer (or take one from one of the showcase solutions) that sends events in this format; modify it to send about 200 events per second, with various userIDs and URLs**

Bulk of script came from a previous assignment, problem 10 in assgnment 4 I believe. I was only going to add snippets from the file below but they tended to be scattered and the file wasn't all that large anyway.

**a6\_p4\_producer.py**

|  |
| --- |
| **...**  **from** kafka **import** KafkaConsumer, KafkaProducer   *# 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'* **return** collections.deque(**'2017-09-12 {hour:02}:{minute:02}:{second:02}'**.format(  hour=randint(0,23), 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** produce\_topic\_events(thread\_number, data, is\_debug=**False**):  *# use same url as passed into the --broker-list arge for kafka-console-producer.sh* producer = KafkaProducer(bootstrap\_servers=**'localhost:9092'**)topic\_name = **'hw6problem4'   for** line **in** data: *# use 3 lines below for first part of problem 10* producer.send(topic\_name, str.encode(line))  **if** is\_debug:  print(**'-- topic {}, message sent: : {}'**.format(topic\_name, line))   producer.flush()  **def** do\_threads(num\_userids, urls, event\_count, thread\_count, total\_num\_events, is\_debug):  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) # skip this for Assignment 4/Problem 10* **pass** log\_threads = []  **for** i **in** range(1, thread\_count + 1):  *#new\_thread = Thread(target=generate\_log, args=(i, output\_dir\_path, visits, is\_debug))* new\_thread = Thread(target=produce\_topic\_events, args=(i, 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]   **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))   **while True**:  print(**'sending batch at {}'**.format(datetime.datetime.now()))  do\_threads(args.num\_userids, urls, args.event\_count, args.thread\_count, total\_event\_count, args.debug)  time.sleep(1) |

The do\_threads() is pretty much an artifact of the old problem on which the script was based and my tests are only with thread count = 1. The main arguments have been discussed in previous assignments and only one combination is used for this problem. The main do\_threads() is called in an infinite loop, sleeping 1 second in between each loop to match problem details.

1. **have at least 5-10 windows completed**
2. **Run your job and demonstrate the counts**

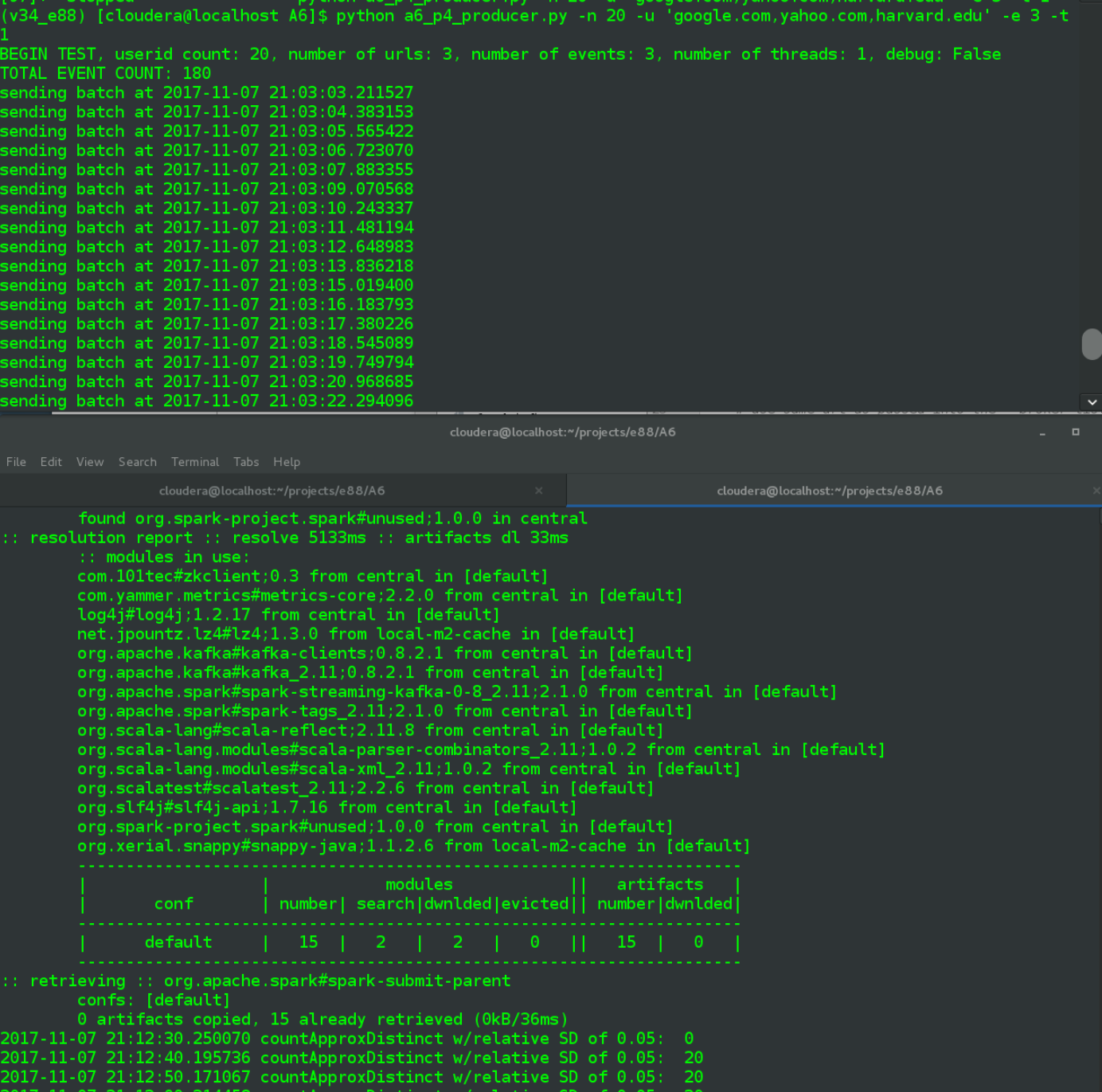
In first window, begin the spark-submit job, passing in the specified topic name along with a window duration of 10 seconds.

spark-submit --packages org.apache.spark:spark-streaming-kafka-0-8\_2.11:2.1.0 a6\_p4\_kafka.py hw6problem4 10

In separate terminal, start the producer script, below will create 180 events (20 userid \* 3 url \* 3 events) every 1 second:

python a6\_p4\_producer.py -n 20 -u 'google.com,yahoo.com,harvard.edu' -e 3 -t 1

Below is a screenshot from my run, where I began the producer script once the first window (with count = 0) had been calculated in spark. Given 180 events a second, each batch with all 20 userids present, it would be expected to see count = 20 and indeed that is what we get.



And the text versions of same, covering several 10 second windows, producer followed by spark

**(v34\_e88) [cloudera@localhost A6]$ python a6\_p4\_producer.py -n 20 -u 'google.com,yahoo.com,harvard.edu' -e 3 -t 1**

BEGIN TEST, userid count: 20, number of urls: 3, number of events: 3, number of threads: 1, debug: False

TOTAL EVENT COUNT: 180

sending batch at 2017-11-07 21:12:32.366108

sending batch at 2017-11-07 21:12:33.566396

sending batch at 2017-11-07 21:12:34.739001

sending batch at 2017-11-07 21:12:35.898303

sending batch at 2017-11-07 21:12:37.089432

sending batch at 2017-11-07 21:12:38.252861

sending batch at 2017-11-07 21:12:39.432614

sending batch at 2017-11-07 21:12:40.609408

sending batch at 2017-11-07 21:12:41.821570

sending batch at 2017-11-07 21:12:43.049879

sending batch at 2017-11-07 21:12:44.205294

sending batch at 2017-11-07 21:12:45.467390

sending batch at 2017-11-07 21:12:46.651176

sending batch at 2017-11-07 21:12:47.855975

sending batch at 2017-11-07 21:12:49.064227

sending batch at 2017-11-07 21:12:50.228329

sending batch at 2017-11-07 21:12:51.457499

sending batch at 2017-11-07 21:12:52.866853

sending batch at 2017-11-07 21:12:54.046278

sending batch at 2017-11-07 21:12:55.230517

sending batch at 2017-11-07 21:12:56.437976

sending batch at 2017-11-07 21:12:57.748470

sending batch at 2017-11-07 21:12:58.928808

sending batch at 2017-11-07 21:13:00.106238

sending batch at 2017-11-07 21:13:01.298430

sending batch at 2017-11-07 21:13:02.512271

sending batch at 2017-11-07 21:13:03.690161

sending batch at 2017-11-07 21:13:04.850029

sending batch at 2017-11-07 21:13:06.068251

sending batch at 2017-11-07 21:13:07.251771

sending batch at 2017-11-07 21:13:08.468341

sending batch at 2017-11-07 21:13:09.619542

sending batch at 2017-11-07 21:13:10.796044

sending batch at 2017-11-07 21:13:12.053473

sending batch at 2017-11-07 21:13:13.237095

sending batch at 2017-11-07 21:13:14.394975

sending batch at 2017-11-07 21:13:15.587455

sending batch at 2017-11-07 21:13:16.785890

sending batch at 2017-11-07 21:13:17.951638

sending batch at 2017-11-07 21:13:19.128653

sending batch at 2017-11-07 21:13:20.311304

sending batch at 2017-11-07 21:13:21.472044

sending batch at 2017-11-07 21:13:22.639942

sending batch at 2017-11-07 21:13:23.831746

sending batch at 2017-11-07 21:13:25.026368

sending batch at 2017-11-07 21:13:26.231770

sending batch at 2017-11-07 21:13:27.388284

sending batch at 2017-11-07 21:13:28.573434

sending batch at 2017-11-07 21:13:29.717678

sending batch at 2017-11-07 21:13:30.863199

sending batch at 2017-11-07 21:13:32.058053

sending batch at 2017-11-07 21:13:33.201160

sending batch at 2017-11-07 21:13:34.349737

sending batch at 2017-11-07 21:13:35.559576

sending batch at 2017-11-07 21:13:36.704087

sending batch at 2017-11-07 21:13:37.862609

sending batch at 2017-11-07 21:13:39.012114

sending batch at 2017-11-07 21:13:40.182701

sending batch at 2017-11-07 21:13:41.458422

sending batch at 2017-11-07 21:13:42.783588

sending batch at 2017-11-07 21:13:43.962880

sending batch at 2017-11-07 21:13:45.113572

sending batch at 2017-11-07 21:13:46.263766

sending batch at 2017-11-07 21:13:47.419079

sending batch at 2017-11-07 21:13:48.628263

sending batch at 2017-11-07 21:13:49.813920

sending batch at 2017-11-07 21:13:50.963409

sending batch at 2017-11-07 21:13:52.135900

sending batch at 2017-11-07 21:13:53.277468

sending batch at 2017-11-07 21:13:54.422538

sending batch at 2017-11-07 21:13:55.563209

sending batch at 2017-11-07 21:13:56.741002

sending batch at 2017-11-07 21:13:57.977688

sending batch at 2017-11-07 21:13:59.129563

sending batch at 2017-11-07 21:14:00.286113

sending batch at 2017-11-07 21:14:01.464253

sending batch at 2017-11-07 21:14:02.656100

spark-submit

**(v34\_e88) [cloudera@localhost A6]$ spark-submit --packages org.apache.spark:spark-streaming-kafka-0-8\_2.11:2.1.0 a6\_p4\_kafka.py hw6problem4 10**

Ivy Default Cache set to: /home/cloudera/.ivy2/cache

The jars for the packages stored in: /home/cloudera/.ivy2/jars

:: loading settings :: url = jar:file:/opt/spark/spark-2.1.1-bin-hadoop2.7/jars/ivy-2.4.0.jar!/org/apache/ivy/core/settings/ivysettings.xml

org.apache.spark#spark-streaming-kafka-0-8\_2.11 added as a dependency

:: resolving dependencies :: org.apache.spark#spark-submit-parent;1.0

confs: [default]

found org.apache.spark#spark-streaming-kafka-0-8\_2.11;2.1.0 in central

found org.apache.kafka#kafka\_2.11;0.8.2.1 in central

...

org.slf4j#slf4j-api;1.7.16 from central in [default]

org.spark-project.spark#unused;1.0.0 from central in [default]

org.xerial.snappy#snappy-java;1.1.2.6 from local-m2-cache in [default]

---------------------------------------------------------------------

| | modules || artifacts |

| conf | number| search|dwnlded|evicted|| number|dwnlded|

---------------------------------------------------------------------

| default | 15 | 2 | 2 | 0 || 15 | 0 |

---------------------------------------------------------------------

:: retrieving :: org.apache.spark#spark-submit-parent

confs: [default]

0 artifacts copied, 15 already retrieved (0kB/36ms)

2017-11-07 21:12:30.250070 countApproxDistinct w/relative SD of 0.05: 0

2017-11-07 21:12:40.195736 countApproxDistinct w/relative SD of 0.05: 20

2017-11-07 21:12:50.171067 countApproxDistinct w/relative SD of 0.05: 20

2017-11-07 21:13:00.314458 countApproxDistinct w/relative SD of 0.05: 20

2017-11-07 21:13:10.172918 countApproxDistinct w/relative SD of 0.05: 20

2017-11-07 21:13:20.186442 countApproxDistinct w/relative SD of 0.05: 20

2017-11-07 21:13:30.288679 countApproxDistinct w/relative SD of 0.05: 20

2017-11-07 21:13:40.288548 countApproxDistinct w/relative SD of 0.05: 20

2017-11-07 21:13:50.152049 countApproxDistinct w/relative SD of 0.05: 20

2017-11-07 21:14:00.291289 countApproxDistinct w/relative SD of 0.05: 20

**Problem 6: (Points: 30)**

1. **Install Cassandra (either locally or on AWS/Docker) and create a "hw6" keyspace**

On my local VM, wget the package from apache and unzip to newly created /opt/cassandra directory

**[cloudera@localhost A6]$ wget http://supergsego.com/apache/cassandra/3.11.1/apache-cassandra-3.11.1-bin.tar.gz**

--2017-11-08 18:02:31-- http://supergsego.com/apache/cassandra/3.11.1/apache-cassandra-3.11.1-bin.tar.gz

Resolving supergsego.com (supergsego.com)... 173.12.119.133

Connecting to supergsego.com (supergsego.com)|173.12.119.133|:80... connected.

HTTP request sent, awaiting response... 200 OK

Length: 38104148 (36M) [application/x-gzip]

Saving to: ‘apache-cassandra-3.11.1-bin.tar.gz’

100%[==========================================================================>] 38,104,148 2.07MB/s in 18s

2017-11-08 18:02:49 (2.04 MB/s) - ‘apache-cassandra-3.11.1-bin.tar.gz’ saved [38104148/38104148]

**[cloudera@localhost A6]$ sudo mkdir /opt/cassandra**

**[sudo] password for cloudera:**

**[cloudera@localhost A6]$ sudo tar zxvf apache-cassandra-3.11.1-bin.tar.gz -C /opt/cassandra/**

apache-cassandra-3.11.1/bin/

apache-cassandra-3.11.1/conf/

apache-cassandra-3.11.1/conf/triggers/

apache-cassandra-3.11.1/doc/

apache-cassandra-3.11.1/doc/cql3/

apache-cassandra-3.11.1/doc/html/

apache-cassandra-3.11.1/doc/html/\_images/

apache-cassandra-3.11.1/doc/html/\_sources/

apache-cassandra-3.11.1/doc/html/\_sources/architecture/

apache-cassandra-3.11.1/doc/html/\_sources/configuration/

apache-cassandra-3.11.1/doc/html/\_sources/cql/

apache-cassandra-3.11.1/doc/html/\_sources/data\_modeling/

apache-cassandra-3.11.1/doc/html/\_sources/development/

apache-cassandra-3.11.1/doc/html/\_sources/faq/

apache-cassandra-3.11.1/doc/html/\_sources/getting\_started/

apache-cassandra-3.11.1/doc/html/\_sources/operating/

apache-cassandra-3.11.1/doc/html/\_sources/tools/

apache-cassandra-3.11.1/doc/html/\_sources/troubleshooting/

...

And following my general pattern, chown the extracted directory to current user = cloudera

[cloudera@localhost A6]$ sudo chown -R cloudera:cloudera /opt/cassandra/apache-cassandra-3.11.1/

After reading around a bit it seems like setting CASSANDRA\_HOME env var should allow a number of default directory locatoins to work without needing to set in cassandra.yaml. To that end I edit my.bashrc and add below, sourcing it afterwards

|  |
| --- |
| export CASSANDRA\_HOME=/opt/cassandra/apache-cassandra-3.11.1  export PATH=$PATH:$CASSANDRA\_HOME/bin |

Source .bashrc after above change

[cloudera@localhost apache-cassandra-3.11.1]$ source ~/.bashrc

Create the root cassandra directories in /var/lib and /var/log, which should work in conjunction with setting $CASSANDRA\_HOME above

[cloudera@localhost A6]$ cd /opt/cassandra/apache-cassandra-3.11.1/ cassandra/

[cloudera@localhost apache-cassandra-3.11.1]$ sudo mkdir /var/lib/cassandra

[cloudera@localhost apache-cassandra-3.11.1]$ sudo mkdir /var/log/cassandra

[cloudera@localhost apache-cassandra-3.11.1]$ sudo chown -R cloudera:cloudera /var/lib/cassandra/

[cloudera@localhost apache-cassandra-3.11.1]$ sudo chown -R cloudera:cloudera /var/log/

In separate terminal window, launch cassandra in the foreground

**[cloudera@localhost ~]$ cd $CASSANDRA\_HOME**

**[cloudera@localhost apache-cassandra-3.11.1]$ ls**

bin CHANGES.txt conf doc interface javadoc lib LICENSE.txt NEWS.txt NOTICE.txt pylib tools

[cloudera@localhost apache-cassandra-3.11.1]$ ./bin/cassandra -f

OpenJDK 64-Bit Server VM warning: Cannot open file /opt/cassandra/apache-cassandra-3.11.1/logs/gc.log due to No such file or directory

CompilerOracle: dontinline org/apache/cassandra/db/Columns$Serializer.deserializeLargeSubset (Lorg/apache/cassandra/io/util/DataInputPlus;Lorg/apache/cassandra/db/Columns;I)Lorg/apache/cassandra/db/Columns;

CompilerOracle: dontinline org/apache/cassandra/db/Columns$Serializer.serializeLargeSubset (Ljava/util/Collection;ILorg/apache/cassandra/db/Columns;ILorg/apache/cassandra/io/util/DataOutputPlus;)V

CompilerOracle: dontinline org/apache/cassandra/db/Columns$Serializer.serializeLargeSubsetSize (Ljava/util/Collection;ILorg/apache/cassandra/db/Columns;I)I

Back to the initial terminal, start up cql client, create the hw6 keyspace and switch context into it

**[cloudera@localhost apache-cassandra-3.11.1]$ ./bin/cqlsh**

Connected to Test Cluster at 127.0.0.1:9042.

[cqlsh 5.0.1 | Cassandra 3.11.1 | CQL spec 3.4.4 | Native protocol v4]

Use HELP for help.

**cqlsh> CREATE KEYSPACE hw6 WITH replication = {'class':'SimpleStrategy','replication\_factor':1};**

**cqlsh> USE hw6;**

cqlsh:hw6>

1. **Given events in the format <uuid><timestamp><url><ua\_country><TTFB> [TTFB == Time To First Byte]**
2. **Design a table, called "hw6\_p6" in Cassandra to store individual events and be able to answer queries:**
   1. **Q1: Get count of events per country per URL for a specified time range [t1 … t2]**
   2. **Q2: Get an average TTFB per country per URL for a specified time range [t1 … t2]**

After trying for some time to get a structure to to work with those apparent aggregation queries and figured out the formulation of the above points is misleading and must be interpreted in context of bullet point 5., where it becomes obvious no group by is intended.

|  |
| --- |
| CREATE TABLE hw6.table1 (country text,  url text,  event\_time timestamp,  event\_uuid uuid,  ttfb\_ms int,  PRIMARY KEY ((country, url), event\_time)  ) WITH CLUSTERING ORDER BY (event\_time ASC); |

1. **Explain design of your table's Primary Key**

The PRIMARY KEY value above is a compound one, with two parts: the partitioning key and the clustering key(s). In my case the partitioning key is a composite one, including country and url, which will cause data with similar values in those fields to be physically co-located, i.e. stored on the same node. Inclusion of event\_time in the clustering key portion will allow for that column to be included in filtering queries, which is a requirement of part 6. Within a given partition on disk the data will be sorted by the clustering key, event\_time for this table.

1. **Using CQLSH, create the table, insert a few events, spanning a few hours in time**

Create table using earlier DD, insert 20 events covering three countries across three urls across 3 hours in time. The exact timestamps are randomly generated within the span of November 9th, 2017 >= 6am and < 9am.

screenshot:



text:

**cqlsh:hw6> CREATE TABLE hw6.hw6\_p6 (country text,**

**... url text,**

**... event\_time timestamp,**

**... event\_uuid uuid,**

**... ttfb\_ms int,**

**... PRIMARY KEY ((country, url), event\_time)**

**... ) WITH CLUSTERING ORDER BY (event\_time ASC);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','google.com','2017-11-09 07:00:52',16896db3-f2a1-4625-8c45-b8c40130d603,298);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','yahoo.com','2017-11-09 07:13:28',1772a47c-267a-4c29-8cce-d39770285b90,388);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','yahoo.com','2017-11-09 08:28:31',08583087-6bcd-4be0-9927-fa62f20eb39d,380);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','google.com','2017-11-09 08:01:45',65d629fe-9a94-4f0e-8f54-3a7d5b65c529,370);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','google.com','2017-11-09 07:52:59',a9b8031a-efed-4c7f-8412-6655e115ea6d,343);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','harvard.edu','2017-11-09 08:57:03',40bea404-1bf5-44f1-b64b-1ac68febce4b,306);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','harvard.edu','2017-11-09 07:55:15',184c2972-1acf-4ea1-b3dc-4ea40f6fea3f,346);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','harvard.edu','2017-11-09 06:54:19',a1c2c3af-1155-4150-a90f-c98a481c3626,309);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','harvard.edu','2017-11-09 06:56:07',912899ff-bd74-4828-a27b-eb7d7775c12f,325);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','google.com','2017-11-09 07:38:23',86b08441-0643-4123-afd6-6b1434b1e909,386);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','harvard.edu','2017-11-09 08:39:08',f90570f0-d583-4ba5-8ef4-2ee7822ec68b,242);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','harvard.edu','2017-11-09 07:40:10',08fdd83b-d66b-4bec-a915-e57118f4526f,236);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','yahoo.com','2017-11-09 08:22:28',55b3f3d7-45d7-4e1b-ac5c-4528bf8f423e,278);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','harvard.edu','2017-11-09 07:04:56',22e1800a-7815-4627-a30f-46f18db1c404,319);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','harvard.edu','2017-11-09 08:48:09',01a36dfb-84ac-4fca-829a-bf619d4b2047,209);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','google.com','2017-11-09 06:47:52',b267355b-a2ce-4233-a4b5-ebcc75c0b73e,302);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','yahoo.com','2017-11-09 07:35:23',56abbbdb-ec80-4cfc-b1bc-7ada1219b5a3,240);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','yahoo.com','2017-11-09 07:58:28',61f200bd-229b-47bb-a802-d640d37d7854,258);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','google.com','2017-11-09 08:17:20',d1fecc52-88fa-4644-9e66-31dfc240ca14,334);**

**cqlsh:hw6> INSERT INTO hw6.hw6\_p6(country, url, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','yahoo.com','2017-11-09 06:28:12',9ad1b8cc-8a32-4942-87f9-79f77cf66952,377);**

1. **Create CQL queries for the above Q1 and Q2 , run them for a few variations of input parameters (URLs, countries, time ranges), demonstrate the results**

Run 3 queries in format of Q1:

**cqlsh:hw6> -- aggregation over 1 row**

**cqlsh:hw6> select country, url, COUNT(\*) FROM hw6.hw6\_p6 WHERE country='Japan' and url = 'yahoo.com' and event\_time > '2017-11-09 07:12:28' AND event\_time < '2017-11-09 07:15:28';**

country | url | count

---------+-----------+-------

Japan | yahoo.com | 1

(1 rows)

**cqlsh:hw6> -- aggregation over 3 rows**

**cqlsh:hw6> select country, url, COUNT(\*) FROM hw6.hw6\_p6 WHERE country='Japan' and url = 'yahoo.com' and event\_time > '2017-11-09 06:12:28' AND event\_time < '2017-11-09 09:15:28';**

country | url | count

---------+-----------+-------

Japan | yahoo.com | 3

(1 rows)

**cqlsh:hw6> -- aggregation over 2 rows**

**cqlsh:hw6> select country, url, COUNT(\*) FROM hw6.hw6\_p6 WHERE country='Jordan' and url = 'harvard.edu' and event\_time > '2017-11-09 08:12:28' AND event\_time < '2017-11-09 09:15:28';**

country | url | count

---------+-------------+-------

Jordan | harvard.edu | 2

(1 rows)

And use the same set of variations for Q2:

**cqlsh:hw6> -- aggregation over 1 row**

**cqlsh:hw6> select country, url, AVG(ttfb\_ms) as Avg\_ttfb\_ms FROM hw6.hw6\_p6 WHERE country='Japan' and url = 'yahoo.com' and event\_time > '2017-11-09 07:12:28' AND event\_time < '2017-11-09 07:15:28';**

country | url | avg\_ttfb\_ms

---------+-----------+-------------

Japan | yahoo.com | 388

(1 rows)

**cqlsh:hw6> -- aggregation over 3 rows**

**cqlsh:hw6> select country, url, AVG(ttfb\_ms) as Avg\_ttfb\_ms FROM hw6.hw6\_p6 WHERE country='Japan' and url = 'yahoo.com' and event\_time > '2017-11-09 06:12:28' AND event\_time < '2017-11-09 09:15:28';**

country | url | avg\_ttfb\_ms

---------+-----------+-------------

Japan | yahoo.com | 381

(1 rows)

**cqlsh:hw6> -- aggregation over 2 rows**

**cqlsh:hw6> select country, url, AVG(ttfb\_ms) as Avg\_ttfb\_ms FROM hw6.hw6\_p6 WHERE country='Jordan' and url = 'harvard.edu' and event\_time > '2017-11-09 08:12:28' AND event\_time < '2017-11-09 09:15:28';**

country | url | avg\_ttfb\_ms

---------+-------------+-------------

Jordan | harvard.edu | 274

(1 rows)

**Problem 7: (Points: 30)**

1. **Assume that you are getting too many events for each URL+country combination now - millions and millions per day; modify your table from Problem6 by adding partitioning by hour  - call it "hw6\_p7"**

Similar to previous DDL only now adding event\_hour column to the table and also to the partitionkey portion of the primary key

|  |
| --- |
| CREATE TABLE hw6.hw6\_p7 (country text,  url text,  event\_hour timestamp,  event\_time timestamp,  event\_uuid uuid,  ttfb\_ms int,  PRIMARY KEY ((country, url, event\_hour), event\_time)  ) WITH CLUSTERING ORDER BY (event\_time ASC); |

1. **Explain design of your table's Primary Key**

The previous table had a compound PRIMARY KEY and a composite partitioning key and htat continues to be the case here, only event\_hour has now been added to the partitioning key. This will be necessary (or at least a very good idea) because there will be millions of events per day and inclusion of event\_hour in the partitioning key will cause the data to be more evenly spread across available nodes as the partitioning key controls where the data will be physically located. Potentially at least, most queries will only run against one or two of these partitions - certainly that is true of the queries for this problem, of course different real-world query patterns might inform a somewhat different definition of the primary key.

The event\_time continues as the clustering key, so data will be sorted by that field within each partition. If there were multiple columns in the clustering key would need to progressively include each one in the list of fields being returned... kind of hard to describe without the presence of an example compound clustering key.

1. **Using CQLSH, create the table, insert a few events, spanning a few hours in time**

in cqlsh client

cqlsh> CREATE TABLE hw6.hw6\_p7 (country text,

... url text,

... event\_hour timestamp,

... event\_time timestamp,

... event\_uuid uuid,

... ttfb\_ms int,

... PRIMARY KEY ((country, url, event\_hour), event\_time)

... ) WITH CLUSTERING ORDER BY (event\_time ASC);

I tried seeing what I could to transform the data from the previous table directly into the new table but Cassandra does not appear to be a feature rich product and I abandoned that attempt after creating, and playing around with, a substring UDF. Instead generated some brand new events in the expected pattern and inserted into the new table:

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','yahoo.com','2017-11-09 06:00:00','2017-11-09 06:02:58',f5eddfe1-79d4-4ba2-8738-252c5b68190f,356);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','harvard.edu','2017-11-09 07:00:00','2017-11-09 07:24:38',b935190e-9009-421c-90ce-6fe30c12ff59,330);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','harvard.edu','2017-11-09 07:00:00','2017-11-09 07:44:59',88c1a335-33f8-4ef0-8bcb-85da2aea2ce3,220);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','harvard.edu','2017-11-09 07:00:00','2017-11-09 07:36:55',a2957ba2-9396-455e-ab48-0b00d25cf8cd,304);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','harvard.edu','2017-11-09 07:00:00','2017-11-09 07:11:14',b6774a17-1fb2-4315-b0a4-a51eef2be825,326);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','google.com','2017-11-09 08:00:00','2017-11-09 08:38:28',a0391054-b3b7-4fb2-ade4-6123f28713d6,284);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','harvard.edu','2017-11-09 07:00:00','2017-11-09 07:21:31',5de7f573-3e64-414e-820a-9de4de1688ba,281);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','google.com','2017-11-09 06:00:00','2017-11-09 06:39:09',5a86971e-f04f-4507-87c8-d7ead78bb320,333);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','google.com','2017-11-09 06:00:00','2017-11-09 06:30:40',66ff2994-d27c-48c7-a968-c54b5f3fe173,242);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','google.com','2017-11-09 08:00:00','2017-11-09 08:23:31',97d1dbb3-d17c-4ab9-9071-0778dc36cf53,286);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','harvard.edu','2017-11-09 08:00:00','2017-11-09 08:14:53',fef5e2f6-0375-4890-b44a-20a45bda4f0e,257);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','google.com','2017-11-09 08:00:00','2017-11-09 08:19:17',f0d222eb-f30c-4b51-9d7a-acb8ff748e26,387);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','google.com','2017-11-09 06:00:00','2017-11-09 06:29:33',269d2fd3-195d-4da4-94bc-ebe080e7b1da,373);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jamaica','google.com','2017-11-09 08:00:00','2017-11-09 08:45:42',d247354f-32e5-4d3d-969c-bca212b87a7f,341);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','yahoo.com','2017-11-09 07:00:00','2017-11-09 07:04:21',f72d1677-393f-4222-9f3c-547aad2da2bf,362);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','yahoo.com','2017-11-09 08:00:00','2017-11-09 08:43:29',8ea320a0-8d05-4d11-9671-fcc7c1a16e93,305);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','google.com','2017-11-09 08:00:00','2017-11-09 08:59:33',bc8c9abc-832b-45ff-b7f7-26cd2fa64715,222);**

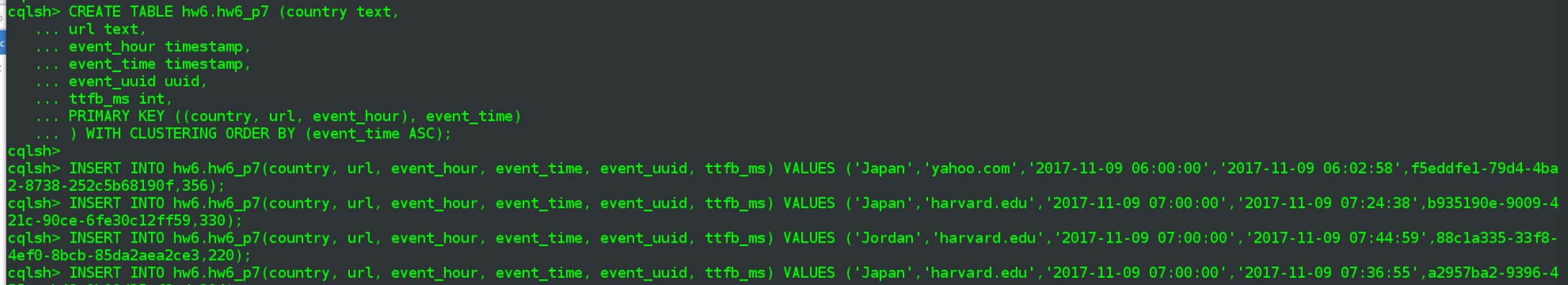
**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Japan','yahoo.com','2017-11-09 07:00:00','2017-11-09 07:03:29',6bb6a8f8-90ae-4808-a663-4053a44698c3,335);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','google.com','2017-11-09 07:00:00','2017-11-09 07:07:57',9b5aaa37-aac4-44a5-972a-b6f6bcdd3127,263);**

**cqlsh> INSERT INTO hw6.hw6\_p7(country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms) VALUES ('Jordan','harvard.edu','2017-11-09 08:00:00','2017-11-09 08:16:19',22fd788f-44f7-4aac-a700-938c417c45b3,330);**

cqlsh>

partial screenshot:



1. **Create CQLSH queries for the above Q1 and Q2 , run them for a few variations of input parameters (URLs, countries, time ranges), demonstrate the results**

Both sets of queries are similar to those from problem 6 but the data is slightly different and the filters needed to be adjusted to match.

Q1 queries

**cqlsh> -- aggregation over 1 row**

**cqlsh> select country, url, event\_hour, COUNT(\*) FROM hw6.hw6\_p7 WHERE country='Japan' and url = 'yahoo.com' and event\_hour = '2017-11-09 07:00:00' and event\_time > '2017-11-09 07:01:28' AND event\_time < '2017-11-09 07:15:28';**

country | url | event\_hour | count

---------+-----------+---------------------------------+-------

Japan | yahoo.com | 2017-11-09 12:00:00.000000+0000 | 2

(1 rows)

**cqlsh> -- aggregation over 2 rows**

**cqlsh> select country, url, event\_hour, COUNT(\*) FROM hw6.hw6\_p7 WHERE country='Japan' and url = 'google.com' and event\_hour = '2017-11-09 08:00:00' and event\_time > '2017-11-09 08:12:28' AND event\_time < '2017-11-09 09:15:28';**

country | url | event\_hour | count

---------+------------+---------------------------------+-------

Japan | google.com | 2017-11-09 13:00:00.000000+0000 | 2

(1 rows)

**cqlsh> -- aggregation over 2 rows**

**cqlsh> select country, url, event\_hour, COUNT(\*) FROM hw6.hw6\_p7 WHERE country='Jamaica' and url = 'harvard.edu' and event\_hour = '2017-11-09 07:00:00' and event\_time > '2017-11-09 07:10:28' AND event\_time < '2017-11-09 07:35:28';**

country | url | event\_hour | count

---------+-------------+---------------------------------+-------

Jamaica | harvard.edu | 2017-11-09 12:00:00.000000+0000 | 2

(1 rows)

cqlsh>

Q2 variations

**cqlsh> -- aggregation over 1 row**

**cqlsh> select country, url, event\_hour, AVG(ttfb\_ms) as Avg\_ttfb\_ms FROM hw6.hw6\_p7 WHERE country='Japan' and url = 'yahoo.com' and event\_hour = '2017-11-09 07:00:00' and event\_time > '2017-11-09 07:01:28' AND event\_time < '2017-11-09 07:15:28';**

country | url | event\_hour | avg\_ttfb\_ms

---------+-----------+---------------------------------+-------------

Japan | yahoo.com | 2017-11-09 12:00:00.000000+0000 | 348

(1 rows)

**cqlsh> -- aggregation over 2 rows**

**cqlsh> select country, url, event\_hour, AVG(ttfb\_ms) as Avg\_ttfb\_ms FROM hw6.hw6\_p7 WHERE country='Japan' and url = 'google.com' and event\_hour = '2017-11-09 08:00:00' and event\_time > '2017-11-09 08:12:28' AND event\_time < '2017-11-09 09:15:28';**

country | url | event\_hour | avg\_ttfb\_ms

---------+------------+---------------------------------+-------------

Japan | google.com | 2017-11-09 13:00:00.000000+0000 | 335

(1 rows)

**cqlsh> -- aggregation over 2 rows**

**cqlsh> select country, url, event\_hour, AVG(ttfb\_ms) as Avg\_ttfb\_ms FROM hw6.hw6\_p7 WHERE country='Jamaica' and url = 'harvard.edu' and event\_hour = '2017-11-09 07:00:00' and event\_time > '2017-11-09 07:10:28' AND event\_time < '2017-11-09 07:35:28';**

country | url | event\_hour | avg\_ttfb\_ms

---------+-------------+---------------------------------+-------------

Jamaica | harvard.edu | 2017-11-09 12:00:00.000000+0000 | 303

(1 rows)

cqlsh>

1. **Explain the main difference between "hw6\_p6" and "hw6\_p7" tables**

The biggest difference is the most obvious one, which is that an event\_hour column, of datatype = timestamp, has been added to the the p7 table. The values in this column will act as reduced form of the more specifiic event\_time values that are specific to the second. Then related to that is the addition of that new column to the compound partitioning key in order to allow more even distribution of data across nodes, and more efficient querying assuming that the queries we are modeling against are like the ones presented for this problem, aggregations against a particular hour-in-time.

**Problem 9: Bonus: +40**

1. **Instead of generating/inserting events manually into your "hw6\_p7" table you created for Problem7 - write a small program that does:**
   1. **generate a few thousand events in the format required in Problem7 (<uuid><timestamp><url><ua\_country><TTFB>), spanning a few hours in time**
   2. **Using Cassandra driver, insert these events into the "hw6\_p7" table - for example, for Java, you could use this driver:** [**https://docs.datastax.com/en/developer/java-driver/3.3/ (Links to an external site.)Links to an external site.**](https://docs.datastax.com/en/developer/java-driver/3.3/)
   3. **Also using the driver, implement the Q1 and Q2 queries and print out the results**

Install the python cassandra driver into my virtual environment using pip

**(v34\_e88) [cloudera@localhost A6]$ pip install cassandra-driver**

Collecting cassandra-driver

Downloading cassandra-driver-3.12.0.tar.gz (222kB)

100% |████████████████████████████████| 225kB 1.3MB/s

Requirement already satisfied: six>=1.9 in /home/cloudera/virtualenvs/v34\_e88/lib/python3.4/site-packages (from cassandra-driver)

Installing collected packages: cassandra-driver

Running setup.py install for cassandra-driver ... done

Successfully installed cassandra-driver-3.12.0

Below is my python script, broken down into two main sections:

1. create events (hard-coded as count=3,000) and write to hw6.hw6\_p7
2. run queries from P7 on hw6.hw6\_p7

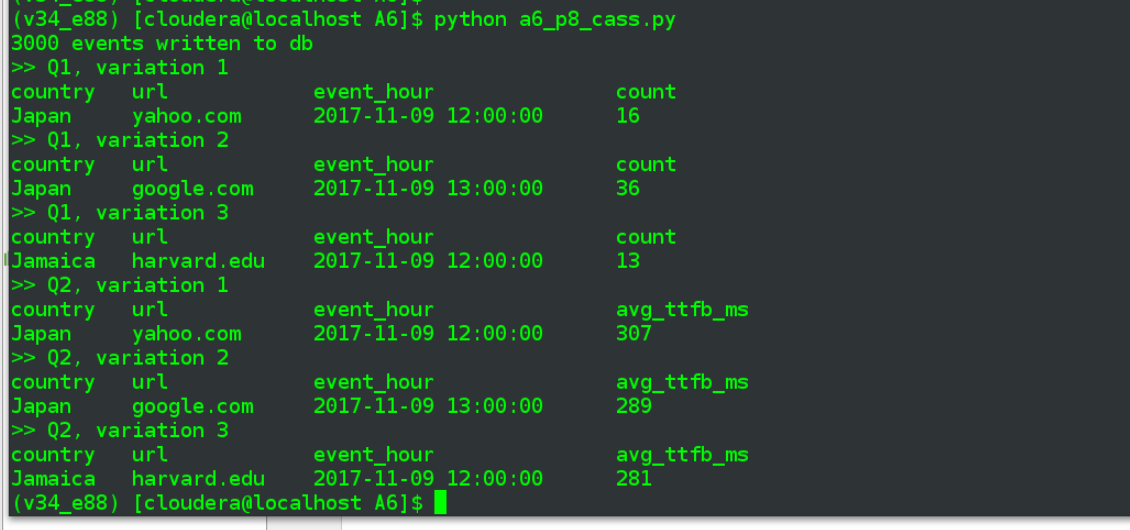
**a6\_p8\_cass.py**

|  |
| --- |
| **from** random **import** randint **import** collections **import** uuid **import** time  **from** cassandra.cluster **import** Cluster  *# return an n count deque of timestamp string values in format of 'YYYY-MM-DD HH:MM:SS'* **def** get\_all\_timestamps(count):  Timepoint = collections.namedtuple(**'Timepoint'**, **'year month day hour minute second'**)  *# generate timepoints on Nov 9th, 2017, >= 6am and < 9am* time\_points = (Timepoint(year=2017, month=11, day=9,  hour=randint(6, 8), minute=randint(0, 59), second=randint(0, 59)) **for** i **in** range(count))   *# below doesn't handle February, but that's ok for this problem* **return** collections.deque(**'{year}-{month:02}-{day:02} {hour:02}:{minute:02}:{second:02}'**.format(  year=t.year, month=t.month, day=t.day **if** t.month **in** [4,6,9,11] **else** randint(1,31), hour=t.hour,  minute=t.minute, second=t.second) **for** t **in** time\_points)  *# generate the list of events and insert them to hw6\_7 table* **def** populate\_table(num\_events):   *# same urls as P7* urls=[**'yahoo.com'**,**'google.com'**,**'harvard.edu'**]  *# add another three countries* countries=[**'Jamaica'**,**'Japan'**,**'Jordan'**,**'Fiji'**,**'Finland'**,**'France'**]   stamps = get\_all\_timestamps(num\_events)  events = []  **for** stamp **in** stamps:  *# initial events are in format of (<uuid><timestamp><url><ua\_country><TTFB>)* Event = collections.namedtuple(**'Event'**, **'event\_uuid event\_time url country ttfb\_ms'**)  *# randomly assign an url, country, and ttfb value to each event* events.append(Event(event\_uuid=uuid.uuid4(),  event\_time=stamp,  url=urls[randint(0, len(urls)-1)],  country=countries[randint(0, len(countries)-1)],  ttfb\_ms=randint(200, 400)))   events\_written = 0  **for** event **in** events:  session.execute(  **"""  INSERT INTO hw6\_p7 (country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms)  VALUES (%(country)s, %(url)s, %(event\_hour)s, %(event\_time)s, %(event\_uuid)s, %(ttfb\_ms)s)  """**, {  **'country'**: event.country,  **'url'**: event.url,  **'event\_hour'**: event.event\_time[:14] + **'00:00'**,  **'event\_time'**: event.event\_time,  **'event\_uuid'**: event.event\_uuid,  **'ttfb\_ms'**: event.ttfb\_ms  }  )  events\_written += 1   print(**'{} events written to db'**.format(events\_written))  *# run each of the queries from P7 on the newly re-populated database, print results* **def** run\_queries():  results = []   *# 3 Q1 queries* rows = session.execute(**'''  select country, url, event\_hour, COUNT(\*)   FROM hw6.hw6\_p7   WHERE country='Japan' and url = 'yahoo.com' and event\_hour = '2017-11-09 07:00:00'   and event\_time > '2017-11-09 07:01:28' AND event\_time < '2017-11-09 07:15:28'  '''**)  results.append(rows)   rows = session.execute(**'''  select country, url, event\_hour, COUNT(\*)   FROM hw6.hw6\_p7   WHERE country='Japan' and url = 'google.com' and event\_hour = '2017-11-09 08:00:00'   and event\_time > '2017-11-09 08:12:28' AND event\_time < '2017-11-09 09:15:28'  '''**)  results.append(rows)   rows = session.execute(**'''  select country, url, event\_hour, COUNT(\*)   FROM hw6.hw6\_p7   WHERE country='Jamaica' and url = 'harvard.edu' and event\_hour = '2017-11-09 07:00:00'   and event\_time > '2017-11-09 07:10:28' AND event\_time < '2017-11-09 07:35:28'   '''**)  results.append(rows)   *# 3 Q2 queries* rows = session.execute(**'''  select country, url, event\_hour, AVG(ttfb\_ms) as avg\_ttfb\_ms   FROM hw6.hw6\_p7   WHERE country='Japan' and url = 'yahoo.com' and event\_hour = '2017-11-09 07:00:00'   and event\_time > '2017-11-09 07:01:28' AND event\_time < '2017-11-09 07:15:28'  '''**)  results.append(rows)   rows = session.execute(**'''  select country, url, event\_hour, AVG(ttfb\_ms) as avg\_ttfb\_ms   FROM hw6.hw6\_p7   WHERE country='Japan' and url = 'google.com' and event\_hour = '2017-11-09 08:00:00'   and event\_time > '2017-11-09 08:12:28' AND event\_time < '2017-11-09 09:15:28'  '''**)  results.append(rows)   rows = session.execute(**'''  select country, url, event\_hour, AVG(ttfb\_ms) as avg\_ttfb\_ms   FROM hw6.hw6\_p7   WHERE country='Jamaica' and url = 'harvard.edu' and event\_hour = '2017-11-09 07:00:00'   and event\_time > '2017-11-09 07:10:28' AND event\_time < '2017-11-09 07:35:28'  '''**)  results.append(rows)   fmt = **'{0:<10}{1:<15}{2:<25}{3}'** *# iterate through each set of results, which will only have one row each and print the corresponding header & values* **for** i, rows **in** enumerate(results):  **if** i < 3:  print(**'>> Q1, variation {}'**.format(i+1))  print(fmt.format(**'country'**, **'url'**, **'event\_hour'**, **'count'**))  **for** row **in** rows:  print(fmt.format(row.country, row.url, str(row.event\_hour), row.count))  **else**:  print(**'>> Q2, variation {}'**.format(i-2))  print(fmt.format(**'country'**, **'url'**, **'event\_hour'**, **'avg\_ttfb\_ms'**))  **for** row **in** rows:  print(fmt.format(row.country, row.url, str(row.event\_hour), row.avg\_ttfb\_ms))    cluster = Cluster() *# connect to the hw6 keyspace* session = cluster.connect(**'hw6'**)  *# add a few thousand rows to hw6.hw6\_p7* populate\_table(3000) *# run the same queries from P7 on the new dataset in Cassandra* run\_queries() |

The code is pretty well annotated I believe but I'll point out the queries being run are the same exact set of six that were used in Problem 7.

Execute the script and review results

python a6\_p8\_cass.py



**Problem 10: Bonus: +15**

* **Modify your app from Problem 9 to use "async" operations writing events into Cassandra (for Java:** [**https://docs.datastax.com/en/developer/java-driver/3.3/manual/async/ (Links to an external site.)Links to an external site.**](https://docs.datastax.com/en/developer/java-driver/3.3/manual/async/)**)**
* **Use QOURUM consistency level for your writes and reads**

The code from problem 9 has been updated to below - much of the async info came from datastax page at <https://datastax.github.io/python-driver/getting_started.html>. I've excerpted some lines that didn't change at all but the remaining edits re async/consistency level wound up being spread throughout the remaining code.

**A6/a6\_p9\_cass\_async.py**

|  |
| --- |
| **...**  **from** cassandra.cluster **import** Cluster **from** cassandra **import** ConsistencyLevel **from** cassandra.query **import** SimpleStatement *# from cassandra import ReadTimeout* **...**  *# generate the list of events and insert them to hw6\_7 table* **def** populate\_table(num\_events):  ...   events\_written = 0  futures = []  **for** event **in** events:   insert\_stmt = SimpleStatement( **"""  INSERT INTO hw6\_p7 (country, url, event\_hour, event\_time, event\_uuid, ttfb\_ms)  VALUES (%(country)s, %(url)s, %(event\_hour)s, %(event\_time)s, %(event\_uuid)s, %(ttfb\_ms)s)  """**, consistency\_level=ConsistencyLevel.QUORUM)  futures.append(session.execute\_async(insert\_stmt, {  **'country'**: event.country,  **'url'**: event.url,  **'event\_hour'**: event.event\_time[:14] + **'00:00'**,  **'event\_time'**: event.event\_time,  **'event\_uuid'**: event.event\_uuid,  **'ttfb\_ms'**: event.ttfb\_ms  }))   *# wait for them to complete by calling .result(), though we don't care about any actual values  # per the docs: query has not yet completed, call result() will block until it has and then return the result  # or raise an Exception if an error occurred.* **for** future **in** futures:  **try**:  result = future.result()  events\_written += 1  **except** Exception **as** e:  print(**'ERROR:'**, e)   print(**'async: {} events written to db'**.format(events\_written))   *# run each of the queries from P7 on the newly re-populated database, print results* **def** run\_queries():  stmts = []   *# for p9 switch to making a list of strings = each select query  # 3 Q1 queries* select = **'''  select country, url, event\_hour, COUNT(\*)   FROM hw6.hw6\_p7   WHERE country='Japan' and url = 'yahoo.com' and event\_hour = '2017-11-09 07:00:00'   and event\_time > '2017-11-09 07:01:28' AND event\_time < '2017-11-09 07:15:28'  '''** stmts.append(select)   select = **'''  select country, url, event\_hour, COUNT(\*)  FROM hw6.hw6\_p7  WHERE country='Japan' and url = 'google.com' and event\_hour = '2017-11-09 08:00:00'  and event\_time > '2017-11-09 08:12:28' AND event\_time < '2017-11-09 09:15:28'  '''** stmts.append(select)   select = **'''  select country, url, event\_hour, COUNT(\*)  FROM hw6.hw6\_p7  WHERE country='Jamaica' and url = 'harvard.edu' and event\_hour = '2017-11-09 07:00:00'  and event\_time > '2017-11-09 07:10:28' AND event\_time < '2017-11-09 07:35:28'  '''** stmts.append(select)   *# 3 Q2 queries* select = **'''  select country, url, event\_hour, AVG(ttfb\_ms) as avg\_ttfb\_ms  FROM hw6.hw6\_p7  WHERE country='Japan' and url = 'yahoo.com' and event\_hour = '2017-11-09 07:00:00'  and event\_time > '2017-11-09 07:01:28' AND event\_time < '2017-11-09 07:15:28'  '''** stmts.append(select)   select = **'''  select country, url, event\_hour, AVG(ttfb\_ms) as avg\_ttfb\_ms  FROM hw6.hw6\_p7  WHERE country='Japan' and url = 'google.com' and event\_hour = '2017-11-09 08:00:00'  and event\_time > '2017-11-09 08:12:28' AND event\_time < '2017-11-09 09:15:28'  '''** stmts.append(select)   select = **'''  select country, url, event\_hour, AVG(ttfb\_ms) as avg\_ttfb\_ms  FROM hw6.hw6\_p7  WHERE country='Jamaica' and url = 'harvard.edu' and event\_hour = '2017-11-09 07:00:00'  and event\_time > '2017-11-09 07:10:28' AND event\_time < '2017-11-09 07:35:28'  '''** stmts.append(select)   fmt = **'{0:<10}{1:<15}{2:<25}{3}'** *# iterate through each set of results, which will only have one row each and print the corresponding header & values* **for** i, stmt **in** enumerate(stmts):  **if** i < 3:  print(**'>> Q1, variation {}'**.format(i+1))  print(fmt.format(**'country'**, **'url'**, **'event\_hour'**, **'count'**))  *# wrap each query in a SimpleStatement in order to use a QUORUM consistency level instead of default LOCAL\_ONE* query = SimpleStatement(stmt, consistency\_level=ConsistencyLevel.QUORUM)  *# execute the query* rows = session.execute(query)  *# iterate through results... there should only be one row* **for** row **in** rows:  print(fmt.format(row.country, row.url, str(row.event\_hour), row.count))  **else**:  print(**'>> Q2, variation {}'**.format(i-2))  print(fmt.format(**'country'**, **'url'**, **'event\_hour'**, **'avg\_ttfb\_ms'**))  *# wrap each query in a SimpleStatement in order to use a QUORUM consistency level instead of default LOCAL\_ONE* query = SimpleStatement(stmt, consistency\_level=ConsistencyLevel.QUORUM)  *# execute the query* rows = session.execute(query)  *# iterate through results... there should only be one row* **for** row **in** rows:  print(fmt.format(row.country, row.url, str(row.event\_hour), row.avg\_ttfb\_ms))    cluster = Cluster() *# connect to the hw6 keyspace* session = cluster.connect(**'hw6'**)  *# add a few thousand rows to hw6.hw6\_p7* populate\_table(3000) *# run the same queries from P7 on the new dataset in Cassandra* run\_queries() |

I've added additional comments re the changes, which were in two places:

* Insert statements were changed from simple session.execute() to being wrapped in SimpleStatement(), which, in combination with the session.execute() were then inserted into a list of "futures. Those were iterated through, with .result() called on each
* the individual session.exeucte()'s for the select queries were turned into a list of strings, which were iterated over in order to wrap each in a Simplestatement() in order to achieve desired consistencyLevel
  + the ConsistencyLevel (and SimpleStatement) come in via new import statements
* **Run the Q1 and Q2 queries only after you have ensured that all write requests are completed**

The page cited earlier has this to say about ensuring async actions have completed, so calling .result() on the SimpleStatement objects will be key to achieving the goal of this point:

*The driver supports asynchronous query execution through* [*execute\_async()*](https://datastax.github.io/python-driver/api/cassandra/cluster.html#cassandra.cluster.Session.execute_async)*. Instead of waiting for the query to complete and returning rows directly, this method almost immediately returns a* [*ResponseFuture*](https://datastax.github.io/python-driver/api/cassandra/cluster.html#cassandra.cluster.ResponseFuture) *object. There are two ways of getting the final result from this object.*

*The first is by calling* [*result()*](https://datastax.github.io/python-driver/api/cassandra/cluster.html#cassandra.cluster.ResponseFuture.result) *on it. If the query has not yet completed, this will block until it has and then return the result or raise an Exception if an error occurred. For example:*

All that remains is to call the script, results of which, unsurprisingly, look very similar to output of previous problem:

