**Yuen, Benjamin**

2019-09-17

Harvard University Extension - Principles of Big Data Processing e88

Homework 2: Vertical and Horizontal Scaling , Shared State Management

This document is a template for your solutions submission. You are free to add additional information in this submission if you would like. Extra screenshots and extra documentation are appreciated. Screenshots must always be viewable. If a screenshot is too blurry or chopped off in a key area you will not receive full credit for it.

**Make sure to also submit all your source code (.java files , .py files or whatever language you are using) - in a separate archive, named <LastName>\_<FirstName>\_HW2.zip**

Please identify which problems were completed. If any were incomplete, please identify where you encountered problems.

|  |
| --- |
| Problem 1: 100% complete  Problem 2: 100% complete  Problem 3: 100% complete  Problem 4: 100% complete  Problem 5 Bonus: 100% complete |

**Problem 1: CPU Analysis** [points: 25]

Paste your source code into the following area [10 points]

|  |
| --- |
| import argparse  import time  import os  import pathlib  import sys  # ProcessPoolExecutor is used here, so the python "multiprocessing" module is used.  from concurrent.futures import ProcessPoolExecutor  def fibonacci(i):      """Computes a number in the Fibonacci seqeunce.      The Fibonacci sequence, f(i), is defined as f(i) = f(i-1) + f(i-2).      For the base cases, f(0) = 0 and f(1) = 1.      Args:          i: A non-negative integer.      Returns:          An integer that is the i-th number in the Fibonacci sequence.      """      f\_1 = 1      f\_2 = 0      if i == 0:          result = 0      elif i == 1:          result = 1      else:          for k in range(2, i + 1):              result = f\_1 + f\_2              f\_2 = f\_1              f\_1 = result      return result  def loop\_forever():      program\_name = pathlib.Path(sys.argv[0]).name      pid = os.getpid()        while True:          time.sleep(1)          print(f'Process Name: {program\_name} Process ID: {pid}')          fibonacci(1000000000)  if \_\_name\_\_ == '\_\_main\_\_':      # We take the number of threads as an argument, defaulting to 4 if not      # provided.      parser = argparse.ArgumentParser(description='CSCI E-88 HW2 Problem 1')      parser.add\_argument('num\_threads', default=[4], nargs=argparse.ZERO\_OR\_MORE,                          metavar='numThreads', type=int, help="Number of Threads.")      args = parser.parse\_args(args=None)      try:          # Calculation is done in its own thread.          with ProcessPoolExecutor(max\_workers=args.num\_threads[0]) as executor:              # Start a thread for every fibonacci calculation.              futures = [executor.submit(loop\_forever) for n in range(args.num\_threads[0])]      except:          executor.shutdown(wait=False) |

Provide your table or graphs demonstrating the results of running this code with 2, 4, and 16 threads on a 4 CPU machine: [5 points]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | Number of Threads | Cores Saturated (from bar charts) | CPU Avg % (up to 100%) | Load Avg (can exceed 1 for multi core systems) | | 2 | 2 | 50 | 1.92 | | 4 | 4 | 99.5 | 3.95 | | 16 | 4 | 99.8 | 15.65 | |

What can you summarize about the results? [3 points]

|  |
| --- |
| Using htop, I added a metric “CPU avg”. This goes from 0-100% regardless of number of cores. In addition, the “Load Avg” was observed, which can go up to the number of cores utilized.  In general, it was observed that the number of threads corresponds to the number of cores that can be saturated. It was observed that the CPU utilization was dominated by “user” mode.  When 2 threads were in use, the program could not saturate all cores, resulting in only 50% “CPU Avg”.  For the 4 core instance in amazon, the t3a.xlarge (4cpu) type was used. This is a “burstable” table and can give some unexpected results in htop “Load avg”, which went over 4 when 16 threads were in use. |

Provide your table or graphs demonstrating the results of running this code with 2, 4, and 16 threads on an 8 CPU machine: [5 points]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | Number of Threads | Cores Saturated (from bar charts) | CPU Avg % (up to 100%) | Load Avg (can exceed 1 for multi core systems) | | 2 | 2 | 25 | 1.82 | | 4 | 4 | 50.1 | 3.65 | | 16 | 8 | 99.9 | 15.35 | |

What can you summarize about the results? How does a 4 CPU machine compare to an 8 CPU machine in this exercise? [2 points]

|  |
| --- |
| For the 8core instance, the t3a.x2large instance was used.  In general, the number of threads corresponds to the number of cores saturated. At 2 and 4 threads, the 8 core system cannot be saturated overall (CPU Avg=50%, Load Avg=4). Only with 16 threads, the system can be saturated (CPU Avg=100%, Load Avg=16 – due to burstable instance). |

**Problem 2: I/O Analysis** [points: 25]

Paste your source code into the following area. Make sure you clarify what you did to programmatically create an I/O intensive process. [10 points]

|  |
| --- |
| import argparse  import time  import os  import pathlib  import sys  # ProcessPoolExecutor is used here, so the python "multiprocessing" module is used.  from concurrent.futures import ProcessPoolExecutor  def do\_io(outchar, pid, bufsize):      repetitions = 20      with open(f'outfile\_{pid}.txt', 'w') as f:          for i in range(repetitions):              f.write(outchar)      with open(f'outfile\_{pid}.txt', 'r') as f:          for i in range(repetitions):              outchar = f.read(bufsize)      os.unlink(f'outfile\_{pid}.txt')  def loop\_forever():      program\_name = pathlib.Path(sys.argv[0]).name      pid = os.getpid()      bufsize = 50000000      outchar = 'a' \* bufsize        while True:          time.sleep(1)          print(f'Process Name: {program\_name} Process ID: {pid}')          do\_io(outchar, pid, bufsize)  if \_\_name\_\_ == '\_\_main\_\_':      # We take the number of threads as an argument, defaulting to 4 if not      # provided.      parser = argparse.ArgumentParser(description='CSCI E-88 HW2 Problem 2')      parser.add\_argument('num\_threads', default=[4], nargs=argparse.ZERO\_OR\_MORE,                          metavar='numThreads', type=int, help="Number of Threads.")      args = parser.parse\_args(args=None)      try:          # Calculation is done in its own thread.          with ProcessPoolExecutor(max\_workers=args.num\_threads[0]) as executor:              # Start a thread for every fibonacci calculation.              futures = [executor.submit(loop\_forever) for n in range(args.num\_threads[0])]      except Exception as e:          print(e)          executor.shutdown(wait=False) |

Provide your table or graphs demonstrating the results of running this code with 2, 4 and 16 threads on a 4 CPU machine: [5 points]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Number of Threads | Actual Write Avg (M/s) | Actual Read Avg (M/s) | CPU Avg % (up to 100%) | Load Avg (can exceed 1 for multi core systems) | | 2 | 80 (max 128) | 0 | 40 | 1.90 | | 4 | 125 (max 128) | 0 | 68 | 3.65 | | 16 | 128 | 30 (max 128) | 5 | 15.65 | |

What can you summarize about the results? [3 points]

|  |
| --- |
| For the IO intensive program, care was taken to choose operations that do not require much CPU to ensure that the program can become IO bound.  For the IO measurements, iotop “actual” read and writes were used (in order not to be distracted by caching effects).  The actual read rates can only be observed when there are 16 threads. For 2 and 4 threads, the reads happened too quickly for my implementation of IO intensive tasks.  For actual writes, the system is saturated when there were 4 or more threads. The write rates are consistently at 128M/s and 100% busy.  The CPU usage was dominated by “system” and “wait” instead of “user” (cpu charts from “nmon” tool) in IO intensive tasks.  Load Avg include “wait” states, taking into account of burstable characteristis of t3 instances, we have htop “Load average” of almost 16 when 16 threads are in use. Most of the time though, the CPU cores are in “wait” states. The “CPU avg” only counts “user” and “system”, and it shows the utilization is only around 4% when 16 threads are in use. The process is very much IO bound. |

Provide your table or graphs demonstrating the results of running this code with 2, 4 and 16 threads on an 8 CPU machine: [5 points]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Number of Threads | Actual Write Avg (M/s) | Actual Read Avg (M/s) | CPU Avg % (up to 100%) | Load Avg (can exceed 1 for multi core systems) | | 2 | 0 | 0 | 22 | 1.85 | | 4 | 30 (max 128) | 0 | 36 | 2.34 | | 16 | 128 | 5 (max 128) | 3 | 15.41 | |

What can you summarize about the results? How does a 4 CPU machine compare to an 8 CPU machine in this exercise? [2 points]

|  |
| --- |
| For the 8CPU machine, it was found that only 16 threads can saturate the actual write at 128M/s. This machine has more memory, it is possible that the system allocated more cache which masked out the IO operations.  Besides “actual” read/write, the “Total” read/write was observed. This may include cached IO and can fluctuate widely (even for 2 threads). This number can go over 1G/s at times.  Just as the 4 Cpu, when the actual write are saturated, the CPU load is very low, indicating that the system is truly IO bound. |

**Problem 3: unique counts** [points: 25]

Paste your source code into the following area [10 points]

|  |
| --- |
| import argparse  import time  import os  import pathlib  import sys  import pandas as pd  import traceback  # ProcessPoolExecutor is used here, so the python "multiprocessing" module is used.  from concurrent.futures import ProcessPoolExecutor  from multiprocessing import Lock, Manager  FILE\_PATTERN = '\*input\*'  def read\_file(input\_name):      with open(input\_name, 'r') as input\_file:          input\_data = pd.read\_csv(input\_file, header=None, parse\_dates=[1], na\_filter=False)          input\_data.columns = ['uuid', 'timestamp','url','userid','country','browser','os','status','TTFB']      return input\_data  def aggregate\_file(infile\_name):      data\_df = read\_file(infile\_name)      print(f'Start processing {infile\_name}...')      local\_dict = {}      for k in data\_df[['timestamp', 'url', 'userid']].iterrows():          hour = k[1].timestamp.floor('h')          if hour not in local\_dict:              local\_dict[hour] = {}          if k[1].url not in local\_dict[hour]:              local\_dict[hour][k[1].url] = {}          if k[1].userid not in local\_dict[hour][k[1].url]:              local\_dict[hour][k[1].url][k[1].userid] = 1          else:              local\_dict[hour][k[1].url][k[1].userid] += 1      print(f'{infile\_name} done!')      return local\_dict  def update\_shared\_state(local\_dict, shared\_dict, lock):      lock.acquire()      try:          for l\_hour in local\_dict:              if l\_hour in shared\_dict:                  d = shared\_dict[l\_hour].copy()                  for url in local\_dict[l\_hour]:                      if url in d:                          for userid in local\_dict[l\_hour][url]:                              if userid in d[url]:                                  d[url][userid] += 1                              else:                                  d[url][userid] = 1                      else:                          d[url] = local\_dict[l\_hour][url]                  shared\_dict[l\_hour] = d              else:                  shared\_dict[l\_hour] = local\_dict[l\_hour]      finally:          lock.release()  def process\_file(infile\_name, shared\_dict, lock):      local\_dict = aggregate\_file(infile\_name)      update\_shared\_state(local\_dict, shared\_dict, lock)  def query1(data\_dict, outfile):      with open(outfile, 'w') as f:          for hour in data\_dict:              count = len(data\_dict[hour])              time\_str = hour.strftime('%Y-%m-%d:%H')              f.write(f'{time\_str}, {count}\n')  def query2(data\_dict, outfile):      with open(outfile, 'w') as f:          for hour in data\_dict:              time\_str = hour.strftime('%Y-%m-%d:%H')              for url in data\_dict[hour]:                  count = len(data\_dict[hour][url])                  f.write(f'{time\_str}:{url}, {count}\n')  def query3(data\_dict, outfile):      with open(outfile, 'w') as f:          for hour in data\_dict:              time\_str = hour.strftime('%Y-%m-%d:%H')              for url in data\_dict[hour]:                  count = sum(data\_dict[hour][url].values())                  f.write(f'{time\_str}:{url}, {count}\n')  if \_\_name\_\_ == '\_\_main\_\_':      # We take the number of threads as an argument, defaulting to 4 if not      # provided.      parser = argparse.ArgumentParser(description='CSCI E-88 HW2 Problem 3')      parser.add\_argument('num\_threads', default=[4], nargs=argparse.ZERO\_OR\_MORE,                          metavar='numThreads', type=int, help="Number of Threads.")      args = parser.parse\_args(args=None)      futures = []      manager = Manager()      lock = manager.RLock()      shared\_dict = manager.dict()      try:          # Calculation is done in its own thread.          with ProcessPoolExecutor(max\_workers=args.num\_threads[0]) as executor:              # Start a thread for every input file.              for filename in pathlib.Path('.').glob(FILE\_PATTERN):                  futures.append(executor.submit(process\_file, filename, shared\_dict, lock))      except Exception as e:          print(e)          executor.shutdown(wait=False)      for f in futures:          if f.exception():              print(f.exception())      query1(shared\_dict, 'query1.out')      query2(shared\_dict, 'query2.out')      query3(shared\_dict, 'query3.out') |

Explain your choice of the data structures for shared state management [5 points]

|  |
| --- |
| For this part, python multiprocessing was used to perform tasks in parallel.  For the shared state, the multiprocessing.Manager was used. At the top level, a proxy dict holds the date/hour entries. Within each entry, a python dict of dict was used to hold url and userid data. In total, there are three levels (including the manager proxy dict).  In each thread, a local version of the multilevel dict is first built when reading a file. The local dict is then merged to the shared dict. A manager.Lock is acquired before then read/update cycle. This is only done after then entire local dict has been built up to minimize lock contention between threads. |

What are the results of your queries for the following specified keys? [10 points]

The expected output for the first value is provided for your reference.

|  |
| --- |
| **Query 1:**  **<date\_hour>, <url\_count>**  2019-09-12:13, 185  2019-09-12:14, 186  2019-09-12:15, 185  2019-09-12:16, 190  2019-09-12:17, 189    **Query 2**  **<date:hour:url>, unique\_user\_count**  2019-09-12:02:http://example.com/?url=003, 1  2019-09-12:02:http://example.com/?url=004, 3  2019-09-12:02:http://example.com/?url=005, 4  2019-09-12:02:http://example.com/?url=006, 10  **Query 3**  **<date:hour:url>, event\_count**  2019-09-12:02:http://example.com/?url=003, 1  2019-09-12:02:http://example.com/?url=004, 3  2019-09-12:02:http://example.com/?url=005, 5  2019-09-12:02:http://example.com/?url=006, 10 |

**Problem 4: time range queries** [points: 25]

Paste your source code into the following area [15 points]

|  |
| --- |
| import argparse  import time  import os  import pathlib  import sys  import pandas as pd  import traceback  # ProcessPoolExecutor is used here, so the python "multiprocessing" module is used.  from concurrent.futures import ProcessPoolExecutor  from multiprocessing import Lock, Manager  FILE\_PATTERN = '\*input\*'  def read\_file(input\_name):      with open(input\_name, 'r') as input\_file:          input\_data = pd.read\_csv(input\_file, header=None, parse\_dates=[1], na\_filter=False)          input\_data.columns = ['uuid', 'timestamp','url','userid','country','browser','os','status','TTFB']      return input\_data  def aggregate\_file(infile\_name, timestamp\_lower, timestamp\_upper):      data\_df = read\_file(infile\_name)      print(f'Start processing {infile\_name}...')      local\_dict = {}      for k in data\_df[['timestamp', 'url', 'country']].iterrows():          hour = k[1].timestamp.floor('h')          # filter data based on time limits          if hour >= timestamp\_lower and hour <= timestamp\_upper:              if hour not in local\_dict:                  local\_dict[hour] = {}              if k[1].country not in local\_dict[hour]:                  local\_dict[hour][k[1].country] = {}              if k[1].url not in local\_dict[hour][k[1].country]:                  local\_dict[hour][k[1].country][k[1].url] = 1              else:                  # This is not strictly necessary, for the Query 4 specification, but                  # we are also storing the unique UUID counts per url in case of future queries                  # that require it.                  local\_dict[hour][k[1].country][k[1].url] += 1      print(f'{infile\_name} done!')      return local\_dict  def update\_shared\_state(local\_dict, shared\_dict, lock):      lock.acquire()      try:          for l\_hour in local\_dict:              if l\_hour in shared\_dict:                  d = shared\_dict[l\_hour].copy()                  for country in local\_dict[l\_hour]:                      if country in d:                          for url in local\_dict[l\_hour][country]:                              if url in d[country]:                                  # This is not strictly necessary, for the Query 4 specification, but                                  # we are also storing the unique UUID counts per url in case of future queries                                  # that require it.                                  d[country][url] += 1                              else:                                  d[country][url] = 1                      else:                          d[country] = local\_dict[l\_hour][country]                  shared\_dict[l\_hour] = d              else:                  shared\_dict[l\_hour] = local\_dict[l\_hour]      finally:          lock.release()  def process\_file(infile\_name, shared\_dict, timestamp\_lower, timestamp\_upper, lock):      local\_dict = aggregate\_file(infile\_name, timestamp\_lower, timestamp\_upper)      update\_shared\_state(local\_dict, shared\_dict, lock)  def query4(data\_dict, outfile):      with open(outfile, 'w') as f:          for hour in data\_dict:              time\_str = hour.strftime('%Y-%m-%d:%H')              for country in data\_dict[hour]:                  count = len(data\_dict[hour][country])                  f.write(f'{time\_str},{country}, {count}\n')  if \_\_name\_\_ == '\_\_main\_\_':      # We take the number of threads as an argument, defaulting to 4 if not      # provided.      parser = argparse.ArgumentParser(description='CSCI E-88 HW2 Problem 4')      parser.add\_argument('num\_threads', default=[4], nargs=argparse.ZERO\_OR\_MORE,                          metavar='numThreads', type=int, help="Number of Threads.")      # Take lower and upper date limits from command line, defaulting to      # "09/13/2019 5PM UTC" and "09/14/2019 9AM UTC".      # The time limits are truncated to the hour.      parser.add\_argument('date\_lower', default=["09/13/2019 5PM UTC"], nargs=argparse.ZERO\_OR\_MORE,                          metavar='dateLower', type=int, help="Lower limit of date range (inclusive and truncated to the hour).")      parser.add\_argument('date\_upper', default=["09/14/2019 9AM UTC"], nargs=argparse.ZERO\_OR\_MORE,                          metavar='dateUpper', type=int, help="Upper limit of date range (inclusive and truncated to the hour).")      args = parser.parse\_args(args=None)      # convert date range spec to pandas timestamp      timestamp\_lower = pd.Timestamp(args.date\_lower[0]).floor('h')      timestamp\_upper = pd.Timestamp(args.date\_upper[0]).floor('h')      futures = []      manager = Manager()      lock = manager.RLock()      shared\_dict = manager.dict()      try:          # Calculation is done in its own thread.          with ProcessPoolExecutor(max\_workers=args.num\_threads[0]) as executor:              # Start a thread for every input file.              for filename in pathlib.Path('.').glob(FILE\_PATTERN):                  futures.append(executor.submit(process\_file, filename, shared\_dict,                                      timestamp\_lower, timestamp\_upper, lock))      except Exception as e:          print(e)          executor.shutdown(wait=False)        for f in futures:          if f.exception():              print(f.exception())      query4(shared\_dict, 'query4.out') |

What are the main differences with the Problem 3 implementation? [5 points]

|  |
| --- |
| For this query, the main difference is that the multilevel dict has date/hour, country and url data. Also, the date/hour is filtered when constructing the local dicts within each thread.  At the bottom level, the program keeps a count of the URL events. This is not necessary for query 4, but is kept so any queries that require this can be done easily. |

What are the results of your query for the specified keys ? [5 points]

The expected output for the first value is provided for your reference.

|  |
| --- |
| <date,hour,country>, url\_count  2019-09-13:19,IQ, 1  2019-09-13:19,IR, 4  2019-09-13:19,IS, 9  2019-09-13:19,IT, 2  2019-09-13:19,JE, 4 |

**Problem 5: Bonus: Top N queries** [15 points]

Paste your source code into the following area [5 points]

|  |
| --- |
| import argparse  import time  import os  import pathlib  import sys  import pandas as pd  import traceback  # ProcessPoolExecutor is used here, so the python "multiprocessing" module is used.  from concurrent.futures import ProcessPoolExecutor  from multiprocessing import Lock, Manager  FILE\_PATTERN = '\*input\*'  def read\_file(input\_name):      with open(input\_name, 'r') as input\_file:          input\_data = pd.read\_csv(input\_file, header=None, parse\_dates=[1], na\_filter=False)          input\_data.columns = ['uuid', 'timestamp','url','userid','country','browser','os','status','TTFB']      return input\_data  def aggregate\_file(infile\_name):      data\_df = read\_file(infile\_name)      print(f'Start processing {infile\_name}...')      local\_dict = {}      for k in data\_df[['timestamp', 'url', 'TTFB']].iterrows():          day = k[1].timestamp.floor('d')          if day not in local\_dict:              local\_dict[day] = {}          if k[1].url not in local\_dict[day]:              local\_dict[day][k[1].url] = {'ttfb\_sum':k[1].TTFB,                                              'event\_count':1}          else:              local\_dict[day][k[1].url]['ttfb\_sum'] += k[1].TTFB              local\_dict[day][k[1].url]['event\_count'] += 1      print(f'{infile\_name} done!')      return local\_dict  def update\_shared\_state(local\_dict, shared\_dict, lock):      lock.acquire()      try:          for l\_day in local\_dict:              if l\_day in shared\_dict:                  d = shared\_dict[l\_day].copy()                  for url in local\_dict[l\_day]:                      if url in d:                          d[url]['ttfb\_sum'] += local\_dict[l\_day][url]['ttfb\_sum']                          d[url]['event\_count'] += local\_dict[l\_day][url]['event\_count']                      else:                          d[url] = local\_dict[l\_day][url]                  shared\_dict[l\_day] = d              else:                  shared\_dict[l\_day] = local\_dict[l\_day]      finally:          lock.release()  def process\_file(infile\_name, shared\_dict, lock):      local\_dict = aggregate\_file(infile\_name)      update\_shared\_state(local\_dict, shared\_dict, lock)  def query5(data\_dict, outfile):      with open(outfile, 'w') as f:          f.write('Date       URL                         AVG TTFB\n')          for day in data\_dict:              ttfb\_means = {}              time\_str = day.strftime('%Y-%m-%d')              for url in data\_dict[day]:                  mean\_ttfb = data\_dict[day][url]['ttfb\_sum'] / data\_dict[day][url]['event\_count']                  ttfb\_means[url] = mean\_ttfb              # sort and get top 5 (min 5)              top\_ttfb\_urls = sorted(ttfb\_means.items(), key = lambda x : x[1])[:5]              for u in top\_ttfb\_urls:                  f.write(f'{time\_str} {u[0]} {u[1]:.4}\n')  if \_\_name\_\_ == '\_\_main\_\_':      # We take the number of threads as an argument, defaulting to 4 if not      # provided.      parser = argparse.ArgumentParser(description='CSCI E-88 HW2 Problem 5')      parser.add\_argument('num\_threads', default=[4], nargs=argparse.ZERO\_OR\_MORE,                          metavar='numThreads', type=int, help="Number of Threads.")      args = parser.parse\_args(args=None)      futures = []      manager = Manager()      lock = manager.RLock()      shared\_dict = manager.dict()      try:          # Calculation is done in its own thread.          with ProcessPoolExecutor(max\_workers=args.num\_threads[0]) as executor:              # Start a thread for every input file.              for filename in pathlib.Path('.').glob(FILE\_PATTERN):                  futures.append(executor.submit(process\_file, filename, shared\_dict, lock))      except Exception as e:          print(e)          executor.shutdown(wait=False)        for f in futures:          if f.exception():              print(f.exception())      query5(shared\_dict, 'query5.out') |

What are the main differences with the Problem 3 and 4 implementation? [5 points]

|  |
| --- |
| The multilevel dict in this query has only “day” and url. At the bottom dict level (url level), the **totals** of TTFB as well as the event counts are recorded for each URL in the dict. This is required when merging from individual threads so that the final totals can be calculated.  Upon query, the means are calculated from the totals. The totals are kept in the data dict so that additional data can be merged. The query also sorts the TTFB means per url per day and then selects the minimum 5 entries per day. |

What are the results of your query? [5 points] The expected 5 values for 9/12 are provided, please fill in the values for avg TTFB and the URLs for 9/13 and 9/14.

|  |
| --- |
| Date URL Average\_TTFB  9/12/19 <http://example.com/?url=114> 0.393101408  9/12/19 <http://example.com/?url=101> 0.402545  9/12/19 <http://example.com/?url=133> 0.413317187  9/12/19 <http://example.com/?url=033> 0.418867857  9/12/19 <http://example.com/?url=157> 0.419289394  2019-09-13 http://example.com/?url=039 0.4028  2019-09-13 http://example.com/?url=110 0.4295  2019-09-13 http://example.com/?url=005 0.4306  2019-09-13 http://example.com/?url=006 0.4314  2019-09-13 http://example.com/?url=028 0.4319  2019-09-14 http://example.com/?url=123 0.4056  2019-09-14 http://example.com/?url=070 0.4186  2019-09-14 http://example.com/?url=124 0.4233  2019-09-14 http://example.com/?url=107 0.4266  2019-09-14 http://example.com/?url=101 0.4294 |