**Choi, Mason**

September 8, 2022

Harvard University Extension - Principles of Big Data Processing e88

**Homework 2: 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.

|  |
| --- |
| **For example:**  Problem 1: 100% complete  Problem 2: 100% complete  Problem 3: 100% complete  Problem 4: 100% complete  Problem 1.2 Bonus: 100% complete  Problem 5 Bonus: 100% complete  Problem 6 Bonus: 100% complete |

**Problem 1: Analyze CPU [and optionally I/O] intensive multi-threaded processes** [points: 35]

Paste your CPU load generating source code into the following area. If using a provided code, just mention that

|  |
| --- |
| Using the “hw2\_cpu.py” code provided. |

Provide your table or graphs demonstrating the results when running hw2-cpu-load app with 2, 4, and 16 threads on a 4 CPU machine. [15 points]

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 Threads:  Graphical user interface, text  Description automatically generated  4 Threads:  Graphical user interface  Description automatically generated  16 Threads:  Graphical user interface, text  Description automatically generated   |  |  |  | | --- | --- | --- | | Threads | Cores active | % utilization per core | | 2 | 2 | 100 | | 4 | 4 | 100 | | 16 | 4 | 25 | |

What conclusions can you make based on the results? [5 points]

|  |
| --- |
| When running with 2 threads, each thread could run on its own CPU core, each one utilizing 100% of their core. With 4 threads and 4 CPU cores available, again each can run on its own core using 100% of it because this process was basically the only user of CPU processing power.  With 16 threads, however, each thread could not map to its own core, meaning that the cores were split up among the 16 giving 4 threads per core. This meant that each thread at most could only utilize 25% of the core, shown in the graph above.  The speed at which a multithreaded process executes is highly dependent on the cores available, since each thread must share among the others equally, ideally. This is where scaling comes into play, and how much is for us to understand and determine! |

**problem 1.2 (Optional Bonus 5 points):**

Explain what I/O load approach you used - provided app and your modifications (if any)

|  |
| --- |
| I used the provided “hw2\_io.py” code and htop+iotop for CPU and IO statistics monitoring. |

Provide your table or graphs demonstrating the results when running an I/O load app with 2, 4 and 16 threads.

|  |
| --- |
| Text  Description automatically generated  Text  Description automatically generated  Graphical user interface, text  Description automatically generated |

What conclusions can you make based on the results?

|  |
| --- |
| I am not sure why the disk write per core changes from 2 threads to 4 threads even though the CPU utilization per core (100%) is the same for both, and why the total disk write remains constant across all thread counts. However, with more IO load it is clear that performance suffers and the load is shared equally among cores. |

**problem 1.3:**

Show/ summarize your analyses for a 8 CPU machine with 4 and 16 threads [10 points]

|  |
| --- |
| Graphical user interface, text  Description automatically generated    I found similar results as in problem 1.1, threads will share all available CPU cores evenly up to 100% utilization of 1 core. |

**Problem 2: GPU research** [points: 15]

Explain the differences between CPUs vs GPUs; what kind of applications would benefit from using GPUs? [5 points]

|  |
| --- |
| CPUs, or central processing units, generally are designed to handle a wide range of tasks with low latency, and usually handle a single sets of tasks sequentially. GPUs, or graphics processing units, on the other hand often perform better with independent tasks running in parallel. GPUs were initially designed to accelerate graphics rendering, but due to their high parallelization capabilities they have evolved to accommodate machine learning tasks such as training models. |

Review the code of the provided IO-intensive and CPU-intensive programs; would any of them benefit from being run on a GPU-powered system? if not - what changes could be made to make them benefit from GPUs?[5 points]

|  |
| --- |
| The programs untouched would most likely not benefit all that much from being run on a GPU-powered system unless more threads were needed. In their current states, the programs would run just fine if not better on CPUs because of their versatility in diverse code execution.  However, we could adapt the programs to run faster by increasing the threads working, taking advantage of the many orders of magnitude more cores on a GPU system than a CPU one. For example, we could optimize the Fibonacci calculator to calculate the n numbers in a couple different threads and split the amount of file line writing among multiple cores. Since the main difference between GPU and CPU is the high parallelization capabilities, tweaking code to [intelligently] utilize more cores would increase efficiency. |

Show your research on AWS EC2 instances with GPU support including most and least powerful?[5 points]

|  |
| --- |
| A complete list of instances can be found at <https://aws.amazon.com/ec2/instance-types/>, with instances with GPU support under the “accelerated computing” tab. Namely the P and G instances.  Graphical user interface, text, application, email  Description automatically generated  It looks like AWS’s p3dn.24xlarge is the most powerful GPU EC2 instance by far (you can tell by than name!), delivering 5,120 CUDA Cores and 640 Tensor Cores per unit, and with 8 GPUs that’s 40,960 CUDA Cores and 5,120 Tensor Cores. That’s over 45,000 cores, more than 2,500X the number my local computer has!  Text  Description automatically generated  The least powerful instance with GPU support looks to be the g5.xlarge sporting a single NVIDIA A10G Tensor Core GPU that has 320 third-generation NVIDIA Tensor Cores.  A picture containing table  Description automatically generated |

**Problem 3: unique counts** [points: 30]

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

|  |
| --- |
| *# Copyright (c) 2022 CSCIE88 Marina Popova*  *# Code adapted by Mason Choi in September of 2022 for CSCIE88*  *# {"2022-09-04:07": {"url1": {user-01, user-02}, "url2": {...}, ...}, ...}*  import argparse  from datetime import date  import multiprocessing  from multiprocessing import Process, Value, Lock  from collections import namedtuple, defaultdict  from threading import local  prog = "event\_counter\_shared\_ctypes\_mem"  desc = "run specified number of threads - use shared event counter"  parser = argparse.ArgumentParser(prog=prog, description=desc)  parser.add\_argument('--thread-count', '-tc', default=4, type=int)  parser.add\_argument('--logs-directory', '-ld', required=False,                      help="Directory where the log files 01-04 are stored. "                           "If not supplied, this program assumes that all 4 log files"                           " are directly in the present working directory.")  parsed\_args = parser.parse\_args()  thread\_count = parsed\_args.thread\_count  logs\_dir = parsed\_args.logs\_directory  Event = namedtuple('Event',                     ['uuid', 'timestamp', 'url', 'userid', 'country', 'ua\_browser', 'ua\_os', 'response\_status', 'TTFB'])  parsed\_args = parser.parse\_args()  thread\_count = parsed\_args.thread\_count  **def** do\_work(shared\_results, lock, file\_name):      local\_dict = {}  *# initialize local data dict*      with open(file\_name) as file\_handle:          events = map(parse\_line, file\_handle)          for event in events:              date\_hour = event[1][:event[1].find(":")]              date\_hour = date\_hour.replace("T", ":")  *# parse datetime into date:hour form*              url = event[2]              user = event[3]  *# aggregate relevant data into dict of dicts*              if date\_hour in local\_dict:                  if url in local\_dict[date\_hour]:                      local\_dict[date\_hour][url].add(user)  *# sets allow only unique values*                  else:                      local\_dict[date\_hour][url] = set([user])              else:                  local\_dict[date\_hour] = {url: set([user])}          with lock:              shared\_results.append(local\_dict.copy())  *# append data to shared list*          print(file\_name, "has finished processing")  **def** parse\_line(line):      return Event(\*line.split(','))  if \_\_name\_\_ == '\_\_main\_\_':      if logs\_dir is None:          opt\_dir = "logs/"      elif logs\_dir.endswith("/"):          opt\_dir = logs\_dir      else:          opt\_dir = logs\_dir + "/"      with multiprocessing.Manager() as manager:  *# declare a shared list to record each file's dict*          shared\_results = manager.list()          lock = Lock()          jobs = []          for thread\_id in range(thread\_count):              file\_name = opt\_dir + "file-input%s.csv" % str(thread\_id + 1)              t = multiprocessing.Process(                  target=do\_work,                  args=(shared\_results, lock, file\_name))              jobs.append(t)              t.start()          for curr\_job in jobs:              curr\_job.join()          print("Threads finished!")  *# threads have finished*          final\_dict = {}  *# initialize final data structure*  *# compile file data into final\_dict*          for file\_dict in shared\_results:              for date\_hour in file\_dict.keys():                  if date\_hour in final\_dict:                      for url in file\_dict[date\_hour]:                          if url in final\_dict[date\_hour]:                              final\_dict[date\_hour][url] |= file\_dict[date\_hour][url]                          else:                              final\_dict[date\_hour][url] = file\_dict[date\_hour][url]                  else:                      final\_dict[date\_hour] = file\_dict[date\_hour]    *# QUERY 1, PROBLEM 3.3.1*          for hour in ["07", "08", "09", "10", "11"]:              date\_hour = "2022-09-04:" + hour              print(**f**"{date\_hour}, {len(final\_dict[date\_hour])}")    *# QUERY 2, PROBLEM 3.3.2*          for url\_id in range(1, 6):              date\_hour = "2022-09-06:13"              url = **f**"http://example.com/?url=00{url\_id}"              print(**f**"{date\_hour}, {url}, {len(final\_dict[date\_hour][url])}") |

Explain your choice of the data structures for shared state management [10 points]

|  |
| --- |
| I chose to go with a dictionary of dictionaries data structure for my shared state management. The first set of keys are the unique timestamps in the form date:hour, corresponding to a dictionary value with unique URLs as keys and a set of userIDs as values. In this way, one data structure can meet the needs of both query requirements mentioned below, with the amount of unique URLs stored as the length of the child dictionaries associated, and the unique user counts stored as the child dictionary’s set length.  My data structure looks as follows:  *{“date\_time”: {“url”: {userID, userID}, “url”: {userID}, …}, “date\_time”: …}*  My actual shared state structure is a list, in which each dict of dicts (one for each file) is stored. After all threads complete, a simple chunk of code executes to combine each of these dicts of dicts into one final data structure, which then can be used for said queries. In this way, the processes can run in basically true parallel, drastically improving runtime over locking the shared structure to modify it in every thread. |

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>**  2022-09-04:07, 184  2022-09-04:08, 185  2022-09-04:09, 187  2022-09-04:10, 192  2022-09-04:11, 190    **Query 2**  **<date:hour:url>, unique\_user\_count**  2022-09-06:13, http://example.com/?url=001, 1  2022-09-06:13, http://example.com/?url=002, 1  2022-09-06:13, http://example.com/?url=003, 6  2022-09-06:13, http://example.com/?url=004, 5  2022-09-06:13, http://example.com/?url=005, 3 |

**Problem 4: Unique Count per URL per day/hour** [points: 20]

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

|  |
| --- |
| *# Copyright (c) 2022 CSCIE88 Marina Popova*  *# Code adapted by Mason Choi in September of 2022 for CSCIE88*  *# query 3: {"2022-09-04:07": {"url1": 4, "url2": 1, "url3": 99}, ...}*  import argparse  from datetime import date  import multiprocessing  from multiprocessing import Process, Value, Lock  from collections import namedtuple, defaultdict  from threading import local  prog = "event\_counter\_shared\_ctypes\_mem"  desc = "run specified number of threads - use shared event counter"  parser = argparse.ArgumentParser(prog=prog, description=desc)  parser.add\_argument('--thread-count', '-tc', default=4, type=int)  parser.add\_argument('--logs-directory', '-ld', required=False,                      help="Directory where the log files 01-04 are stored. "                           "If not supplied, this program assumes that all 4 log files"                           " are directly in the present working directory.")  parsed\_args = parser.parse\_args()  thread\_count = parsed\_args.thread\_count  logs\_dir = parsed\_args.logs\_directory  Event = namedtuple('Event',                     ['uuid', 'timestamp', 'url', 'userid', 'country', 'ua\_browser', 'ua\_os', 'response\_status', 'TTFB'])  parsed\_args = parser.parse\_args()  thread\_count = parsed\_args.thread\_count  **def** do\_work(shared\_results, lock, file\_name):      local\_dict = {}  *# initialize local data dict*      with open(file\_name) as file\_handle:          events = map(parse\_line, file\_handle)          for event in events:              date\_hour = event[1][:event[1].find(":")]              date\_hour = date\_hour.replace("T", ":")  *# parse datetime into date:hour form*              url = event[2]  *# aggregate relevant data into dict of dicts*              if date\_hour in local\_dict:                  if url in local\_dict[date\_hour]:                      local\_dict[date\_hour][url] += 1                  else:                      local\_dict[date\_hour][url] = 1              else:                  local\_dict[date\_hour] = {url: 1}          with lock:              shared\_results.append(local\_dict.copy())  *# append data to shared list*          print(file\_name, "has finished processing")  **def** parse\_line(line):      return Event(\*line.split(','))  if \_\_name\_\_ == '\_\_main\_\_':      if logs\_dir is None:          opt\_dir = "logs/"      elif logs\_dir.endswith("/"):          opt\_dir = logs\_dir      else:          opt\_dir = logs\_dir + "/"      with multiprocessing.Manager() as manager:  *# declare a shared list to record each file's dict*          shared\_results = manager.list()          lock = Lock()          jobs = []          for thread\_id in range(thread\_count):              file\_name = opt\_dir + "file-input%s.csv" % str(thread\_id + 1)              t = multiprocessing.Process(                  target=do\_work,                  args=(shared\_results, lock, file\_name))              jobs.append(t)              t.start()          for curr\_job in jobs:              curr\_job.join()          print("Threads finished!")  *# threads have finished*          final\_dict = {}  *# initialize final data structure*  *# compile file data into final\_dict*          for file\_dict in shared\_results:              for date\_hour in file\_dict.keys():                  if date\_hour in final\_dict:                      for url in file\_dict[date\_hour]:                          if url in (final\_dict[date\_hour]):                              final\_dict[date\_hour][url] += file\_dict[date\_hour][url]                          else:                              final\_dict[date\_hour][url] = file\_dict[date\_hour][url]                  else:                      final\_dict[date\_hour] = file\_dict[date\_hour]    *# QUERY 3, PROBLEM 4.2*          for url\_id in ["06", "07", "09", "10", "11"]:              date\_hour = "2022-09-05:06"              url = **f**"http://example.com/?url=0{url\_id}"              print(**f**"{date\_hour}, {url}, {final\_dict[date\_hour][url]}") |

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 3**  **<date:hour:url>, event\_count**  2022-09-05:06, http://example.com/?url=006, 1  2022-09-05:06, http://example.com/?url=007, 2  2022-09-05:06, http://example.com/?url=009, 2  2022-09-05:06, http://example.com/?url=010, 4  2022-09-05:06, http://example.com/?url=011, 4 |

**Problem 5: [Bonus] time range queries** [points: 10]

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

|  |
| --- |
| *# Copyright (c) 2022 CSCIE88 Marina Popova*  *# Code adapted by Mason Choi in September of 2022 for CSCIE88*  *# {"2022-09-04:07": {"country": {url1, url2}, "country": {...}, ...}, ...}*  import argparse  from datetime import date, datetime  import multiprocessing  from multiprocessing import Process, Value, Lock  from collections import namedtuple, defaultdict  from threading import local  from dateutil.parser import parse  prog = "event\_counter\_shared\_ctypes\_mem"  desc = "run specified number of threads - use shared event counter"  parser = argparse.ArgumentParser(prog=prog, description=desc)  parser.add\_argument('--thread-count', '-tc', default=4, type=int)  parser.add\_argument('--time-range', '-tr', required=True, type=str, nargs=2,                      help="Time range for query, in form YEAR-MONTH-DAYTHOUR. Inclusive.")  parser.add\_argument('--logs-directory', '-ld', required=False,                      help="Directory where the log files 01-04 are stored. "                           "If not supplied, this program assumes that all 4 log files"                           " are directly in the present working directory.")  parsed\_args = parser.parse\_args()  thread\_count = parsed\_args.thread\_count  logs\_dir = parsed\_args.logs\_directory  time\_range = [parse(parsed\_args.time\_range[0]), parse(parsed\_args.time\_range[1])]  Event = namedtuple('Event',                     ['uuid', 'timestamp', 'url', 'userid', 'country', 'ua\_browser', 'ua\_os', 'response\_status', 'TTFB'])  parsed\_args = parser.parse\_args()  thread\_count = parsed\_args.thread\_count  **def** do\_work(shared\_results, lock, file\_name):      local\_dict = {}  *# initialize local data dict*      with open(file\_name) as file\_handle:          events = map(parse\_line, file\_handle)          for event in events:              date\_hour = event[1][:event[1].find(":")]  *# check if time is within specified range*              if parse(date\_hour) >= time\_range[0] and parse(date\_hour) <= time\_range[1]:                  date\_hour = date\_hour.replace("T", ":")  *# parse datetime into date:hour form*                  country = event[4]                  url = event[2]  *# aggregate relevant data into dict of dicts*                  if date\_hour in local\_dict:                      if country in local\_dict[date\_hour]:                          local\_dict[date\_hour][country].add(url)  *# sets allow only unique values*                      else:                          local\_dict[date\_hour][country] = set([url])                  else:                      local\_dict[date\_hour] = {country: set([url])}          with lock:              shared\_results.append(local\_dict.copy())  *# append data to shared list*          print(file\_name, "has finished processing")  **def** parse\_line(line):      return Event(\*line.split(','))  if \_\_name\_\_ == '\_\_main\_\_':      if logs\_dir is None:          opt\_dir = "logs/"      elif logs\_dir.endswith("/"):          opt\_dir = logs\_dir      else:          opt\_dir = logs\_dir + "/"      with multiprocessing.Manager() as manager:  *# declare a shared list to record each file's dict*          shared\_results = manager.list()          lock = Lock()          jobs = []          for thread\_id in range(thread\_count):              file\_name = opt\_dir + "file-input%s.csv" % str(thread\_id + 1)              t = multiprocessing.Process(                  target=do\_work,                  args=(shared\_results, lock, file\_name))              jobs.append(t)              t.start()          for curr\_job in jobs:              curr\_job.join()          print("Threads finished!")  *# threads have finished*          final\_dict = {}  *# initialize final data structure*  *# compile file data into final\_dict*          for file\_dict in shared\_results:              for date\_hour in file\_dict.keys():                  if date\_hour in final\_dict:                      for country in file\_dict[date\_hour]:                          if country in final\_dict[date\_hour]:                              final\_dict[date\_hour][country] |= file\_dict[date\_hour][country]                          else:                              final\_dict[date\_hour][country] = file\_dict[date\_hour][country]                  else:                      final\_dict[date\_hour] = file\_dict[date\_hour]    *# QUERY 5, PROBLEM 5.2.1*          queries = [              ("2022-09-05", "19", "AE"),              ("2022-09-05", "19", "AF"),              ("2022-09-05", "19", "UG"),              ("2022-09-05", "23", "PN"),              ("2022-09-06", "07", "FO"),              ("2022-09-06", "08", "TK"),          ]          for query\_params in queries:              query = final\_dict[**f**"{query\_params[0]}:{query\_params[1]}"][query\_params[2]]              print(**f**"{query\_params[0]}, {query\_params[1]}, {query\_params[2]} : {len(query)}") |

What are the results of your queries for the following time range and specified keys? [5points]

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

|  |
| --- |
| **Query 4**  Time range [t1, t2]=[2022-09-05 19:00:00, 2022-09-06 08:00:00]  **<date hour country>, url\_count**  Text  Description automatically generated |

**Problem 6: Bonus: Top N queries** [10 points]

Paste your source code into the following area [3 points]

|  |
| --- |
| *# Copyright (c) 2022 CSCIE88 Marina Popova*  *# Code adapted by Mason Choi in September of 2022 for CSCIE88*  *# {"2022-09-04": {"url": (TTFB, TTFB), "url": {...}, ...}, ...}*  import argparse  from datetime import date  import multiprocessing  from multiprocessing import Process, Value, Lock  from collections import namedtuple, defaultdict  from threading import local  from typing import final  prog = "event\_counter\_shared\_ctypes\_mem"  desc = "run specified number of threads - use shared event counter"  parser = argparse.ArgumentParser(prog=prog, description=desc)  parser.add\_argument('--thread-count', '-tc', default=4, type=int)  parser.add\_argument('--logs-directory', '-ld', required=False,                      help="Directory where the log files 01-04 are stored. "                           "If not supplied, this program assumes that all 4 log files"                           " are directly in the present working directory.")  parsed\_args = parser.parse\_args()  thread\_count = parsed\_args.thread\_count  logs\_dir = parsed\_args.logs\_directory  Event = namedtuple('Event',                     ['uuid', 'timestamp', 'url', 'userid', 'country', 'ua\_browser', 'ua\_os', 'response\_status', 'TTFB'])  parsed\_args = parser.parse\_args()  thread\_count = parsed\_args.thread\_count  **def** do\_work(shared\_results, lock, file\_name):      local\_dict = {}  *# initialize local data dict*      with open(file\_name) as file\_handle:          events = map(parse\_line, file\_handle)          for event in events:              date\_hour = event[1][:event[1].find("T")]  *# parse datetime into date*              url = event[2]              TTFB = float(event[8])  *# aggregate relevant data into dict of dicts*              if date\_hour in local\_dict:                  if url in local\_dict[date\_hour]:                      local\_dict[date\_hour][url].append(TTFB)  *# sets allow only unique values*                  else:                      local\_dict[date\_hour][url] = [TTFB]              else:                  local\_dict[date\_hour] = {url: [TTFB]}          with lock:              shared\_results.append(local\_dict.copy())  *# append data to shared list*          print(file\_name, "has finished processing")  **def** parse\_line(line):      return Event(\*line.split(','))  if \_\_name\_\_ == '\_\_main\_\_':      if logs\_dir is None:          opt\_dir = "logs/"      elif logs\_dir.endswith("/"):          opt\_dir = logs\_dir      else:          opt\_dir = logs\_dir + "/"      with multiprocessing.Manager() as manager:  *# declare a shared list to record each file's dict*          shared\_results = manager.list()          lock = Lock()          jobs = []          for thread\_id in range(thread\_count):              file\_name = opt\_dir + "file-input%s.csv" % str(thread\_id + 1)              t = multiprocessing.Process(                  target=do\_work,                  args=(shared\_results, lock, file\_name))              jobs.append(t)              t.start()          for curr\_job in jobs:              curr\_job.join()          print("Threads finished!")  *# threads have finished*          final\_dict = {}  *# initialize final data structure*  *# compile file data into final\_dict*          for file\_num, file\_dict in enumerate(shared\_results):              for date in file\_dict.keys():                  if date in final\_dict:                      for url in file\_dict[date]:                          if url in final\_dict[date]:                              final\_dict[date][url] += file\_dict[date][url]                          else:                              final\_dict[date][url] = file\_dict[date][url]                  else:                      final\_dict[date] = file\_dict[date]                    if file\_num == thread\_count-1:                      for url in final\_dict[date]:                          final\_dict[date][url] = sum(final\_dict[date][url]) / len(final\_dict[date][url])    *# QUERY 6, PROBLEM 6.3.1*          sorted(final\_dict["2022-09-03"].items(), key=**lambda** item: item[1])  *# sort value ascending*          dates = ["2022-09-03", "2022-09-04", "2022-09-05"]          for date in dates:              print("\nDate       URL                         Average\_TTFB")              sorted\_dict = sorted(final\_dict[date].items(), key=**lambda** item: item[1])              for TTFB in sorted\_dict[:5]:  *# print lowest 5 TTFBs' information*                  print(**f**"{date} {TTFB[0]} {TTFB[1]}") |

What are the main differences with the Problem 3 and 4 implementation? [2 points]

|  |
| --- |
| The largest difference between this solution and the problem 3/4 solutions is the extra post-processing required to average and sort the data. Since my approach required each thread to write to its own local data structure and then copy that into the shared state list, during the combination code, after multithreading is complete, I must also average each URL’s TTFBs in addition to combining them. Finally, I implemented a clever “sort by value” line to sort the URLs by TTFBs ascending, meaning that I can easily change the amount of points to pull (ie. Top 15 instead of top 5). |

What are the results of your query? [5 points] The expected 5 values for 9/03 are provided, please fill in the values for avg TTFB and the URLs for 9/04 and 9/05. [5 points]

|  |
| --- |
| Date URL Average\_TTFB  2022-09-03 http://example.com/?url=006 0.345907142  2022-09-03 http://example.com/?url=133 0.372253846  2022-09-03 http://example.com/?url=197 0.382582758  2022-09-03 http://example.com/?url=094 0.396904545  2022-09-03 http://example.com/?url=054 0.3975125    Date URL Average\_TTFB  2022-09-04 http://example.com/?url=043 0.41260892857142856  2022-09-04 http://example.com/?url=105 0.4236718749999999  2022-09-04 http://example.com/?url=037 0.42634266666666665  2022-09-04 http://example.com/?url=121 0.4273459016393443  2022-09-04 http://example.com/?url=170 0.4288125  Date URL Average\_TTFB  2022-09-05 http://example.com/?url=129 0.38792285714285707  2022-09-05 http://example.com/?url=095 0.4184578947368421  2022-09-05 http://example.com/?url=103 0.4220188405797102  2022-09-05 http://example.com/?url=043 0.42509841269841264  2022-09-05 http://example.com/?url=177 0.4350882352941177 |