**Merritt, Jessica**

09/17/2019

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.

|  |
| --- |
| *for example:*  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]

|  |
| --- |
| # Merritt, Jessica Homework 2 Problem 1  # Creates given number of threads and preforms CPU intensive work  import argparse  from time import sleep  import multiprocessing  import os  prog = "hw2\_question1"  desc = "Creates and starts a specified number of threads and preforms CPU intensive work on loop"  parser = argparse.ArgumentParser(prog=prog, description=desc)  parser.add\_argument('--numThreads', '-l', default=4, type=int)  parsed\_args = parser.parse\_args()  numThreads = parsed\_args.numThreads  def calcuate\_fibonacci\_infinite():  while True:  sleep(1) # sleep for 1 sec  print(os.getpid()) #print thread process id  fibonacci(1000000000) #calculate 1 billonith fibonacci number  #compute fibonacci to nth decimal  def fibonacci(n):  a = 0  b = 1  if n < 0:  print("Incorrect input")  elif n == 0:  return a  elif n == 1:  return b  else:  for i in range(2,n):  c = a + b  a = b  b = c  return b  jobs = []  for thread in range(numThreads):  t = multiprocessing.Process(target=calcuate\_fibonacci\_infinite, args=())  jobs.append(t)  t.start() # new child process is started at this point, it has its own execution flow  for curr\_job in jobs:  curr\_job.join()  print "Process Completed" |

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]

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Utilization at start | Utilization at middle | Utilization after a few minutes |
| 2 | 2 CPU at 62% | 2 CPU at 100% | 1 CPU at 100%  Then process dies |
| 4 | 4 CPU at ~50% | 4 CPU at 100% | 1 CPU at 100%  Then process dies |
| 16 | 4 CPU at ~100% | 4 CPU at ~70% as threads are dropped | 1 CPU at 100%  Then process dies |

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

|  |
| --- |
| The CPU is quickly overwhelmed by the threads at each level causing it to fall over and kill all the threads. It initially tries to spread the work across as many CPUS as possible (2 for the 2 threaded process) but as usage reaches 100% the individual processes are killed. This knocked over my instances eventually as it struggled with deleting the last process. |

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]

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Utilization at start | Utilization at middle | Utilization after a few minutes |
| 2 | 2 CPU at 100% | 1 CPU at 100% | 1 CPU at 100% |
| 4 | 4 CPU at 100% | 2 CPU at 100% | 1 CPU at 100% |
| 16 | 8 CPU at ~ 95% | 8 CPU bouncing between 100% and ~50% as threads drop | 1 CPU at 100% |

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

|  |
| --- |
| The 8 CPU machine attempts to utilize as many CPUs as possible, but ends up slowly killing the threads. Unlike the 4 CPU machine it does not kill all other processes on the machine as the last thread dies. The CPU utilization follows the same pattern across both machines |

**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]

|  |
| --- |
| # Merritt, Jessica Homework 2 Problem 2  # Creates given number of threads and preforms IO intensive work  import argparse  from random import randint  from time import sleep  import multiprocessing  import os  NEW\_LINE\_CHAR = "\n"  prog = "hw2\_question1"  desc = "Creates and starts a specified number of threads and preforms IO intensive work on loop"  parser = argparse.ArgumentParser(prog=prog, description=desc)  parser.add\_argument('--numThreads', '-t', default=4, type=int)  parsed\_args = parser.parse\_args()  numThreads = parsed\_args.numThreads  def infinite\_io():  while True:  sleep(1) # sleep for 1 sec  print(os.getpid()) #print thread process id  io\_work() #calculate 1 billonith fibonacci number  # Write 10000 random numbers to a file and delete it  def io\_work():  fileName = "random\_num\_files{}.txt".format(os.getpid())  file = open(fileName, "w")  for i in range(10000):  file.write("{}{}".format(randint(0, 10000), NEW\_LINE\_CHAR))  file.close()  fileRead = open(fileName, "r")  contents = fileRead.read()  fileRead.close()  os.remove(fileName)  jobs = []  for thread in range(numThreads):  t = multiprocessing.Process(target=infinite\_io, args=())  jobs.append(t)  t.start() # new child process is started at this point, it has its own execution flow  for curr\_job in jobs:  curr\_job.join()  print "Process Completed" |

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]

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Utilization at start | Utilization at middle | Utilization after a few minutes |
| 2 | 2 CPU at 1.7%  No DISK READ  Total DISK WRITE : 94.44 K/s  Actual DISK WRITE: 0.00B/s | 3 CPU sharing between .7 - 1.7%  No DISK READ  Total DISK WRITE : 94.36 K/s  Actual DISK WRITE: 5.36 B/s | 2-3 CPU bouncing at between 1-3%  No DISK READ  Total DISK WRITE : 94.50 K/s  Actual DISK WRITE: 0.00 B/s |
| 4 | 4 CPU at ~2.7%  No DISK READ  Total DISK WRITE : 189.04 K/s  Actual DISK WRITE: 2.95 K/s | 4 CPU at 1.3%  No DISK READ  Total DISK WRITE : 188.36 K/s  Actual DISK WRITE: 2.95 K/s | 4 CPU at.07-2.7%  No DISK READ  Total DISK WRITE : 188.90 K/s  Actual DISK WRITE: 2.95 K/s |
| 16 | 4 CPU at ~9-12%  No DISK READ  Total DISK WRITE : 755.17 K/s  Actual DISK WRITE: 8.85 K/s | 4 CPU at ~4-7%  No DISK READ  Total DISK WRITE : 740.00 K/s  Actual DISK WRITE: 11.80 K/s | 4 CPU at ~7%  No DISK READ  Total DISK WRITE : 754.48 K/s  Actual DISK WRITE: 11.81 K/s |

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

|  |
| --- |
| CPU usage increases inline with thread number increase. Disk read also increase as thread number increases and more files are being created. These processes are so low in CPU usage that sometimes random CPU usage from running the instance shows on a CPU. |

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]

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Utilization at start | Utilization at middle | Utilization after a few minutes |
| 2 | 2 CPU at 2%  No DISK READ  Total DISK WRITE : 94.34 K/s  Actual DISK WRITE: 0.00 B/s | 4 CPUs sharing ~3% utilization per thread  No DISK READ  Total DISK WRITE : 94.36 K/s  Actual DISK WRITE: 11.34 K/s | 2-5 CPUs sharing total ~2.5 utilization per thread  No DISK READ  Total DISK WRITE : 94.36 K/s  Actual DISK WRITE: 0.00 B/s |
| 4 | 6 CPU at 1.7%  No DISK READ  Total DISK WRITE : 188.72 K/s  Actual DISK WRITE: 2.95 K/s | 3-5 CPU sharing about 2.7 % per thread  No DISK READ  Total DISK WRITE : 188.72 K/s  Actual DISK WRITE: 2.95 K/s | 3-5 CPU sharing about 2.7 % per thread  No DISK READ  Total DISK WRITE : 188.72 K/s  Actual DISK WRITE: 2.95 K/s |
| 16 | 8 CPU between 2-7 %  No DISK READ  Total DISK WRITE : 754.48 K/s  Actual DISK WRITE: 3.93 K/s | 8 CPU between 2-7 %  No DISK READ  Total DISK WRITE : 754.08 K/s  Actual DISK WRITE: 3.93 K/s | 8 CPU between 2-7 %  No DISK READ  Total DISK WRITE : 754.38 K/s  Actual DISK WRITE: 3.93 K/s |

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

|  |
| --- |
| The program wasn’t very read intensive, but write went up with thread number. CPU usage remained low and distributed. The two machines preformed equally well here with the 8 CPU having a slight advantage with the 16 thread CPU usage, but similar write speeds. This shows that threading can only help so much here as the limit is on the i/o resources like the hard drive. |

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

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

|  |
| --- |
| # Merritt, Jessica Homework 2 Problem 3  # Creates 4 threads to reads files and answer 3 queries  import multiprocessing  def readFileAndStoreData(name, urlData, userData, visitData, lock):  file = open(name, "r")  line = file.readline()  while line:  contents = line.strip().split(",")  time = contents[1][:13].replace("T", ":")  url = contents[2]  user = contents[3]  with lock:  storeData(time, url, user, urlData, userData, visitData)  line = file.readline()  file.close()  def storeData(time, url, user, urlData, userData, visitData):  #Update url count  urlDict = urlData  userDict = userData  visitDict = visitData  if time in urlData:  copyUrl = urlDict[time]  copyUrl.add(url)  urlDict[time] = copyUrl  else:  urlDict[time] = set([url])  urlData = urlDict  timeUrlPair = "{}:{}".format(time, url)  if timeUrlPair in userDict:  copyUser = userDict[timeUrlPair]  copyUser.add(user)  userDict[timeUrlPair] = copyUser  visitDict[timeUrlPair]+=1  else:  userDict[timeUrlPair] = set([user])  visitDict[timeUrlPair] = 1  userData = userDict  visitData = visitDict  def calculateQuery(urlData, userData, visitData):  # URL per hour per day  file1 = open("Query1.txt", "w")  # Unique visitors per URL per hour per day  file2 = open("Query2.txt", "w")  # unique (by line) events/clicks per URL per hour per day  file3 = open("Query3.txt", "w")  for hour in urlData :  file1.write("{}, {}\n".format(hour, len(urlData[hour])))  for urlTimePair in userData :  file2.write("{}, {}\n".format(urlTimePair, len(userData[urlTimePair])))  file3.write("{}, {}\n".format(urlTimePair, visitData[urlTimePair]))  file1.close()  file2.close()  file3.close()  #Start 4 threads to process all 4 files  if \_\_name\_\_ == '\_\_main\_\_':  jobs = []  lock = multiprocessing.Lock()  manager = multiprocessing.Manager()  hourToUrlDict = manager.dict()  hourAndUrlToUserDict = manager.dict()  hourAndUrlToVisitsDict = manager.dict()  for i in range(4):  fileName = "file-input{}.csv".format(i + 1)  t = multiprocessing.Process(target=readFileAndStoreData, args=(fileName, hourToUrlDict, hourAndUrlToUserDict, hourAndUrlToVisitsDict, lock))  jobs.append(t)  t.start()  # wait for all threads to finish  for curr\_job in jobs:  curr\_job.join()  calculateQuery(hourToUrlDict, hourAndUrlToUserDict, hourAndUrlToVisitsDict) |

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

|  |
| --- |
| Due to issues with thread safe nested dicts I opted for a separate dicts for each query.  The first query is a map of time to a set of URLs at that time. This means calculating the query is just calculating the length of each key.  The second query uses a map of time+url to a set of users.  The third query uses a map of time+url to a count of visits. These two queries go hand in hand and can be updated and read together reducing time spent. The data structure for Query 2 allows for a quick length calculation for the answer, and the data structure for 3 is essentially a list of answers.  All shared data updating is done within a lock to prevent race conditions |

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]

|  |
| --- |
| # Merritt, Jessica Homework 2 Problem 3  # Creates 4 threads to reads files and answer 3 queries  import multiprocessing  import datetime as dt  def readFileAndStoreData(name, q4Data, lock):  file = open(name, "r")  line = file.readline()  while line:  contents = line.strip().split(",")  time = contents[1][:13].replace("T", ":")  url = contents[2]  user = contents[3]  country = contents[4]  date = dt.datetime.strptime(time, '%Y-%m-%d:%H')  t1 = dt.datetime.strptime("2019-09-13:17", '%Y-%m-%d:%H')  t2 = dt.datetime.strptime("2019-09-14:09", '%Y-%m-%d:%H')  if (t1 <= date <= t2) :  with lock:  storeQuery4(url, country, time, q4Data)  line = file.readline()  file.close()  def storeQuery4(url, country, time, data):  timeCountryString = "{},{}".format(time, country)  dictCopy = data  if timeCountryString in data:  copyUrl = dictCopy[timeCountryString]  copyUrl.add(url)  dictCopy[timeCountryString] = copyUrl  else:  dictCopy[timeCountryString] = set([url])  data = dictCopy  def calculateQuery(q4Data):  file1 = open("Query4.txt", "w")  for timeCountryPair in q4Data :  file1.write("{}, {}\n".format(timeCountryPair, len(q4Data[timeCountryPair])))  file1.close()  #Start 4 threads to process all 4 files  if \_\_name\_\_ == '\_\_main\_\_':  jobs = []  lock = multiprocessing.Lock()  manager = multiprocessing.Manager()  countryToUrlDict = manager.dict()  for i in range(4):  fileName = "file-input{}.csv".format(i + 1)  t = multiprocessing.Process(  target=readFileAndStoreData,  args=(fileName,  countryToUrlDict,  lock))  jobs.append(t)  t.start()  # wait for all threads to finish  for curr\_job in jobs:  curr\_job.join()  calculateQuery(countryToUrlDict) |

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

|  |
| --- |
| This is mostly similar with the exception being I needed to parse dates and also grab country codes. Here there is more processing the occurs before a lock is acquired, but the data structure is very similar to the one used for Query1. |

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]

|  |
| --- |
| # Merritt, Jessica Homework 2 Problem 5  # Creates 4 threads to reads files and answers Query 5  import multiprocessing  import operator  def readFileAndStoreData(name, q5Data, lock):  file = open(name, "r")  line = file.readline()  while line:  contents = line.strip().split(",")  date = contents[1][:10]  url = contents[2]  ttfb = contents[8]  with lock:  storeQuery5(url, date, ttfb, q5Data)  line = file.readline()  file.close()  def storeQuery5(url, date, ttfb, data):  dateUrlString = "{},{}".format(date, url)  dataCopy = data  if dateUrlString in data:  arrayOfTtfb = dataCopy[dateUrlString]  arrayOfTtfb.append(float(ttfb))  dataCopy[dateUrlString] = arrayOfTtfb  else:  dataCopy[dateUrlString] = [float(ttfb)]  data = dataCopy  def calculateQuery(q5Data):  averageDict = {}  #Get averages  for dateUrl in q5Data :  splitKey = dateUrl.split(",")  date = splitKey[0]  url = splitKey[1]  ttfbArray = q5Data[dateUrl]  average = sum(ttfbArray) / len(ttfbArray)  if date in averageDict:  averageDict[date][url] = average  else:  averageDict[date] = {url : average}  #print output  file1 = open("Query5.txt", "w")  file1.write("Date url AVG TTFB\n")  for date in averageDict:  #Sort averages per day and take top 5  sortedAverages = sorted(averageDict[date].iteritems(), key=operator.itemgetter(1))[:5]  for pair in sortedAverages:  file1.write("{} {} {}\n".format(date, pair[0], pair[1]))  file1.write("\n")  file1.close()  #Start 4 threads to process all 4 files  if \_\_name\_\_ == '\_\_main\_\_':  jobs = []  lock = multiprocessing.Lock()  manager = multiprocessing.Manager()  data = manager.dict()  for i in range(4):  fileName = "file-input{}.csv".format(i + 1)  t = multiprocessing.Process(  target=readFileAndStoreData,  args=(fileName,  data,  lock))  jobs.append(t)  t.start()  # wait for all threads to finish  for curr\_job in jobs:  curr\_job.join()  calculateQuery(data) |

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

|  |
| --- |
| This implementation does far more of the calculation after exiting the threads due to a need to select the top 5 from each day. The averages could have had a running calculation using sum and count, but in the end some of the data processing needed to be done on one thread |

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> 0.402814285714  2019-09-13 <http://example.com/?url=110> 0.429469354839  2019-09-13 <http://example.com/?url=005> 0.430635849057  2019-09-13 <http://example.com/?url=006> 0.43138115942  2019-09-13 <http://example.com/?url=028> 0.431906451613  2019-09-14 <http://example.com/?url=123> 0.405598076923  2019-09-14 <http://example.com/?url=070> 0.418589333333  2019-09-14 <http://example.com/?url=124> 0.423315873016  2019-09-14 <http://example.com/?url=107> 0.426570588235  2019-09-14 <http://example.com/?url=101> 0.429436923077 |