**Cheng, Loi**

9/19/2019

Harvard University Extension - Principles of Big Data Processing e88

Homework 3: Shared state with Redis; HDFS

* **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>\_HW3.zip**
* **Make sure to add full result files into that archive as well**

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: Bonus: 100% complete  Problem 4: 100% complete  Problem 5: 100% complete |

**Problem 1: unique counts** [points: 40]

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

|  |
| --- |
| #hw3\_problem1.py  #Loi Cheng  #9/19/2019  import argparse  import redis  import pandas as pd  def processData(fileName, redisClient):      '''      reduces a set of data to useful lists, which will then be uploaded to a redis server      '''      df = pd.read\_csv(fileName,header=None)      df['DateTime'] = pd.to\_datetime(df[1])      df['DateHour'] = df['DateTime'].dt.date.astype('str') + ':' + df['DateTime'].dt.strftime('%H').astype('str')      df['DateHour\_URL'] = df['DateHour'].astype('str') + ':' + df[2].astype('str')      df = df.rename(columns={0:'UUID', 2: 'url', 3: 'user'})      # Query1      # reduce to list with hour and unique url for that hour      dfQueryOne = df[['DateHour','url']].drop\_duplicates()      hourUrlList = dfQueryOne.values.tolist()      # update redis with the new data, with 'DateHour' as the key, and the corresponding set of unique urls as the value      # get the number of unique urls after the update and store it in another key      for [DateHour, url] in hourUrlList:          key = DateHour + ':all\_unique\_urls'          redisClient.sadd(  key , url  )          uniqueUrlCount = redisClient.scard(  key  )          redisClient.set(  DateHour + ':unique\_url\_count', uniqueUrlCount  )      # Query2      # reduce data to list with hour unique URL, and unique visitor list      dfQueryTwo = df[['DateHour\_URL','user']].drop\_duplicates()      hourUrlUserList = dfQueryTwo.values.tolist()      # update redis with the new data, using similar structure and process as Query 1      for [DateHour\_URL, user] in hourUrlUserList:          key = DateHour\_URL + ':all\_unique\_users'          redisClient.sadd(  key , user  )          uniqueUserCount = redisClient.scard(  key  )          redisClient.set(  DateHour\_URL + ':unique\_user\_count' , uniqueUserCount  )      # Query3      # Reduce data to table with count of unique (by UUID) events/clicks per URL per hour per day      # <date:hour:url>,  Total Event\_count      dfQueryThree = df[['DateHour\_URL','UUID']].groupby('DateHour\_URL').count().reset\_index()      hourUrlUUIDList = dfQueryThree.values.tolist()      # update redis with the new data, using similar structure and process as Query 1      # do not update if the file has already been processed into redis      if redisClient.sismember(  'Query3:Files' , fileName  ) != 1 :          for [DateHour\_URL, UUIDcount] in hourUrlUUIDList:              redisClient.incrby(  DateHour\_URL + ':event\_count' , UUIDcount  )              redisClient.sadd(  'Query3:Files' , fileName  )      print('file processed: ' + fileName)      return  def main():      # get input arguments      parser = argparse.ArgumentParser(description='reads in one input file, processes it line by line, and updates some counters/data structures in Redis')      parser.add\_argument('file', type=str, help='name of file to process')      args = parser.parse\_args()      redisClient = redis.StrictRedis(host='redis\_server', port=6379, password='1234')      # process data from file      processData(args.file, redisClient)  if \_\_name\_\_ == '\_\_main\_\_':      main()  #hw3\_redis\_docker-compose.yml  version: '3'  services:    redis\_server:      image: redis      container\_name: redis\_server      command: redis-server --requirepass 1234      ports:      - "6379:6379"      redis\_client:      image: redis      container\_name: redis\_client      depends\_on:        - redis\_server  #hw3\_p1\_docker-compose.yml  #the hw3\_python docker image was made to run hw3\_problem1.py and process the input\_files  #four containers are created from the hw3\_python image, and each of them processes one input file  version: '3'  services:    python\_client1:      image: hw3\_python      container\_name: python\_client1      command: bash -c "python hw3\_problem1.py 'input\_files/file-input1.csv' "    python\_client2:      image: hw3\_python      container\_name: python\_client2      command: bash -c "python hw3\_problem1.py 'input\_files/file-input2.csv' "    python\_client3:      image: hw3\_python      container\_name: python\_client3      command: bash -c "python hw3\_problem1.py 'input\_files/file-input3.csv' "    python\_client4:      image: hw3\_python      container\_name: python\_client4      command: bash -c "python hw3\_problem1.py 'input\_files/file-input4.csv' " |

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

|  |
| --- |
| For Query1 and Query2, the redis SETS structure was used for the shared state management. Because it stores only unique values, we avoid the problem of adding duplicate urls or users to the redis server after each file is processed. We can then get the unique url counts and unique user counts by getting the number of elements in each of the SETS. For Query3, integers are sufficient because every UUID is unique so we can just count the number of UUID’s in each file and update the sum in redis after each file is processed. |

Show a screenshot of your Redis server running, and 4 instances of your application [5 points]

|  |
| --- |
| Docker Redis server    Docker Redis server and Python clients    Docker Python clients    Redis monitor |

Show results of your queries in the Redis CLI, for the specified keys [15 points]

|  |
| --- |
| Query 1:  <date\_hour>, <url\_count>  2019-09-12:12 186  2019-09-13:21 187  2019-09-13:17 185  2019-09-14:02 185  2019-09-13:03 189  Query 2  <date:hour:url>, unique\_user\_count  2019-09-14:14:http://example.com/?url=042, 3  2019-09-12:19:http://example.com/?url=013, 4  2019-09-14:03:http://example.com/?url=162, 3  2019-09-13:01:http://example.com/?url=035, 4  2019-09-14:10:<http://example.com/?url=043>, 1  Query 3  <date:hour:url>, event\_count  2019-09-14:14:http://example.com/?url=042, 3  2019-09-12:19:http://example.com/?url=013, 4  2019-09-14:03:http://example.com/?url=162, 3  2019-09-13:01:http://example.com/?url=035, 4  2019-09-14:10:<http://example.com/?url=043>, 1 |

**Problem 2: time range queries** [points: 30]

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

|  |
| --- |
| #hw3\_problem2.py  #Loi Cheng  #9/19/2019  import argparse  import redis  import pandas as pd  def processData(fileName,t1,t2,redisClient):      '''      reduces a set of data to useful lists, which will then be uploaded to a redis server      '''      #read file, add the DateTime      df = pd.read\_csv(fileName,header=None)      df['DateTime'] = pd.to\_datetime(df[1])        #trim file by t1 t2      df = df.set\_index(df['DateTime'])      df = df.loc[t1:t2]      #create the DateHour\_Country column      df = df.rename(columns={2: 'url',4:'country'})      df['DateHour'] = df['DateTime'].dt.date.astype('str') + ':' + df['DateTime'].dt.strftime('%H').astype('str')      df['DateHour\_Country'] = df['DateHour'].astype('str') + ',' + df['country'].astype('str')      # Query4      # reduce to list with hour,country and unique url      dfQueryFour = df[['DateHour\_Country','url']].drop\_duplicates()      hourCountryUrlList = dfQueryFour.values.tolist()        # update redis with the new data, with 'DateHour\_Country' as the key, and the corresponding set of unique urls as the value      # get the number of unique urls after the update and store it in another key      for [DateHour\_Country, url] in hourCountryUrlList:          key = DateHour\_Country + ':all\_unique\_urls'          redisClient.sadd(  key , url  )          uniqueUrlCount = redisClient.scard(  key  )          redisClient.set(  DateHour\_Country + ':unique\_url\_count', uniqueUrlCount  )      print('file processed: ' + fileName)      return  def main():      # get input arguments      parser = argparse.ArgumentParser(description='reads in one input file, processes it line by line, and updates some counters/data structures in Redis')      parser.add\_argument('file', type=str, help='name of file to process')      args = parser.parse\_args()      redisClient = redis.StrictRedis(host='redis\_server', port=6379, password='1234')      #specify time range      t1 = '2019-09-13 05:00:00'      t2 = '2019-09-14 09:00:00'      # process data from file      processData(args.file, t1, t2, redisClient)  if \_\_name\_\_ == '\_\_main\_\_':      main()  #hw3\_p2\_docker-compose.yml  #the hw3\_python docker image was made to process the input\_files  #four containers are created from the hw3\_python image, and each of them processes one input file  #this docker-compose file should be run after hw3\_redis\_docker-compose.yml is up  version: '3'  services:    python\_client1:      image: hw3\_python      container\_name: python\_client1      command: bash -c "python hw3\_problem2.py 'input\_files/file-input1.csv' "    python\_client2:      image: hw3\_python      container\_name: python\_client2      command: bash -c "python hw3\_problem2.py 'input\_files/file-input2.csv' "    python\_client3:      image: hw3\_python      container\_name: python\_client3      command: bash -c "python hw3\_problem2.py 'input\_files/file-input3.csv' "    python\_client4:      image: hw3\_python      container\_name: python\_client4      command: bash -c "python hw3\_problem2.py 'input\_files/file-input4.csv' " |

Explain your choice of Redis data structures [5 points]

|  |
| --- |
| Query4 is very similar to Query1 and Query2, so the redis SETS structure was again used for the shared state management. Because it stores only unique values, we avoid the problem of adding duplicate urls or users to the redis server after each file is processed. We can then get the unique url counts and unique user counts by getting the number of elements in each of the SETS. |

Show results of your query for the specified keys [10 points]

|  |
| --- |
| <date,hour,country>, url\_count  2019-09-14:02,BW,4  2019-09-13:05,KI, 6  2019-09-13:08,DJ, 2  2019-09-13:16,AS, 3  2019-09-14:00,VE, 4 |

**Problem 3: Bonus: Top N queries** [15 points]

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

|  |
| --- |
| #hw3\_problem3.py  #Loi Cheng  #9/19/2019  import argparse  import redis  import pandas as pd  def processData(fileName,redisClient):      '''      reduces a set of data to useful lists, which will then be uploaded to a redis server      '''      #read file, add the DateTime      df = pd.read\_csv(fileName,header=None)      df['DateTime'] = pd.to\_datetime(df[1])        #create the DateURL column      df = df.rename(columns={2:"url",8:"TTFB"})      df['Date'] = df['DateTime'].dt.date      df['DateURL'] = df['Date'].astype('str') + "::" + df['url'].astype('str')      # Query5      # reduce to list DateURL and Sum TTFB and Count TTFB      dfDateUrlTTFB = df[['DateURL','TTFB']]      dfDateUrlTTFBSum = dfDateUrlTTFB.groupby('DateURL').sum().reset\_index()      dfDateUrlTTFBSum = dfDateUrlTTFBSum.rename(columns={'TTFB':'TTFBSum'})      dfDateUrlTTFBCount = dfDateUrlTTFB.groupby('DateURL').count().reset\_index()      dfDateUrlTTFBCount = dfDateUrlTTFBCount.rename(columns={'TTFB':'TTFBCount'})      dfQueryFive = dfDateUrlTTFBSum.set\_index('DateURL').join(dfDateUrlTTFBCount.set\_index('DateURL')).reset\_index()      dateUrlTTFBList = dfQueryFive.values.tolist()        # update redis with the new data      # update the TTFBSum and TTFBCount for each DateURL      # compute the new average TTFB and update it in the sorted set      for [DateURL, TTFBSum, TTFBCount] in dateUrlTTFBList:          key = DateURL + ':TTFBSum:TTFBCount'          redisClient.hincrbyfloat(  key , 'TTFBSum' , TTFBSum  )          redisClient.hincrby(  key , 'TTFBCount' , TTFBCount  )          averageTTFB = float( redisClient.hmget(key , 'TTFBSum')[0] ) / float( redisClient.hmget(key , 'TTFBCount')[0] )          DateURLSplit = DateURL.split('::')          Date = DateURLSplit[0]          URL = DateURLSplit[1]          redisClient.zadd( Date +":Average\_TTFB" , {URL:averageTTFB} )      print('file processed: ' + fileName)      return  def main():      # get input arguments      parser = argparse.ArgumentParser(description='reads in one input file, processes it line by line, and updates some counters/data structures in Redis')      parser.add\_argument('file', type=str, help='name of file to process')      args = parser.parse\_args()      redisClient = redis.StrictRedis(host='redis\_server', port=6379, password='1234')      # process data from file      processData(args.file, redisClient)  if \_\_name\_\_ == '\_\_main\_\_':      main()  #hw3\_p3\_docker-compose.yml  #the hw3\_python docker image was made to process the input\_files  #four containers are created from the hw3\_python image, and each of them processes one input file  #this docker-compose file should be run after hw3\_redis\_docker-compose.yml is up  version: '3'  services:    python\_client1:      image: hw3\_python      container\_name: python\_client1      command: bash -c "python hw3\_problem3.py 'input\_files/file-input1.csv' "    python\_client2:      image: hw3\_python      container\_name: python\_client2      command: bash -c "python hw3\_problem3.py 'input\_files/file-input2.csv' "    python\_client3:      image: hw3\_python      container\_name: python\_client3      command: bash -c "python hw3\_problem3.py 'input\_files/file-input3.csv' "    python\_client4:      image: hw3\_python      container\_name: python\_client4      command: bash -c "python hw3\_problem3.py 'input\_files/file-input4.csv' " |

What are the main differences with the Problem 1 and 2 implementation? [5 points]

|  |
| --- |
| The main difference is the addition of using a sorted set to store the average TTFB for each url. The sorted set allows speedy retrieval of the top fastest sites for the given days. |

Show results of your query [5 points]

|  |
| --- |
|  |

**Problem 4: [15 points] AWS EMR cluster and HDFS**

Include screenshot of the created EMR cluster - with all cluster information; clearly identify master and slave nodes - either from the EC2 instances views or from the cluster view [5 points]

|  |
| --- |
| Cluster Information    Slave Node    Slave Node    Master node    Inbound rules for master node  My local machine’s IP address (68.9.133.147) was added |

List commands you used to ssh to your master node and view HDFS version; include screenshots of the hdfs version output [5 points]

|  |
| --- |
| SSH to master node with command  ssh -i BigDataHarvard.pem hadoop@ec2-18-222-226-109.us-east-2.compute.amazonaws.com  Welcome Screen    HDFS version |

List commands you used to demo basic (create, get/view, delete file) HDFS operations - include a screenshot of the terminal where you were running the commands, with their output [5 points]

|  |
| --- |
| List root directory    List apps folder    Create new file “bigbigfile.csv”    Get file “bigbigfile.csv”, edit it locally and put it back on HDFS  Make file “anotherfile.txt” and put it on HDFS    List root directory    View contents of “bigbigfile.csv” in HDFS    Delete “anotherfile.txt” from HDFS    Terminating EMR Cluster |

**Problem 5: [15 points]**

explain the difference between idempotent and non-idempotent operations [5 points]

|  |
| --- |
| An idempotent operation does not make any new changes to the results if it is called more than once. Conversely, a non-idempotent operation does change the results if it is called more than once. |

give an example of a non-idempotent operation [5 points]

|  |
| --- |
| Given a variable count = 0, an operation to increment count by 1 is a non-idempotent operation. |

give an example of an idempotent operation [5 points]

|  |
| --- |
| Given a set with items [1,2,4,5], where each item must be unique. An operation to add 6 to the set would result in the set being updated to [1,2,4,5,6]. The set would still be [1,2,4,5,6] if this operation were to be repeated. So this operation is idempotent |