**Harvard Extension School Fall, 2017**

**CSCI E-88: Principles of Big Data Processing**

**Marina Yu Popova**  
The goal of this course is to learn core principles of building highly distributed, highly available systems for processing large volumes of data with historical and near real-time querying capabilities. We cover the stages of data processing that are common to most real-world systems, including high-volume, high-speed data ingestion, historical and real-time metrics aggregation, unique counts, data de-duplication and reprocessing, storage options for different operations, and principles of distributed data indexing and search. We review approaches to solving common challenges of such systems and implement some of them. The focus of this course is on understanding the challenges and core principles of big data processing, not on specific frameworks or technologies used for implementation. We review a few notable technologies for each area with a deeper dive into a few select ones. The course is structured as a progression of topics covering the full, end-to-end data processing pipeline typical in real-world scenarios.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Assignment1**   * Due Sep 7 by 11:59pm * Points 100 * Submitting a file upload * Available Aug 31 at 12am - Sep 13 at 11:59pm 14 days   This assignment was locked Sep 13 at 11:59pm.  **Assignment 1** [1 week]  Due: Sept 7, Thu, midnight EST  **Goals:**   * AWS account is fully functional and you are able to create/access EC2 instances * LAMP stack setup is functional * submission requirements are clear   **Pre-requisites:** create AWS account or setup VMWare on your local laptop  **Submission requirements** - common to all assignments   * Solution Document - a Word or PDF file, named <LastName>\_<FirstName>\_HW1.docx[pdf] (for example: Popova\_Marina\_HW1.docx)  that contains:   + answers to posted questions (if any)   + explanation of choices/approaches you made in your solution   + all execution steps and commands with screenshots of running the commands and screenshots of the results   + snippets of the most important code [if applicable] to demonstrate your solution   + full results [or a subset of results if more than 50 lines] * Full source code (no compiled classes, or generated by IDEs artifacts like .project etc.) - submitted as either separate files (say, hw1\_problem1.py or Problem1.java) or archived into a zip/tar archive; archive should be named: <LastName>\_<FirstName>\_HW<number>.zip[tar] * Result files (if any) - can be placed into the same archive with the source code       **Problem 1:**  -- setup your AWS account if you have not done so already, create an EC2 instance , CentOS-based one (latest version is fine)  -- OR install VMWare and make sure you can create and start a CentOS-based VM  -- implement Problem 1 from HW0 if you have not already done that  -- deploy your Problem 1 solution on an EC2 instance or VMWare VM  -- run your Problem 1 and demonstrate the results    **Problem 2:**  -- create an EC2 instance with LAMP stack OR  -- create a CentOS-based VM and install Apache and MySQL DB there  -- demonstrate that you can run Apache web server and hit a welcome page  -- demonstrate that you can see the corresponding requests logged in the log file  -- demonstrate that you can start your DB  -- demonstrate that you can create a table, insert some data, query the data - using either command-line shell or GUI of your choice (PHPMyAdmin is a nice tool)    **Grading Rubrik**  Total points: 100       |  |  |  | | --- | --- | --- | | Problem | Main requirements |  | |  | All submission requirements are fulfilled (solution document with all required information is provided and correctly named, results and code is included) |  | | Problem 1 |  |  | |  | Correct EC2 instance (or VM) is created and described |  | |  | HW0 Problem 1 is correctly implemented |  | |  | Execution of the HW0 P1 on the EC2 instance , as well as correct results (output file) are demonstrated |  | | Problem 2 |  |  | |  | Correct EC2 instance (or VM) is created; installation of the required stack (LAMP) is demonstrated - all steps are described |  | |  | Apache server is started; access logs with the logs of test requests are demonstrated |  | |  | MySQL DB is installed and started |  | |  | CRUD operations (create, read, update, delete) are demonstrated | 10 | |
| **Assignment2**   * Due Sep 22 by 12:01pm * Points 200 * Submitting a file upload * Available Sep 5 at 12am - Sep 25 at 12:01pm 21 days   This assignment was locked Sep 25 at 12:01pm.  Assignment 2 [ 2 weeks]  Submission requirements - common to all assignments   * Solution Document - a Word or PDF file, named <LastName>\_<FirstName>\_HW1.docx[pdf] (for example: Popova\_Marina\_HW1.docx)  that contains:   + answers to posted questions (if any)   + explanation of choices/approaches you made in your solution   + all execution steps and commands with screenshots of running the commands and screenshots of the results   + snippets of the most important code [if applicable] to demonstrate your solution   + full results [or a subset of results if more than 50 lines] * Full source code (no compiled classes, or generated by IDEs artifacts like .project etc.) - submitted as either separate files (say, hw1\_problem1.py or Problem1.java) or archived into a zip/tar archive; archive should be named: <LastName>\_<FirstName>\_HW<number>.zip[tar] * Result files (if any) - can be placed into the same archive with the source code     [NOTE: you may use example implementations of HW0 Problem1 which will be made available on Sat, 09/09, as the basis for your multi-threaded apps. We've added multi-threading to the example implementations as well, to make it easier to get started for those who have not used threads extensively. Example implementations will be in Java, Python and Scala. Python and Scala implementations will be demonstrated and explained in the Labs on Sat, 09/09, 9AM EST, and on Mon, 09/11, 8PM EST; Java implementation will be discussed in the Lab on 09/16]    Problem 1a: [points: 40]  write a simple program in your choice of language that does:   * creates specified number of threads - based on the input parameter * starts each of those threads * each thread does:   + sleep for 10-15 ms   + do some CPU-intensive work - like parsing a RegExp [your choice]   + do above 2 things forever * start your app and let it run * using Unix tools like ‘top’, ‘ps’ , 'htop' and Java specific ones like ‘jps’ - try to figure out how your threads are mapped to available CPUs - include the results into your solution document (screenshots and explanation) * change the input parameter (number of threads to start) - see how it affects the CPU usage. Include your results into your solution document  (screenshots and explanation) * create a graph of # of threads vs. CPU utilization [bonus: +5 points]     Problem 1b: [bonus: +10]   * Write a slightly modified version of the Problem1 program: * Instead of CPU-intensive work, let the threads do some I/O intensive work - like scanning a directory * Using the same [or other] tools - explore how adding more threads affects I/O utilization of your server * Include your results into the solution document     Problem 2a: [points: 20]   * Create two EC2 instances in AWS , or two VMs, with different hardware profiles: * 1st VM: small number of CPUs (4) * 2nd VM: high number of CPUs (8) * Run the program from Problem 1a on both VMs, with the same variations of number of threads as in Problem 1a; monitor CPU and I/O utilization * Report the results (screenshots, your conclusions and , for a bonus 5 points - graphs)     Problem 2b: [bonus: +5]   * Run the program from Problem 1b on both VMS; monitor CPU and I/O utilization * Try to run both programs at the same time - monitor CPU and I/O * Report the results (screenshots, your conclusions and , for a bonus 5 points - graphs)     Problem 3: [points: 30]  write a multi-threaded program to generate "log" files as following:   * specify application parameters: number of distinct userIds, list of URLs , number of "events" per URL per user; number of threads in the app * number of log lines should be: [number of distinct userIds] X [number of URLs] X [number of "events" per URL per user] * create specified number of threads - "writers" * each writer thread should generate one output file with logs in the format (one line == one event)   + <timestamp> <url> <userId> * name of the file should be:   + <threadID>\_events.txt     Problem 4: [points: 60]   * implement Example from Lecture 2 (getting unique click and visitor counts): one app multiple threads * decide how to provide separate sets of input files for each thread   *Lecture 2, Example re-cap: getting unique click and visitor counts*  *--Input: files with events (one line is one event == one user click on a URL) - generated in Problem 3*  *-- write a program , with multiple threads; each thread reads in and processes one input file*  *-- your program has to keep the shared state in memory*  *-- after processing all data files, the program should  output results to the following queries:*  *Query 1: get count of unique URLs*  *Query 2: get count of unique visitors per URL*  *Query 3: get count of unique (by userId) clicks per URL*    Problem 5: [bonus: +30]   * modify implementation of Problem 4 for the use case of **multiple apps on one machine** * decide how to provide separate sets of input files for each thread and per each app * use shared DB for state. Use the MySQL server setup in HW1 for the DB * decide how to control completion of "mappers" and start/execution of the "reducer"     Problem 6: [bonus: +15] - demonstrate contention on the same raw in the DB   * generate a new set of input files with the same URL in the data set * find a way to time your app performance * run multiple instances of your app on the same (or different ) server * demo how performance of your app degrades as you keep adding more instances - due to locking on the same raw/URL in your DB     Problem 7: [points: 30]   * setup a 3[or 5]-node AWS EMR cluster (alternatively, you could build a 3[5]-node cluster on VMWare VMs - using Cloudera QuickStart VMs or installing Hadoop manually, but it is much harder to do) * identify your master and slave nodes - include screenshots that show that * ssh to your master node and verify version of HDFS by issuing a command like: "hdfs -version" * demonstrate basic file operations in HDFS (create, get, delete a file) - using HDFS commands documentation for reference     Grading Rubrik  Total points: 200  Required (non-bonus) problems are highlighted   |  |  |  | | --- | --- | --- | | **Problem** | **Main requirements** | **Points** | |  | All submission requirements are fulfilled (solution document with all required information is provided and correctly named, results and code is included) | 20 | | **Problem 1a** |  | **40** | |  | Required app is created, number of threads can be specified as parameter and the configured number of threads is started/working, doing CPU-intensive operations | 20 | |  | Mapping of threads to cores is demonstrated, with different number of running threads | 20 | |  | graph of # of threads vs. CPU utilization | bonus: +5 | | Problem 1b | Same as Problem1a, but with threads doing  CPU-intensive operations | bonus: +10 | | **Problem 2a** |  | **20** | |  | 2 EC2s with 4 and 8 cores created, results of running Problem1a on both machines are demonstrated, with variations in thread numbers |  | |  | graph of # of threads vs. CPU utilization | bonus: +5 | | Problem 2b | Same as Problem 2a, but running Problem1b app | bonus: +5 | |  | graph of # of threads vs. CPU and I/O utilization | bonus: +5 | | **Problem 3** |  | **30** | |  | Application is correctly implemented, with required parameters; each thread is creating one output log file | 20 | |  | Correctly named result files , with specified content, are generated and demonstrated | 10 | | **Problem 4** |  | **60** | |  | Correct in-memory data structures are created and explained | 10 | |  | separate sets of input files are provided for and processed by each thread | 5 | |  | Each lines is parsed and correct fields are extracted and saved into the in-memory data structures | 5 | |  | Handling of parsing errors is defined | bonus: +5 | |  | Correct strategy for concurrent shared data access is implemented and explained | 10 | |  | Aggregation functions for answering the three questions are implemented correctly and results are demonstrated | 20 | | Problem 5 |  | bonus: +30 | |  | Strategy for providing input to all apps/threads is explained and implemented (how is it changed from Problem 1 ?) | 5 | |  | Correct DB tables are defined; access to DB is implemented | 10 | |  | strategy for controlling completion of mappers from multiple apps and separate reducer  app is defined and implemented | 10 | |  | Aggregation results are demonstrated for all three queries | 5 | | Problem 6 |  | bonus: + 15 | |  | demonstrate the impact by running more and more instances for the same URL/raw in the table | 15 | | **Problem 7** |  | **30** | |  | AWS EMR cluster is created, master and slave nodes are identified | 10 | |  | remote login to the master node is successful,  HDFS version is verified | 10 | |  | HDFS basic file commands demonstrated | 10 | |
| **Assignment3**   * Due Oct 1 by 11:59pm * Points 100 * Submitting a file upload * Available Sep 21 at 9am - Oct 4 at 11:59pm 14 days   This assignment was locked Oct 4 at 11:59pm.  Assignment 3 - 1 week  **Submission Requirements** - please review them carefully as we will now be quite strict about deducting points if not followed. If you’re unsure of how to format your solutions doc, please check the samples or the showcase solutions we have posted.  **Solution Document** - a Word or PDF file, titled <LastName>\_<FirstName>\_HW3.docx[pdf] (for example: Popova\_Marina\_HW3.docx)  that contains:  - answers to posted questions (if any)  - explanation of choices/approaches you made in your solution  - all execution steps and commands with screenshots of running the commands and screenshots of the results. Remember, you need to prove that you did the work you were asked to do.  - snippets of the most important code [if applicable] to demonstrate your solution  - full output results [or a subset of results if more than 50 lines]. You may submit a screenshot from a console or window or the raw text output, but make sure it is included in your solutions document.  - Full source code with no compiled classes, IDE generated artifacts like .project etc.  Submit either as separate files (say, hw1\_problem1.py or Problem1.java) or archived into a zip/tar archive. Archives should be named: <LastName>\_<FirstName>\_HW<number>.zip[tar]  Result files (if any) - can be placed into the same archive with the source code    **Total points: 100**  **Problem 1: (Points: 30)**  Write an application that calculates answers to the specified below queries using the Hadoop MapReduce APIs:   * As input, use a few (2-5) files; each file has multiple events (one line is one event == one user click on a URL); you can generate these files using your app developed in HW2 Problem3, or you can create them manually; input files should be located in HDFS * create a separate MR job for each Query:   + Query 1: get count of unique URLs   + Query 2: get count of unique visitors per URL   + Query 3: get count of unique (by userId) clicks per URL * run your jobs on an EMR cluster     **Problem 2: (Points: 30)**   1. Modify data generation script to generate events with timestamps that span 3-4 hours in time; place your generated data files into HDFS 2. write a new set of jobs, that process these generated files and answer the same 3 queries, but with added "by hour" aggregation. Modified queries will look as following:   Query 1: get count of unique URLs by hour  Query 2: get count of unique visitors per URL by hour  Query 3: get count of unique (by userId) clicks per URL by hour        3. run your jobs on an EMR cluster    **Problem 3:** You may choose which version you would prefer to work on:  Problem 3A: (Points: 40)   * define an Avro schema for your data; explain your schema design * write a MR job that reads your generated in Problem 2.1 input files (as text) and writes them out in Avro format * run the job on your EMR cluster - demonstrate the results     OR  Problem 3B (alternative easier option) (Points: 20)   * define an Avro schema for your data; explain your schema design * write an app (non-MapReduce) that reads your generated in Problem 2.1 input files (as text) and writes them out in Avro format, on your local server (or EC2 instance if you like)     Below are Bonus Problems ! Work on these at your own risk !  Problem 4: (Bonus: +25)   * based on your Problem 2's job, write a new job, called Problem4Avro, that uses the Avro file[s] (generated in Problem 3) as the input and calculates answers to the same 3 queries * execute the job on your EMR cluster     Problem 5: (Bonus: +15)   * write an app (either MR or standalone) that reads data in your Avro format (output of your Problem 3's job) and writes its out in the Parquet format       Grading Rubrik  Total Points: 100     |  |  |  | | --- | --- | --- | | **Problem** | **Main requirements** | **Points** | | **Problem 1** |  | **30** | |  | MR jobs for each of 3 queries are created correctly | 20 | |  | Jobs implementation is explained in the solution document, execution and output results are demonstrated | 10 | |  | Jobs execution on EMR cluster is demonstrated , along with the results | 10 | | **Problem 2** |  | **30** | |  | MR jobs for each of 3 queries are created correctly | 20 | |  | Jobs implementation is explained in the solution document, execution and output results are demonstrated | 10 | |  | Jobs execution on EMR cluster is demonstrated , along with the results | 10 | | **Problem 3A** |  | **40** | |  | Avro schema is designed and explained | 10 | |  | MR job (text → Avro format) is correctly implemented and explained; execution and results are demonstrated | 20 | |  | Job execution on EMR cluster is demonstrated , along with the results | 10 | | **OR Problem 3B** |  | **20** | |  | Avro schema is designed and explained | 10 | |  | application (text → Avro format) is correctly implemented and explained; execution and results are demonstrated | 10 | | Problem 4 |  | bonus: +25 | |  | MR jobs, that take Avro input, for each of 3 queries are created correctly and explained; execution and output results are demonstrated | 15 | |  | Jobs execution on EMR cluster is demonstrated , along with the results | 10 | | Problem 5 |  | bonus: + 15 | |  | Application that reads Avro and outputs Parquet data is implemented correctly; execution and results are demonstrated | 15 | |
| **Assignment4**   * Due Oct 15 by 11:59pm * Points 200 * Submitting a file upload * Available Sep 28 at 9am - Oct 19 at 12:15am 21 days   This assignment was locked Oct 19 at 12:15am.  Assignment 4 - 2 weeks  **Submission Requirements** - please review them carefully as we will now be quite strict about deducting points if not followed. If you’re unsure of how to format your solutions doc, please check the samples or the showcase solutions we have posted.  **Solution Document** - a Word or PDF file, titled <LastName>\_<FirstName>\_HW4.docx[pdf] (for example: Popova\_Marina\_HW4.docx)  that contains:  - answers to posted questions (if any)  - explanation of choices/approaches you made in your solution  - all execution steps and commands with screenshots of running the commands and screenshots of the results. Remember, you need to prove that you did the work you were asked to do.  - snippets of the most important code [if applicable] to demonstrate your solution  - full output results [or a subset of results if more than 50 lines]. You may submit a screenshot from a console or window or the raw text output, but make sure it is included in your solutions document.  - Full source code with no compiled classes, IDE generated artifacts like .project etc.  Submit either as separate files (say, hw1\_problem1.py or Problem1.java) or archived into a zip/tar archive. Archives should be named: <LastName>\_<FirstName>\_HW<number>.zip[tar]  Result files (if any) - can be placed into the same archive with the source code    **Total points: 200**  **Part 1: Flume (100 points)**    **Problem 1: (Points: 20) simplest Flume setup**   * Setup Flume on a server (can be any server: EC2 instance, local laptop, VMWere VM) * Configure "Spooling Directory" Source - to read data from some specified directory * Configure a "File Roll" sink (local directory sink) for some output directory on your local server * Use memory channel * Create 5 files with 10 lines of text each (anything goes) - named file1.txt, file2.txt,.... File5.txt * Place them into the spooling source directory * Observe them copied to the destination directory - demo the results * Try to copy the same files again into the spooling source directory - explain and demo the results     **Problem 2: (Points: 20)** setup web server and a client to issue requests to the server   * Install Tomcat or Nginx web server on your server (can be local server, EC2 instance or VMWare CentOS VMs) - you can use port 80 or any other port you want to use * Create a simple static html page and make it index.html (home page of the web server) * Test your page by issuing 'curl' requests to the index.html page; something like "curl <http://localhost:80/index.html>" * Find and examine logs generated by the server * Write a script on an app that would issue 'curl' requests to your web server constantly, with a  configurable rate (number of requests to send per second); you script/app should run "forever" when started, until you stop it * Make sure you see all generated requests being logged in the web server log     **Problem 3**: experiment with the Memory Channel   1. Install and configure Flume on the same server (where your web server is running) 2. Configure a "tail -F"  Exec source for web server logs 3. Configure a "File Roll" sink (local directory sink) for some output directory on your local server 4. Configure a memory channel with the capacity of events 5. Start your script - to start generating log entries in the web server log 6. Start your Flume agent - observe creation of local files with the events from the web server log 7. Simulate Flume Sink becoming unavailable (for example, set permissions on your sink directory to 444) (do NOT stop the event generation app) 8. Wait for 5 min - make the Sink available again 9. Stop data generation app 10. Once all log data is processed by Flume (monitor the destination directory) - stop Flume 11. Compare destination logs with the source web logs - did you loose any data? Report the results 12. Bonus: [+5] experiment with the event generation rates and memory channel configuration - how many events would you need to generate per second to see your memory channel filling up ? what happens then?     **Problem 4**: experiment with the File Channel   * Do the same exercise as in Problem 3 - but use the File Channel this time * Configure checkpoint directory , and the 'dataDirs' File Channel log directory * When Flume sink is down (step 7) - observe behavior of the 'dataDir' and 'checkpoint' dirs content - what do you see there? * make your Flume sink available again, and compare destination and source data again. Do you see any data loss now? * Bonus: [+5] Experiment with the event generation rates - do you see any impact of file channel on performance?     **Problem 5:** (Points: 20) HDFS sink   * Create an AWS EMR cluster with Hadoop installed * replicate the Web server +Flume setup on one of the EC2 instances from the cluster (does not matter if this is a master or slave node) -  install web server + Flume manually on that instance (you can also do it on a separate EC2 instance if you like - but beware of connectivity setup) * Configure a "tail -F"  Exec source for web server logs * Configure HDFS sink - define some time-based dir/file name structure e.g,  flume-out/problem5/events/%y-%m-%d/ * Start your event generation app (either from your local server , hitting your EC2 instance - have to open ports), or run it locally on the same EC2 server * Observe correct directories/files being created in HDFS     Problem 6: Bonus (+20)   * Do the same setup as in Problem 5, but this time - setup Web server and Flume on a separate EC2 instance, not on one of the EMR nodes * Configure your Flume agent to write data into the EMR cluster - make sure the setup works * Use file channel for the HDFS sink * Start pushing events - observe them being written into HDFS * Simulate your HDFS sink failure - make your HDFS cluster unavailable - either close the ports for your EMR cluster or stop hadoop processes (hdfs namenodes and data nodes) on that cluster (be creative :) ) * Observer what happens with the Flume agent * Make HDFS available again * Observe what happens with the Flume agent - do you have to restart it? Did you loose any of the data?     Problem 7: Bonus (+20)   * Use AWS EMR cluster again * Setup one Flume agent with the Spooling Directory Source and an Avro sink * Create a file with a few events in Avro format (you can use the classic 'users.avsc' schema from the Avro docs and just create a few records of data in that format) * Setup another Flume agent with Avro source and an HDFS sink * Wire the pipeline to read the original Avro data and transport them into HDFS     **Part 2: Kafka: 100 points**  **Problem 8: (Points: 20) Basic Kafka**   * Install a simple Kafka cluster - it can be one node cluster , on your local laptop * Create a topic "problem8" with 2 partitions * Describe the cluster and show its state via kafka-topics --describe command * Verify that you can send message and receive messages to/from the topic by running Kafka console producer and consumer from two different terminals * Examine kafka commit logs, by using the kafka.tools.DumpLogSegments tool , for each partition of the "problem8" topic     **Problem 9: (Points: 20) Kafka + Flume integration**   * Use your local Web server + Flume setup, where Flume is getting data from your web server logs via "tail -f" source * Create a new topic, "problem9" , with 2 partitions * Add Kafka Sink to your Flume - to start pushing events into the "problem9" topic * Start your app to generate CURL requests to the web app * Start a Kafka console consumer for "problem9" topic - verify you are getting your web server logs * Add FLume's UUID Interceptor - to add UUID to each event * Verify you are getting events with UUIDs now in Kafka (in your console consumer)     **Problem 10: (Points: 25) Basic Kafka producers and consumers**   * Create a new topic, "problem10" , with 3 partitions * Using Kafka API, create a new Problem10Producer application that generates and sends events to the "problem10" topic - generate the same type of log events we've worked with so far, plus add a UUID: one event is one line in the format: <uuid> <timestamp> <url> <userId>; make sure you have at least 3 different userIDs   + note: you can use your existing event generation app - just convert it into a Kafka producer * Verify your producer works correctly by receiving generated events via Kafka console consumer * Using Kafka APIs, create a Problem10Consumer (single-threaded is OK), with the "p10consumer" consumer group.id,  that listens to the "problem10" topic - demonstrate that it recieves events generated by the Problem10Producer   + For each event received the Problem10Consumer should print out partition number, event key and event body   + Demonstrate that the Problem10Consumer receives events from all 3 partitions and they are equally distributed * Modify Problem10Producer to send events with the key partitioned based on the userIDs:   + Events with the same userID should go to the same partition number   + Make sure that there is at least one userID corresponding to each partition (you have 3 partitions, remember) * In your Problem10Consumer, verify that the events for the same userID are received from the same partition     **Problem 11: (Points: 15) consumer groups**   * Building up on the Problem 10's setup: start a second instance of the Problem10Consumer, with the same consumer group ID * Demonstrate how partitions are re-balanced between the two consumer instances - which partitions are processed by which instance now? * Start two more instances of the Problem10Consumer with the same group ID - to the total of 4 instances now. Demonstrate the partition re-balancing and explain the results. Which instances are processing which partitions, if any? * Stop two instances of the Problem10Consumer - demonstrate the partition re-balancing again     **Problem 12: (Points: 20) multi-node cluster**   * Setup a 3-node Kafka cluster - you can do it either in AWS or using Docker   + make sure each node has a distinct broker.id, has privateIPs to talk to Zookeeper, but uses publicIPs as advertised hosts in the server.properties for each node; myid file for zookeeper should have the same id as the broker.id * Create a topic, "problem12", with 4 partitions and replication factor 2 * Examine the state of the cluster and show how all replicas are spread over the cluster nodes - show all ISRs * Using console producer, send a few events to the "problem12" topic; verify you have received them using the console consumer * Stop Kafka on one of the nodes (do not stop Zookeeper), wait for a few seconds - show the state of the cluster now, and all ISRs of the "problem12" topic * inspect Kafka's application logs - what kind of errors/messages do you see? * Run console producer and consumer again, see if you can still send/receive messages * re-start Kafka on that node - observe how fast the cluster gets into the full ISR state     Problem 13: (Bonus: +20)   * Convert your Problem10Consumer into a multi-threaded Problem11Consumer - where you start multiple Consumer threads per specified consumer group. * Do not use offset autocommit anymore (if you were using it before) - make sure each cnsumer thread commits its offset via commitAsync() operation * Make the number of consumer threads configurable via your Problem11Consumer input parameter * Create a new topic, "problem13", with 10 partitions * Start one Problem11Consumer instance with 5 consumer threads, joining consumer group "p13consumers" * Keep pushing events into the "problem13" topic * Demo partition distribution among all consumer threads * Run kafka-consumer-group tool and show offset movement for the consumer group * Start one more instance of the Problem11Consumer, also with 5 threads, joining the same consumer group "p13consumers" * Demo partition re-balancing between all consumers     Grading Rubrik  Total Points: 200     |  |  |  | | --- | --- | --- | | **Problem** | **Main requirements** | **Points** | | **Problem 1** |  | **20** | |  | correct source, sink and channel are created; flow is working and is demonstrated by showing Flume logs, execution and data appearing in the correct destination | 15 | |  | effect of using non-unique file names wit spooling directory source is demonstrated | 5 | | **Problem 2** |  | **20** | |  | web server setup is correct, described and logs are demonstrated | 10 | |  | request generator script (CURL commands to the web server) is implemented and demonstrated | 10 | | **Problem 3** |  | **20** | |  | correct source, sink and **memory channel** are created; flow is working and is demonstrated by showing Flume logs, execution and data appearing in the correct destination | 10 | |  | data loss due to memory channel usage is demonstrated | 10 | |  | experiments with the event generation rates and memory channel configuration are demonstrated and results are explained | bonus: +5 | | **Problem 4** |  | **20** | |  | correct source, sink and **file channel** are created; flow is working and is demonstrated by showing Flume logs, execution and data appearing in the correct destination | 10 | |  | data durability due to memory channel usage is demonstrated | 10 | |  | experiments with the event generation rates and file channel configuration are demonstrated and results are explained | bonus: +5 | | **Problem 5** |  | **20** | |  | web server + EMR cluster + HDFS + request generation script setup is completed , verified to be working and demonstrated in the solution doc | 10 | |  | HDFS sink is correctly configured with the specified directory names pattern and corresponding data storage is demonstrated | 10 | | Problem 6 |  | bonus: +20 | |  | web server + Flume on separate [EC](mailto:EC@,)2,  EMR cluster + HDFS setup is completed and demonstrated | 10 | |  | sink durability in the event of HDFS failure is demonstrated | 10 | | Problem 7 |  | bonus: + 20 | |  | spool dir -> Avro -> HDFS  pipeline is implemented and demonstrated | 20 | | **Problem 8** |  | **20** | |  | Kafka cluster is setup, all steps are demonstrated, topic is created and cluster state is shown | 10 | |  | console producer and consumer are working | 5 | |  | logs are inspected with the DumpLogSegments tool | 5 | | **Problem 9** |  | **20** | |  | Flume's Kafka source is setup and log sending to Kafka is demonstrated | 15 | |  | Flume UUID Interceptor is setup and events with UUIDs are received in Kafka | 5 | | **Problem 10** |  | **25** | |  | Problem10Producer is correctly implemented and its work is demonstrated | 10 | |  | Problem10Consumer is correctly implemented and its work is demonstrated | 10 | |  | partitioning by userID is correctly implemented in the Problem10Producer and demonstrated | 5 | | **Problem 11** |  | **15** | |  | partition re-balancing for a topic with 3 partitions is demonstrated with a different number of consumers in the same consumer group (2 and 4) | 15 | | **Problem 12** |  | **20** | |  | 3-node Kafka cluster is correctly setup and its work is demonstrated via console producer and consumer | 15 | |  | one node failure is demonstrated; ISRs are shown before, during and after the failure; time the cluster took to recover is noted | 5 | | Problem 13 |  | Bonus: +20 | |  | multi-threaded Consumer , with commitAsync(), is correctly implemented, explained and its work is demonstrated | 15 | |  | partition re-balancing is demonstrated when adding more consumer instances/threads | 5 | |
| **Assignment5**   * Due Oct 29 by 11:59pm * Points 200 * Submitting a file upload * Available Sep 28 at 9am - Nov 1 at 11:59pm about 1 month   This assignment was locked Nov 1 at 11:59pm.  **Assignment 5 - 2 weeks**  **Spark Batch processing + Batch Views**  **Submission Requirements** - please review them carefully as we will now be quite strict about deducting points if not followed. If you’re unsure of how to format your solutions doc, please check the samples or the showcase solutions we have posted.  **Solution Document** - a Word or PDF file, titled <LastName>\_<FirstName>\_HW4.docx[pdf] (for example: Popova\_Marina\_HW5.docx)  that contains:  - answers to posted questions (if any)  - explanation of choices/approaches you made in your solution  - all execution steps and commands with screenshots of running the commands and screenshots of the results. Remember, you need to prove that you did the work you were asked to do.  - snippets of the most important code [if applicable] to demonstrate your solution  - full output results [or a subset of results if more than 50 lines]. You may submit a screenshot from a console or window or the raw text output, but make sure it is included in your solutions document.  - Full source code with no compiled classes, IDE generated artifacts like .project etc.  Submit either as separate files (say, hw1\_problem1.py or Problem1.java) or archived into a zip/tar archive. Archives should be named: <LastName>\_<FirstName>\_HW<number>.zip[tar]  Result files (if any) - can be placed into the same archive with the source code    **Total points: 200**  Part 1: Spark Batch processing  - 120 points  **Problem 1A: (Points: 30) Local Spark setup**   1. Input data format: <timestamp> <URL> <userID> 2. generate 2-5 files, with a few (10-20) events per user per URL per hour in each file, in the specified  format, with timestamps that span 5-20 days in time 3. write 3 Spark jobs, that process these generated files and answer the "hourly" aggregates queries:   Query 1: get count of unique URLs by hour  Query 2: get count of unique visitors per URL by hour  Query 3: get count of unique (by userId) clicks per URL by hour  4. compare DAGs, jobs and RDDs of the three jobs, explain the differences  The Jobs need to use the RDD Interface only .  Please **do not** use dataframe or Dataset API’s .  You can run your jobs  locally (from Eclipse/Notebooks ) or via spark-shell .  **Problem 1B - Spark on EMR setup**   1. Place your input data into HDFS 2. Modify your jobs to use data from HDFS 3. Run your jobs on an EMR cluster with Spark installed 4. compare DAGs, jobs and RDDs of the jobs run locally vs on AWS cluster, explain the differences     **Problem 2: (Points: 30)**   1. Modify your generated data to also contain UUID for each event: <uuid> <timestamp> <URL> <userID>; generate 2-5 files, with a few (10-20) events per user per URL per hour in each file, in the specified  format, with timestamps that span 5-20 days in time 2. place your input data into a S3 bucket and create a folder ; lets call it Dataset1 3. Make sure you have at least 20 unique userIDs 4. Duplicate your input files - create a copy of each file, with a different name. We are trying to simulate having "duplicate" events, with the same UUID. In our case it is an extreme - each event will have a duplicate event with the same UUID; Let call it Dataset 2. Place  these files also in S3 in the same bucket but under a different folder. 5. Modify your Spark jobs to de-duplicate the data and execute the same queries 6. Run your jobs on Dataset1 first 7. Run your jobs on both Dataset1 and Dataset2 - compare the results with those from Step 6, they should be the same because you have added the de-dup logic 8. Run these jobs on an EMR cluster with Spark 9. compare DAGs, jobs and RDDs of the jobs run locally vs on AWS cluster, explain the differences     **Problem 3: (Points: 30)**   1. Define an Avro schema for your data (with UUIDs); explain your schema design 2. Write a Spark Job to read the input data (both Dataset1 and Dataset2 you generated for Problem 2) from S3 and convert it to Avro format (DatasetAvro) -  and write the results back into S3 in a different folder. 3. Modify your Problem 2 Spark jobs to read input data in the Avro format, de-dupe the events and give answers to the same 3 queries 4. Write a Spark job that reads DatasetAvro, de-dupes this data and  writes it back to S3 but in a Parquet format. Call it DatasetParquet 5. Run the jobs on an EMR cluster with Spark 6. compare DAGs, jobs and RDDs of the three jobs, explain the differences     **Problem 4: (Points: 30)**   1. Generate a new dataset (DatasetCommunity) in the format: <userID> <communityID>; make sure that you have a few userIDs per each "communityID"; use the same 20 userIDs you have generated for the Dataset1 in Problem2 2. For example:   user01 community01  user02 community01  user03 community02  user04 community02  ...  3. Write a new Spark job that loads both DatasetParquet and DatasetCommunity  4. Implement the following query:  get counts of clicks per URL per communityID  5. Run the jobs on an EMR cluster with Spark  6. compare DAGs, jobs and RDDs of this job with those from Problem 2, explain the differences    Problem 5: Bonus: 30 points   1. Take/modify the generated data/files you produced for Problem1, make sure you have at least 100 K records spread over 10-20 files; the Input data format is  <timestamp> <URL> <userID> 2. You want to have at least 3K records or more for the 3 URL’s and 3 ID’s below , we will focus on these 3 URL’s only for this problem    1. <timestamp> <[www.cnn.com (Links to an external site.)Links to an external site.](http://www.cnn.com)> <cnnssoadm>    2. <timestamp> <[www.angieslist.com (Links to an external site.)Links to an external site.](http://www.cnn.com)> <aglssoadm>    3. <timestamp> <[www.nytimes.com (Links to an external site.)Links to an external site.](http://www.cnn.com)> <nytssoadm> 3. Task 1: modify your Spark job from Problem 1 to calculate hourly counts of events (number of records) for the 3 specified target URLs only; the URLs can be passed as your job arguments via main() method, or hardcoded in a list in your code. Note the job execution time 4. Task 2: modify your job to use "broadcast variables" to specify the target 3 URLS. Note the job execution time 5. Report the time differences between #4 and #5;   You can run the jobs on either EMR Spark Cluster or locally (make sure to have local[\*] so all cores are used)    Problem 6: Bonus: 30 points   1. Take/modify the generated data/files you produced for Problem1, make sure you have at least 1 Mil records spread over 20 files; the Input data format is  <timestamp> <URL> <userID> 2. Modify your Problem 1 Spark jobs to use transformations specified below:   2a - use **groupbykey** , record the time it takes to run the same 3 queries as in Problem 1  2b - use **reducebykey instead** , record the time it takes to run the same 3 queries as in Problem 1  3. Next for both 2a and 2b modify the code to force the “number of partitions” to be 200 - record the time to execute the same 3 queries  4. Report the time differences between #2 (a and b) and #3    Part 2: Batch Views - 80 points  **Problem 7: (Points: 50)** Batch Views with NoSQL DBs (MongoDB)   1. Install MongoDB locally (or on Docker) 2. design and create tables to store whatever information is needed to calculate the following reports [most likely you will need to store the results of your batch views, computed in Problem 1,  and some additional information] :   Report 0: the same 3 original queries (hourly counts)  Report 1: Daily count of unique URLs  Report 2: Daily count of unique visitors   1. Modify your Problem1 Spark job to store required data into the new tables in the Mongo DB 2. write and execute Mongo DB queries/aggregates to generate the specified reports     **Problem 8: (Points: 30)**  Batch Views with RDBMS   1. Install a local MariaDB 2. Do the same tasks as in Problem 7, with the goal of running the same reports, but using the MariaDB this time - you have to modify your Spark job to write the results to MariaDB 3. explain the differences in table designs and aggregation/reporting views calculation approaches between MongoDB and MariaDB 4. *[Bonus: +20]* repeat the same , running your Spark job on an AWS Spark cluster and having your MariaDB running on a separate EC2 instance     Problem 9: Bonus: (Points: +20)   1. given the same input dataset as in Problem1, decide what would be the simplest Batch View to be calculated and stored, in order to run the following type of reports/queries:    1. what is the busiest hour of the day for a specified URL for a specified day? [URL and the 'day date' are input parameters for the query]    2. "busiest hour" is defined as the hour that has the most events for that URL (total count, not uniques) 2. use can choose which DB to use (MongoDB or MariaDB) 3. modify your Spark job from Problem1 to calculate an store the identified required information into the DB 4. run the specified query for some URL and some day (you pick, depending on what dates/urls you have in your input data)     Problem 10: Bonus: (Points: +50) Spark + Dynamo DB on AWS   1. Instead of using MariaDB or MongoDB, use Dynamo DB on AWS for your Batch Views storage 2. Define appropriate hash key, range key and necessary attributes to compute the same reports 3. Find an approach to calculate the same Reports as in Problem 7 and 8 4. explain what you could and could not achieve with DynamoDB (easily) as opposed to MongoDB and MariaDB, and why   Grading Rubrik  Total Points: 200     |  |  |  | | --- | --- | --- | | **Problem** | **Main requirements** | **Points** | | **Problem 1A** |  | **30** | |  | correct Spark jobs (using RDDs) are created and run; execution and results are demonstrated | 20 | | **Problem 1B** | jobs are executed on an EMR cluster with Spark | 5 | |  | DAGs, jobs and RDDs analyses is presented in the solution document | 5 | | **Problem 2** |  | **30** | |  | correct datasets are created (with UUIDs) and located in S3 | 5 | |  | correct Spark jobs (using RDDs) are created and run; execution and results of running it on one and two datasets are demonstrated | 15 | |  | jobs are executed on an EMR cluster with Spark | 5 | |  | DAGs, jobs and RDDs analyses is presented in the solution document | 5 | | **Problem 3** |  |  | |  | correct Spark job converting plain test logs into Avro and storing into S3 is implemented and demonstrated | 10 | |  | correct Spark jobs reading Avro data and executing 3 queries are implemented and demonstrated | 5 | |  | correct Spark job converting Avro logs into Parquet and storing them into S3 is implemented and demonstrated | 5 | |  | jobs are executed on an EMR cluster with Spark | 5 | |  | DAGs, jobs and RDDs analyses is presented in the solution document | 5 | | **Problem 4** |  | **30** | |  | correct DatasetCommunity input data is created; correct Spark job is implemented and demonstrated | 20 | |  | jobs are executed on an EMR cluster with Spark | 5 | |  | DAGs, jobs and RDDs analyses is presented in the solution document | 5 | | Problem 5 |  | Bonus: 30 | |  | Task1 is implemented correctly and results are demonstrated for the specified 3 URLs | 15 | |  | Task 2 is implemented correctly with broadcast variables; results are demonstrated; results are compared and explained | 15 | | Problem 6 |  | Bonus: 30 | |  | Jobs using reduceByKey and groupByKey transformations are implemented; results are demonstrated and compared | 20 | |  | number of partitions is changed to 200 and the results are compared with the previous jobs | 10 | | **Problem 7** |  | **50** | |  | MongoDB is installed; correct tables are designed and created for the batch views results | 15 | |  | Spark jobs to store the results into the new tables are implemented and demonstrated | 20 | |  | correct reports are calculated and demonstrated | 15 | | **Problem 8** |  | **30** | |  | MariaDB is installed; correct tables are designed and created for the batch views results; Spark jobs to store the results into the new tables are implemented and demonstrated | 20 | |  | correct reports are calculated and demonstrated; differences in table designs and aggregation/reporting views calculation approaches are explained | 10 | |  | setup on AWS Spark cluster + MariaDB on EC2 is working and demonstrated | Bonus: +20 | | Problem 9 |  | Bonus: 20 | |  | correct batch view (table structure) is defined and the Spark job is executed and data is stored into the table; an example query asking for the "busiest" hour of a day for a URL is run and results are demonstrated |  | | Problem 10 |  | Bonus: 50 | |  | DynamoDB on AWS is setup; correct tables for batch views are designed (hash key, range key and necessary attributes) and created | 10 | |  | Spark jobs to store the results into the new tables are implemented and demonstrated | 20 | |  | correct reports are calculated and demonstrated; differences in table designs and aggregation/reporting views calculation approaches are explained (in comparison to MongoDB and MariaDB) | 20 | |
| **Assignment6**   * Due Sunday by 11:59pm * Points 200 * Submitting a file upload * Available Sep 28 at 9am - Nov 15 at 11:59pm about 2 months   **Assignment 6 - 2 weeks**  **Spark Stream processing + Cassandra**  **Submission Requirements** - please review them carefully as we will now be quite strict about deducting points if not followed. If you’re unsure of how to format your solutions doc, please check the samples or the showcase solutions we have posted.  **Solution Document** - a Word or PDF file, titled <LastName>\_<FirstName>\_HW6.docx[pdf] (for example: Popova\_Marina\_HW6.docx)  that contains:  - answers to posted questions (if any)  - explanation of choices/approaches you made in your solution  - all execution steps and commands with screenshots of running the commands and screenshots of the results. Remember, you need to prove that you did the work you were asked to do.  - snippets of the most important code [if applicable] to demonstrate your solution  - full output results [or a subset of results if more than 50 lines]. You may submit a screenshot from a console or window or the raw text output, but make sure it is included in your solutions document.  - Full source code with no compiled classes, IDE generated artifacts like .project etc.  Submit either as separate files (say, hw1\_problem1.py or Problem1.java) or archived into a zip/tar archive. Archives should be named: <LastName>\_<FirstName>\_HW<number>.zip[tar]  Result files (if any) - can be placed into the same archive with the source code    **Total points: 200**  You can run all your jobs  locally (from Eclipse/Notebooks ) or on AWs EMR via spark-shell  The following online materials should be very helpful:  [https://spark.apache.org/docs/latest/streaming-programming-guide.html (Links to an external site.)Links to an external site.](https://spark.apache.org/docs/latest/streaming-programming-guide.html)  [https://spark.apache.org/docs/latest/streaming-kafka-0-10-integration.html (Links to an external site.)Links to an external site.](https://spark.apache.org/docs/latest/streaming-kafka-0-10-integration.html)    NOTE: for Problems 1-3 - you can also use netcat as the streaming data source, and modify your data generation scripts to push lines into your locally running netcat server  **Problem 1: (Points: 30)**   1. Configure your Spark Streaming job to read data from an input directory using the textFileStream source, with the batch window of 1 sec 2. You can use the same files with input data in the format <timestamp> <URL> <userID> that you've generated for Homework5. Make sure you have at least 20 unique URLs 3. Implement a job that :    1. counts the number of clicks per URL per batch and prints the results after each batch    2. counts the running total of clicks per URL and prints the results after each batch as well 4. Keep adding a new data file (with a few hundred events) into the Spark job's input directory every 1-2 seconds or so - we are simulating a "stream" of events 5. observe your job execution and results - you should have at least 5-10 batches completed     **Problem 2: (Points: 30)**   1. Modify the job from Problem 1 to do a "windowed" count of events per URL - per 5 second windows, with the same 1 second batch duration 2. Use tumbling windowing, not sliding windows 3. Keep feeding new input files into the job's input directory every 1-2 seconds - show the results of your job - have at least 5-10 windows completed     **Problem 3: (Points: 40)**   1. Use the same input data source (files fed into the input directory) 2. Calculate the number of unique users **per 10 sec tumbling windows** using two approaches:  * Using regular Spark aggregation methods * Using HyperLogLog algorithm * Try  2-3 different accuracy settings for HLL, and have at least 5-10 windows completed * Compare the results of HLL with different accuracy settings and the regular aggregation results * Compare the execution flow (number of jobs) for HLL with different accuracy settings - explain the results     **Problem 4: (Points: 40)**   1. Modify your Problem 3 job to take input from Kafka , instead of an input directory; use only HLL method of counting unique users - you are still counting unique users **per 10 sec tumbling windows** 2. Setup Kafka (either locally or remote) and create a topic "hw6problem4" with 4 partitions for that 3. Test the results first by manually sending events (strings in the specified format) via the Kafka console producer 4. Next, use your previously developed KafkaProducer (or take one from one of the showcase solutions) that sends events in this format; modify it to send about 200 events per second, with various userIDs and URLs 5. have at least 5-10 windows completed 6. Run your job and demonstrate the counts     Problem 5: Bonus: +30: Spark Streaming with Watermarking   1. Using the same input data (files fed into the input directory) - make sure your events have UUIDs and you have duplicate events 2. Write a Spark job that:    1. De-dupes events based on UUID, and up to 10 min "age" - this means that if you get an event with the same UUID that is within 10 min of each other - they are considered as one event; if the same two events happen more than 10 minutes apart - they are considered as distinct events    2. count the running total of clicks per URL and prints the results after each batch (you can pick whatever batch duration is convinient for you) 3. observe your job execution and results - demonstrate how events are de-dupes within the 10 min window, but not de-duped outside of the window     Part 2: Real-Time Views with Cassandra    **Problem 6: (Points: 30)**   1. Install Cassandra (either locally or on AWS/Docker) and create a "hw6" keyspace 2. Given events in the format <uuid><timestamp><url><ua\_country><TTFB> [TTFB == Time To First Byte] 3. Design a table, called "hw6\_p6" in Cassandra to store individual events and be able to answer queries:    1. Q1: Get count of events per country per URL for a specified time range [t1 … t2]    2. Q2: Get an average TTFB per country per URL for a specified time range [t1 … t2] 4. Explain design of your table's Primary Key 5. Using CQLSH, create the table, insert a few events, spanning a few hours in time 6. Create CQL queries for the above Q1 and Q2 , run them for a few variations of input parameters (URLs, countries, time ranges), demonstrate the results     **Problem 7: (Points: 30)**   1. Assume that you are getting too many events for each URL+country combination now - millions and millions per day; modify your table from Problem6 by adding partitioning by hour  - call it "hw6\_p7" 2. Explain design of your table's Primary Key 3. Using CQLSH, create the table, insert a few events, spanning a few hours in time 4. Create CQLSH queries for the above Q1 and Q2 , run them for a few variations of input parameters (URLs, countries, time ranges), demonstrate the results 5. Explain the main difference between "hw6\_p6" and "hw6\_p7" tables     Problem 8: (Bonus: +40) Spark Streaming with Cassandra   1. Create a Spark Streaming job to read data from an input directory, in the format <uuid><timestamp><url><TTFB> [TTFB == Time To First Byte]. Make sure you have a few unique URLs 2. Implement a job that :    1. counts the number of clicks per URL, and AVG TTFB per URL per batch (print the results after each batch for extra validation) - you can pick whatever batch duration you prefer    2. Stores data into a new Cassandra table, "hw6\_p8" 3. Design the "hw6\_p8" table to be capable of answer the following questions:    1. Q1: Get count of events per URL for a specified time range [t1 … t2]    2. Q2: Get an average TTFB per URL for a specified time range [t1 … t2] 4. Using CQLSH, create the table 5. Create CQLSH queries for the above Q1 and Q2 , run them for a few variations of input parameters (URLs and time ranges), demonstrate the results 6. Explain differences between "hw6\_p8" and "hw6\_p7" tables     Problem 9: Bonus: +40   1. Instead of generating/inserting events manually into your "hw6\_p7" table you created for Problem7 - write a small program that does:    1. generate a few thousand events in the format required in Problem7 (<uuid><timestamp><url><ua\_country><TTFB>), spanning a few hours in time    2. Using Cassandra driver, insert these events into the "hw6\_p7" table - for example, for Java, you could use this driver: [https://docs.datastax.com/en/developer/java-driver/3.3/ (Links to an external site.)Links to an external site.](https://docs.datastax.com/en/developer/java-driver/3.3/)    3. Also using the driver, implement the Q1 and Q2 queries and print out the results     Problem 10: Bonus: +15   * Modify your app from Problem 9 to use "async" operations writing events into Cassandra (for Java: [https://docs.datastax.com/en/developer/java-driver/3.3/manual/async/ (Links to an external site.)Links to an external site.](https://docs.datastax.com/en/developer/java-driver/3.3/manual/async/)) * Use QOURUM consistency level for your writes and reads * Run the Q1 and Q2 queries only after you have ensured that all write requests are completed     Problem 11: Bonus: +50   * Create a 3-node Cassandra cluster - either on AWS or using Docker/VMs * Create the same table, "hw6\_p7", from Problem 7 and insert a few events * Using 'nodetool' utility inspect:   + Status of the cluster   + Which nodes hold some specific row (by partition key, pick one that you inserted into your table)     Grading Rubrik  Total Points: 200     |  |  |  | | --- | --- | --- | | **Problem** | **Main requirements** | **Points** | | **Problem 1** |  | **30** | |  | correct Spark streaming job is implemented and run; execution and results are demonstrated - counts per each batch are printed out, for at least 5-10 batchs |  | | **Problem 2** |  | **30** | |  | correct Spark streaming job with 5 sec tumbling windows is implemented and run; execution and results are demonstrated - counts for at least 5-10 windows |  | | **Problem 3** |  | **40** | |  | correct Spark streaming job with 10 sec tumbling windows is implemented and run; execution and results are demonstrated and compared - counts of unique users using Spark aggregations and using HLL algorithm; for at least 5-10 windows | 25 | |  | 2-3 different accuracy settings for HLL are tried and results are demonstrated | 10 | |  | jobs analyses for different HLL settings is performed and presented in the solution document | 5 | | **Problem 4** |  | **40** | |  | correct Spark streaming job , with input form Kafka and with 10 sec tumbling windows is implemented and run; execution and results are demonstrated and compared - counts of unique users using HLL algorithm | 30 | |  | events are generated at 200 events/sec rate via KafkaProducer , and the results of the job are demonstrated for at least 5-10 windows | 10 | | Problem 5 |  | Bonus: 30 | |  | correct Spark streaming job with watermarking is implemented and run; running total of clicks per URL are printed for each window | 15 | |  | results are demonstrated: events are de-dupes within the 10 min window, but not de-duped outside of the window | 15 | | **Problem 6** |  | **30** | |  | Correct Cassandra table is created and keys design is explained | 15 | |  | using CQLSH, demo data is populated into the table, required queries are created and their results are demonstrated for a few input parameters | 15 | | **Problem 7** |  | **30** | |  | Correct Cassandra table is created and keys design is explained; difference between "hw6\_p6" and "hw6\_p7" tables is explained | 15 | |  | using CQLSH, demo data is populated into the table, required queries are created and their results are demonstrated for a few input parameters | 15 | | Problem 8 |  | Bonus: +40 | |  | Correct Cassandra table is created and keys design is explained; main differences between "hw6\_p8" and "hw6\_p7" tables are explained | 10 | |  | correct Spark Streaming job is implemented that calculates number of clicks per URL, and AVG TTFB per URL per batch | 10 | |  | Cassandra connector is used to store the required data into the Casssandra table | 10 | |  | using CQLSH, required queries are created and their results are demonstrated for a few input parameters | 10 | | Problem 9 |  | Bonus: +40 | |  | program that uses Cassandra driver is correctly implemented; results of storing events into the Cassandra table are demonstrated | 20 | |  | Q1 and Q2 queries are implemented, using Cassandra driver; and the results are demonstrated | 20 | | Problem 10 |  | Bonus: +15 | |  | application is modified to use ASYNC writes; results are demonstrated | 15 | | Problem 11 |  | Bonus: +50 | |  | 3-node Cassandra cluster is installed and working; specified table with some test data is created | 40 | |  | 'nodetool' utility use is demonstrated for cluster status and partition key location detection | 10 | |
| **Assignment7**  **Problem 1: (Points: 20)**   * Install ElasticSearch, Kibana and X-Pack - follow instructions from ElasticSeach official site: [https://www.elastic.co/guide/en/elasticsearch/reference/5.6/install-elasticsearch.html (Links to an external site.)Links to an external site.](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/install-elasticsearch.html) -- either locally or on EC2/VM/Docker * Demonstrate your cluster state by exploring "Monitoring" tab     **Problem 2: (Points: 20)**   * We will be using events in the format:   <uuid><timestamp><url><ua\_country><userId><ua\_browser><ua\_os><response\_status><TTFB>:  Where new fields are:  <ua\_browser>: should have values like: "Firefox", "Chrome", "IE", …  <ua\_os>:  should have values of a few operating systems like: "Mac OSX 10.2", "Windows ", "CentOS 5.3" …  <reponse\_status>: values of HTTP status like "200", "400", "401", "404", "500", "503", …   * design and create a mapping in ES; create an index with this mapping * create/index a few events using CURL commands (PUT ....) - write a script to generate events with a few different values for each of specified event fields * demo the events by querying the index and showing a few events     **Problem 3: (Points: 30)**   * Write a program to generate and index events specified in Problem 2 - this time using Java API (or Python API) calls to ES to index the data * Make sure you have at least 5-15 different values for each of the event fields (except UUID - UUID has to be, well, a UUID for each event :-) ), but vary it in time - say, 5 variations of URLs for one hour, than 10 for another hour, then back to 5… to simulate more "realistic" data where you do not see the same users/URLs all the time * demo the generated events by querying the index and showing a few events     **Problem 4: (Points: 30)**   * Using "Dev Tools" tab/utility in Kibana (or using Postman or other HTTP request handling tools), create/execute/demo results of the following queries: * Q1:  Show popularity of different browsers in different countries (count of events per country per browser) * Q2: find top 5 URLs that resulted in most error response codes (response code is in 500-599 range) (top 5 URLs where the max count of events with response code 5xx)     Problem 4\_Extra: Bonus: +50  [Tip: you need to use pipeline aggregations:  [https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-pipeline.html (Links to an external site.)Links to an external site.](https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-pipeline.html) ]   * Q3: given a time range of a few hours [t1 … t2], find which hour had the max total number of requests for a specific country (say, "China" or "Russia" - whatever countries you have in your generated events) * Q4: given a time range of a few hours [t1 … t2], find which URL had the worst (max) hourly average TTFB     **Problem 5: (Points: 15)**   * using Dev Tools from X-Pack - profile the Problem 4 Q1 and Q2 queries     **Problem 6: (Points: 15)**   * you can use "Dev Tools" for this problem, or any other HTTP-request handling tool * create two indices - one per day: for example:   + date1 = Nov 17, 2018 → index name: problem6\_11172017   + date2 = Nov 18, 2017 → index name: problem6\_11182017 * generate a few events in the same format as in Problem2 and with event "timestamp" values falling into either date1 or date2 * Index events for date1 into the problem6\_11172017, and with date2 into the problem6\_11182017 indices * Implement and demo results of the following query: * Q5: find all distinct response codes - across both indices     Problem 6\_Extra: Bonus: +10   * using indices from Problem 6, implement and demo results of the following query: * Q6: find all distinct countries per hour - across both problem6\_xxx indices     **Problem 7: (Points: 30)**   * add geo-location fields to your events:   (useful reference for mappings: [https://www.elastic.co/guide/en/elasticsearch/reference/current/geo-point.html (Links to an external site.)Links to an external site.](https://www.elastic.co/guide/en/elasticsearch/reference/current/geo-point.html))  <uuid><timestamp><url><ua\_country>**<location>**<userId><ua\_browser><ua\_os><response\_status><TTFB>:  where the new field 'location' should contain two coordinates: 'lt' and 'ln'   * create two new indices for events in this format - similar to Problem 6:   + problem7\_11172017 and problem7\_11182017 * generate a few events in the new format and with event "timestamp" values falling into either date1 or date2; make sure you have some good variation of values for TTFB field (say, between 0.04 and 10.6 seconds) * Index events for date1 into the problem7\_11172017, and with date2 into the problem7\_11182017 indices * Using Kibana: create geohash grid visualization which shows locations where TTFB > 5 seconds - for all data from the problem7\_xxx indices     **Problem 8: (Points: 40)**   * we will use the same indices /data created in Problem 7 * create Kibana Dashboard with the following visualizations:   + geohash grid created in Problem 7   + average TTFB   + pie chart with count of request per response code * using Kibana's date/time picker, pick different time ranges within the date1-date2 timeframe; demo the results     Problem 8\_Extra: Bonus: +20   * add one more visualization to your P8 Dashboard:   + visualization of the Q2 query from Problem 4 ( top 5 URL’s with the most error response codes)     Grading Rubrik  Total Points: 200   |  |  |  | | --- | --- | --- | | **Problem** | **Main requirements** | **Points** | | **Problem 1** |  | **20** | |  | ElasticSearch, Kibana and X-Pack are installed and demoed via Monitoring |  | | **Problem 2** |  | **20** | |  | correct mapping is created in ES; index with a few events is created and demonstrated |  | | **Problem 3** |  | **30** | |  | a program to index events into ES via ES API is implemented and the results are demonstrated |  | | **Problem 4** |  | **30** | |  | Q1 is correctly implemented and the results are demonstrated | 15 | |  | Q2 is correctly implemented and the results are demonstrated | 15 | | Problem 4\_Extra |  | Bonus: +50 | |  | Q3 is correctly implemented and the results are demonstrated | 20 | |  | Q4 is correctly implemented and the results are demonstrated | 30 | | **Problem 5** |  | **15** | |  | Q1 and Q2 queries are profiled and the results are demonstrated |  | | **Problem 6** |  | **15** | |  | Two required indices are create and populated with events; Q5 is correctly implemented and the results are demonstrated |  | | Problem 6\_Extra |  | Bonus: +10 | |  | Q6 is correctly implemented and the results are demonstrated |  | | **Problem 7** |  | **30** | |  | Two required indices with geo-location data are create and populated with events; geohash grid visualization is correctly implemented and demonstrated |  | | **Problem 8** |  | **40** | |  | Kibana Dashboard with the 3 required visualizations is created and demonstrated | 30 | |  | different time ranges are demonstrated | 10 | | Problem 8\_Extra |  | Bonus: +20 | |  | Visualization of the Q2 (top 5 URL’s) is added to the Dashboard and demonstrated |  | |