The Cloud Computing Capstone Task 2 Report

Andrei Karpau

Video report: https://vimeo.com/155666522?utm\_source=email&utm\_medium=vimeo-cliptranscode-201504&utm\_campaign=29220

Give a brief overview of how you integrated each system

In order to deal with the Spark Streaming framework I decided to integrate it with Kafka. I find it rather efficient, although at first I had some problems with spark performance on big data set. For installing the system I used HDP 2.3 (Hortonwork) and Ambari client. I was not able to successfully install HDP on Rhel 7 because of different errors during installation. However, the problem was solved by using Rhel 6 version.

Cassandra is installed manually on the separate cluster. The result data is written from Spark to Cassandra using DataStax Java Driver. This time I decided to save the results of all tasks to Cassandra, since I find it more efficient to work with data this way. Anyway, it is not a problem to save the data to a text file from Cassandra, if it is really needed.

To make Spark + Kafka work fast enough, I deployed a cluster with 4 m4.2xlarge instances and installed HDP 2.3 on it. For feeding data into Kafka, I have created the producer that reads data for each input file and sends it to Kafka. Each file reading process has its own thread, which is stored into the thread pool that is running 8 thread at the same time. It means that each producer processes 8 files simultaneously. For efficiency, I divided the data to 8 chunks and ran 8 different producers on different instances. So, the process of sending the data into Kafka was quicker.

What approaches and algorithms did you use to answer each question in each system?

For all the tasks I used the same algorithms that were used in Capstone part 1. The main logic was not changed much, nevertheless the implementation was changed due to using Spark.

Moreover, due to using Spark Streaming + Kafka there are some methods that were used for each task. The most important among them are:

* KafkaUtils.createStream method, for creating a stream from Kafka. For efficiency I use "auto.offset.reset = smallest" option, in order to read data from the beginning of topic. So, there is no need to write data to Kafka separately for each task.
* For combining previous rdd with the new one, I use checkpoint and updateStateByKey after counting basic sums.
* For deciding if the job is already finished, I tried to check if nothing was changed during the last time, but this method does not work well enough. So, I have also configured a time limit in order to ensure, that job is finished automatically and flush rdd flag will be set.
* The last transform method, which calculates the final values, makes sorting, filtering, etc., is evaluated only on the last step, when flush rdd flag is set to true.
* For writing into Cassandra I use DataStax driver. With the help of driver, the connection to Cassandra cluster is opened and data is stored into Cassandra Table.

Task 1.1 (top 10 popular airports)

This is classic sum counting task. At first flatMapToPair finds all non-empty origins and destinations and writes Airport/1 pair. Then reduceByKey counts the sums. After it updateStateByKey is called and count the sums as well. The last step is transform to pair function that sorts and gets top items and sends them to Cassandra, when flushRDD flag is set to true.

Task 1.3 (Rank the days of the week by on-time arrival performance)

This tasks count the sums as well. flatMapToPair method gets dayOfWeek and isdelayed values and writes key = day, value = tuple: isdelayed/1. Afterwards reduceByKey and updateStateByKey phases sums the tuples so result contains the number of delayed flights as well as the total number of flight. The final processing and sorting is called on the last stage, before sending data into Cassandra.

Task 2.1 (rank the top-10 carriers by on-time departure performance from X)

In this task flatMapToPair selects origin, airline id (I use airline ID, since in column description is written that airline code can be different through the years) and departure delayed columns and filters the lines based on input. Afterwards it writes key = origin + airline id and value = is delayed/1. Then reduceByKey and updateStateByKey make the same summing as in previous example. On the last stage, before sending results, it makes the sorting and finds top values.

Task 2.3 (for each source-destination pair X-Y, rank the top-10 carriers by on-time arrival performance)

The flatMapToPair stage filters the lines based on XY input and selects origin, destination, airline id and is delayed columns. Then writes origin+destination+airline id into the key and tuple IsDelayed/1 into the value. After it reduceByKey and updateStateByKey stages counts the sums and at the end the final performance counting and sorting is done.

Task 2.4 (For each source-destination pair X-Y, determine the mean arrival delay (in minutes) for a flight from X to Y)

The flatMapToPair stage selects origin, destination, delay in minutes values from, filters them and writes key = origin + destination and value = delay/1. The summing is made by reduceByKey and updateStateByKey. The final means counting is made on the last stage, before sending values to Cassandra.

Task 3.2 (Tom’s travel)

The flatMapToPair processes XYZ values saves them into collection. For each XYZ it makes two collection instances, which contain unique sample number (e.g. 1 for first XY and ZY, 2 for second XY and YZ etc.), origin, destination, date and flag that shows if it is the first or second part of the trip. Then each line is combined with all the collection items and if it has right origin, destination, date and time values, then it is written further (key is sample number, value – all needed information about the flight). Since, the key value is unique for flights for the same XYZ sample, the reduceByKey and updateStateByKey tasks get all the needed information to find out the best flights (by arrival time) and filter out all the rest. In the last stage the results are sent to Cassandra.

What system-level or application-level optimizations (if any) did you employ?

Due to many problems that I had, when running my implementations on whole data, I had to make a lot of improvements and optimizations. Finally, it takes around two minutes to run the whole job on four m4.2xlarge instances.

The main improvements and optimizations are:

* Writing to Kafka topic only once and then resetting the offset to smallest by spark stream connection, in order to read it from the beginning.
* Setting the Kafka max partition rate in order to avoid out of memory exceptions.
* Using direct Kafka stream instead of receiver, in order to avoid overfilling the spark memory
* Using a lot of Kafka producers that are writing to Kafka from different instances. Also, using the multithreaded implementation for reading the files simultaneously.
* Using checkpoint as well as updateStateByKey for combining RDDs.
* Evaluating the final computations such as sorting, final counting, finding top instances, etc. as well as writing to Cassandra only once, when the application finishes.

Your opinion about whether the results make sense and are useful in any way.

In my view, the results can be very useful for analyzing the way, the companies and airports work. It can be very important to deal with concrete values of popularities of airports or statistics of delay. Moreover, such a data can be used for providing the information to passengers, so they can easily imagine, how long they need for reaching another city and what the possible delay can be.

How did the different stacks (Hadoop and Spark) from Task 1 and Task 2 compare? Which stack did you find the easiest to use? The fastest?

For dealing with concrete task the Hadoop stack seems easier for me. I had much troubles with integrating Spark and Kafka. Moreover, some of them such as finishing the job when stream finishes, are rather tricky in Spark Streaming. However, the experience, which I got when working with Kafka and Spark Streaming is very important for me. Of course, these systems are much more efficient than Hadoop, when working on real streaming data. And for dealing with non-streaming data common Spark can be used with all its pros. All in all, I find that every system has own advantages and disadvantages and is more or less efficient when working with different data.

Results

In two first tasks my results differ a little bit from Hadoop results (Hadoop results differ a little bit from sample results, so I do not know which of them are right). Probably, some values were missing when writing/reading them in Kafka. Anyway, the results do not differ much and do not influence on the order.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Task 1.1   |  |  | | --- | --- | | airport | popularity | | ORD | 12345086 | | ATL | 11426777 | | DFW | 10715625 | | LAX | 7661829 | | PHX | 6533596 | | DEN | 6208962 | | DTW | 5592679 | | IAH | 5437194 | | MSP | 5161624 | | SFO | 5125817 | | Task 1.3   |  |  | | --- | --- | | day | perf | | 6 | 0.8277588138208978 | | 2 | 0.8112275161919221 | | 1 | 0.8035704684812311 | | 7 | 0.8016547179049915 | | 3 | 0.7951472163043267 | | 4 | 0.7695368050525832 | | 5 | 0.7608271916690015 | |
| Task 2.1   |  |  |  | | --- | --- | --- | | origin | fromairline | perf | | BOS | 20384 | 0.9145663379195584 | | BOS | 20295 | 0.90625 | | BOS | 20312 | 0.8836385115180153 | | BOS | 20366 | 0.8732394366197183 | | BOS | 19790 | 0.8668548589502433 | | BOS | 20374 | 0.8517727466894489 | | BOS | 19386 | 0.8502937274966837 | | BOS | 19707 | 0.8407105522862678 | | BOS | 19805 | 0.8404401541017438 | | BOS | 20363 | 0.8322456813819578 | | CMH | 20295 | 0.9355590062111802 | | CMH | 19822 | 0.9322139303482587 | | CMH | 19805 | 0.9208626025445793 | | CMH | 19386 | 0.9038944616278474 | | CMH | 19707 | 0.9005714285714286 | | CMH | 19790 | 0.8952539015445565 | | CMH | 20404 | 0.8812600969305331 | | CMH | 20355 | 0.8770326238245578 | | CMH | 20211 | 0.8752641482124764 | | CMH | 19704 | 0.8617649884704074 | | SRQ | 19805 | 0.9203970022280737 | | SRQ | 20312 | 0.9167927382753404 | | SRQ | 19977 | 0.9106699751861043 | | SRQ | 20374 | 0.9062351261304141 | | SRQ | 19790 | 0.9000851708148008 | | SRQ | 19386 | 0.8942533045842317 | | SRQ | 20355 | 0.8938418440064875 | | SRQ | 19707 | 0.888243831640058 | | SRQ | 20211 | 0.8858653397605349 | | SRQ | 20295 | 0.8674121405750799 | | JFK | 19977 | 0.886757190522448 | | JFK | 19805 | 0.8397495026225357 | | JFK | 20374 | 0.8346153846153846 | | JFK | 20404 | 0.8284935023263276 | | JFK | 19704 | 0.8258696104978129 | | JFK | 19386 | 0.798867523888168 | | JFK | 19790 | 0.7866691934916209 | | JFK | 20384 | 0.7845234787440956 | | JFK | 20355 | 0.7841796462791778 | | JFK | 20409 | 0.7835102290682572 | | SEA | 20304 | 0.90546331976807 | | SEA | 19391 | 0.9020152883947186 | | SEA | 19690 | 0.880690737833595 | | SEA | 20378 | 0.8759124087591241 | | SEA | 19386 | 0.8722235369687954 | | SEA | 19790 | 0.8681046042205953 | | SEA | 19805 | 0.8671830015003447 | | SEA | 20355 | 0.8608240637135522 | | SEA | 20404 | 0.8520408163265306 | | SEA | 19704 | 0.8512417014998771 | | Task 2.3   |  |  |  |  | | --- | --- | --- | --- | | origin | airline | dest | perf | | OKC | 20211 | DFW | 0.9442922374429223 | | OKC | 20366 | DFW | 0.8791208791208791 | | OKC | 19805 | DFW | 0.8786586637621376 | | OKC | 20304 | DFW | 0.856140350877193 | | OKC | 19790 | DFW | 0.8517975522692504 | | OKC | 20398 | DFW | 0.8198711857248443 | | OKC | 20417 | DFW | 0.0 | | BOS | 20211 | LGA | 1.0 | | BOS | 19790 | LGA | 0.8870055145519576 | | BOS | 20355 | LGA | 0.8453371671873704 | | BOS | 20384 | LGA | 0.837960832438513 | | BOS | 19707 | LGA | 0.8063567628115373 | | BOS | 20398 | LGA | 0.69798207644943 | | BOS | 19386 | LGA | 0.6605019815059445 | | BOS | 19805 | LGA | 0.5 | | BOS | 20417 | LGA | 0.3793103448275862 | | BOS | 20312 | LGA | 0.0 | | LGA | 20211 | BOS | 1.0 | | LGA | 20384 | BOS | 0.8916429153094463 | | LGA | 19790 | BOS | 0.8636311225901669 | | LGA | 20355 | BOS | 0.8630209641850208 | | LGA | 19707 | BOS | 0.8306328763329178 | | LGA | 20398 | BOS | 0.7170909991946974 | | LGA | 19386 | BOS | 0.6974564926372155 | | LGA | 20417 | BOS | 0.6060606060606061 | | LGA | 19805 | BOS | 0.0 | | MSP | 19707 | ATL | 0.8671875 | | MSP | 20304 | ATL | 0.8202479338842975 | | MSP | 20437 | ATL | 0.797150230001122 | | MSP | 19790 | ATL | 0.7689207482156042 | | MSP | 19386 | ATL | 0.7524721195463305 | | MSP | 20417 | ATL | 0.7513812154696132 | | MSP | 20366 | ATL | 0.7235976183014728 |   Task 2.4   |  |  |  | | --- | --- | --- | | origin | dest | mean | | OKC | DFW | 8.413198308479858 | | BOS | LGA | 8.523786828952353 | | LGA | BOS | 7.708941633259005 | | MSP | ATL | 12.100188746620415 |   Task 3.2   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | date1 | origin1 | dest1 | date2 | origin2 | dest2 | flightid1 | flightid2 | time1 | time2 | | 2008-05-16 | LAX | MIA | 2008-05-18 | MIA | LAX | 231 | 203 | 1406 | 1758 | | 2008-01-24 | DFW | STL | 2008-01-26 | STL | ORD | 1336 | 1835 | 0831 | 1450 | | 2008-09-07 | PHX | JFK | 2008-09-09 | JFK | MSP | 118 | 369 | 1406 | 1547 | | 2008-04-03 | BOS | ATL | 2008-04-05 | ATL | LAX | 270 | 1185 | 0855 | 1518 | |