16 February 2016

# Cloud Computing Capstone Project: Apache Spark Streaming + Kafka

## CLUSTER INFORMATION & NOTES ON INTEGRATION

Cloud Computing Capstone Task 2 starts off where Task 1 ends - in a 4-node Hadoop cluster (1 name node, 3 data nodes), with Cassandra installed. The following components were added to this software pack: Zookeeper (required by Kafka, running on 3 nodes only, as an odd number of nodes is needed for proper quorum), Kafka (running on all 4 nodes acting as message brokers) and Spark (with Master process on the name node and Worker processes everywhere). The 'spark-submit' script was ran from the name node as well (for the complete job running process, see the companion video), in YARN cluster mode (thus being able to take advantage of all data nodes).

The data was cleaned with the same Python script as in Task 1, with the only alterations of adding flight numbers and negative delays to the output, as was suggested by Task 1 peer review. No manual gzipping was done now, as the data needed to be streamed line by line.

I followed Prof. Farivar's advice and chose Scala as the main programming language for solving problems with Spark. I made a Bitbucket repository for my code, private during the development time, but now open for grading purposes.

As every question involved Kafka stream reading and input line parsing, I put them into separate files to be re-used constantly (see code for streaming and CSV parsing). And the particular solutions themselves were implemented as simple Scala objects with methods for data processing.

Using these technologies, I found Spark to be much more concise and convenient than Hadoop in terms of programming, as I could utilise Scala's functional programming capabilities and its built-in serialisable datatypes, such as tuples, thus avoiding the Hadoop's chore of writing manual serialisation of custom datatypes to strings. And I also liked the ability to combine batch and streaming processing in one system (the batch part, which I tried before the assignment switched to Spark Streaming, was even faster than Hadoop, as it's less disk-intensive).

But the streaming part comes for a price - as all computation is done gradually (and many state-updating calculations are redone with each new value), Spark Streaming requires more memory and CPU than Hadoop's batch processing does. On the other hand, it requires less HDD space by itself, but, as the same nodes were running Kafka producer scripts (with partitioning and [sometimes] replication), I still needed a lot of disk space for Kafka message logs. So I had to resize my data nodes to m4.xlarge and increase their disk space up to 90 GB. My name node, however, remained the same, as it could maintain its control functions without having to process heavy data computation. And I applied the following optimisations to get the most of these nodes:

## OPTIMISATIONS EMPLOYED

### Configuration:

It may sound stupid, but initially I didn't pay much attention to Hadoop's configuration in Task 1, so I thought that the '8 vcores' I saw in YARN's web interface were the actual cores it had 'found' (just because, coincidentally, my datanodes had 8GB RAM each back then). And when I resized my nodes (and saw in the instance info that m4.xlarge have only 4 vcores each), and saw that I had to make the configuration changes in order for my system to see the values, and that 8 vcores and 8GB were just arbitrary default settings. That explained why my Hadoop processing was quite sub-optimal and tended to consume all system resources with heavy spilling (as there weren't physically enough resources to provide the requested number of parallel containers). So I re-configured my system to use 3 cores and 12GB RAM per node for

YARN containers (1 core and 4GB RAM to be left for other tasks, like running the OS processes, Cassandra or streaming).

I also changed the CapacityScheduler settings in YARN configuration, setting **yarn.scheduler.capacity.resource-calculator** property to DominantResourceCalculator, instead of DefaultResourceCalculator, so that it would take both cores and RAM into consideration when assigning containers (DefaultResourceCalculator uses only RAM). This helped me increase the productivity of the cluster at least by 30%.

**Stream running:**

From this article I knew that I needed to split the Kafka stream into multiple partitions, as the number of partitions sets the upper bound on maximum parallelisation Spark can achieve with its executor containers, and it's actually OK to 'overpartition'. So I configured the topic 'airline-ontime' with 18 partitions, feeding it data from HDFS with Kafka's standard console producer and UNIX pipes (as shown in the video).

I also experimented with replication, but found it to create a very serious performance decrease (about 1.5 times as much). So I decided to optimise for speed, by setting the topic replication factor to 1 (with this amount of data a few lost entries would not have harmed total statistics seriously), and, in fact, no messages were actually lost in my case, as the calculations proved.

As for the consumer (Spark) side, I ran it with 8 executors (+ 1 control process), each with 1 vcore and 3GB RAM (+ some container overhead), running tasks in parallel.

**Code:**

There are 2 mechanisms of maintaining the persistent state for streams in Apache Spark: the old but stable **updateStateByKey** and the new and experimental **mapWithState** added in Spark 1.6. The former depends on the size of accumulated state, the latter depends on the size of the current data stream part. So I used **updateStateByKey** for Group 1 questions and Question 3.1 (where I needed to save the whole state to a text file), and **mapWithState** for the questions where I needed to update the data in the SB gradually, e.g., Group 2 and Question 3.2 (which could even crash my whole cluster without this optimisation, as the size of the accumulated state was too big to be saved to Cassandra at once).

However, my main mistake, I think, was running Spark on YARN, Kafka streaming and Cassandra saving on the same nodes. All these distributed systems together created serious processing delays on the cluster, so, even with optimisation techniques mentioned above, the performance of tasks that involved all these three systems simultaneously is still not perfect (as the video demonstrates). If I do a production cloud system later, I'll separate virtual servers by their tasks, so that I'd have at least the streaming part with its heavy messaging/partitioning/replication overhead moved elsewhere - and this, I think, is the main lesson I learned regarding cluster productivity.

And here are my solutions for the questions:

### 1.1 TOP 10 POPULAR AIRPORTS

Decided to start with the simplest one. The code is very straightforward, mirroring the 'airport, 1' => 'sum for each airport' Hadoop approach (but in more 'functional' Scala manner), and yielded the identical results:

(ORD,12449354)
(ATL,11540422)
(DFW,10799303)
(LAX,7723596)
(PHX,6585534)
(DEN,6273787)
(DTW,5636622)
(IAH,5480734)

(MSP,5199213)
(SFO,5171023)

### 1.3 WEEKDAY ARRIVAL PERFORMANCE RATING

This solution code is simple as well, mostly mirroring the Hadoop approach: count the total numbers of non-cancelled flights in a weekday and the number of flights that arrived on time (I used the standard metric from the original CSV files: to be 'on-time', the flight has to have an arrival delay of less than 15 minutes), then calculate the percentage of on-time flights.

(6,82.82717)
(2,81.16304)
(1,80.4086)
(7,80.21672)
(3,79.57186)
(4,77.02254)
(5,76.128426)

### 2.1 TOP-10 CARRIERS FOR EVERY AIRPORT

This solution in its processing algorithm is very similar to the next one, so I made a basic Scala trait with state-updating functions used in both questions. As a main comparison metric here (and in the next question) I used the Bayesian 'weighted rating' formula, so that the comparison would be more 'fair', taking carrier/airport size (in # of flights) into consideration (it is not fair to judge a company with 1000 flights and a company with 1000000 flights by the same percentage metric). The code for the formula is placed in a separate class, just like I did in Hadoop.

In comparison with my previous Hadoop solution (where I used a separate app to import large text files from HDFS to Cassandra), here (and in the subsequent 2 tasks as well) using the Datastax Spark driver for Cassandra was a great relief. I was no longer bound to the linear CSV structure and could take advantage of Cassandra's list data types. Here (see code) I saved the Top-10 records as frozen (=immutable, in order to avoid appending and item duplication) sorted lists of tuples, thus reducing the number of queries ran and guaranteeing the order of items after all updates:

| | |
|---|---|
| SRQ | [('DL', 0.8907809), ('AA', 0.8806869), ('US', 0.87609977), ('NW', 0.86808634), ('TW', 0.86628085), ('UA', 0.8657496), ('EA', 0.8581809), ('XE', 0.85149986), ('TZ', 0.8489666), ('FL', 0.8432086)] |
| CMH | [('AA', 0.90496165), ('NW', 0.8949941), ('DL', 0.8890142), ('US', 0.87431777), ('TW', 0.8678417), ('EA', 0.8586777), ('CO', 0.8575338), ('PI', 0.8538337), ('DH', 0.85366505), ('ML (1)', 0.85090363)] |
| JFK | [('UA', 0.8823546), ('AA', 0.83999676), ('XE', 0.8391709), ('CO', 0.83541656), ('DH', 0.83361524), ('EA', 0.8167953), ('NW', 0.8142169), ('EV', 0.8081295), ('YV', 0.8040785), ('US', 0.80123216)] |
| SEA | [('OO', 0.88840246), ('NW', 0.8698513), ('DL', 0.8667623), ('AA', 0.86486244), ('US', 0.85772127), ('HA', 0.8511665), ('CO', 0.8500006), ('PS', 0.8478014), ('YV', 0.8418658), ('TZ', 0.84117365)] |
| BOS | [('PA (1)', 0.8940371), ('DL', 0.86643934), ('TZ', 0.8510384), ('NW', 0.8497486), ('ML (1)', 0.84735066), ('EA', 0.8446346), ('XE', 0.8421649), ('EV', 0.84199643), ('AA', 0.84084076), ('9E', 0.8395463)] |

### 2.2 TOP-10 DESTINATIONS FOR EVERY AIRPORT

This task code uses the same trait with state-updating functions, and its processing flow differs only in field names and initial threshold for the quantity of items. And it yielded the following results:

| | |
|---|---|
| SRQ | [('MCO', 0.94469714), ('RDU', 0.92649144), ('TPA', 0.9189564), ('IAH', 0.91137), ('RSW', 0.90549207), ('MEM', 0.89414185), ('CLT', 0.89281374), ('DTW', 0.8855565), ('ATL', 0.88067275), ('BWI', 0.87865245)] |
| CMH | [('CLE', 0.9213497), ('IND', 0.90469223), ('DFW', 0.9034613), ('MSP', 0.90003765), ('BNA', 0.8986719), ('CLT', 0.8984127), ('DTW', 0.8958545), ('IAH', 0.8954783), ('MEM', 0.8930693), ('ATL', 0.89012766)] |
| JFK | [('STT', 0.9076781), ('BQN', 0.88204795), ('SNA', 0.8638568), ('UCA', 0.8638195), ('CHS', 0.84899163), ('SRQ', 0.8488688), ('STX', 0.8464052), ('LGB', 0.84632796), ('SFO', 0.84595186), ('BUR', 0.8458677)] |
| SEA | [('CVG', 0.91396666), ('PSC', 0.89877605), ('MEM', 0.89175504), ('DTW', 0.8844285), ('IAH', 0.8733711), ('MSP', 0.8725524), ('DFW', 0.8707409), ('IAD', 0.87020487), ('PIT', 0.86754817), ('CLE', 0.86734176)] |
| BOS | [('LGA', 0.9241785), ('AUS', 0.8859828), ('BDL', 0.8851336), ('SJU', 0.8664461), ('CVG', 0.86484027), ('OAK', 0.8644439), ('SJC', 0.86419624), ('LGB', 0.8640638), ('MSY', 0.8631643), ('DCA', 0.86140704)] |

### 2.4 MEAN ARRIVAL DELAYS FOR ORIGIN-DESTINATION PAIRS

The algorithm is similar to the weekday arrival performance rating, the only difference is counting actual arrival delays for every origin-destination pairs instead of ‘on-time’ flags count. Then I divided each route’s sum of delays by its number of flights, with the following results:

| | |
|---|---|
| LGA-BOS | 1.4838648 |
| BOS-LGA | 3.7841182 |
| OKC-DFW | 4.969055 |
| MSP-ATL | 6.737008 |

### 3.1 AIRPORT POPULARITY DISTRIBUTION

I am not very well-versed in statistics, so in Task 1 I made a graphical plot of data which showed that the Zipf distribution of airport popularity was highly unlikely. But as I saw the mentioning of Kolmogorov-Smirnov test in the Task 1 example solutions, I decided not to use any graphics this time and run this test instead.

Spark itself has K-S tests functionality, but for now it only supports testing for normal distribution ‘out-of-the-box’ (for everything else one has to hand-code the distribution’s CDF, and, again, I’m not very good in statistics to be sure I can do it properly). So I resorted to using Python scripts, as SciPy module for Python has K-S tests for many distributions. And there is an openly available automated script to test a given dataset for all of them with this library. All I had to do was to edit it slightly, so that it would read a CSV file produced by the task code (which is trivial and re-uses most of Question 1.1 methods) and output the following results:

```
----------------------------------------------------------------------
Top 10 after 1 iteration(s)
----------------------------------------------------------------------
1    mielke               p:  0.234134138541   D:  0.0548031754774
2    genextreme           p:  0.0658617917864  D:  0.0692416184761
3    invweibull           p:  0.0658617098157  D:  0.069241631172
4    lomax                p:  0.0542312207169  D:  0.0711961462832
5    genpareto            p:  0.0542147752065  D:  0.0711991553419
6    burr                 p:  0.022657276847   D:  0.0793851410933
7    invgamma             p:  0.0219732268653  D:  0.0796574902217
8    loglaplace           p:  0.0136912449228  D:  0.0837479874598
9    lognorm              p:  0.00322038343632 D:  0.0951754614787
10   johnsonsu            p:  0.0030894765637  D:  0.0954829203903
```

According to this output, I'd honestly argue that in fact the log-normal distribution suggested by the example answers may be more probable than Zipf (which is not on the list at all), but still is not the best fit for the data - as Mielke's Beta-Kappa distribution seems to fit a lot better.

### 3.2. TOM'S TRAVELLING OPTIONS

Due to the streaming nature of the system (which results in the requirement of processing streams in linear or at least near-linear time), I could not afford the luxury of saving the complete cross-product of flights, like I did in Hadoop (as it would have required me $O(n^2)$ time). So I had to save only the single best flights for the first and for the second half of the day for each origin, destination and date, and then query the database by the separate Python script (so that the user won't need to calculate dates and run two DB queries manually, all he/she needs to input is the route and its start date, the rest is done by the script). The Spark algorithm itself remained quite trivial (see code) - all I had to do was to choose (for each origin, destination and date) the flight with the smallest arrival delay, both in the current stream part and in the global state, then save the resulting flights to the database. Here are the results I got:

| Route | Date 1 | Date 2 | Total delay | Flight 1 | Flight 2 |
|---|---|---|---|---|---|
| BOS-ATL-LAX | 2008-04-03 | 2008-04-05 | 5.0 | FL 270, 05:48 | FL 40, 18:57 |
| PHX-JFK-MSP | 2008-09-07 | 2008-09-09 | -42.0 | B6 178, 11:27 | NW 609, 17:47 |
| DFW-STL-ORD | 2008-01-24 | 2008-01-26 | -19.0 | AA 1336, 06:57 | AA 2245, 16:54 |
| LAX-MIA-LAX | 2008-05-16 | 2008-05-18 | -9.0 | AA 280, 08:17 | AA 456, 19:25 |

### CONCLUSIONS

The processing result in the end was very similar to Hadoop's (with the only difference of allowing negative delays for 3.2 and 2.4, and excluding cancelled flights from top-10 ratings), so it makes me think that the process is OK, and the result still makes sense (while I may doubt about negative delays, they make the verification process a lot easier). And I am very likely to employ AWS, Hadoop and Spark in my further professional life.