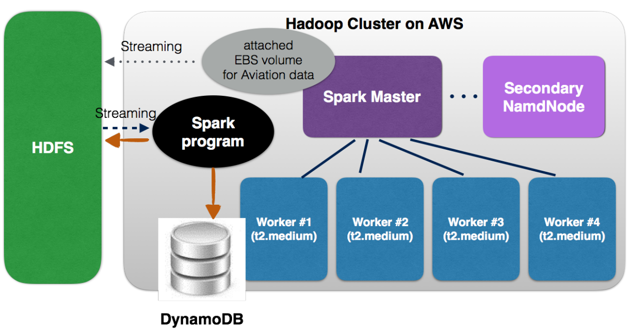
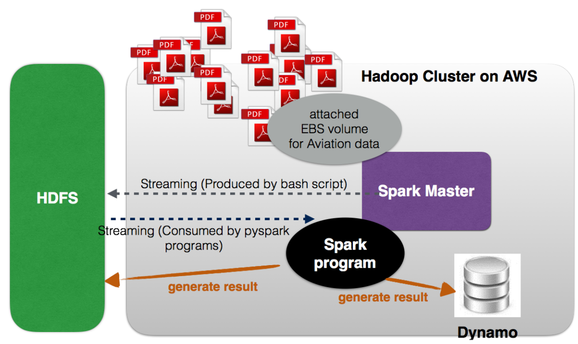
Cloud Computing Capstone Task 2 Report

Paul Lo paullo0106@gmail.com 2016-02-23

I. System Architecture





I deployed 6 EC2 instances in Hadoop cluster for Spark tasks

- One master and secondary NameNode, plus 4 *t2.medium* instances as Spark workers.
- EBS Volume for **Aviation** dataset is attached, corresponding csv files were unzipped and copied to HDFS.
- **DynamoDB** is running on one of Slave DataNode for saving results of Group2 and 3.

II. Technology stack and tools

I mainly used Python for executing the tasks and integrating HDFS, DynamoDB:

- *pyspark* for implementing spark jobs executing on Hadoop cluster. *matplotlib pyplot* for drawing popularity visualization in task 3-1.
- AWS python SDK *boto3*(https://github.com/boto/boto3) for DynamoDB CRUD operations.
- For simulating **streaming scenario**, I implement a bash script continuously copying csv files to a HDFS directory which is registered to Spark Streaming Context:
  `ssc.textFileStream('hdfs://MASTER_URL/data/on_time/streaming/')`

  Script example:
  https://github.com/paullo0106/cloud_computing_capstone/tree/master/pyspark/streaming_producer.sh

III. Data processing process and Assumption

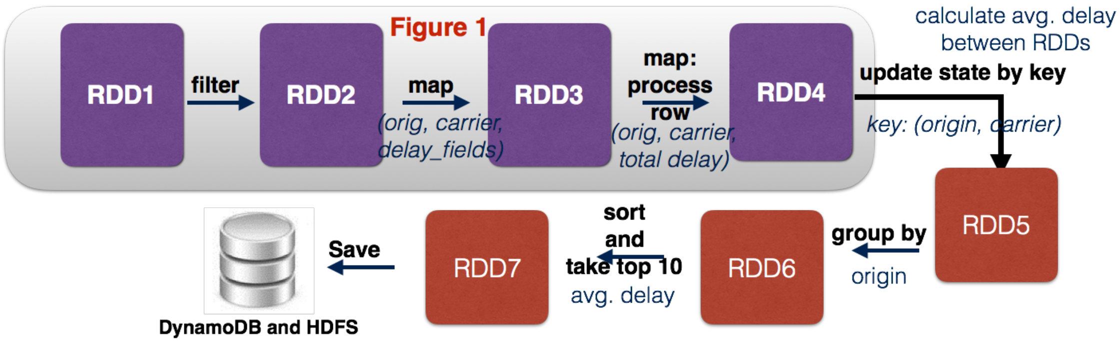
- When sorting the on time performance, I ignored rows with empty delay fields, as I considered them as missing values (noticed some fields have '0' value) rather than 0. In addition, I ignored negative delays.
- I have kept only useful columns including ***FlightDate***, ***AirlineID***, ***Carrier***, ***FlightNum***, ***Origin***, ***Dest***, and delay fields before putting to HDFS for saving storage.

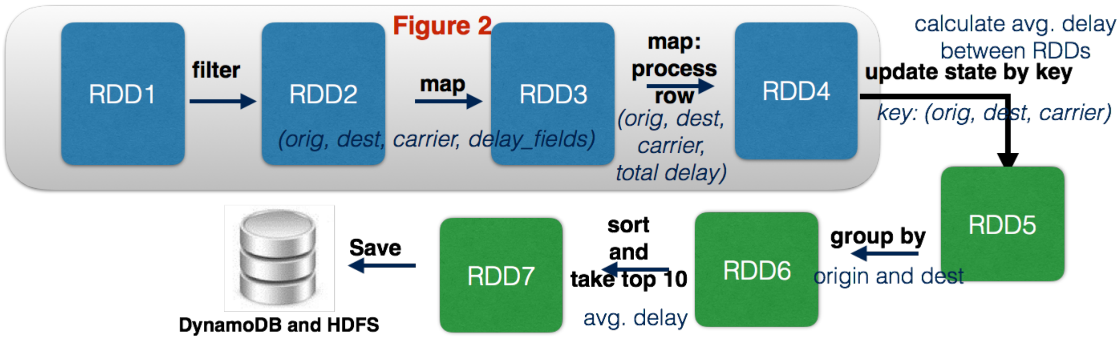
IV. Code, algorithm and optimization

All my spark streaming tasks following the paradigm as shown in https://github.com/paullo0106/cloud_computing_capstone/blob/master/pyspark/streaming_consumer.py *(except that I wasn't able to finish 3-2 due to performance bottleneck)*, this python program has three major parts:

- Have Spark Stream Context monitor a HDFS directory by invoking *textFileStream('hdfs://MASTER_URL/streaming/dir/')* and then execute my streaming_producer.py which keep copying new csv files to the directory.
- To keep aggregating the status of **RDDs** in each window while new data comes in. I universally take advantage of *updateStateByKey(updateFunc)* and *transform(sortOrOtherFunc)* in my programs, and I invoke *foreachRDD(process)* for saving the status periodically to **DynamoDB** and **HDFS** in *process function*. When the data keep coming, there are intermediate results being generated, **I inserted only the final results to DynamoDB** for saving time and storage, and the DB schema created in Task 1 was reused: https://github.com/paullo0106/cloud_computing_capstone/tree/master/dynamodb-crud
- For the Spark Streaming Context to stop itself, I maintain a dfstream_num to check if all of those csv files are processed, after it reach the threshold and the progress stuck for a while the streaming receiver will exit: *ssc.stop(stopSparkContext=True, stopGraceFully=True)*.

The general idea of RDD flow for Group 2-1 and 2-2 are depicted as *Figure 1* (they only differ in replacing **carrier** with **dest** in RDD flow), while Group 2-3 is shown in *Figure 2*.
Detailed code in python can be found in:
https://github.com/paullo0106/cloud_computing_capstone/blob/master/pyspark/streaming2-1.py
https://github.com/paullo0106/cloud_computing_capstone/blob/master/pyspark/streaming2-2.py
https://github.com/paullo0106/cloud_computing_capstone/blob/master/pyspark/streaming2-3.py





I applied several mechanisms to enhance the capability and performance of my Hadoop cluster both on system and application level:

- **Rebalance the cluster**: I increased my DataNode number for faster execution, and I executed *start-balancer.sh* to re-balance my cluster after the addition of DataNode to ensure the efficiency of every node, and I found that using "**2g**" *spark.executor.memory* configuration for my 4 workers slightly speed up the analyzing process by about 10% in Group 2 questions.
- **Block replication and network delay**: I changed *dfs.replication* configuration from default value 3 to 2, so I have more spaces available and less data trans. time, as the tasks
- **Data insertion and query**: I'm a beginner on DynamoDB, so the table schema I created is quick and dirty. I transferred some query tasks to python side as workaround. Improving the schema is my next to-do list.

V. Video and Result

Please refer to http://youtu.be/Aiv9oiZ8B00 for the **video** and detailed results in **appendix** section. My streaming pace is at 1 csv every 5 seconds, to keep the clip concise and to the point, **in the video I would only show:**

(A) How my spark cluster and data structure like

(B) How my streaming producer/consumer generally work (demonstrate a year of 12 files)

(C) Query DynamoDB with pre-executed results on each task.

- Group 1-1 and Group 1-2

	Command	Result
Group 1-1	ubuntu@ec2:~$ python mr_job_capstone1-1.py -r hadoop hdfs://<url>:/data/orig_dest/*.csv -o ubuntu@ec2:~$ python mr_job_capstone1-1.py -r hadoop hdfs://<url>:/data/orig_dest/*.csv -o hdfs://<url>:/data/q1_1_output/ hadoop fs -cat /data/q1_1_output/part-00000	Popularity, Airport 12546419 "ATL" 9760533 "ORD" 8209285 "DFW" 6630709 "DEN" 5401766 "MSP" 5359079 "CLT" 5212033 "PHX" 4935139 "LAX" 4882625 "DTW" 4750123 "IAH"
Group 1-2	ubuntu@ec2:~$ python mr_job_capstone1-2.py -r hadoop hdfs://<url>:/data/on_time/*.csv -o hdfs://<url>:/data/q1_2_output/ ubuntu@ec2:~$ hadoop fs -cat /data/q1_2_output/part-00000	Avg delay, Carrier 41.17 "AQ" 41.45 "F9" 44.46 "HP" 46.12 "WN" 48.21 "NW" 48.80 "US" 49.23 "HA" 49.45 "DL" 50.42 "AS" 52.14 "OO"

- **Group 2-1** (see **appendix** for results)

  Task execution: python mr_job_capstone2-1.py -r hadoop hdfs://<url>:/data/on_time/*.csv -o hdfs://<url>:/data/q2_1_output

  Query: python dynamodb-crud/query_table2-1.py <airport>
- **Group 2-2** (see **appendix** for results)

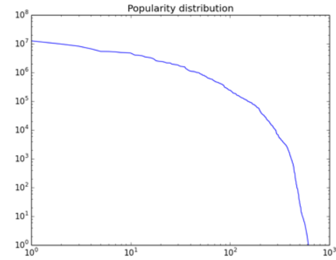
  Task execution: python mr_job_capstone2-2.py -r hadoop hdfs://<url>:/data/on_time/*.csv -o hdfs://<url>:/data/q2_2_output

  Query: python dynamodb-crud/query_table2-2.py <airport>
- Group 2-3 (see **appendix** for results)

  Task execution: python mr_job_capstone2-3.py -r hadoop hdfs://<url>:/data/on_time/*.csv -o hdfs://<url>:/data/q2_3_output

  Query: python dynamodb-crud/query_table2-3.py <airport>
- Group 3-1 – Airport popularity distribution

  Reusing most of code in Group 1-1, I depicted the top 10,000 popular airports by



matplotlib.pyplot as shown below. It is an originally **long-tail distribution** but **it's not a straight line** on log-log plot, therefore it's not Zipf distribution. Example solutions suggests us use ***powerRlaw*** to further verify.
https://github.com/paullo0106/cloud_computing_capstone/blob/master/popularity_plot.py

- Group 3-2 (see **appendix** for results)
  note: I reused what I had in Task 1 for month 1, 4, 5, 9 in 2008 because I failed to finish an efficient version on spark. (I fixed missing airline number pointed by peers)

VI. Comparisons, Ideas and Future Work

For the **stack comparison** between task 1 and 2:

- **Spark** run slightly faster (10-20% in my case) than **MapReduce**. It's not as fast as I expected, I think I should create another set of powerful workers to take advantage of its in-memory computation characteristics. (haven't checked the **cost** part though)
- The code is shorter – this would benefit development speed, although for beginner like me would have a steep learning curve in the beginning, I believe it would be worthwhile in the long run by enjoying new paradigm of computing.
- Some people in the forum argue about the anti-pattern on streaming existing files as an exercise, I found it a little bit tricky but did help me quickly adapt new toys on old tasks.

For **future work**, I had some difficulties estimating the cluster size (and optimize the cost), and think **Amazon EMR** should be a good option to scale the computation/storage easily. I wasn't able to make it work for some reasons, I will continue studying it. In addition, I would want to try **Cassandra** as many people recommend the easiness of its python driver. **Kafka** is another thing I didn't have enough time to play with.

VII. Reference

- PySpark streaming functions:
  http://spark.apache.org/docs/latest/api/python/pyspark.streaming.html#pyspark.streaming.DStream.updateStateByKey
  http://spark.apache.org/docs/latest/api/python/pyspark.streaming.html#pyspark.streaming.DStream.transform
- Spark Streaming - Maintaining calculation status
  https://class.coursera.org/cloudcapstone-001/forum/thread?thread_id=198

Appendix

Group 2-1 results

ubuntu@ec2:~$ python dynamodb-crud/query_table2-1.py **SRQ**
Top 10 carriers From SRQ:
US delays 39.31 on average
YV delays 41.07 on average
OH delays 49.52 on average
DL delays 52.32 on average
RU delays 52.54 on average
TZ delays 54.96 on average
9E delays 55.17 on average
FL delays 56.86 on average
XE delays 59.49 on average
NW delays 62.25 on average

ubuntu@ec2:~$ python dynamodb-crud/query_table2-1.py **CMH**
Top 10 carriers From CMH:
US delays 47.94 on average
HP delays 50.24 on average
WN delays 50.8 on average
NW delays 51.05 on average
OO delays 52.99 on average
DL delays 53.05 on average
YV delays 56.06 on average
9E delays 56.46 on average
AA delays 56.62 on average
OH delays 56.81 on average

ubuntu@ec2:~$ python dynamodb-crud/query_table2-1.py **JFK**
Top 10 carriers From JFK:
HP delays 49.73 on average
UA delays 53.86 on average
CO delays 55.68 on average
B6 delays 56.17 on average
US delays 56.22 on average
EV delays 57.07 on average
DL delays 57.7 on average
RU delays 60.21 on average
AA delays 60.69 on average
OH delays 63.78 on average

ubuntu@ec2-:~$ python dynamodb-crud/query_table2-1.py **SEA**
Top 10 carriers From SEA:
EV delays 38.75 on average
WN delays 41.62 on average
XE delays 41.82 on average
F9 delays 42.34 on average
OO delays 42.88 on average
DL delays 42.9 on average

```
US delays 47.14 on average
AS delays 47.68 on average
HP delays 48.85 on average
CO delays 48.98 on average
```

```
ubuntu@ec2:~$ python dynamodb-crud/query_table2-1.py BOS
Top 10 carriers From BOS:
EV delays 43.22 on average
HP delays 46.8 on average
AS delays 47.01 on average
DL delays 48.28 on average
RU delays 48.44 on average
US delays 49.73 on average
NW delays 52.23 on average
TZ delays 55.23 on average
MQ delays 55.93 on average
OH delays 57.81 on average
```

Group 2-2 results

```
ubuntu@ec2:~$ python dynamodb-crud/query_table2-2.py SRQ
Top 10 destination From SRQ:
DCA delays 27.56 on average
MSP delays 33.83 on average
CLT delays 40.62 on average
CLE delays 41.93 on average
MDW delays 43.86 on average
CVG delays 47.88 on average
BOS delays 49.3 on average
LGA delays 55.41 on average
IND delays 56.41 on average
ATL delays 57.54 on average
```

```
ubuntu@ec2:~$ python dynamodb-crud/query_table2-2.py CMH
Top 10 destination From CMH:
OMA delays 36.0 on average
FLL delays 41.12 on average
MCI delays 41.54 on average
PHX delays 43.29 on average
CLT delays 44.08 on average
STL delays 44.63 on average
BNA delays 45.39 on average
LAX delays 45.63 on average
TPA delays 46.47 on average
MCO delays 47.95 on average
```

```
ubuntu@ec2:~$ python dynamodb-crud/query_table2-2.py JFK
Top 10 destination From JFK:
BHM delays 34.67 on average
MLB delays 43.85 on average
LGB delays 45.05 on average
RSW delays 45.1 on average
```

PNS delays 45.6 on average
BQN delays 46.6 on average
OAK delays 47.24 on average
DEN delays 48.36 on average
PSE delays 49.61 on average
CHS delays 49.77 on average

ubuntu@ec2:~$ python dynamodb-crud/query_table2-2.py **SEA**
Top 10 destination From SEA:
MIA delays 35.37 on average
MCI delays 36.43 on average
BNA delays 36.76 on average
ABQ delays 37.73 on average
SNA delays 39.86 on average
CVG delays 39.94 on average
ATL delays 41.17 on average
MCO delays 41.93 on average
LIH delays 42.87 on average
MKE delays 43.16 on average

ubuntu@ec2:~$ python dynamodb-crud/query_table2-2.py **BOS**
Top 10 destination From BOS:
ACK delays 35.0 on average
SLC delays 42.06 on average
DAY delays 43.96 on average
CLT delays 44.07 on average
AUS delays 45.17 on average
PHX delays 45.32 on average
SAV delays 46.6 on average
MYR delays 46.73 on average
RSW delays 47.26 on average
ISP delays 47.56 on average

Group 2-3 results

ubuntu@ec2:~$ python dynamodb-crud/query_table2-3.py **LGA BOS**
From LGA to BOS:
DL delays 36.71 on average
US delays 43.96 on average
MQ delays 57.95 on average
OH delays 70.54 on average

ubuntu@ec2:~$ python dynamodb-crud/query_table2-3.py **BOS LGA**
From BOS to LGA:
DL delays 43.19 on average
US delays 45.62 on average
OH delays 49.83 on average
MQ delays 59.61 on average
TZ delays 133.0 on average

ubuntu@ec2:~$ python dynamodb-crud/query_table2-3.py **OKC DFW**
From OKC to DFW:
OH delays 47.5 on average
MQ delays 55.22 on average
AA delays 59.63 on average
EV delays 63.88 on average
OO delays 144.26 on average

ubuntu@ec2:~$ python dynamodb-crud/query_table2-3.py **MSP ATL**
From MSP to ATL:
NW delays 45.32 on average
DL delays 50.02 on average
OH delays 50.94 on average
FL delays 58.04 on average
OO delays 59.73 on average
EV delays 60.57 on average

Group 3-2 results

Analyze task execution:

ubuntu@ec2:~$ python mr_job_capstone3-2.py -r hadoop hdfs://ec2-xx-xx-xx-xx.compute-1.amazonaws.com:/data/on_time/**On_Time_On_Time_Performance_2008_1.csv** -o hdfs://ec2- xx-xx-xx-xx.compute-1.amazonaws.com:/data/q3_2_output1/
ubuntu@ec2:~$ python mr_job_capstone3-2.py -r hadoop hdfs://ec2-xx-xx-xx-xx.compute-1.amazonaws.com:/data/on_time/**On_Time_On_Time_Performance_2008_4.csv** -o hdfs://ec2- xx-xx-xx-xx.compute-1.amazonaws.com:/data/q3_2_output4/
ubuntu@ec2:~$ python mr_job_capstone3-2.py -r hadoop hdfs://ec2-xx-xx-xx-xx.compute-1.amazonaws.com:/data/on_time/**On_Time_On_Time_Performance_2008_5.csv** -o hdfs://ec2- xx-xx-xx-xx.compute-1.amazonaws.com:/data/q3_2_output5/
ubuntu@ec2:~$ python mr_job_capstone3-2.py -r hadoop hdfs://ec2-xx-xx-xx-xx.compute-1.amazonaws.com:/data/on_time/**On_Time_On_Time_Performance_2008_9.csv** -o hdfs://ec2- xx-xx-xx-xx.compute-1.amazonaws.com:/data/q3_2_output9/

Result queries:
ubuntu@ec2:~$ python dynamodb-crud/query_table3-2.py DFW STL ORD 20080124
LAX->ORD (by AA) then ORD->JFK (by B6), historical delay: 0

ubuntu@ec2:~$ python dynamodb-crud/query_table3-2.py BOS ATL LAX 20080403
Route BOS->ATL->LAX on 20080403:
BOS->ATL (by FL 270) then ATL->LAX (by DL 1185), historical delay: 0

ubuntu@ec2:~$ python dynamodb-crud/query_table3-2.py LAX MIA LAX 20080516
Route LAX->MIA->LAX on 20080516:
LAX->MIA (by AA 280) then MIA->LAX (by AA 203), historical delay: 0

ubuntu@ec2:~$ python dynamodb-crud/query_table3-2.py PHX JFK MSP 20080907
Route PHX->JFK->MSP on 20080907:
PHX->JFK (by B6 178) then JFK->MSP (by NW 609), historical delay: 0