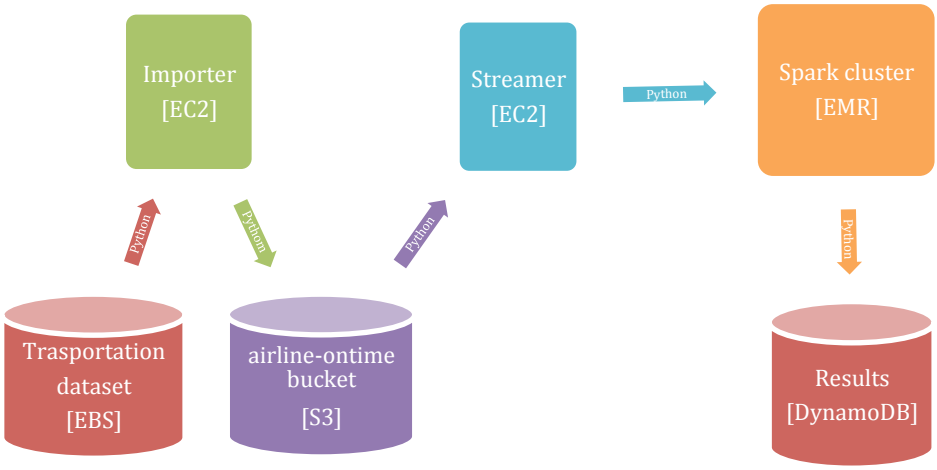
Cloud Computing Capstone

Task 2
Stream processing with Spark Streaming

This document describes the solution adopted to solve the second task of the Coursera Cloud Computing Capstone.

The next figure gives a brief overview of the architecture, which is explained in the following sections of the document:



The stack of technologies used for this task is:

- EC2: Linux instances used to run the importer and streaming scripts
- S3: persistent storage of imported data
- EMR: managed Hadoop cluster to perform calculations on data
- Python: scripts used to clean and stream the source dataset.
- Pyspark: Python implementation of the Spark API
- Spark streaming: processing engine running on EMR cluster through YARN. Its processes are implemented in pyspark, a Python library.
- DynamoDB: persistent key-value storage of task results.

Systems integration

During task 1, aviation dataset was already cleaned and imported to S3 using a Python script. In order to transform this batch data into streaming data, a custom Python streaming server was developed. It listens for connections on a given port, reads data from the source S3 bucket and streams this data line by line to each connected client.

In order to run Spark, an EMR cluster (Amazon managed Hadoop service) is launched with one master and two core nodes. Spark is already integrated in this solution via YARN, and includes a web console that allows monitoring Spark tasks and logs.

Each of the task 2 exercises is solved using a spark program, which is submitted to the Spark master node for its execution. These programs connect to the streaming server, processing the streaming data and dumping the accumulated results to the master

terminal (group 1) and DynamoDB tables (group 2 and 3).

Algorithms to answer each question

Each of the questions was answered using pyspark scripts, which are explained next:

- Common parts:

Every script instantiate a StreamingContext and listen to incoming data via a socketTextStream:

```
lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))
```

After that, each line of data is splitted and mapped to a custom Flight object, and cancelled flights are filtered beforehand:

```
lines.map(lambda line: line.split(","))\
     .map(lambda fields: Flight(fields))\
     .filter(lambda fl: fl.Cancelled == 0)
```

- Group 1 ex 2:

Each flight is flat-mapped to an origin and a destination, which are then used as keys to update the current state map by adding 1 to each airport.

```
lines.flatMap(lambda fl: [(fl.Origin, 1), (fl.Dest, 1)])\
     .updateStateByKey(lambda new, last: sum(new) + (last or 0))
```

The resulting RDDs are then ordered by descending frequency and the 10 first are printed to stdout:

```
filtered.foreachRDD(lambda rdd: print_rdd(rdd))

def print_rdd(rdd):
    airports = rdd.takeOrdered(10, key = lambda x: -x[1])
```

- Group 1 ex 3:

Each flight is mapped to a key (day of week) and a pair of values (arrival delay and 1), which are then used to update the current state map by counting the total delay and number of flights for each day.

```
lines.map(lambda fl: (fl.DayOfWeek, (fl.ArrDelay, 1)))\
     .updateStateByKey(updateFunction)
```

The resulting RDDs are then ordered by ascending medium delay and printed to stdout:

```
filtered.foreachRDD(lambda rdd: print_rdd(rdd))

def print_rdd(rdd):
    daysOfWeek = rdd.takeOrdered(10, key = lambda x: x[1][0]/x[1][1])
```

- Group 2 ex 1:

Each flight is mapped to a key formed by the origin airport and the carrier and a value formed by the departure delay and frequency, which are then used as keys to update

the current state map.

```
lns.map(lambda fl: ((fl.Origin, fl.UniqueCarrier), (fl.DepDelay, 1)))
   .updateStateByKey(updateFunction)
```

Each resulting RDD partition is then saved to DynamoDB:

```
filtered.foreachRDD(lambda rdd: rdd.foreachPartition(save_partition))

def save_partition(rdd):
    item = Item(out_table, data={
            "airport": record[0][0],
            "carrier": record[0][1],
            "mean_delay": int(record[1][0] / record[1][1])
        })
    item.save(overwrite=True)
```

- Group 2 ex 2:

Each flight is mapped to a key composed by the origin and destination airports and a value formed by the departure delay and frequency, which are then used as keys to update the current state map.

```
lns.map(lambda fl: ((fl.Origin, fl.Dest), (fl.DepDelay, 1)))
   .updateStateByKey(updateFunction)
```

Each resulting RDD partition is then saved to DynamoDB.

- Group 2 ex 3:

Each flight is mapped to a key formed by the origin and destination airports and a value formed by the arrival delay and frequency, which are then used as keys to update the current state map.

```
lns.map(lambda fl: ((fl.Origin, fl.Dest), (fl.ArrDelay, 1)))
   .updateStateByKey(updateFunction)
```

Each resulting RDD partition is then saved to DynamoDB.

- Group 3 ex 2:

Each flight is filtered and mapped through two different streams, to classify XY and YZ flights. The mapper uses (XY date + Y airport) as the key, and the flight as the value.

```
flights_xy = filtered.filter(lambda fl: fl.CRSDepTime < "1200")\
                     .map(lambda fl: map_flight(fl, False))
flights_yz = filtered.filter(lambda fl: fl.CRSDepTime > "1200")\
                     .map(lambda fl: map_flight(fl, True))
```

These streams are then joined by key, obtaining all flights that meet the conditions:

```
xyz = flights_xy.join(flights_yz)
```

Finally, all flights are saved to DB, keeping flights with lower total delay and overwriting flights with higher delay.

```
xyz.foreachRDD(lambda rdd: rdd.foreachPartition(save_partition))
```

Questions results

Question 1.1

Airport	Total flights
ORD	12051796
ATL	11323515
DFW	10591818
LAX	7586304
PHX	6505078
DEN	6183518
DTW	5504120
IAH	5416653
MSP	5087036
SFO	5062339

Question 1.2

Carrier	Mean delay
HA	-1.01 min
AQ	1.15 min
PS	1.45 min
ML(1)	4.74 min
PA (1)	5.32 min
F9	5.46 min
NW	5.55 min
WN	5.56 min
OO	5.73 min
9E	5.86 min

Question 1.3

Weekday	Mean delay
6	4.30 min
2	5.99 min
7	6.61 min
1	6.71 min
3	7.20 min
4	9.09 min
5	9.72 min

Question 2.1

Origin	Top 10 carriers by on time departure from origin
SRQ	TZ, XE, US, AA, UA, YV, NW, DL, TW, FL
CMH	AA, DH, NW, DL, ML(1), US, EA, PI, TW, YV
JFK	UA, CO, DH, XE, AA, B6, PA(1), NW, DL, MQ
SEA	OO, PS, YV, US, AA, HA, NW, DL, TZ, EV
BOS	TZ, PA(1), ML(1), EV, NW, DL, US, AA, EA, XE

Question 2.2

Origin	Top 10 destinations by on time departure from origin
SRQ	EYW, IAH, MEM, TPA, MDW, RDU, BNA, FLL, MCO, BWI
CMH	SYR, OMA, AUS, CLE, SDF, MSN, SLC, CAK, DFW, DTW
JFK	SWF, ISP, ABQ, MYR, ANC, UCA, BQN, BGR, STT, CHS
SEA	EUG, PIH, PSC, CVG, MEM, LIH, DTW, IAH, CLE, SNA
BOS	SWF, ONT, GGG, AUS, MSY, LGA, BDL, MDW, LGB, OAK

Question 2.4

Origin	Destination	Mean arrival delay
LGA	BOS	1 min
BOS	LGA	3 min
OKC	DFW	4 min
MSP	ATL	6 min

Question 3.2

X	Y	Z	DATE	Flight X-> Y	Flight Y->Z
BOS	ATL	LAX	2008-04-03	FL270	DL75
PHX	JFK	MSP	2008-07-09	B6178	NW609
DFW	STL	ORD	2008-01-24	AA1336	AA2245
LAX	MIA	LAX	2008-05-16	AA280	AA456

Employed optimizations

In order to improve performance, lower network traffic and obtain results faster, several optimisations have been employed on the process:

- Removal of unused data columns: only needed columns were imported from the original dataset and thus streamed to the Spark cluster.
- Spark properties tweaking: several Spark configuration options were modified to achieve better results:

```
"spark.dynamicAllocation.enabled": "true"
"spark.executor.memory": "2G"
"spark.yarn.executor.memoryOverhead": "2G"
```

- Use of DynamoDB ranges and indexes: result data can be efficiently queried and ordered in DynamoDB via the use of range keys and local secondary indexes.

Results analysis

As with task 1, results obtained from task 2 exercises do make sense and can be useful, as they transform a huge amount of information into particular answers to specific questions.

These answers summarize data that might be useful for individuals to plan routes or make travelling decisions and for companies to improve flights and airport connections performance.

Stack comparison

Hadoop and Spark Streaming stacks are not easily compared, as their goals are different: while the former is used for batch processing, the latter is designed for stream processing.

Though we have used the same source dataset for both systems, it made little sense to used stored (batch) data to feed a stream-processing engine, and this makes it harder to obtain conclusions.

Regarding ease of use, Hadoop was much easier and friendly and I was able to leverage Hive (a SQL-like language that is compiled into map-reduce tasks) and Hue (a graphical Hive query interface), allowing me to perform higher-level operations, while I had to do all the programming (including a streaming server) in order to run Spark Streaming.

On the other hand, the fastest stack was Spark, probably because it is designed with real-time data processing in mind. Also, Hive queries compilation to map-reduce might not be as optimal as processes directly implemented in Python.

Video report

https://youtu.be/j3ewvE2jzJ0