Cloud Computing Capstone project

Part 2

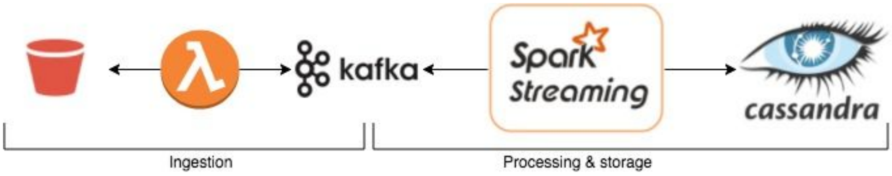
Johan Kielbaey - September 2nd 2017

Introduction

This report covers the second part of the Capstone project, which focuses on using streaming data systems to solve the presented questions. The video demonstration is available on https://youtu.be/uvE7usqvV_U.

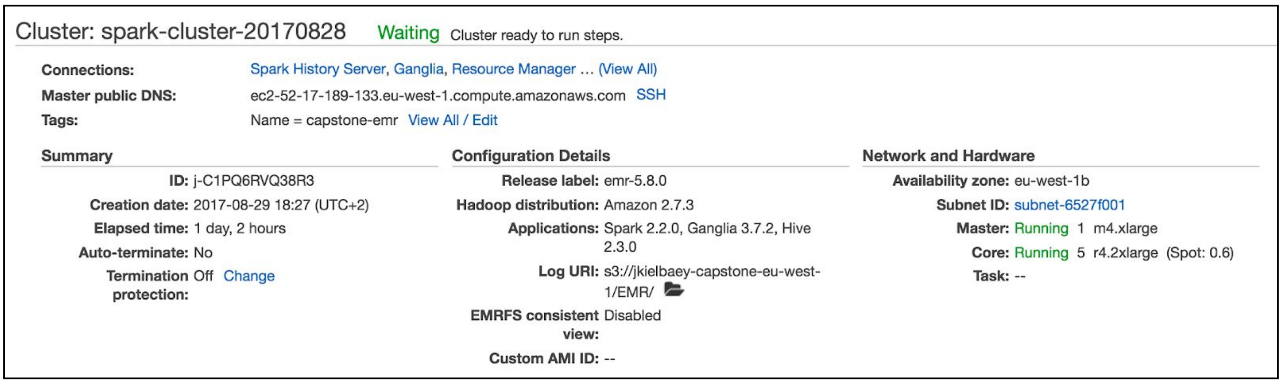
System overview

The schema below gives a high-level overview of the different components and their data flows.



To ingest the data into Kafka, I used a similar approach as on part 1 of this project: the already cleaned data was read from S3 storage and sent to Kafka via AWS Lambda functions. The messages in Kafka are in JSON format. To process the data, I used Spark Streaming as required. Where necessary, I stored the results in a Cassandra table. All spark programs were written in Python.

An EMR cluster with Spark 2.2.0 was deployed with 1 m4.large master node and 5 r4.2xlarge nodes. In order to reduce cost of running this cluster, I opted to use spot instances.


Cluster: spark-cluster-20170828 Waiting Cluster ready to run steps.

Connections: Spark History Server, Ganglia, Resource Manager ... (View All)
Master public DNS: ec2-52-17-189-133.eu-west-1.compute.amazonaws.com SSH
Tags: Name = capstone-emr View All / Edit

Summary
ID: j-C1PQ6RVQ38R3
Creation date: 2017-08-29 18:27 (UTC+2)
Elapsed time: 1 day, 2 hours
Auto-terminate: No
Termination protection: Off Change

Configuration Details
Release label: emr-5.8.0
Hadoop distribution: Amazon 2.7.3
Applications: Spark 2.2.0, Ganglia 3.7.2, Hive 2.3.0
Log URI: s3://jkielbaey-capstone-eu-west-1/EMR/
EMRFS consistent view: Disabled
Custom AMI ID: --

Network and Hardware
Availability zone: eu-west-1b
Subnet ID: subnet-6527f001
Master: Running 1 m4.xlarge
Core: Running 5 r4.2xlarge (Spot: 0.6)
Task: --

Both the Kafka and Cassandra clusters were deployed using CloudFormation templates available on the aws-quickstart GitHub project. The Kafka cluster was composed of 3 m4.2xlarge EC2 instances with 200GB of EBS storage / node. The Cassandra cluster consisted of 3 c4.xlarge instances with 128GB/node.

Name	Instance ID	Instance Type	Availability Z	Instance St	Status Checks	IPv4 Public IP
awsqs-broker-0	i-096835aa744236352	m4.2xlarge	eu-west-1b	running	2/2 checks ...	54.77.173.230
awsqs-broker-2	i-0de74979558c64f79	m4.2xlarge	eu-west-1b	running	2/2 checks ...	34.248.182.98
awsqs-broker-1	i-0fb0b1486e24efc7c	m4.2xlarge	eu-west-1b	running	2/2 checks ...	52.214.52.29
cassandra-opscenter	i-099610c98146bb89e	t2.medium	eu-west-1b	running	2/2 checks ...	52.212.194.92
cassandra-node	i-0b09d2f073ed6acf0	c4.xlarge	eu-west-1a	running	2/2 checks ...	-
cassandra-node	i-0a9950daa07c881e1	c4.xlarge	eu-west-1b	running	2/2 checks ...	-
cassandra-node	i-0182ea4cc33d06f82	c4.xlarge	eu-west-1c	running	2/2 checks ...	-

Optimizations

1. Initially, the Kafka topic was created with 5 partitions. The documentation mentions that the createDirectStream method of the KafkaUtils package will automatically map Kafka partitions onto RDD partitions and read data from these partitions in parallel. The Kafka topic was recreated with 50 partitions. Tests with data of 1 month showed a reduction in time to read the data from 75sec to 15 sec.
2. In order to reduce the amount of data to be read, solutions for different questions are combined into a single spark program. The program reads the data only once and persists it. The different solutions use this persisted DStreams to process the data and calculate the solutions.

Results

This section presents the solutions to each of the different questions. Each program starts by streaming the data from Kafka using `createDirectStream()`, which was converted from JSON format into python dictionaries and persisted into memory (`.map(lambda msg: json.loads(msg[0])).cache()`).

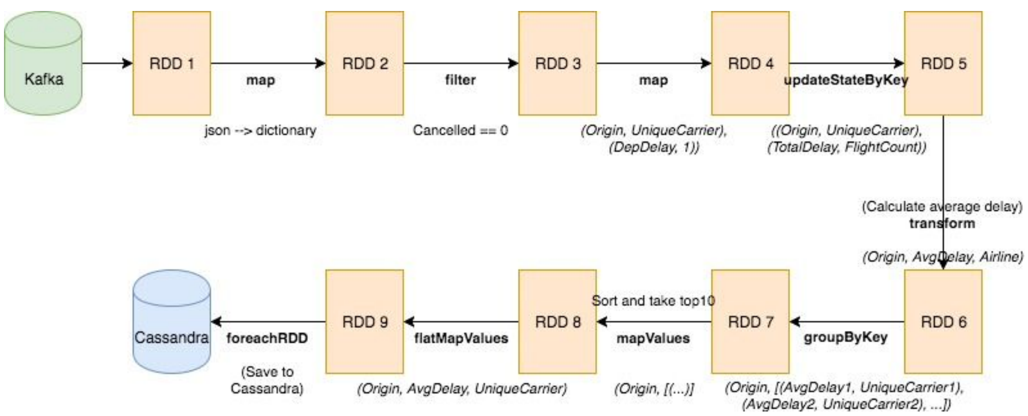
Group 1

For question 1, the persisted stream of flights is converted into a stream of tuples (`(Origin, 1)` and `(Dest, 1)`) using `flatmap`. An aggregation of these tuples is maintained (`updateStateByKey`). On every iteration `takeOrdered` takes the top 10 most popular airports, which are printed on the screen.

For question 2 the cancelled flights are filtered out of the stream (`filter`). The filtered stream is converted into tuples (`UniqueCarrier, (ArrDelay, 1)`) using `map`. Of these tuples a aggregation of flights per airline and total delay is maintained (`updateStateByKey`). At the end of every iteration, the average delay per airline is calculated, which is then sorted and only the top 10 are displayed.

Question 1.1		**Question 1.2**	
Airport	**Total number of flights**	**Airlines**	**Average delay (minutes)**
ORD	12.449.354	HA	-1.01
ATL	11.540.422	AQ	1.16
DFW	10.799.303	PS	1.45
LAX	7.723.596	ML (1)	4.74
PHX	6.585.534	PA (1)	5.30
DEN	6.273.787	F9	5.46
DTW	5.636.622	WN	5.54
IAH	5.480.734	NW	5.55
MSP	5.199.213	OO	5.73
SFO	5.171.023	9E	5.85

Group 2



The solution for each of the questions in group 2 starts similar to question 1.2. The schema above gives an overview of each of the different transformations used for question 2.1.

This sequence consists of 4 stages:

1. Organise the tuples to efficiently aggregate the state of the current iteration with the previous iterations (RDD1 to RDD5) to aggregate all data since the program started running.
2. Calculate the average delay for each combination of Origin and UniqueCarrier; and transform the stream of tuples (RDD6).
3. Group the values of all tuples with the same key into a single tuple, sort these values and take the top 10 (RDD6 to RDD9).
4. Save the results into Cassandra.

For sequence of transformations for the questions 2.2 and 2.3 differ in the fields used to generate RDD4.

The answers for question 2.1:

Airport	Airline	Average delay (minutes)	Airport	Airline	Average delay (minutes)	Airport	Airline	Average delay (minutes)
SRQ	TZ	-0,3820	JFK	UA	5,9683	BOS	TZ	3,0638
	XE	1,4898		XE	8,1137		PA (1)	4,4472
	YV	3,4040		CO	8,2012		ML (1)	5,7348
	AA	3,6335		DH	8,7430		EV	7,2081
	UA	3,9521		AA	10,0807		NW	7,2452
	US	3,9684		B6	11,1271		DL	7,4453
	TW	4,3047		PA (1)	11,5235		XE	8,1029
	NW	4,8564		NW	11,6378		US	8,6879
	DL	4,8692		DL	11,9867		AA	8,7336
	MQ	5,3506		TW	12,6391		EA	8,8914
CMH	DH	3,4911	SEA	OO	2,7058			
	AA	3,5139		PS	4,7206			
	NW	4,0416		YV	5,1223			
	ML (1)	4,3665		TZ	6,3450			
	DL	4,7134		US	6,4124			
	PI	5,2013		NW	6,4988			
	EA	5,9374		DL	6,5356			
	US	5,9933		HA	6,8555			
	TW	6,1591		AA	6,9392			
	YV	7,9612		CO	7,0965			

The answers for question 2.2:

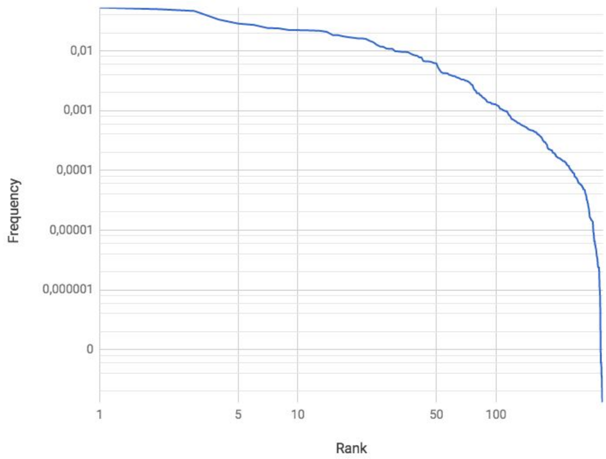
Airport	Destination airport	Average delay (minutes)	Airport	Destination airport	Average delay (minutes)	Airport	Destination airport	Average delay (minutes)
SRQ	EYW	0,0000	JFK	SWF	-10,5000	BOS	SWF	-5,0000
	TPA	1,3289		ABQ	0,0000		ONT	-3,0000
	IAH	1,4446		ANC	0,0000		GGG	1,0000
	MEM	1,7030		ISP	0,0000		AUS	1,2087
	FLL	2,0000		MYR	0,0000		LGA	3,0541
	BNA	2,0623		UCA	1,9170		MSY	3,2465
	MCO	2,3645		BGR	3,2103		LGB	5,1362
	RDU	2,5354		BQN	3,6062		OAK	5,7832
	MDW	2,8381		CHS	4,4027		MDW	5,8956
	CLT	3,3584		STT	4,4928		BDL	5,9827
CMH	AUS	-5,0000	SEA	EUG	0,0000			
	OMA	-5,0000		PIH	1,0000			
	SYR	-5,0000		PSC	2,6505			
	MSN	1,0000		CVG	3,8787			
	CLE	1,1050		MEM	4,2602			
	SDF	1,3529		CLE	5,1702			
	CAK	3,7004		BLI	5,1982			
	SLC	3,9393		YKM	5,3796			
	MEM	4,1520		SNA	5,4063			
	IAD	4,1581		LIH	5,4811			

The answers for question 2.3:

X - Y	Airline	Average delay (minutes)	X - Y	Airline	Average delay (minutes)
LGA - BOS	TW	-3	BOS - LGA	TW	-11
	US	-2,9		US	1,09
	PA (1)	-0,42		DL	2,02
	DL	1,75		PA (1)	6,07
	EA	4,82		EA	9,46
	MQ	9,86		MQ	12,63
	NW	14,44		NW	15,21
	OH	27,98		AA	28
	AA	28,5		OH	30,45
MSP - ATL	9E	0		TZ	133
	EA	4,12	OKC - DFW	TW	0,1
	OO	4,76		EV	1,36
	FL	6,27		AA	4,57
	DL	6,33		MQ	4,68
	NW	6,99		DL	6,73
	OH	8,3		OO	12,84
	EV	10,08		OH	47,5

Group 3

For question 3.1, I re-used the logic of question 1.1. The graph on the right presents a log-log graph of the calculated popularity of the different airports in the dataset. As in task 1, the graph clearly doesn't show a straight downward line and thus the popularity is not a zipf distribution. Obviously the results and conclusion haven't changed from task 1.

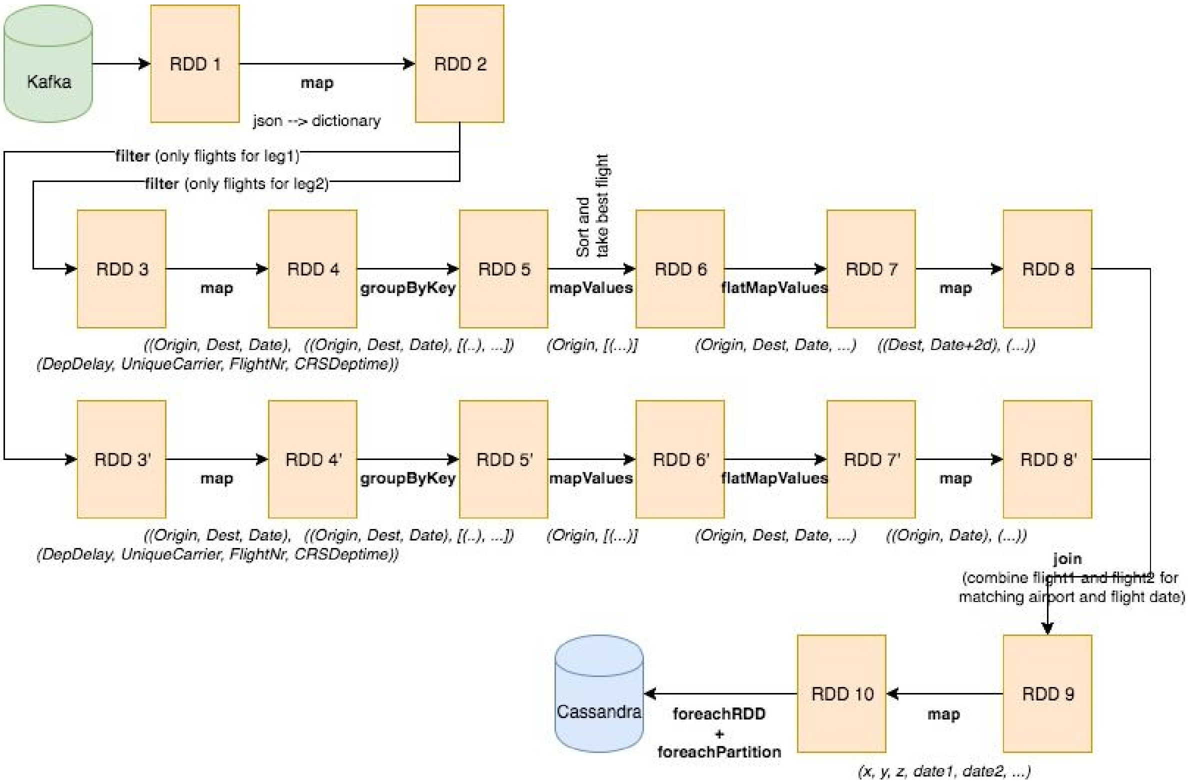


The sequence of transformations used for question 3.2, re-uses many of the transformations used in previous questions. The schema below gives an overview of all transformations. It consists of 3 major stages:

1. Stream data from Kafka and convert from JSON into dictionaries (RDD1 to RDD2).
2. Processing the stream of tuples to generate a set of candidate flights for part 1 (RDD2 to RDD8) and a similar set of transformations to generate a set of candidate flights for part 2 (RDD2 to RDD8'). Both sets of transformation use the same source (RDD2) and produce tuples in form of ((airport, date), (flight details)).
3. Combining candidate flights from RDD8 and RDD8' with matching (airport, date) and transforming this into a stream of itineraries (x, y, z, date1, date2, airline/flight1, airline/flight2, delay1, delay2, total delay). The stream of itineraries is send to Cassandra.

The answers for question 3.2:

	Leg 1			Leg 2			
Date	X - Y	Flight number	Delay	Y - Z	Flight number	Delay	Total delay
03-04-2008	BOS - ATL	FL 270	7.0	ATL - LAX	FL 40	-2.0	5.0
07-09-2008	PHX - JFK	B6 178	-25.0	JFK - MSP	NW 609	-17.0	-42.0
14-01-2008	DFW - STL	AA 1336	-14.0	STL - ORD	AA 2245	-5.0	-19.0
16-05-2008	LAX - MIA	AA 280	10.0	MIA - LAX	AA 456	-19.0	-9.0



Conclusion

Given both task 1 and task 2 use the same data set and same questions, my conclusions on whether the results make sense and regarding its usefulness are obviously the same. The potential for financial impact due to delayed flights, as well as impact on customer satisfaction and airline reputation, will drive airlines to maintain a timely schedule. In that sense these results are as expected. I was surprised to see negative averages, which meant that some flights are structurally leaving or arriving early.

Travel agencies can use this type of information to give better recommendations to customers. Airlines can use it to detect and remediate structural delays in flight schedules.

In the first part of this project I used hive queries to answer the questions. To my surprise spark streaming was between 5 to 10 times slower than hive. (In both cases EC2 instance types were used with the same CPU capacity and similar network bandwidth (c4.2xlarge vs r4.2xlarge). The r4 family has more memory, which should be an advantage for spark).

I expect this difference in performance to be a consequence of the interface and programming language I used. Hive SQL queries are a higher level language to express business logic. The Hive engine translates the queries into execution plan that run in the native language of the Hadoop framework (java). My spark programs were written in python, which require regular serialization/deserialization of data between scala and python. In spark I used the RDD API, which is a 5x slower compared to the DataFrame API[1].

[1] https://www.slideshare.net/databricks/2015-0616-spark-summit (slide 16)