Cloud computing capstone Task 2

Setting up the environment

To perform the capstone tasks I choose Amazon Elastic Compute Cloud services, which enables us to use scalable and reliable infrastructure.

My architecture in Amazon Cloud consists of a cluster with three c3.xlarge instances, which are optimized for a computing purposes. As a server operation system, I choose Red Hat Linux Amazon Machine Images. After Amazon instances were up and running, I connected them into a network to create a cluster.

To manage Hadoop cluster and integrate it into Amazon environment I choose Ambari Hortonworks framework.

The process of creating a cluster and setupping Hadoop and Ambari Hortonworks server shown as series of screenshots on the video.

Extract data

The data for processing is provided as EBS Volume and I choose to mount this volume to my infrastructure as read-only volume. There are a lot of data on provided volume and we need to choose with what specific data we will work. I choose «airline_ontime» data set as it is providing us with all the data we needed. Also all data are zipped so I wrote a bash-script which recursively unzip all data sets for all years.

Clean data

To clean up the data I choose PIG, which is a platform for analyzing large data sets. As PIG is used on the top of Hadoop, I copied all the extracted data to Hadoop file system.

I prepared a PIG-script, which queries all the data sets and extract only the necessary data.

In screenshots in the video, it is shown how PIG-script can be lunched from Ambari Hortonworks interface.

On-time performance definition

To compute on-time performance I decided to use DepDelay and ArrDelay data, which can be negative. In addition, I skipped the data, which are applied to canceled flights.

Streaming context

As a source for Spark Streaming I decided to use File Streams. Spark Streaming monitors particular folder on HDFS and it begins to process every file which are putted in that directory. To automatically move the files which are needed to be processed I developed a simple script. During stream processing I periodically update state by calling updateStateByKey with checkpointing.

As a pros of file streams is that they do not require running a receiver, like it required for Kafka.

Results

Q 1-1: Rank the top 10 most popular airports by numbers of flights to/from the airport.

1. Load data from the stream context
2. Map to pair <Origin, 1>, <Destination, 1>
3. Reduce by key to prepare <Airport, Number of flights>
4. Transform data and sort by flight count
5. Take top 10 results

Airport	Flights
"ORD"	12446029
"ATL"	11537389
"DFW"	10794558
"LAX"	7721119
"PHX"	6582439
"DEN"	6270398
"DTW"	5635407
"IAH"	5478221
"MSP"	5197627
"SFO"	5168871

Q 1-2: Rank the top 10 airlines by on-time arrival performance.

1. Load data from the stream context
2. Map to pair <Airline, Delay>
3. Group data by key and map to pair to prepare <Airline, Number of flights>
4. Reduce by key to prepare <Airline, Sum of delays>
5. Join <Airline, Number of flights> and <Airline, Sum of delays> and calculate <Airline, Avg delay>
6. Transform and sort data by avg delay
7. Take top 10 result

Airline	Delay
"HA"	-1.01
"AQ"	1.15
"PS"	1.45
"ML"	4.74
"PA"	5.35
"F9"	5.46
"NW"	5.56
"WN"	5.56
"OO"	5.73
"9E"	5.86

Q 2-1: For each airport X, rank the top-10 carriers in decreasing order of on-time departure performance from X.

1. Load data from the stream context
2. Map to pair <{Origin, Carrier}, Delay>
3. Prepare dataset to calculate <{Origin, Carrier}, Flight count>
4. Prepare dataset to calculate <{Origin, Carrier}, Sum delay>
5. Join datasets from 3 and 4 to calculate <{Origin, Carrier}, Avg delay>
6. Map to pair <Origin, {Carrier, Avg delay}>
7. Reduce by key and produce <Origin, Top 10 {Carrier, Avg delay}>

Origin	Minimum delays
"SRQ"	[(-0.38, "TZ"), (-0.09, "RU"), (3.4, "YV"), (3.65, "AA"), (3.95, "UA"), (3.97, "US"), (4.3, "TW"), (4.86, "NW"), (4.87, "DL"), (5.04, "XE")]
"CMH"	[(3.49, "DH"), (3.52, "AA"), (4.04, "NW"), (4.37, "ML"), (4.71, "DL"), (5.2, "PI"), (5.94, "EA"), (5.99, "US"), (6.02, "AL"), (6.1, "RU")]
"JFK"	[(5.07, "RU"), (5.97, "UA"), (8.2, "CO"), (8.74, "DH"), (10.08, "AA"), (11.13, "B6"), (11.53, "PA"), (11.64, "NW"), (11.99, "DL"), (12.41, "AL")]
"SEA"	[(2.71, "OO"), (4.72, "PS"), (5.12, "YV"), (6.01, "AL"), (6.35, "TZ"), (6.43, "US"), (6.5, "NW"), (6.54, "DL"), (6.86, "HA"), (6.94, "AA")]
"BOS"	[(2.12, "RU"), (3.06, "TZ"), (4.45, "PA"), (5.73, "ML"), (7.21, "EV"), (7.25, "NW"), (7.45, "DL"), (8.62, "AL"), (8.69, "US"), (8.73, "AA")]

Q 2-2: For each airport X, rank the top-10 airports in decreasing order of on-time departure performance from X.

1. Load data from the stream context
2. Map to pair <{Origin, Destination}, Delay>
3. Prepare dataset to calculate <{Origin, Destination }, Flight count>
4. Prepare dataset to calculate <{Origin, Destination }, Sum delay>
5. Join datasets from 3 and 4 to calculate <{Origin, Destination }, Avg delay>
6. Map to pair <Origin, {Destination, Avg delay}>
7. Reduce by key and produce <Origin, Top 10 {Destination, Avg delay}>

Origin	Minimum delays
"SRQ"	[(0.0, "EYW"), (1.33, "TPA"), (1.44, "IAH"), (1.7, "MEM"), (2.0, "FLL"), (2.06, "BNA"), (2.36, "MCO"), (2.54, "RDU"), (2.84, "MDW"), (3.36, "CLT")]
"CMH"	[(-5.0, "AUS"), (-5.0, "OMA"), (-5.0, "SYR"), (1.0, "MSN"), (1.1, "CLE"), (1.35, "SDF"), (3.7, "CAK"), (3.94, "SLC"), (4.15, "MEM"), (4.16, "IAD")]
"JFK"	[(-10.5, "SWF"), (0.0, "ABQ"), (0.0, "ANC"), (0.0, "ISP"), (0.0, "MYR"), (1.92, "UCA"), (3.21, "BGR"), (3.61, "BQN"), (4.4, "CHS"), (4.5, "STT")]
"SEA"	[(0.0, "EUG"), (1.0, "PIH"), (2.65, "PSC"), (3.88, "CVG"), (4.26, "MEM"), (5.17, "CLE"), (5.2, "BLI"), (5.38, "YKM"), (5.41, "SNA"), (5.48, "LIH")]
"BOS"	[(-5.0, "SWF"), (-3.0, "ONT"), (1.0, "GGG"), (1.21, "AUS"), (3.05, "LGA"), (3.25, "MSY"), (5.14, "LGB"), (5.78, "OAK"), (5.9, "MDW"), (5.98, "BDL")]

Q 2-4: For each source-destination pair X-Y, determine the mean arrival delay (in minutes) for a flight from X to Y.

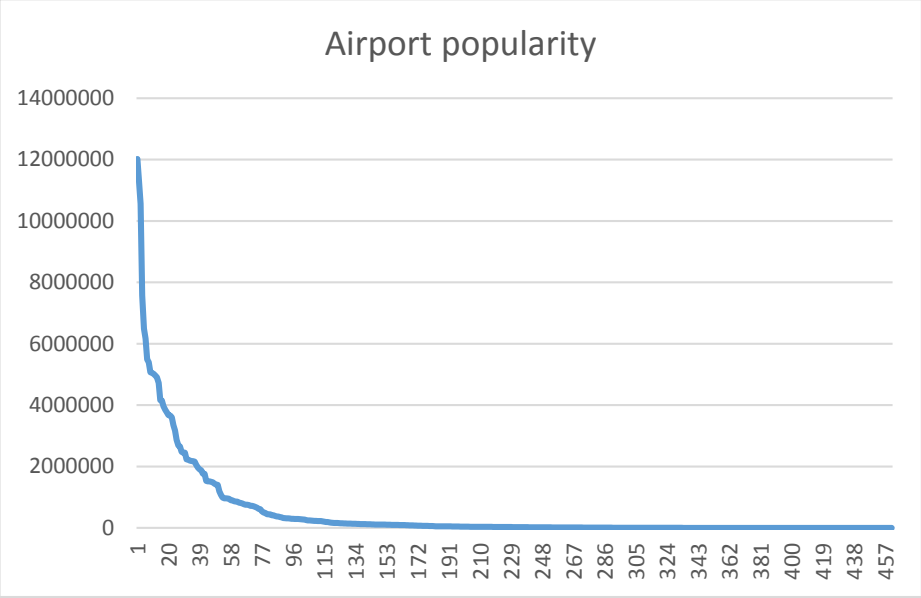
1. Load data from the stream context
2. Map to pair <{Origin, Destination}, Delay>
3. Prepare dataset to calculate <{Origin, Destination }, Flight count>
4. Prepare dataset to calculate <{Origin, Destination }, Sum delay>
5. Join datasets from 3 and 4 to calculate <{Origin, Destination }, Avg delay>

Origin	Destination	Delay
"LGA"	"BOS"	1.48
"BOS"	"LGA"	3.78
"OKC"	"DFW"	5.07
"MSP"	"ATL"	6.74

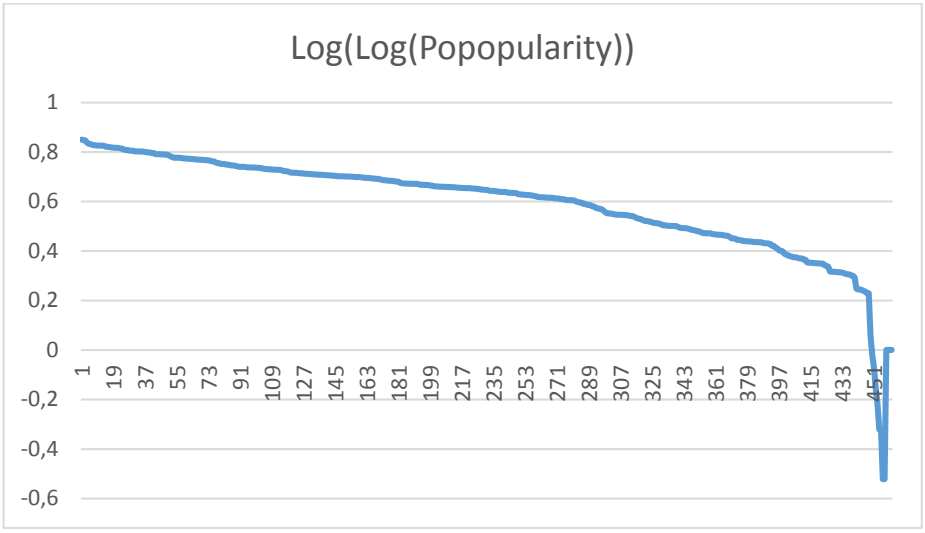
Q 3-1: Does the popularity distribution of airports follow a Zipf distribution? If not, what distribution does it follow?

1. Load data from the stream context
2. Map to pair <Origin, 1>, <Destination, 1>
3. Reduce by key to prepare <Airport, Number of flights>
4. Transform data and sort by flight count

When we have a dataset, we can build a graph of popularity against rank.



To check the Zipf distribution we need to build log-log graph and it seems that graph does not follow Zipf distribution, but Lognormal:



Q 3-2: Tom wants to travel from airport X to airport Z. However, Tom also wants to stop at airport Y for some sightseeing on the way.

1. Load data from the stream context
2. Prepare two data streams: first leg flights which are before 12:00 and second leg flights wich are after 12:00
3. Calculate minimum arrival delay of pair <Origin, Destination> of both data streams
4. Join two streams: the key for the first let is Destination, the key for the second leg is Origin

Route	Origin	Destination	Date (yyyy-mm-dd)	DepTime	Flight	Delay
BOS - ATL - LAX	BOS	ATL	2008-04-03	0548	FL-270	7
	ATL	LAX	2008-04-05	1857	FL-40	-2
PHX - JFK - MSP	PHX	JFK	2008-09-07	1127	B6-178	-25
	JFK	MSP	2008-09-09	1747	NW-609	-17
DFW - STL - LAX	DFW	STL	2008-01-24	0657	AA-1336	-14
	STL	ORD	2008-01-26	1654	AA-2245	-5
LAX – MIA - LAX	LAX	MIA	2008-05-16	0817	AA-280	10
	MIA	LAX	2008-05-18	1925	AA-456	-19

Integration

We can integrate Ambari Hortonworks with various others systems, for example Apache Cassandra. We can easily download additional Cassandra Ambari service, install it from the Ambari dashboard with the help of Service Wizard and make additional setups from the console.

For the first task, I choose Cassandra as data storage, so it was necessary to use Cassandra drivers to connect to the Cassandra instance.

For the second task I choose Amazon DynamoDB, so I use Amazon Client to connect to Amazon instance.

Optimization

Besides infrastructure optimizations of building cluster and changing instance type, I used some other code and system optimizations:

1. For PIG scripts which are used for cleaning the data I used PARALLEL clause.
2. It is necessary to filter data before performing JOIN.
3. Decrease the file size, which are using in file streaming.
4. For task 3.2 use only data for 2008 year.
5. Tuning of YARN, Hadoop, Spark and PIG parameters to allocate more memory and increase parallelism.
6. Script to automatically move files to the file streaming source directory.
7. Send result to Cassandra and DynamoDB to be able perform queries.
8. Run spark-submit with additional parameters to tune memory allocations.
9. Tune batchDuration setting of StreamingContext.

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

The results are useful to analyze performance of airlines and airports and make further forecasts. However, results are hardly depends on cleaning strategy that was chose at the beggining and because of it can be not accurate with others results.

Different stacks comparison

Both Hadoop and Spark are great tools for processing big data sets. However, after task 2 I found that for me the better tool is Spark, because it is possible to use elements of functional programming in distributed calculations. In addition, another advantage of Spark is that it is possible to use stream-based approach and as a source of streaming in this task, I used a file stream. As for speed of calculations, it is hard to say because during stream processing I used different batchDuration time interval at which streaming data will be divided into batches.

Summary

The Capstone was very challenging and sometimes there was not enough time, but obtained practical and theoretical experience is priceless.