Martin Ovington

Cloud Computing Specialization Capstone – Task 1

Overview

The capstone project involves using cloud computing services to process and analyse a publicly available dataset. The dataset is from the Bureau of Transportation Statistics and we will be using data pertaining to airline traffic to answer a series of questions. In this first task we will use *batch processing* techniques to process the data. Specifically, we will use tools that are included in Apache Hadoop toolset and these will be run on the Amazon Web Services (AWS) platform.

A key consideration throughout this project will be the efficient and appropriate use of the various services that are offered by AWS. The services must be combined in a way that maximizes ease of use and minimizes costs. AWS services often have a 'Free Tier', that allow you to use that service up to some limit e.g. storage, run-time etc. and we aim to minimize the costs that are incurred as a result of exceeding these limits.

Cleaning and storing the data

The transportation dataset was initially stored as a 15GB Elastic Block Storage (EBS) snapshot. The snapshot was used to create an Elastic Compute (EC2) volume and was then mounted to an EC2 instance. The 'aviation' subfolder was uploaded to Amazon's cheap, long term storage solution called S3. Amazon S3 offers 5GB of free storage and by uploading only the 3.9GB aviation arrival/departure time data, we remained below this threshold. The data was then downloaded to a local machine, where it could be explored and cleaned before being re-uploaded to S3.

We used a shell script to unzip the many downloaded files on the local machine. The Pandas package for Python was used to clean the files sequentially and produce a cleaned set of files. Records were dropped if the flight was cancelled or they were an exact duplicate of another record. There was a significantly amount of rows dropped due to duplication. Subsequently, all columns that were not necessary for answering the questions for task 1 were dropped. Then, any rows that had any missing values in the remaining columns were dropped. The cleaning procedure reduced the file sizes by a factor of ~7. The files were then all aggregated into one file by appending 30 files worth of data at a time in order to avoid excessive memory usage. The total uncompressed file size was ~5.8GB. The aggregated and cleaned .csv file was uploaded back to S3 storage.

AWS Architecture

Amazon EMR was configured to run with the default three node configuration using m5.xlarge instance types - 4 vCore, 16 GB memory, 64 GB EBS storage. These instances were estimated to have sufficient memory to hold our PySpark dataframes based on testing on a local machine with 16GB memory. One instance operated as the Master node type and the remaining two nodes operated as Core node types. We used EMR release version 5.36.0 and had the following services added to our configuration: Hadoop 2.10.1, JupyterHub 1.4.1, Spark 2.4.8 and Livy 0.7.1. We used a boostrap .sh file to install the necessary Python 3 modules on our instances.

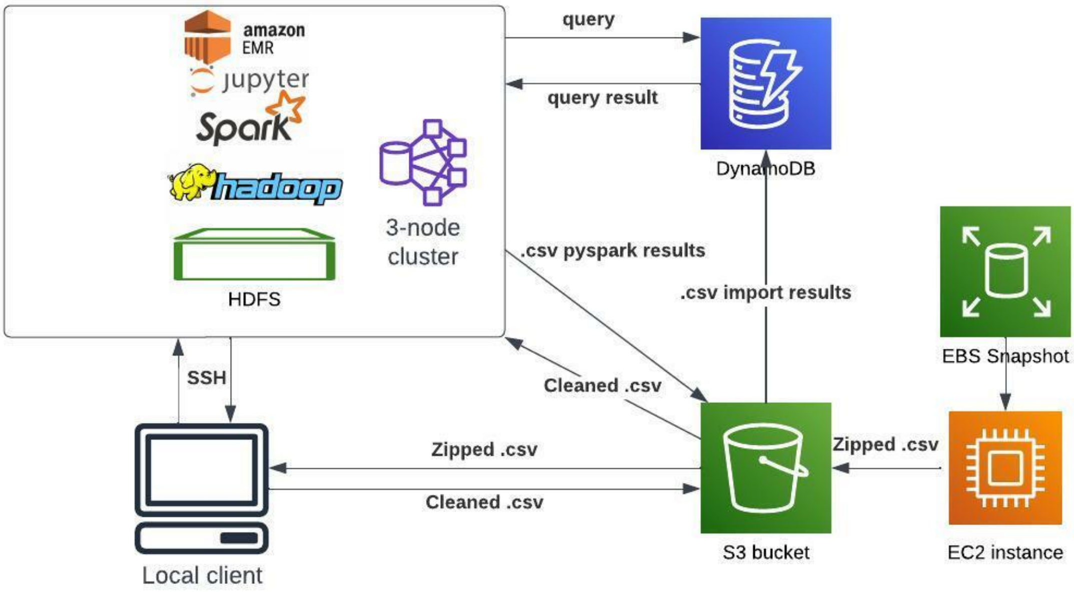


Figure 1. AWS architecture with data flow

General Data Analysis Strategy

We decided to use PySpark within a Jupyter Notebook to do our data analysis, due to familiarity with the Python programming language. The cleaned data was uploaded to the EMR cluster from S3 and was loaded into a PySpark dataframe. Once the data was in a dataframe we could perform necessary data manipulation to answer the questions for Task 1. Before launching the cluster, the Python code was tested locally on a smaller data subset to ensure correctness and to informally observe the performance of different methods. The strategy helped to reduce the amount of testing required once the cluster was running, which had the result of reducing the total running time of the cluster so that costs could be minimized. It was found that using the .show() method on dataframes was slow, so was used whilst debugging and otherwise only when necessary to produce the final result. Similarly, when producing .csv files of the results for the necessary questions, using the .coalesce(1) and .toPandas() functions required higher processing time and could not be used on Question 3.2 due the size of the dataframe. Instead, on this last part we used a separate Python script to combine the .csv files produced by the .to_csv() method, which was still slow. Producing these .csv files was necessary to import the data into DynamoDB due to difficulty with transferring the data directly from the PySpark dataframes. In this way, using Hive would have been simpler, but would have required more time to gain familiarity with.

We found that some of the results differed slightly from the published reference result, although generally not by much. The difference is likely due to the presence of several corrupted .csv files from the 2008 series and due to a difference in cleaning techniques. In particular, we removed many duplicate flight records from the initial data set and it could be that this was not done with the reference results.

Group 1 Question 1

```
df_pyspark.groupBy('Origin').count().orderBy('count', ascending = [False]).show(n=10)
```

Group 1 Question 2

```
df_pyspark.groupBy('Carrier').avg('ArrDelay').orderBy(['avg(ArrDelay)'], ascending = [True]).show(n=10)
```

Group 1 Question 3

```
df_pyspark.groupBy('DayOfWeek').avg('ArrDelay').orderBy(['avg(ArrDelay)'], ascending = [True]).show(n=10)
```

G1Q1

Origin	Count
ORD	5998002
ATL	5659166
DFW	5270525
LAX	3788336
PHX	3246442
DEN	3078009
DTW	2738373
IAH	2700746
MSP	2533996
SFO	2530684

G1Q3

DayOfWeek	avg(ArrDelay)
Saturday	4.30165
Tuesday	5.990445
Sunday	6.61328
Monday	6.716091
Wednesday	7.203647
Thursday	9.094438
Friday	9.720996

G1Q2

Carrier	avg(ArrDelay)
HA	-1.0118
AQ	1.156923
PS	1.450639
ML	4.747609
PA	5.322431
F9	5.466049
NW	5.557783
WN	5.560774
OO	5.735951
9E	5.867185

Group 2 Question 1

```
window = Window.partitionBy('Origin').orderBy('avg(DepDelay)')
df_dep_perf_by_carrier = df_pyspark.groupBy('Origin', 'Carrier').avg('DepDelay')\
.withColumn('row_number', pyfuncs.row_number().over(window))
df_dep_perf_by_carrier = df_dep_perf_by_carrier.filter(df_dep_perf_by_carrier['row_number'] <= 10).drop('row_number')
```

```
q2_airports = ['CMI', 'BWI', 'MIA', 'LAX', 'IAH', 'SFO']
df_dep_perf_by_carrier.filter(df_dep_perf_by_carrier.Origin.isin(q2_airports)).show(n=100)
```

BWI

Carrier	avg(DepDelay)
F9	0.746493
PA	4.761905
CO	5.113806
YV	5.504666
NW	5.655729
AL	5.728207
AA	5.913444
9E	7.220703
US	7.471462
DL	7.641127

CMI

Carrier	avg(DepDelay)
OH	0.612005
US	1.82441
TW	3.744589
PI	4.369928
DH	6.08016
EV	6.665138
MQ	7.992782

IAH

Carrier	avg(DepDelay)
NW	3.515589
PA	3.901583
PI	3.952167
RU	4.778435
AL	4.962449
US	5.031701
F9	5.549463
AA	5.616927
TW	6.027508
WN	6.213109

LAX

Carrier	avg(DepDelay)
RU	1.948387
MQ	2.399978
OO	4.214037
FL	4.687786
TZ	4.758884
PS	4.805362
NW	5.092928
F9	5.719679
HA	5.817645
YV	6.036279

MIA

Carrier	avg(DepDelay)
9E	-3.66667
EV	1.202643
RU	1.283429
TZ	1.782244
XE	2.745645
PA	4.026508
NW	4.423478
US	5.990862
UA	6.808275
ML	7.512146

SFO

Carrier	avg(DepDelay)
TZ	3.935139
MQ	4.782694
F9	5.155881
PA	5.283914
NW	5.717548
PS	6.256748
DL	6.512475
CO	7.047357
US	7.347713
TW	7.763071

Group 2 Question 2

```
window = Window.partitionBy('Origin').orderBy('avg(DepDelay)')
df_dep_perf_by_dest = df_pyspark.groupBy('Origin', 'Dest').avg('DepDelay')\
.withColumn('row_number', pyfuncs.row_number().over(window))
df_dep_perf_by_dest = df_dep_perf_by_dest.filter(df_dep_perf_by_dest['row_number'] <= 10).drop('row_number')
```

```
q2_airports = ['CMI', 'BWI', 'MIA', 'LAX', 'IAH', 'SFO']
df_dep_perf_by_dest.filter(df_dep_perf_by_carrier.Origin.isin(q2_airports)).show(n=100)
```

BWI

Dest	avg(DepDelay)
MLB	1.161473
DAB	1.508475
SRQ	1.550026
IAD	1.797203
UCA	3.325903
CHO	3.744928
BGM	3.795134
DCA	3.970642
GSP	4.192835
SJU	4.203397

CMI

Dest	avg(DepDelay)
PIT	1.107826
CVG	1.902128
DAY	2.928049
PIA	3.412568
STL	3.841727
DFW	5.978074
ATL	6.665138
ORD	8.163504

LAX

Dest	avg(DepDelay)
SDF	-16
LAX	-2
BZN	-0.72727
MAF	0
MFE	1.177515
IYK	1.269825
MEM	1.867979
PIE	1.926209
SNA	2.094492
SBA	2.274377

IAH

Dest	avg(DepDelay)
MSN	-2
MLI	-1
AGS	-0.6173
EFD	1.887708
HOU	2.172037
JAC	2.422619
MTJ	2.902516
RNO	3.227283
GUC	3.537374
BPT	3.599533

SFO

Dest	avg(DepDelay)
SDF	-10
MSO	-4
LGA	-1.75758
PIE	-1.34104
OAK	-0.81371
FAR	0
BNA	2.379209
MEM	3.262416
SJC	3.985614
MKE	3.988028

MIA

Dest	avg(DepDelay)
SAN	1.710383
BUF	2
SLC	2.410788
HOU	2.888491
ISP	3.649123
MEM	3.686327
GNV	3.960685
PSE	3.975845
TLH	4.234642
MCI	4.612245

Group 2 Question 4

```
df_g2q4 = df_pyspark.groupBy('Origin', 'Dest').avg('ArrDelay').orderBy(['avg(ArrDelay)'], ascending = [True])
q4_pairs = [('CMI', 'ORD'), ('IND', 'CMH'), ('DFW', 'IAH'), ('LAX', 'SFO'), ('JFK', 'LAX'), ('ATL', 'PHX')]
for pair in q4_pairs:
    print(pair)
    df_g2q4.filter((df_g2q4['Origin'] == pair[0]) & (df_g2q4['Dest'] == pair[1])).show(n=10)
```

Origin	Dest	avg(ArrDelay)
CMI	ORD	10.14366
IND	CMH	2.899904
DFW	IAH	7.654443
LAX	SFO	9.589283
JFK	LAX	6.635119
ATL	PHX	9.021342

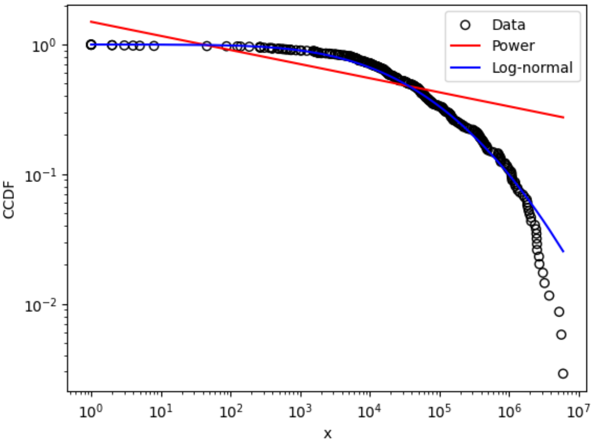
Group 3 Question 1

```
airport_freqs = df_pyspark.groupBy('Origin').count().select("count").rdd.flatMap(lambda x: x).collect()
```

```
x = np.sort(airport_freqs)
n = len(airport_freqs)
y = 1 - np.arange(n) / float(n)

def power_func(x, a, b):
    return a * x ** (-b)

popt, _ = curve_fit(power_func, x, y)
power_y = power_func(x, *popt)
plt.scatter(x, y, marker='o', facecolors='none', edgecolors='black')
plt.plot(x, power_y, color='r')
shape, loc, scale = lognorm.fit(x, 2)
lognorm_y = lognorm.sf(x, shape, loc=loc, scale=scale)
plt.plot(x, lognorm_y, color='b')
plt.xlabel('x')
plt.ylabel('CCDF')
plt.yscale('log')
plt.xscale('log')
plt.legend(['Data', 'Power', 'Log-normal'])
print('Data - Power Kolmogorov-Smirnov:')
print(ks_2samp(y, power_y))
print('Data - Log-normal Kolmogorov-Smirnov:')
print(ks_2samp(y, lognorm_y))
print('CCDF of data and distributions:')
```




```
Data - Power Kolmogorov-Smirnov:
KstestResult(statistic=0.28034682080924855, pvalue=2.2333610336795968e-12)
Data - Log-normal Kolmogorov-Smirnov:
KstestResult(statistic=0.06647398843930635, pvalue=0.42964381800204454)
```

We plotted the Complementary Cumulative Distribution Function (CCDF) of the number of flights from each airport using log-log scales. Using the SciPy library we fitted a Power-law and a Log-normal curve to the airport popularity data. From the plot we can clearly see that the CCDF does not follow the Power-law curve, but does seem to follow a Log-normal curve. We confirmed this using a Kolmogorov-Smirnov test which had a P-value of < 0.05 for the Power-law test, but a P-value of 0.43

for the Log-normal curve. Using a cut-off of P = 0.05, we reject the null hypothesis for the Power-law curve and conclude that it is significantly different to the data. We fail to reject the null hypothesis for the log-normal curve and conclude that it is not significantly different to the data.

Group 3 Question 2

```
df_pyspark = df_pyspark.withColumn('Year', date_format('FlightDate', 'y'))
df_2008 = df_pyspark.filter(df_pyspark['Year'] == 2008).drop('Year')
df_2008.show(n=10)
w = Window.partitionBy('FlightDate', 'Origin', 'Dest')

df_bestXY = df_2008.alias('df1').filter(df_2008['DepTime'] < 1200)
df_bestXY = df_bestXY.withColumn('minArrDelay', pyfuncs.min('ArrDelay').over(w))\
.where(col('ArrDelay') == col('minArrDelay')).drop('minArrDelay')
df_bestYZ = df_2008.alias('df2').filter(df_2008['DepTime'] >= 1200)
df_bestYZ = df_bestYZ.withColumn('minArrDelay', pyfuncs.min('ArrDelay').over(w))\
.where(col('ArrDelay') == col('minArrDelay')).drop('minArrDelay')
col_names = df_2008.columns
df_bestXY = df_bestXY.selectExpr([col + ' as XY_' + col for col in col_names])
df_bestYZ = df_bestYZ.selectExpr([col + ' as YZ_' + col for col in col_names])
df_g3q2 = df_bestXYZ\
.select('XY_FlightDate', 'XY_Origin', 'XY_Dest', 'XY_ArrDelay', 'YZ_FlightDate', 'YZ_Origin', 'YZ_Dest', 'YZ_ArrDelay')
df_g3q2.write.option('header', 'true').csv("./g3q2")
routes = [
    ('CMI', 'ORD', 'LAX', datetime.datetime(2008, 3, 4, 0, 0, 0)),
    ('JAX', 'DFW', 'CRP', datetime.datetime(2008, 9, 9, 0, 0, 0)),
    ('SLC', 'BFL', 'LAX', datetime.datetime(2008, 4, 1, 0, 0, 0)),
    ('LAX', 'SFO', 'PHX', datetime.datetime(2008, 7, 12, 0, 0, 0)),
    ('DFW', 'ORD', 'DFW', datetime.datetime(2008, 6, 10, 0, 0, 0)),
    ('LAX', 'ORD', 'JFK', datetime.datetime(2008, 1, 1, 0, 0, 0))
]

for route in routes:
    df_bestXYZ.where((df_bestXYZ.XY_FlightDate == route[3]) & (df_bestXYZ.XY_Origin == route[0]) \
                   & (df_bestXYZ.XY_Dest == route[1]) & (df_bestXYZ.YZ_Dest == route[2])).show()
```

XY_FlightDate	XY_Carrier	XY_Origin	XY_Dest	XY_DepTime	XY_ArrDelay	YZ_FlightDate	YZ_Carrier	YZ_Dest	YZ_DepTime	YZ_ArrDelay
04/03/2008	MQ	CMI	ORD	710	-14	06/03/2008	AA	LAX	1952	-24
09/09/2008	AA	JAX	DFW	722	1	11/09/2008	MQ	CRP	1648	-7
01/04/2008	OO	SLC	BFL	1101	12	03/04/2008	OO	LAX	1509	6
12/07/2008	WN	LAX	SFO	650	-13	14/07/2008	US	PHX	1916	-19
10/06/2008	UA	DFW	ORD	658	-21	12/06/2008	AA	DFW	1650	-10
01/01/2008	UA	LAX	ORD	700	1	03/01/2008	B6	JFK	1853	-7

Conclusion

Using Jupyter notebooks and PySparks was a convenient and easy to use methodology. By using the local machine to extensively test code before using it on the EMR cluster we significantly reduced running costs. The downside of this methodology was the minor difficulty in getting the results that had to be placed into persistent storage to DynamoDB. Using intermediate .csv files, uploading them to s3 and then importing them to DynamoDB was not difficult, but it was a relatively slow procedure. Other options included using boto3's batch writing, which would require sequentially going through the PySpark dataframes, which would also be slow. Using Hive may have been more efficient in this regard, but unfamiliarity with the framework made this a less attractive option.