Martin Ovington

Cloud Computing Specialization Capstone – Task 2

Overview

The second task of the capstone project follows on from Task 1, where we utilized Amazon Web Services (AWS) to batch process an air traffic dataset from the Bureau of Transportation Statistics. In this part we answered the same set of questions with slightly different parameters, but this time we used *stream processing*. Answering the same questions using stream processing allowed us to compare the two techniques. Specifically, we compared differences in performance and differences in the ease of use of the two frameworks. In Task 1 we utilized Apache Spark with the Python 3 API through PySpark and we do the same in Task 2 by using Spark Structured Streaming[1]. Similarly to Task 1, we aimed to optimize not only for performance, but also for the cost of using the various AWS services.

The performance comparisons were treated somewhat informally due to the artificial nature of the comparison. Given that the full dataset was available from the beginning, batch processing will always be the superior for this purpose and we are simply putting in place an artificial bottleneck in place in order to test the streaming framework, which will be discussed in more detail throughout this report. Running a more formal comparison would require the computation to be repeated multiple times under similar circumstances, which is hindered by the experiment being run on shared cloud infrastructure and the necessity of running slightly different queries on each due to limitations in the output mode/format options when using Spark Streaming.

Cleaning and storing the data

The transport dataset was retrieved from EBS snapshot and cleaned locally and this was described in the task 1 document, so will not be repeated here. We decided to use the Apache Parquet[2] file format for our cleaned dataset, rather than CSV as we did in Task 1. The Parquet format is a compressed and efficient columnar format that is designed by Apache for Hadoop. We had 45 separate Parquet file parts, which allowed us to ingest files separately and prevented some of the memory over-usage crashes that were experienced using the streaming framework. The Parquet format required only 1GB of storage compared to 5.7GB for the CSV format, which reduced the upload time from the local machine to Amazon Simple Storage S3 as well as reducing the network traffic required from S3 to the Amazon Elastic MapReduce (EMR) cluster. The lower storage requirement reduced the amount of S3 storage used and kept the total storage space used below the 6GB free tier limit.

Using the Parquet format significantly improve the performance of the queries. This was also observed to be true when the task 1 queries were re-tested, so for all performance comparisons we repeated the task 1 queries using the Parquet format. The improved performance meant that the AWS cluster did not need to run for as long, which reduced the cost of using this service.

AWS Architecture

The architecture used is much the same as in Task 1, with some minor changes. The EMR cluster was comprised of three m5.xlarge instances with 4 vCore, 16 GB memory, 64 GB EBS storage. The default application configuration was used from the Create notebook menu, in order to simplify the configuration. These included Hadoop, Spark, Livy, Hive and JupyterEnterpriseGateway.

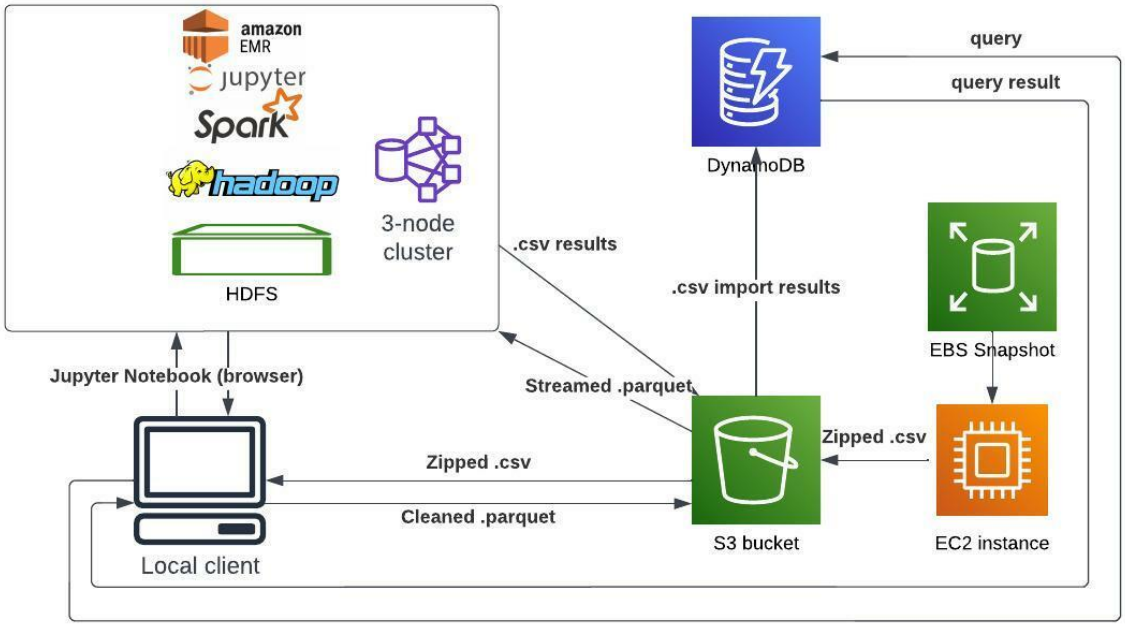

Figure 1 AWS architecture with data flow

Usability Comparison

The Spark Structured Streaming PySpark functions are much the same as their batch processing variants. Setting up the stream required some trial and error due to memory-overuse problems and was relatively straight forward, with the requirement that the stream must be defined using a schema and having to specify how many files should be read in at a time. Setting this option through 'maxFilesPerTrigger', eight was found to be a reasonable configuration that avoided memory-overuse of higher values and the poorer performance of lower values. Queries were performed on the variable that defines the incoming stream using much the same API as batch processing and these are performed using lazy evaluation as with batch processing.

The key differences become apparent when the queries have to be evaluated in order to produce output. When batch processing, the algorithms used have the advantage of being able to look through the entire dataset from the start of the query. With stream processing, the algorithms do not have the ability to see the entire dataset and do not, at least by default, know when the data stream will end. There are some options to timestamp the incoming data and essentially mark a cut-off point, but in our use case this would require predicting how long it would take to read in all the files, which would be tricky as it risks leaving out data by setting the wait time to be too low or hurting performance by setting the wait time to be too high.

The result of these limitations meant that Spark imposes restrictions on which output mode/format combination can be used with particular types of queries. For example, if a query requires sorting, then it is clear that a file cannot be written sequentially as the data arrives, because data incoming later may need to be written before earlier data. Similarly, with aggregation queries, later results may need to be combined with earlier results, so these cannot be written to a file prematurely. Technically, these limitations could be overcome in at least in some file formats, by for example simply overwriting earlier parts of a file, but Spark likely imposes restrictions in order use cases where the algorithm used would be prohibitively inefficient, for example requiring many small, scattered re-writes to a file. With outputting to the console, there are slightly less restrictions as the query can be outputted in 'complete' or 'update' mode where it simply re-outputs the entire or updated results as more micro-batches are processed.

Many of the queries were re-written in order to account for the limitations and generally the output to the console was more cluttered due to there being five batches (45 files in batches of eight) to process for each query. In order to write the CSV files required for some of the questions, the dataframe had to be first written to memory as a table, an output format which had fewer restrictions than the other formats. Once this was done the table was then converted back to a non-streaming dataframe, which could be written to a CSV file without limitation. Uploading the CSV files to S3 and then importing to DynamoDB was then the same as in Task 1.

Performance Comparison

In the final form of the queries, the general impression was the length of processing time for the streaming queries was comparable to that in batch processing. However, getting to this stage required some manual optimizing, trial and error and restructuring of the queries. Setting the 'maxFilesPerTrigger' too low made the queries prohibitively slow, to the point where they often had to be terminated as they seemed to make little progress. Some of the questions where specific routes, origins, destinations etc. were required, the for-loop that ran the specific filters was done after a non-streaming dataframe was produced in order to have acceptable performance. The technique of getting the CSV files by writing to memory first was acceptable in terms of performance and did not seem too different to the Task 1 file writing. Locally, a modified version of the Task 1 notebook took 179 seconds to run completely, whereas the Task 2 notebook took notable longer at 313 seconds. On a one node cluster, the Task 1 notebook took 735 seconds and the Task 2 notebook took 646 seconds. It appeared that the Task 1 Group 2 Question 2 query performed very poorly on EMR, although this could have been due to a change made to the query to make it equivalent to how the same Task 2 query was run. The poorer performance for both tasks on EMR vs. the local machine was likely due to the bottleneck of needing to get data from S3 rather than having the data on a local SSD and the cluster instance having slower hardware than the local machine.

Stream setup

```
spark = SparkSession.builder.config("spark.driver.memory", "12g").config('spark.driver.maxResultSize', '8g')\
.appName('readfromparquet').getOrCreate()
spark
```

```
airport_schema = StructType([StructField('id', IntegerType(), True),
                  StructField('FlightDate', TimestampType(), True),
                  StructField('Carrier', StringType(), True),
                  StructField('Origin', StringType(), True),
                  StructField('Dest', StringType(), True),
                  StructField('DepTime', DoubleType(), True),
                  StructField('ArrTime', DoubleType(), True),
                  StructField('DepDelay', DoubleType(), True),
                  StructField('ArrDelay', DoubleType(), True)])
```

```
airports = spark.readStream.format('parquet').schema(airport_schema).option('header', True)\
.option('maxFilesPerTrigger', 8).load('s3://aws-emr-resources-926489384541-us-east-1/airline_data')
```

Group 1 Question 1

```
query_g1q1 = airports.groupBy('Origin').count().orderBy('count', ascending = [False])
```

```
query_g1q1\
.writeStream \
.outputMode("complete") \
.format("console") \
.start()
```

G1Q1

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

Group 1 Question 2

```
query_g1q2 = airports.groupBy('Carrier').avg('ArrDelay').orderBy(['avg(ArrDelay)'], ascending = [True])
```

```
query_g1q2\
.writeStream \
.outputMode("complete") \
.format("console") \
.start()
```

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)')
query_g2q1 = airports.groupBy('Origin', 'Carrier').avg('DepDelay')\
.orderBy(['Origin', 'avg(DepDelay)'], ascending = [True, True])
```

```
write_query_g2q1 = query_g2q1.writeStream.queryName('g2q1_table')\
.outputMode('complete')\
.format('memory')\
.start()
```

```
df_g2q1 = spark.sql('select * from g2q1_table')
df_g2q1.coalesce(1).write.save('./g2q1_task2.csv', format='csv', header=True)
```

```
q2_airports = ['SRQ', 'CMH', 'JFK', 'SEA', 'BOS']
```

```
for airport in q2_airports:
    df_g2q1.filter(df_g2q1['Origin'] == airport).show(n=10)
```

G2Q1

SRQ		CMH		JFK		SEA		BOS	
Carrier	avg(DepDelay)	Carrier	avg(DepDelay)	Carrier	avg(DepDelay)	Carrier	avg(DepDelay)	Carrier	avg(DepDelay)
TZ	-0.382	AA	3.454508	RU	5.042491	OO	2.689929	RU	1.917834
RU	-0.10727	DH	3.49535	UA	5.947758	PS	4.692201	TZ	3.065583
AA	3.292173	NW	3.989499	CO	8.15346	YV	5.122263	PA	4.44195
YV	3.404022	ML	4.369852	DH	8.750582	AL	6.016399	ML	5.686495
UA	3.83052	DL	4.670345	AA	10.01494	TZ	6.345004	NW	7.141695
US	3.94595	PI	5.185142	B6	11.11261	US	6.398544	EV	7.264984
TW	4.206478	EA	5.315813	PA	11.29325	NW	6.466075	DL	7.412062
NW	4.66595	US	5.833841	NW	11.55464	DL	6.499248	EA	8.094742
DL	4.833183	AL	5.952381	DL	11.90219	HA	6.684251	AL	8.431211
XE	5.109375	RU	6.102676	AL	12.37893	AA	6.866121	US	8.543394

Group 2 Question 2

```
window = Window.partitionBy('Origin').orderBy('avg(DepDelay)')
query_g2q2 = airports.groupBy('Origin', 'Dest').avg('DepDelay').orderBy(['avg(DepDelay)'], ascending = [True])
```

```
query_g2q2.writeStream.queryName('g2q2_table')\
.outputMode('complete')\
.format('memory')\
.start()
```

```
df_g2q2 = spark.sql('select * from g2q2_table')
df_g2q2.coalesce(1).write.save('./g2q2_task2.csv', format='csv', header=True)
```

```
for airport in q2_airports:
    df_g2q2.filter(df_g2q2['Origin'] == airport).show(n=10)
```

G2Q2

SRQ		CMH		JFK		SEA		BOS	
Dest	avg(DepDelay)	Dest	avg(DepDelay)	Dest	avg(DepDelay)	Dest	avg(DepDelay)	Dest	avg(DepDelay)
EYW	0	AUS	-5	SWF	-10.5	EUG	0	SWF	-5
TPA	0.674123	OMA	-5	ABQ	0	PSC	2.594943	ONT	-4
IAH	1.460621	SYR	-5	ANC	0	CVG	3.789409	AUS	1.216609
MEM	1.6875	CLE	1.04727	MYR	0	MEM	4.210309	LGA	3.031499
FLL	2	SDF	1.352941	ISP	0	CLE	5.154082	MSY	3.283912
BNA	2.068862	CAK	3.497368	UCA	1.90795	SNA	5.195117	LGB	4.986319
MCO	2.313877	SLC	3.922939	BGR	3.21028	BLI	5.199197	OAK	5.393519
RDU	2.44759	IND	4.140255	BQN	3.614684	YKM	5.379648	MDW	5.837131
MDW	2.777999	DFW	4.149068	STT	4.336942	LIH	5.481081	DCA	5.888031
RSW	3.351498	MEM	4.157089	CHS	4.339147	DTW	5.650895	BDL	5.957978

Group 2 Question 4

```
query_g2q4 = airports.groupBy('Origin', 'Dest').avg('ArrDelay')\
.orderBy(['Origin', 'avg(ArrDelay)'], ascending = [True, True])
```

```
query_g2q4.writeStream.queryName('g2q4_table')\
.outputMode('complete')\
.format('memory')\
.start()
```

```
df_g2q4 = spark.sql('select * from g2q4_table')
df_g2q4.coalesce(1).write.save('./g2q4_task2.csv', format='csv', header=True)
```

```
q4_pairs = [('LGA', 'BOS'), ('BOS', 'LGA'), ('OKC', 'DFW'), ('MSP', 'ATL')]
for pair in q4_pairs:
    df_g2q4.filter((df_g2q4['Origin'] == pair[0]) & (df_g2q4['Dest'] == pair[1]))\
    .show()
```

G2Q4

Origin	Dest	avg(ArrDelay)
LGA	BOS	1.483865
BOS	LGA	3.784118
OKC	DFW	4.969055
MSP	ATL	6.737008

Group 3 Question 2

```
query_2008 = airports.withColumn('Year', date_format('FlightDate', 'y'))
query_2008 = query_2008.filter(query_2008['Year'] == 2008).drop('Year')
```

```
query_XY = query_2008.filter(query_2008['DepTime'] < 1200)
query_YZ = query_2008.filter(query_2008['DepTime'] >= 1200)
```

```
col_names = query_2008.columns
query_XY = query_XY.selectExpr([col + ' as XY_' + col for col in col_names])
query_YZ = query_YZ.selectExpr([col + ' as YZ_' + col for col in col_names])
```

```
query_XYZ = query_XY\
.join(query_YZ, (col('XY_Dest') == col('YZ_Origin')) & (datediff(col('YZ_FlightDate'), col('XY_FlightDate')) == 2))
```

```
query_g3g2 = query_XYZ\
.select('XY_FlightDate', 'XY_Origin', 'XY_Dest', 'XY_ArrDelay', 'YZ_FlightDate', 'YZ_Origin', 'YZ_Dest', 'YZ_ArrDelay')\
.withColumn('TotalArrDelay', query_XYZ.XY_ArrDelay + query_XYZ.YZ_ArrDelay)\
.filter(((query_XYZ.XY_FlightDate == datetime.datetime(2008, 4, 3, 0, 0, 0)) & (query_XYZ.XY_Origin == 'BOS') \
                    & (query_XYZ.XY_Dest == 'ATL') & (query_XYZ.YZ_Dest == 'LAX')) \
| ((query_XYZ.XY_FlightDate == datetime.datetime(2008, 9, 7, 0, 0, 0)) & (query_XYZ.XY_Origin == 'PHX') \
                    & (query_XYZ.XY_Dest == 'JFK') & (query_XYZ.YZ_Dest == 'MSP')) \
| ((query_XYZ.XY_FlightDate == datetime.datetime(2008, 1, 24, 0, 0, 0)) & (query_XYZ.XY_Origin == 'DFW') \
                    & (query_XYZ.XY_Dest == 'STL') & (query_XYZ.YZ_Dest == 'ORD')) \
| ((query_XYZ.XY_FlightDate == datetime.datetime(2008, 5, 16, 0, 0, 0)) & (query_XYZ.XY_Origin == 'LAX') \
                    & (query_XYZ.XY_Dest == 'MIA') & (query_XYZ.YZ_Dest == 'LAX'))) \
.writeStream.queryName('g3q2_table')\
.outputMode('append')\
.format('memory')\
.start()
```

```
df_g3q2 = spark.sql('select * from g3q2_table').orderBy('TotalArrDelay')
df_g3q2.coalesce(1).write.save('./g3q2_task2.csv', format='csv', header=True)
```

```
routes = [
    ('BOS', 'ATL', 'LAX', datetime.datetime(2008, 4, 3, 0, 0, 0)),
    ('PHX', 'JFK', 'MSP', datetime.datetime(2008, 9, 7, 0, 0, 0)),
    ('DFW', 'STL', 'ORD', datetime.datetime(2008, 1, 24, 0, 0, 0)),
    ('LAX', 'MIA', 'LAX', datetime.datetime(2008, 5, 16, 0, 0, 0)),
]

for route in routes:
    df_g3q2.filter((df_g3q2.XY_FlightDate == route[3]) & (df_g3q2.XY_Origin == route[0]) \
                    & (df_g3q2.XY_Dest == route[1]) & (df_g3q2.YZ_Dest == route[2])) \
    .orderBy('TotalArrDelay') \
    .show(n=1)
```

G3Q2

XY_FlightDate	XY_Origin	XY_Dest	XY_ArrDelay	YZ_FlightDate	YZ_Dest	YZ_ArrDelay	TotalArrDelay
03/04/2008	BOS	ATL	7	05/04/2008	LAX	-2	5
07/09/2008	PHX	JFK	-25	09/09/2008	MSP	-17	-42
24/01/2008	DFW	STL	-14	26/01/2008	ORD	-5	-19
16/05/2008	LAX	MIA	10	18/05/2008	LAX	-19	-9

Conclusion

Using Spark Structured Streaming was more difficult than Spark batch processing, largely due to unfamiliarity with the nuances of the limitations there were on producing output. There were issues early on in development with memory over-use and performance problems, but these became non-issues once the proper configurations were found. Given the size of the dataset, the queries that needed to be run and the infrastructure used, the performance from the perspective of an end user appeared to be similar to batch processing, but differences may have become more apparent with a larger dataset. For the actual use case of having the data available from the start of processing, batch processing is clearly superior given the more efficient algorithms that it can use and offers greater flexibility in output formats with the need for the tricks that had to be used to accomplish this with the streaming framework. Yet, even given the artificial nature of the setup for streaming, the experience of experimenting with streaming would be very valuable for use cases where data would be entering the system at different time intervals and up to date data analysis was desired.

References

[1] Apache Structured Streaming Programming Guide
https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[2] Apache Parquet File Format
https://parquet.apache.org/