Nathaniel Rupsis

CS-598 Cloud Computing Capstone

Task 2

Video Link: https://mediaspace.illinois.edu/media/t/1_hdb917l1

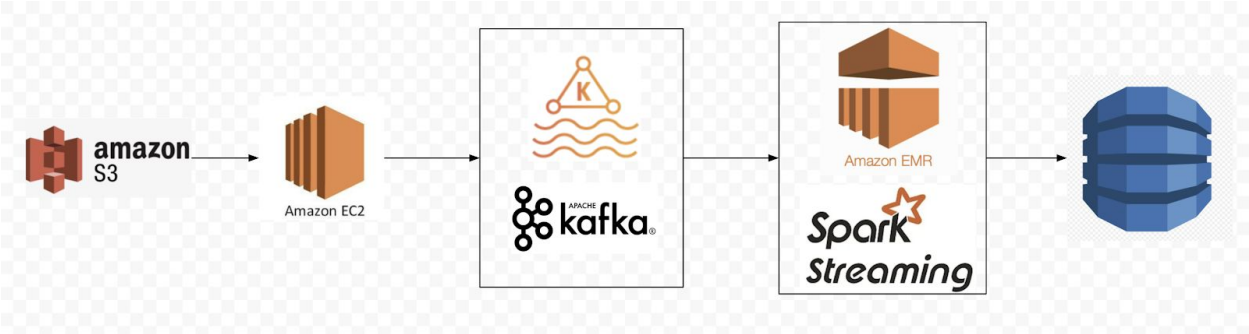
Data Extraction and Cleaning

For task 2, the data has already been downloaded, cleaned, and injected into S3 per task one.

System Integration

For task 2, my streaming implementation of choice is Spark streaming. I've used spark in CS-498 Cloud Computing Applications, so the decision to use spark was made out of familiarity, rather than technical ability / specification.

Figure 1 showcases the overall system pipeline:



(Figure 1)

S3 stores all of the cleaned data, which is then read by an EC2 instance. The EC2 instance acts as a producer, reading in each line of the clean data, parsing, serializing, and then producing kafka events onto the kafka cluster. (see appendix A)

The kafka cluster is a managed kafka service using AWS's MSK service. This service makes it incredibly easy to build and scale kafka clusters, and allows for a substantial amount of flexibility within the system.

Once the data is in kafka, I've utilized AWS's EMR service to run Spark stream, and the

managed jupyter notebooks to implement the logic needed to answer each question.

The final part of the system is to save the answers in persistent storage. For this I am using DynamoDB.

Solution Approach

For each question, the initial spark context set up is identical:

```
  conf.setAppName("Problem_2-1")

  conf.set("spark.streaming.kafka.maxRatePerPartition", 50000)

  conf.set("spark.executor.memory", "2g")

  conf.set("spark.python.worker.memory", "1g")

  sc = SparkContext(conf=conf)

  sc.setLogLevel("WARN")

  ssc = StreamingContext(sc, 2)

  ssc.checkpoint("/tmp/streaming")



  brokers = "******"

  topic = "cs598-task2"kafka_consumer_group = str(uuid.uuid4())

  kafka_client_params = {

      "metadata.broker.list": brokers,

      "group.id": kafka_consumer_group,

      "auto.offset.reset": "smallest",

  }

  events = KafkaUtils.createDirectStream(ssc, [topic], kafka_client_params)

  parsed = events.map(lambda line: json.loads(line[1]))
```

Additionally, for saving to DynamDB, the same helper method looks almost identical:

```
def save_results(items):

  dynamodb = boto3.resource('dynamodb', region_name="us-east-1")
```

```
    table = dynamodb.Table('cs598_question_2_1')

    for item in items:

        origin = item['origin']

        title = item['carrier']

        avg_depdelay = item['avg_depdelay']

        table.put_item(Item=item)
```

To save space on the report, I'll be posting the "Query" portions of the solutions

Question 2.1

For question 2.1 & 2.2, aggregate based on average departure delays and grab top 10

```
    parsed = events.map(lambda line: json.loads(line[1]))

    # Filter based on airport list
    filtered_airports = parsed.filter(lambda item: item['Dest'] in airports)

    # compute averages
    filtered_airports = filtered_airports.updateStateByKey(calculateAverage)

    # for each airport, save to ten items
    for airport in airports:
        saveTopCarriers(filtered_airports.filter(lambda item: item['Dest'] == airport))

    print("Completed")
```

```
def saveTopCarriers(carriers):
    sorted = carriers.sortByKey(True)
    for toSave in sorted.take(10):
        save_results(toSave)


def calculateAverage(newVal, accumlativeAvg):
    if accumlativeAvg is None:
        accumlativeAvg = (0.0, 0, 0.0)
    total = sum(newVal, accumlativeAvg[0])
    count = accumlativeAvg[1] + len(newVal)
    avg = total / float(count)
    return (prod, count, avg)
```

Results:

Airport Code	Carrier	Average Departure Delay
CMI	1) OH 2) US 3) TW 4) PI 5) DH 6) EV 7) MQ	1) 0.6116264687693259 2) 2.033047346679377 3) 4.735353535353536 4) 5.587096774193548 5) 6.027888446215139 6) 6.665137614678899 7) 8.016004886988393
BWI	F9 PA CO YV NW AA 9E US UA FL	1) 0.7562437562437563 2) 4.761904761904762 3) 5.1485301783373085 4) 5.496503496503497 5) 5.791685678899681 6) 6.103507291505722 7) 7.239805825242718 8) 7.508907895286403 9) 7.734484766155638 10) 7.747100147034798
MIA	1) 9E 2) EV 3) RU 4) TZ 5) XE 6) PA 7) NW 8) US 9) UA 10) ML	1) -3.0 2) 1.2026431718061674 3) 1.3021664766248575 4) 1.782243551289742 5) 2.745644599303136 6) 3.7389675499972452 7) 4.5040642076502735 8) 6.061599120445843 9) 6.88091441401217 10) 7.504550050556118
LAX	1) RU 2) MQ 3) OO 4) FL 5) TZ 6) NW 7) F9 8) HA 9) YV 10) EA	1) 1.9483870967741936 2) 2.407221858260434 3) 4.2219592877139975 4) 4.725127379994636 5) 4.763940985246312 6) 4.951674289186682 7) 5.729155372438469 8) 5.813645621181263 9) 6.024156085475379 10) 6.4884189325276935
IAH	1) NW 2) PA 3) RU 4) US 5) F9 6) PI 7) AA 8) WN 9) TW 10) OO	1) 3.51326380228999 2) 3.9847272727272727 3) 4.798695924258635 4) 5.042346298619824 5) 5.545243619489559 6) 5.745321399511798 7) 5.785937409374508 8) 6.449623532414497 9) 6.457104557640751 10) 6.58795822240426

SFO	TZ MQ F9 PA NW DL CO US AA TW	1) 3.952415634862831 2) 4.853923777799549 3) 5.162444663059518 4) 5.305942773294204 5) 5.590084712688538 6) 6.570832797840895 7) 6.837172913980928 8) 7.137510103662098 9) 8.02075832871468 10) 8.069147860259486

Question 2.2

```
events = KafkaUtils.createDirectStream(ssc, [topic], kafka_client_params)
parsed = events.map(lambda line: json.loads(line[1]))

# Filter based on airport list
filtered_airports = parsed.filter(lambda item: item['Dest'] in airports)

# compute averages
filtered_airports = filtered_airports.updateStateByKey(calculateAverage)

# for each airport, save to ten items
for airport in airports:
    saveTopCarriers(filtered_airports.filter(lambda item: item['Dest'] == airport))
```

Same save method as in 2.1, but saving destination instead of carrier

Results:

Airport Code	Destination	Average Departure Delay
CMI	1) ABI 2) PIT 3) PIA 4) CVG 5) DAY 6) STL 7) DFW 8) ATL 9) ORD	1) -7.0 2) 1.1024305555555556 3) 1.4523809523809523 4) 1.8947616800377536 5) 2.89157004073958 6) 5.039735099337748 7) 5.944142746314973 8) 6.665137614678899 9) 8.194098143236074
BWI	1) SAV 2) MLB 3) DAB 4) SRQ 5) IAD 6) UCA 7) GSP 8) BGM	1) -7.0 2) 1.155367231638418 3) 1.4695945945945945 4) 1.5884838880084522 5) 2.262295081967213 6) 3.6748726655348047 7) 4.197686645636172 8) 4.3842260649514975

	9) SJU 10) OAJ	9) 4.409805634417995 10) 4.453125
MIA	1) SHV 2) BUF 3) SAN 4) SLC 5) HOU 6) ISP 7) MEM 8) PSE 9) TLH 10) MCI	1) 0.0 2) 1.0 3) 1.710382513661202 4) 2.5371900826446283 5) 2.8840482573726542 6) 3.647398843930636 7) 3.7935208981468955 8) 3.975845410628019 9) 4.389016018306636 10) 5.5
LAX	1) SDF 2) IDA 3) DRO 4) RSW 5) LAX 6) BZN 7) MAF 8) PIH 9) IYK 10) MFE	1) -16.0 2) -7.0 3) -6.0 4) -3.0 5) -2.0 6) -0.7272727272727273 7) 0.0 8) 0.0 9) 1.2698247440569148 10) 1.3764705882352941
IAH	1) MSN 2) AGS 3) MLI 4) EFD 5) JAC 6) HOU 7) MTJ 8) RNO 9) BPT 10) VCT	1) -2.0 2) -0.6187904967602592 3) -0.5 4) 1.8877082136703045 5) 2.570588235294118 6) 2.631741821396994 7) 2.9501569858712715 8) 3.22158438576349 9) 3.5995325282430852 10) 3.6119087837837838
SFO	1) SDF 2) MSO 3) PIH 4) OAK 5) LGA 6) PIE 7) FAR 8) BNA 9) MEM 10) SJC	1) -10.0 2) -4.0 3) -3.0 4) -2.6271820448877805 5) -1.7575757575757576 6) -1.3410404624277457 7) 0.0 8) 2.425966447848286 9) 3.165533496509836 10) 4.089209855564996

Question 2.3

```
def save_results(items):
   dynamodb = boto3.resource('dynamodb', region_name="us-east-1")

   table = dynamodb.Table('cs598_question_2_3')
   for item in items:
       origin = item['origin']
```

```
        carrier = item['carrier']
        avg_depdelay = item['avg_depdelay']
        dest = item['dest']
        table.put_item(Item=item)

def saveTopCarriers(carriers):
    sorted = carriers.sortByKey(True)
    for toSave in sorted.take(10):
        save_results(toSave)


def calculateAverage(newVal, accumlativeAvg):
    if accumlativeAvg is None:
        accumlativeAvg = (0.0, 0, 0.0)
    total = sum(newVal, accumlativeAvg[0])
    count = accumlativeAvg[1] + len(newVal)
    avg = total / float(count)
    return (prod, count, avg)
```

```
# Source / Dest pairs
    routes = [('CMI', 'ORD'), ('IND', 'CMH'), ('DFW', 'IAH'), ('LAX', 'SFO'), ('JFK',
'LAX'), ('ATL', 'PHX')]


# Filter based on airport to / from
    filtered_airports = parsed.filter(lambda item: (item['origin'], item['Dest']) in
routes)

    # compute averages
    filtered_airports = filtered_airports.updateStateByKey(calculateAverage)

    # for each airport, save to ten items
    for route in routes:
        saveTopCarriers(filtered_airports.filter(lambda item: (item['origin'],
item['Dest']) == route))


    print("Completed")
```

Results:

To / From	Carrier	Average Arrival Delay
CMI -> ORD	1) MQ	1) 10.14366290643663
IND -> CMH	1) CO 2) NW 3) AA 4) HP 5) US 6) DL	1) -2.6579673776662482 2) 3.941798941798942 3) 5.5 4) 5.697254901960784 5) 6.237020316027088 6) 10.6875
DFW -> IAH	1) PA	1) -1.5964912280701755

	2) EV 3) CO 4) OO 5) RU 6) XE 7) AA 8) DL 9) MQ	2) 5.0925133689839575 3) 6.543978685024906 4) 7.564007421150278 5) 7.7914915966386555 6) 8.442865779927448 7) 8.545200789587318 8) 8.904884655714785 9) 9.103211009174313
LAX -> SFO	1) TZ 2) F9 3) EV 4) MQ 5) AA 6) WN 7) US 8) CO 9) UA 10) PA	1) -7.619047619047619 2) -2.028685790527018 3) 6.964630225080386 4) 7.8077634011090575 5) 8.181292220175184 6) 8.79205149734117 7) 8.822730864755023 8) 9.659957627118644 9) 10.364480979470095 10) 10.379324462640737
JFK -> LAX	1) UA 2) HP 3) AA 4) DL 5) PA 6) TW	1) 3.329564447940222 2) 6.680599369085174 3) 7.214716627006736 4) 7.934460351304701 5) 9.1992700729927 6) 12.125
ATL -> PHX	1) FL 2) US 3) HP 4) EA 5) DL	1) 4.552631578947368 2) 6.28811524609844 3) 8.481436314363144 4) 8.662650602409638 5) 9.900493798805696

Question 3.2

```
from __future__ import print_function

# spark-submit --packages org.apache.spark:spark-streaming-kafka-0-10_2.11:2.2.1
--executor-memory 20G --master yarn --num-executors 50  ~/task2/question-2_3.py

import sys
import uuid
import json
import boto3
from pyspark import SparkConf
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils


def save_results(items):
   dynamodb = boto3.resource('dynamodb', region_name="us-east-1")

   table = dynamodb.Table('cs598_question_3_2')
```

```
    for item in items:
        origin = item['origin']
        dest_1 = item['dest_1']
        arrDelay_1 = item['arrDelay_1']
        arrDelay_2 = item['arrDelay_2']
        carrier_1 = item['carrier_1']
        carrier_2 = item['carrier_2']
        depTime_1 = item['depTime_1']
        depTime_2 = item['depTime_2']
        dest_2 = item['dest_2']
        flightDate_1 = item['flightDate_1']
        flightDate_2 = item['flightDate_2']
        flightNum_1 = item['flightNum_1']
        flightNum_2 = item['flightNum_2']
        origin_2 = item['origin_2']
        total_delay = item['total_delay']
        table.put_item(Item=item)


def saveTopCarriers(carriers):
    sorted = carriers.sortBy(lambda item: item['total_delay'])
    for toSave in sorted.take(1):
        save_results(toSave)


if __name__ == '__main__':
    conf = SparkConf()
    conf.setAppName("Problem_3-2")
    conf.set("spark.streaming.kafka.maxRatePerPartition", 50000)
    conf.set("spark.executor.memory", "2g")
    conf.set("spark.python.worker.memory", "1g")

    # airports
    airports = ['CMI', 'ORD', 'LAX', 'JAX', 'DFW', 'CRP', 'SLC', 'BFL', 'SFO', 'PHX',
'JFK']
    year = "2008"

    sc = SparkContext(conf=conf)
    sc.setLogLevel("WARN")
    ssc = StreamingContext(sc, 2)
    ssc.checkpoint("/tmp/streaming")

    brokers =
"b-1.cs598-tast2.n69c9p.c2.kafka.us-east-1.amazonaws.com:9092,b-2.cs598-tast2.n69c9p.c
2.kafka.us-east-1.amazonaws.com:9092"
    topic = "cs598-task2"
    kafka_consumer_group = str(uuid.uuid4())
    kafka_client_params = {
        "metadata.broker.list": brokers,
        "group.id": kafka_consumer_group,
        "auto.offset.reset": "smallest",
    }

    events = KafkaUtils.createDirectStream(ssc, [topic], kafka_client_params)
    parsed = events.map(lambda line: json.loads(line[1]))

    # filter to year
    correct_year = parsed.filter(lambda item: item['year'] == year)

    # Filter to only airports we're looking for
```

```
   filtered_airports = correct_year.filter(lambda item: item['origin'] in airports or
item['Dest'] in airports)

   flights = []

   # filter and Join for each critera
   flights.append(
       filtered_airports.filter(lambda item: item['Orgin'] == 'CMI' and item['Dest']
== 'ORD' and item['Month'] == '3' and item['DayofMonth'] == '4' and item['CRSDepTime']
< 1200) \
       .union(filtered_airports.filter(lambda item: item['Orgin'] == 'ORD' and
item['Dest'] == 'DFW' and item['Month'] == '3' and item['DayofMonth'] == '6' and
item['CRSDepTime'] > 1200))
   )

   flights.append(
       filtered_airports.filter(lambda item: item['Orgin'] == 'JAX' and item['Dest']
== 'DFW' and item['Month'] == '9' and item['DayofMonth'] == '9' and item['CRSDepTime']
< 1200) \
       .union(filtered_airports.filter(lambda item: item['Orgin'] == 'DFW' and
item['Dest'] == 'CRP' and item['Month'] == '9' and item['DayofMonth'] == '11' and
item['CRSDepTime'] > 1200))
   )

   flights.append(
       filtered_airports.filter(lambda item: item['Orgin'] == 'SLC' and item['Dest']
== 'BFL' and item['Month'] == '4' and item['DayofMonth'] == '1' and item['CRSDepTime']
< 1200) \
       .union(filtered_airports.filter(lambda item: item['Orgin'] == 'BFL' and
item['Dest'] == 'LAX' and item['Month'] == '4' and item['DayofMonth'] == '3' and
item['CRSDepTime'] > 1200))
   )

   flights.append(
       filtered_airports.filter(lambda item: item['Orgin'] == 'LAX' and item['Dest']
== 'SFO' and item['Month'] == '7' and item['DayofMonth'] == '12' and
item['CRSDepTime'] < 1200) \
       .union(filtered_airports.filter(lambda item: item['Orgin'] == 'SFO' and
item['Dest'] == 'PHX' and item['Month'] == '7' and item['DayofMonth'] == '14' and
item['CRSDepTime'] > 1200))
   )

   flights.append(
       filtered_airports.filter(lambda item: item['Orgin'] == 'DFW' and item['Dest']
== 'ORD' and item['Month'] == '6' and item['DayofMonth'] == '10' and
item['CRSDepTime'] < 1200) \
       .union(filtered_airports.filter(lambda item: item['Orgin'] == 'ORD' and
item['Dest'] == 'DFW' and item['Month'] == '6' and item['DayofMonth'] == '12' and
item['CRSDepTime'] > 1200))
   )

   flights.append(
       filtered_airports.filter(lambda item: item['Orgin'] == 'LAX' and item['Dest']
== 'ORD' and item['Month'] == '1' and item['DayofMonth'] == '1' and item['CRSDepTime']
< 1200) \
       .union(filtered_airports.filter(lambda item: item['Orgin'] == 'ORD' and
item['Dest'] == 'JFK' and item['Month'] == '1' and item['DayofMonth'] == '3' and
item['CRSDepTime'] > 1200))
   )
```

```
   flights.withColumn("total_delay", flights.map(lambda item: item['ArrDelay'] +
item['ArrDelay_1'])

   # for each flight
   for flight in flights:
       saveTopCarriers(filtered_airports.filter(lambda item: (item['origin'],
item['Dest']) == route))

   print("Completed")

   ssc.start()
   ssc.awaitTermination()
```

Results:

Leg 1	Leg 2
CMI → ORD Carrier: MQ Flight Number: 4278 Sched Depart: 2008-03-04 7:10 Arrival Delay: -14.0	ORD -> LAX: Carrier: AA Flight Number: 607 Sched Depart: 2008-03-06 19:50 Arrival Delay: -24.0 Total arrival delay: -38.0
JAX → DFW Carrier: AA Flight Number: 845 Sched Depart: 2008-09-09 7:25 Arrival Delay: 1.0	DFW -> CRP: Carrier: MQ Flight Number: 3627 Sched Depart: 2008-09-11 16:45 Arrival Delay: -7.0 Total arrival delay: -6.0
SLC → BFL Carrier: OO Flight Number: 3755 Sched Depart: 2008-04-01 11:00 Arrival Delay: 12.0	BFL -> LAX: Carrier: OO Flight Number: 5429 Sched Depart: 2008-04-03 14:55 Arrival Delay: 6.0 Total arrival delay: 18.0
LAX → SFO Carrier: WN Flight Number: 3534 Sched Depart: 2008-07-12 6:50	SFO -> PHX: Carrier: US Flight Number: 412 Sched Depart: 2008-07-15 19:25

Arrival Delay: -13.0 WRONG	Arrival Delay: -19.0 Total arrival delay: -32.0
DFW → ORD Carrier: UA Flight Number: 1104 Sched Depart: 2008-06-10 7:00 Arrival Delay: -21.0	ORD -> DFW: Carrier: AA Flight Number: 2341 Sched Depart: 2008-06-12 16:45 Arrival Delay: -10.0 Total arrival delay: -31.0
LAX → ORD Carrier: UA Flight Number: 944 Sched Depart: 2008-01-01 7:05 Arrival Delay: 1.0	ORD -> JFK: Carrier: B6 Flight Number: 918 Sched Depart: 2008-01-03 19:00 Arrival Delay: -7.0 Total arrival delay: -6.0

Implementation trade offs.

Each system has their pros and cons. The batch processing system allows for (in my opinion) greater freedom to “play” with the data, and draw better conclusions. Batch processing has the advantage of being able to retroactively draw conclusions about your data, and even make predictions. This is why it's the cornerstone of business analytics. However, it lacks the ability to analyze in real time. This is where the streaming system shines. Having the ability to analyze your data in as real time as possible, means you can make decisions faster.

Ultimately, each approach should mirror the needs of the organization. It seems like the “needs” for this capstone project are to just obtain analytics about flight data, and there isn’t any precedence for real time. As such, the batch processing system (in my opinion) fits the needs of the “client” better, and would provide the organization with a better platform to answer their questions.

Conclusion

In summary, running batch processing against a stream of data seems counter intuitive to me. It very much feels like shoving a square peg in a round hole. If the focus for task 2 was actually utilizing real time data (dash boards, real time metrics, etc), then I think the streaming system would make a lot more sense. However, since it’s still running the same batch queries against a

stream, it just adds development overhead, and could be solved much simpler.

With all that being said, AWS provides plenty of streaming tools. The route I chose was one of many possible system configurations that are possible with all that AWS has to offer. If needed, hybrid systems (batch & stream) could easily be developed using some foundation tools (S3, Glue, Lambda, EMR).

The approach I took makes it easy to spin up a kafka cluster, and start ingesting my stream with data. EMR made it incredibly easy to launch a managed spark instance to start processing that data, and then dumping the results into DynamoDB. Working with streaming applications in AWS is fun, and rewarding.

Appendix

A.

```
import json
import boto3
import os
import sys
import csv
import codecs
from kafka import KafkaProducer
from kafka.errors import KafkaError

bucket = 'cs598task1'
s3 = boto3.resource(u's3')
bucket = s3.Bucket(u'cs598task1')

producer = KafkaProducer(
   bootstrap_servers=['b-1***.amazonaws.com:9092',
'b-2.***.amazonaws.com:9092'],
   value_serializer=lambda v: json.dumps(v).encode('utf-8'))

def sendEvent(item):
   data = {}
   data['Year'] = item[0]
   data['Month'] = item[2]
```

```
    data['DayofMonth'] = item[3]
    data['CRSDepTime'] = item[23]
    data['ArrDelay'] = item[36]
    data['Carrier'] = item[8]
    data['Origin'] = item[11]
    data['Dest'] = item[17]
    producer.send('cs598-task2', data)



for i in range(1988, 2009):
    obj = bucket.Object(key=u'{}_clean.csv'.format(i))
    response = obj.get()
    lines = response[u'Body']

    fileLines = 0

    for ln in codecs.getreader('utf-8')(lines):
        fileLines += 1
        print(fileLines)
        line = ln.split(',')
        if(line[0] != 'Year'):
            sendEvent(line)
```