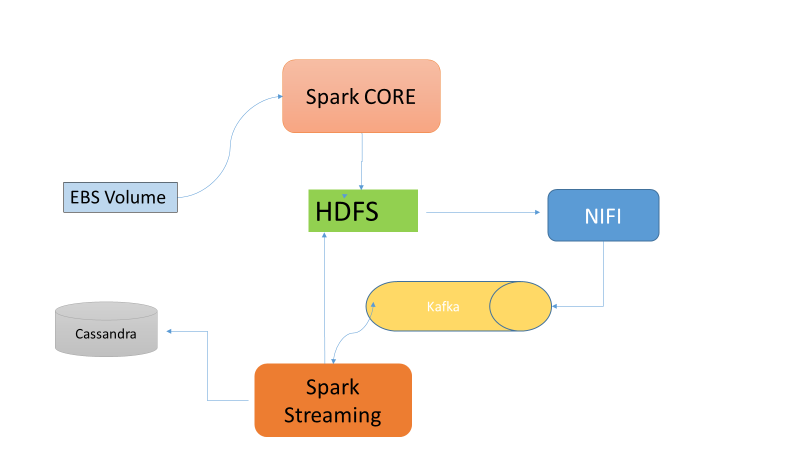
Cloud Computing Capstone Task 2 Report- Sharad Narang

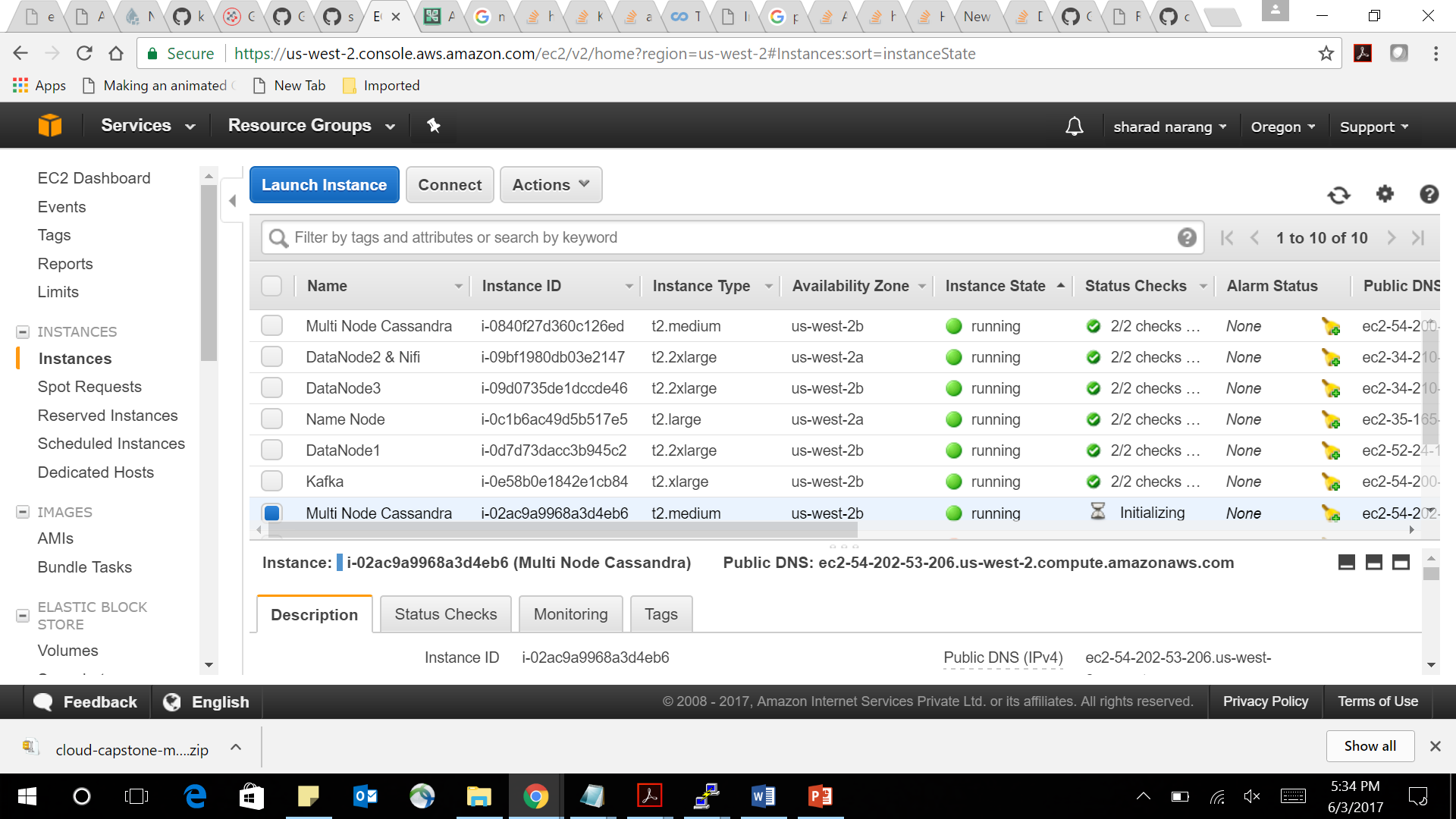
* Give a brief overview of how you integrated each system.

Conceptual Solution: Input data being read from EBS volume using Spark Core Program for trimming/pruning and results are persisted in HDFS. It is picked by NIFI and directed to Kafka cluster. Spark Steaming jobs consume the stream from Kafka, applies the transformations on it for the given questions, writes the intermittent results/checkpoints into HDFS and pushes the data back to Kafka before saving it to Cassandra.

Below is the high-level view of the solution detailing the data flow.



Deployment : Below are the deployment details HDFS, YARN & Spark are on a 1 Name Node + 3 Data Node cluster. Nifi is set up on one of a data node) Kafka is on a separated node. Cassandra is on a multi node cluster (3).



Installation & Set up:

* HDFS Set Up & Configuration - . Referred following link: <https://blog.insightdatascience.com/spinning-up-a-free-hadoop-cluster-step-by-step-c406d56bae42>
* Spark Set Up & configuration Referred following link: <http://blog.insightdatalabs.com/spark-cluster-step-by-step/>
* Cassandra Set Up & configuration Referred Link : <http://datascale.io/how-to-create-a-cassandra-cluster-in-aws-part-2/>
* Nifi Configuration: Referred Link : <http://ijokarumawak.github.io/nifi/2017/01/27/nifi-s2s-local-to-aws/>
* Kafka Set up : Referred Link : : <http://kafka.apache.org/quickstart>
* Integration of Spark & Hadoop - HADOOP\_CONF\_DIR or YARN\_CONF\_DIR points to the directory which contains the (client side) configuration files for the Hadoop cluster. These configs are used to write to HDFS and connect to the YARN Resource Manager. Set HADOOP\_CONF\_DIR in $SPARK\_HOME/spark-env.sh to a location containing the configuration files
* Integration of Cassandra & Spark – Used the datastax:spark-cassandra-connector:2.0.1-s\_2.11 Datastax - Spark & Cassandra connector Refer Link : <https://github.com/datastax/spark-cassandra-connector>
* What approaches and algorithms did you use to answer each question?
* 1.1 Rank the top 10 most popular airports by numbers of flights to/from the airport – Key Code Extract

def processInput(line):

fields = line[1].split(",")

return ((str(fields[12]), 1), (str(fields[18]), 1))

digest = ks.flatMap(processInput)\

.updateStateByKey(updateFunction)\

.transform(lambda rdd: rdd.sortBy(lambda x: x[1], ascending=False))

Processed the data stream, flat map it for key value pair (Origin and Destination Airport with Count 1), used update function to maintain the key’s running count and sort the data on count

* 1.3 Rank the days of the week by on-time arrival performance. - Key Code Extract:

def updateFunction(newValues, movingAvg):

prevAvg, prevN = movingAvg or (0,0)

currentN = len(newValues)

return (float(prevAvg\*prevN + sum(newValues)) / (prevN + currentN), prevN + currentN)

digest = ks.map(producePerDay)\

.updateStateByKey(updateFunction)\

.map(lambda x: (x[0], x[1][0]))\

.transform(lambda rdd: rdd.sortBy(lambda x: x[1], ascending=True))

Processed the data stream, map it for key value pair (WeekNum and ArrivalDelay), used update function to calculate arrival average delay along with maintaining the key’s moving average and print the data sorted on avg delay value

* 2.1 For each airport X, rank the top-10 carriers in decreasing order of on-time departure performance from X.

First we use the updateStateByKey function with Spark checkpoints to count average departure delays for all airport-carrier pairs. The update function calculates three values for each: sum, count and sum/count.

airports\_and\_carriers.updateStateByKey(updateFunction)

def updateFunction(newValues, runningAvg):

if runningAvg is None:

runningAvg = (0.0, 0, 0.0)

# calculate sum, count and average.

prod = sum(newValues, runningAvg[0])

count = runningAvg[1] + len(newValues)

avg = prod / float(count)

return (prod, count, avg)

Then we use the aggregateByKey to have an ordered list of top ten performing carrier for each airport. Aggregate contains top ten carriers and departure delays

airports = airports.transform(lambda rdd: rdd.aggregateByKey([],append,combine))

def append(aggr, newCarrierAvg):

aggr.append(newCarrierAvg)

aggr.sort(key=lambda element: element[1])

return aggr[0:10]

def combine(left, right):

"""

Combine two aggregates. Aggregate contains top ten carriers and departure delays.

Sample: [('TZ',-0.0001), ('AQ',0.025), ('MS',0.3)]

"""

for newElement in right:

left.append(newElement)

left.sort(key=lambda element: element[1])

return left[0:10]

When this is done, all continuously refined top ten performing carriers are delivered to a separate topic called top\_carriers\_by\_airports This topic is then consumed by another Spark Streaming job, which saves and updates values to Cassandra.

* 2.2 For each airport X, rank the top-10 airports in decreasing order of on-time departure performance from X. Key Code Extract:
  + Logic is same as 2.1 , to count average departure delays for all airport-dest airport pairs using update function and then aggregate the records for destination airport.
  + # Count averages
  + airports\_and\_carriers = airports\_and\_carriers.updateStateByKey(updateFunction)
  + # Change key to just airports
  + airports = airports\_and\_carriers.map(lambda row: (row[0][0], (row[0][1], row[1][2])))
  + # Aggregate to just top 10 destination airports
  + airports = airports.transform(lambda rdd: rdd.aggregateByKey([],append,combine))
* 2.4 For each source-destination pair X-Y, determine the mean arrival delay (in minutes) for a flight from X to Y.

We calculate the mean arrival delay for all the airport from-to pairs. The average calculation method is the same as in Question 2.1.

airports\_fromto = airports\_fromto.updateStateByKey(updateFunction)

Then we just filter out for all relevant from-to pairs and save it to airports\_airports\_arrival topic in Kafka. Another Spark Streaming job deals with updating results in Cassandra from this topic.

* 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. More concretely, Tom has the following requirements (for specific queries, see the[**Task 1 Queries**](https://www.coursera.org/learn/cloud-computing-project/supplement/xxQie/task-1-queries) and [**Task 2 Queries**](https://www.coursera.org/learn/cloud-computing-project/supplement/0EylL/task-2-queries)):
  + a) The second leg of the journey (flight Y-Z) must depart two days after the first leg (flight X-Y). For example, if X-Y departs on January 5, 2008, Y-Z must depart on January 7, 2008.
  + b) Tom wants his flights scheduled to depart airport X *before* 12:00 PM local time and to depart airport Y *after* 12:00 PM local time.
  + c) Tom wants to arrive at each destination with as little delay as possible. Y ou can assume you know the actual delay of each flight

At first step we map every item from input\_2008 topic and form a key that holds the airport from-to, flight date, and AM or PM according to what is the SCHEDULED DEPARTURE\_TIME of the flight.

airports\_fromto = rows.map(lambda row: ( \

(row[0], row[1], row[2], AMOrPM(row[5])), \

(row[3], row[4], departureTimePretty(row[5]), float(row[8])) \

) \

)

Next, we filter out all unnecessary data that is not relevant for answering question 3.2 and do a minimum search for each key. Minimum search is based on the

arrival performance of the given flight. This way for each key-value pair we just keep tracking of the best flights.

# Filtering just necessary flights

airports\_fromto = airports\_fromto.filter(lambda row: row[0] == ('BOS', 'ATL', '2008-04-03', 'AM')) \

.union(airports\_fromto.filter(lambda row: row[0] == ('ATL', 'LAX', '2008-04-05', 'PM'))) \

.union(airports\_fromto.filter(lambda row: row[0] == ('PHX', 'JFK', '2008-09-07', 'AM'))) \

.union(airports\_fromto.filter(lambda row: row[0] == ('JFK', 'MSP', '2008-09-09', 'PM'))) \

.union(airports\_fromto.filter(lambda row: row[0] == ('DFW', 'STL', '2008-01-24', 'AM'))) \

.union(airports\_fromto.filter(lambda row: row[0] == ('STL', 'ORD', '2008-01-26', 'PM'))) \

.union(airports\_fromto.filter(lambda row: row[0] == ('LAX', 'MIA', '2008-05-16', 'AM'))) \

.union(airports\_fromto.filter(lambda row: row[0] == ('MIA', 'LAX', '2008-05-18', 'PM')))

# Minimum search

airports\_fromto = airports\_fromto.updateStateByKey(getMinimum)

Results are saved to Kafka topic and then to Cassandra by a different Spark Streaming job.

* What are the results of each question? Use only the provided subset for questions from Group 2 and Question 3.2.
* 1.1 Rank the top 10 most popular airports by numbers of flights to/from the airport.

| ORD| 12449354|

| ATL| 11540422|

| DFW| 10799303|

| LAX| 7723596|

| PHX| 6585534|

| DEN| 6273787|

| DTW| 5636622|

| IAH| 5480734|

| MSP| 5199213|

| SFO| 5171023|

| EWR| 5136971|

| STL| 5125336|

| LAS| 4962958|

| CLT| 4824711|

| LGA| 4337167|

| BOS| 4311116|

| PHL| 4079651|

| PIT| 3936220|

| SLC| 3815114|

| SEA| 3736761|

* 1.3 Rank the days of the week by on-time arrival performance.

| Sunday | 6.613280292442754|

| Wednesday | 7.203656394670348|

| Friday | 9.721032337585571|

| Saturday | 4.301669926076596|

| Monday | 6.716102802585582|

| Thursday | 9.094441008336657|

| Tuesday | 5.990458841319885|

* 2.1 For each airport X, rank the top-10 carriers in decreasing order of on-time departure performance from X.

select airport,carrier,departuredelay from demodb.ques2i where airport= 'SRQ' limit 10;

airport | carrier | departuredelay

---------+---------+----------------

SRQ | TZ | -0.381997

SRQ | XE | 1.48977

SRQ | YV | 3.40402

SRQ | AA | 3.6335

SRQ | UA | 3.95212

SRQ | US | 3.9684

SRQ | TW | 4.30468

SRQ | NW | 4.85636

SRQ | DL | 4.86918

SRQ | MQ | 5.35059

select airport,carrier,departuredelay from demodb.ques2i where airport= 'CMH' limit 10 ;

airport | carrier | departuredelay

---------+---------+----------------

CMH | DH | 3.49111

CMH | AA | 3.51393

CMH | NW | 4.04155

CMH | ML (1) | 4.36646

CMH | DL | 4.71344

CMH | PI | 5.20129

CMH | EA | 5.93739

CMH | US | 5.9933

CMH | TW | 6.1591

CMH | YV | 7.96119

select airport,carrier,departuredelay from demodb.ques2i where airport= 'JFK' limit 10 ;

airport | carrier | departuredelay

---------+---------+----------------

JFK | UA | 5.96833

JFK | XE | 8.11374

JFK | CO | 8.20121

JFK | DH | 8.74298

JFK | AA | 10.08074

JFK | B6 | 11.1271

JFK | PA (1) | 11.52348

JFK | NW | 11.63782

JFK | DL | 11.98667

JFK | TW | 12.63907

select airport,carrier,departuredelay from demodb.ques2i where airport= 'SEA' limit 10 ;

airport | carrier | departuredelay

---------+---------+----------------

SEA | OO | 2.70582

SEA | PS | 4.72064

SEA | YV | 5.12226

SEA | TZ | 6.345

SEA | US | 6.41238

SEA | NW | 6.49876

SEA | DL | 6.53562

SEA | HA | 6.85545

SEA | AA | 6.93915

SEA | CO | 7.09646

select airport,carrier,departuredelay from demodb.ques2i where airport= 'BOS' limit 10 ;

airport | carrier | departuredelay

---------+---------+----------------

BOS | TZ | 3.06379

BOS | PA (1) | 4.44717

BOS | ML (1) | 5.73478

BOS | EV | 7.20814

BOS | NW | 7.24519

BOS | DL | 7.44534

BOS | XE | 8.10292

BOS | US | 8.68792

BOS | AA | 8.73351

BOS | EA | 8.89143

* 2.2 For each airport X, rank the top-10 airports in decreasing order of on-time departure performance from X.
  1. select org\_airport,dest\_airport,departuredelay from demodb.ques2ii where org\_airport = 'SRQ' limit 10;

org\_airport | dest\_airport | departuredelay

-------------+--------------+----------------

SRQ | EYW | 0

SRQ | SJU | 0

SRQ | TPA | 1.32885

SRQ | IAH | 1.44456

SRQ | MEM | 1.70296

SRQ | FLL | 2

SRQ | BNA | 2.06231

SRQ | MCO | 2.36454

SRQ | RDU | 2.5354

SRQ | MDW | 2.83812

select org\_airport,dest\_airport,departuredelay from demodb.ques2ii where org\_airport = 'CMH' limit 10;

org\_airport | dest\_airport | departuredelay

-------------+--------------+----------------

CMH | AUS | -5

CMH | OMA | -5

CMH | SYR | -5

CMH | MSN | 1

CMH | CLE | 1.10499

CMH | SDF | 1.35294

CMH | CAK | 3.70039

CMH | SLC | 3.93929

CMH | MEM | 4.15202

CMH | IAD | 4.1581

select org\_airport,dest\_airport,departuredelay from demodb.ques2ii where org\_airport = 'JFK' limit 10;

org\_airport | dest\_airport | departuredelay

-------------+--------------+----------------

JFK | SWF | -10.5

JFK | ABQ | 0

JFK | ANC | 0

JFK | ISP | 0

JFK | MYR | 0

JFK | UCA | 1.91701

JFK | BGR | 3.21028

JFK | BQN | 3.60623

JFK | CHS | 4.40271

JFK | STT | 4.49277

select org\_airport,dest\_airport,departuredelay from demodb.ques2ii where org\_airport = 'SEA' limit 10;

org\_airport | dest\_airport | departuredelay

-------------+--------------+----------------

SEA | EUG | 0

SEA | PIH | 1

SEA | PSC | 2.65052

SEA | CVG | 3.87874

SEA | MEM | 4.26022

SEA | CLE | 5.17017

SEA | BLI | 5.19825

SEA | YKM | 5.37965

SEA | SNA | 5.40625

SEA | LIH | 5.48108

select org\_airport,dest\_airport,departuredelay from demodb.ques2ii where org\_airport = 'BOS' limit 10;

org\_airport | dest\_airport | departuredelay

-------------+--------------+----------------

BOS | SWF | -5

BOS | ONT | -3

BOS | GGG | 1

BOS | AUS | 1.20871

BOS | LGA | 3.05402

BOS | MSY | 3.24647

BOS | LGB | 5.13618

BOS | OAK | 5.78321

BOS | MDW | 5.89564

BOS | BDL | 5.9827

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

select org\_airport,dest\_airport,arrivaldelay from demodb.ques2iv where org\_airport = 'LGA' and dest\_airport= 'BOS' limit 10;

org\_airport | dest\_airport | arrivaldelay

-------------+--------------+--------------

LGA | BOS | 1.48386

select org\_airport,dest\_airport,arrivaldelay from demodb.ques2iv where org\_airport = 'BOS' and dest\_airport= 'LGA' limit 10;

org\_airport | dest\_airport | arrivaldelay

-------------+--------------+--------------

BOS | LGA | 3.78412

select org\_airport,dest\_airport,arrivaldelay from demodb.ques2iv where org\_airport = 'OKC' and dest\_airport= 'DFW' limit 10;

org\_airport | dest\_airport | arrivaldelay

-------------+--------------+--------------

OKC | DFW | 4.96906

select org\_airport,dest\_airport,arrivaldelay from demodb.ques2iv where org\_airport = 'MSP' and dest\_airport= 'ATL' limit 10;

org\_airport | dest\_airport | arrivaldelay

-------------+--------------+--------------

MSP | ATL | 6.73701

3.1 This has been covered in Task1

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. More concretely, Tom has the following requirements (for specific queries, see the[**Task 1 Queries**](https://www.coursera.org/learn/cloud-computing-project/supplement/xxQie/task-1-queries) and [**Task 2 Queries**](https://www.coursera.org/learn/cloud-computing-project/supplement/0EylL/task-2-queries)):

a) The second leg of the journey (flight Y-Z) must depart two days after the first leg (flight X-Y). For example, if X-Y departs on January 5, 2008, Y-Z must depart on January 7, 2008.

b) Tom wants his flights scheduled to depart airport X *before* 12:00 PM local time and to depart airport Y *after* 12:00 PM local time.

c) Tom wants to arrive at each destination with as little delay as possible. You can assume you know the actual delay of each flight.

**BOS ➝ ATL ➝ LAX, 03/04/2008:**

select \* from demodb.ques3ii where st\_airport= 'BOS' and intrm\_airport= 'ATL' and conn\_dst = 'LAX' and st\_flt\_dt ='2008-04-03' limit 1;

st\_airport | intrm\_airport | conn\_dst | st\_flt\_dt | tot\_delay | conn\_airport | conn\_arln | conn\_dely | conn\_flt | conn\_flt\_dt | conn\_schd\_dep | st\_airline | st\_dep\_tm | st\_dly | st\_flt

------------+---------------+----------+------------+-----------+--------------+-----------+-----------+----------+-------------+---------------+------------+-----------+--------+--------

BOS | ATL | LAX | 2008-04-03 **| 5 |** ATL | 20437 | -2 | 40 | 2008-04-05 | 1852 | 20437 | 0600 | 7 | 270

**PHX ➝ JFK ➝ MSP, 07/09/2008:**

select \* from demodb.ques3ii where st\_airport= 'PHX' and intrm\_airport= 'JFK' and conn\_dst = 'MSP' and st\_flt\_dt ='2008-09-07' limit 1;

st\_airport | intrm\_airport | conn\_dst | st\_flt\_dt | tot\_delay | conn\_airport | conn\_arln | conn\_dely | conn\_flt | conn\_flt\_dt | conn\_schd\_dep | st\_airline | st\_dep\_tm | st\_dly | st\_flt

------------+---------------+----------+------------+-----------+--------------+-----------+-----------+----------+-------------+---------------+------------+-----------+--------+--------

PHX | JFK | MSP | 2008-09-07 **| -42 |** JFK | 19386 | -17 | 609 | 2008-09-09 | 1750 | 20409 | 1130 | -25 | 178

**DFW ➝ STL ➝ ORD, 24/01/2008:**

select \* from demodb.ques3ii where st\_airport= 'DFW' and intrm\_airport= 'STL' and conn\_dst = 'ORD' and st\_flt\_dt ='2008-01-24' limit 1;

st\_airport | intrm\_airport | conn\_dst | st\_flt\_dt | tot\_delay | conn\_airport | conn\_arln | conn\_dely | conn\_flt | conn\_flt\_dt | conn\_schd\_dep | st\_airline | st\_dep\_tm | st\_dly | st\_flt

------------+---------------+----------+------------+-----------+--------------+-----------+-----------+----------+-------------+---------------+------------+-----------+--------+--------

DFW | STL | ORD | 2008-01-24 **| -19 |** STL | 19805 | -5 | 2245 | 2008-01-26 | 1655 | 19805 | 0705 | -14 | 1336

**LAX ➝ MIA ➝ LAX, 16/05/2008:**

select \* from demodb.ques3ii where st\_airport= 'LAX' and intrm\_airport= 'MIA' and conn\_dst = 'LAX' and st\_flt\_dt ='2008-05-16' limit 1;

st\_airport | intrm\_airport | conn\_dst | st\_flt\_dt | tot\_delay | conn\_airport | conn\_arln | conn\_dely | conn\_flt | conn\_flt\_dt | conn\_schd\_dep | st\_airline | st\_dep\_tm | st\_dly | st\_flt

------------+---------------+----------+------------+-----------+--------------+-----------+-----------+----------+-------------+---------------+------------+-----------+--------+--------

LAX | MIA | LAX | 2008-05-16 **| -9 |** MIA | 19805 | -19 | 456 | 2008-05-18 | 1930 | 19805 | 0820 | 10 | 280

* What system- or application-level optimizations (if any) did you employ?
* Data Ingestion & Storage– Used Spark program to read all the data files from HDFS and trim/prune only the needed data elements to answer the given questions, this leads to ~40-50% overall reduction of data size to be stored and processed.
* Data Processing – For tuning the performance following were considered
  + Spark Streaming
    - Increase parallelism: To have multiple parallel spark streaming receivers the number of partitions in Kafka were set to an optimal number 3
    - Enabling Backpressure - Spark Streaming has trouble with situations where the batch-processing time is larger than the batch interval. In other words, Spark will not be able to read data from the topic faster than it arrives—the Kafka receiver for the executor won’t be able to keep up. If this throughput is sustained for long enough, it leads to an unstable situation where the memory of the receiver’s executor overflows. This was managed by setting up the configuration parameter spark.streaming.backpressure.enabled=true
    - Cluster Resource Tuning Example
      * Dynamic allocation allows Spark to dynamically scale the cluster resources allocated to your application based on the workload. When dynamic allocation is enabled and a Spark application has a backlog of pending tasks, it can request executors. When the application becomes idle, its executors are released and can be acquired by other applications. Enable it by setting proposert spark.dynamicAllocation.enabled =’True’
    - Limiting Spark receiver: With Spark's spark.streaming.receiver.maxRate set up to 500 , the number of messages pulled by the stream receiver were limited . Window size, reduce the number of incoming messages per second to a point where the processing time for this window stays within the window and the scheduling delay stays at zero .
  + Kafka:
    - Going with 3 partitions help to get good write and read performance
    - Keeping only relevant data in the topics
      * On each topic, only relevant rows and records are stored. All unnecessary columns are stripped in CSVs from ontime\_perf dataset. Topics that are populated to prepare data for Cassandra storage only have small portion of relevant data.
      * Reducing storage space: Kafka topics need only 16BG of EBS storage to store every message necessary for all computations. Even with replication factor of 2.
      * Reducing network traffic: Since Kafka messages are noticeably smaller, less network bandwidth is needed during streaming operation.
* Read Performance -DB model Cassandra
  + Dedicated Commit Log Disk: Cassandra write operations are occurred on a commit log on disk and then to an in-memory table structure called Memtable. When thresholds are reached, that Memtable is flushed to a disk in a format called SSTable. So we separate out Commit Log locations, it will isolate Commit Log I/O traffics from other Cassandra Reads, Memtables and SSTables traffics
    - Mount a separate partition for commit log
    - Changed CommitLogDirectory: /mnt/commitlog in cassandra.yaml
  + Increasing Java Heap Size: To avoid out of memory issues when we run a heavy load on Cassandra. Followed following rule and updated the heap size as 2 GB in cassandra-env.sh f considering node memory (8GB)
    - Heap Size = 1/2 of System Memory when System Memory < 2GB
    - Heap Size = 1GB when System Memory >= 2GB and <= 4GB
    - Heap Size = 1/4 of System Memory(but not more than 8GB) when System Memory >4GB
  + Tune Concurrent Reads and Writes:  Concurrent readers and writers control the maximum number of threads allocated to a particular stage. So having an optimal concurrent reads and concurrent writes value will improve Cassandra performance .Changed two parameters ConcurrentReaders and ConcurrentWriters in cassandra.yaml by following the rule 4 concurrent reads per processor core so for t2.large it will be 16
* Give your opinion about whether the results make sense and are useful in any way.

Results are useful as it gives insights on the flights data in terms of popularity of airport and could help coming up with the optimized itinerary with different constraints and conditions based on the past 20-year data.

Further data analysis will help in understanding the problematic airports or carrier having arrival and departure delay issues for root cause analysis and subsequent corrective and preventive actions.

* How did the different stacks (Hadoop and Spark) from Task 1 and Task 2 compare? Which stack did you find the easiest to use? The fastest?
  1. Considering type of data

Data has historical characteristics, so using batch processing suits better in this case. There's no necessity for near real-time data generation that stream

processing provides.

* 1. Considering type of questions & computations

Spark Core (Batch) jobs provide better performance on these kind of calculations (minimum search, average, best from a given category). Since they're

doing one big-bang map and reduce operation on the whole dataset. Spark's streaming iterative micro-batch approach causes more computation overhead. The

results take longer to produce with stream processing, because all the data should flow through the pipelines. Only after that we get the precise

computation.

Spark Streaming would provide better results, if we would investigate best performing airports, carriers on a given short time window (e.g. today or

yesterday), or if we would do a different time of computation (e.g. estimated delay based on real-time data. Weather condition, air traffic, historical

information from HDFS etc.)

* 1. Considering development effort

Streaming provides faster feedback, than big batch processing jobs. It's easier to spot if something goes wrong. Since Kafka allows consumption of

messages multiple times, by just using offsets and offset resets, streaming jobs can be executed on production-like big datasets on the fly. Mistakes are

cheaper and fixes are easier.

Batch processes, like Spark batch jobs, are often debugged and tested over a small amount of dataset. Only after that they can be executed on larger

portion of data. From that point developers have to rely on logging functionality to determine job execution history and fine-tune the computation