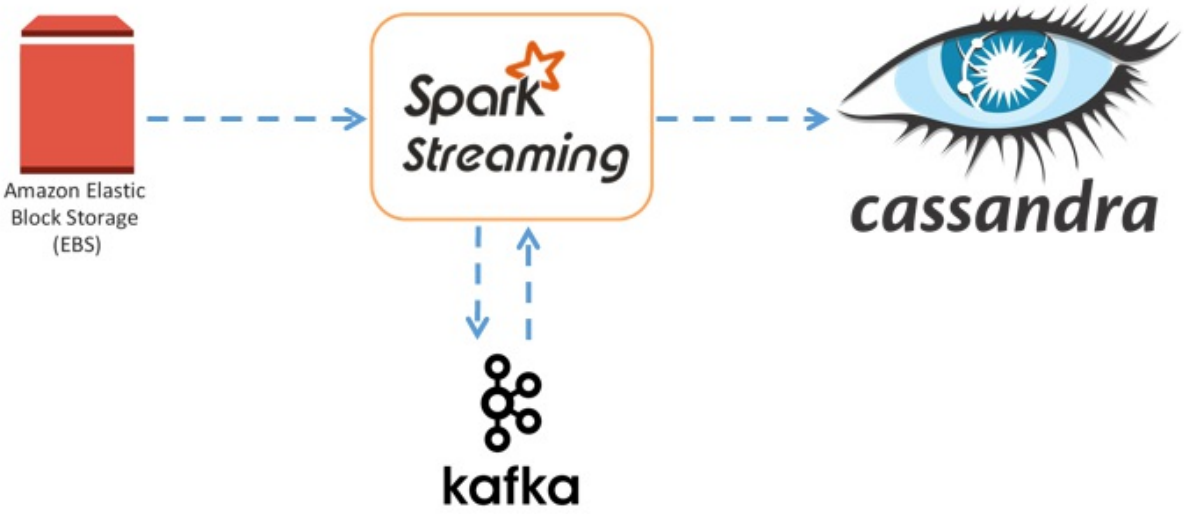
System Integration

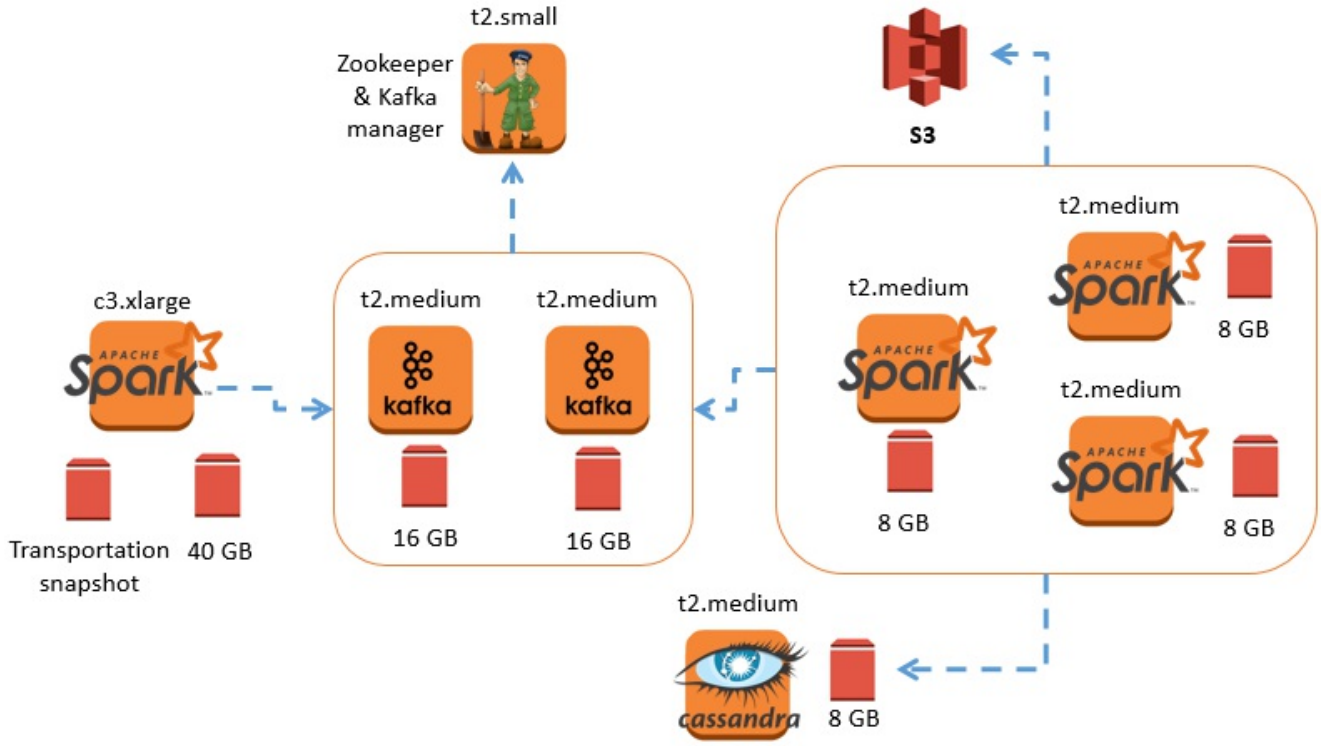
Components and Data Flow

Input is being read from EBS volume using Spark Streaming. The results are trimmed and directed to Kafka cluster. Separate Spark Steaming jobs are reusing this stream from Kafka, continuously refining the data flow before saving it to Cassandra.



Deployment View

Transportation dataset is mounted as EBS volume under a c3.xlarge instance. This also has a pretty big 40 GB local EBS volume for the extracted CSV files. These are ingested by local Spark Streaming job running in 4 threads parallel. Kafka cluster has 2 t2.medium nodes and coordinated by Zookeeper and kafka-manager. These are installed on a separate t2.small node. Spark Streaming cluster contains 1 driver and 2 worker nodes. They're using an S3 bucket for saving checkpoints. For simplicity Cassandra is only installed to one t2.medium.



Cassandra Migration

Migrating to Apache Cassandra is done by using Spark Cassandra Connector. This allows shifting loaded DataFrames from Spark to Apache Cassandra.

References

- Spark Cassandra Connector
- kafka-manager

Solution Approach

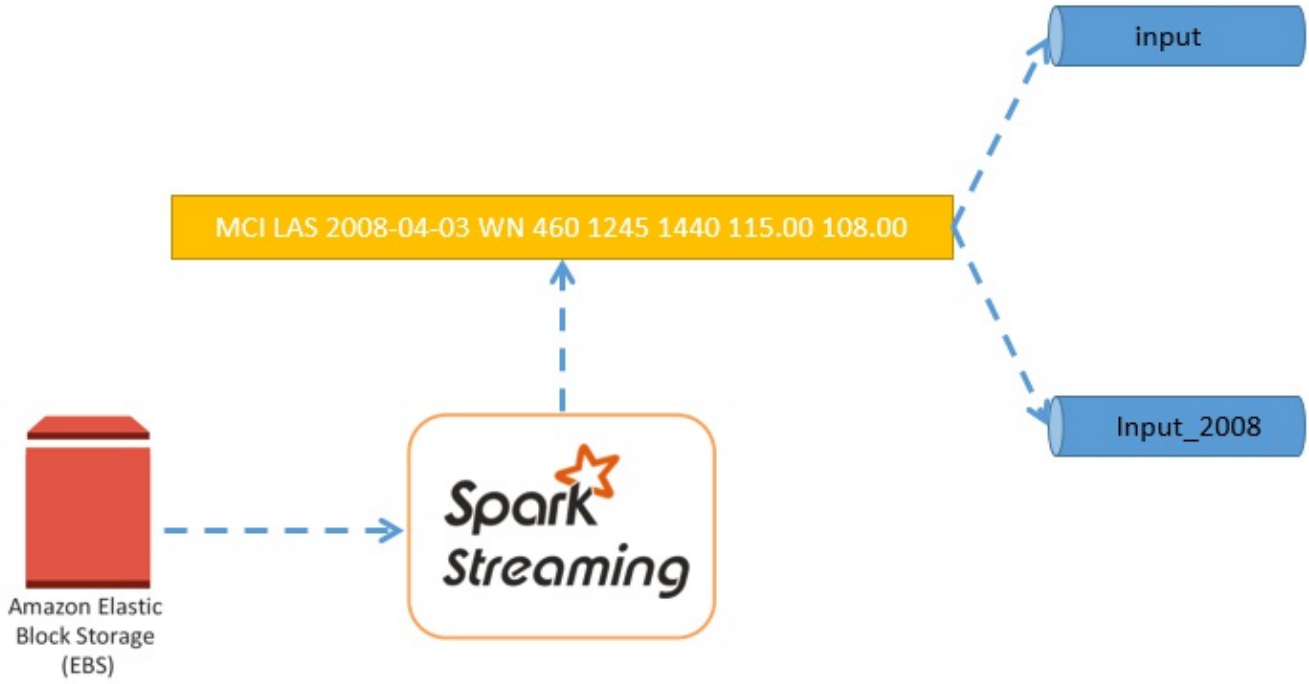
Feeding data into Kafka

First we start a Spark job locally, that watches a directory for incoming files. That directory is populated with CSV files from the `airline_ontime` folder of transportation dataset. The CSV extraction is done by a `bash` script.

The Spark Streaming job will cut off all unnecessary columns from the on-time performance CSV files. The structure for one message is the following:

AIRPORT_FROM | AIRPORT_TO | DEPARTURE_DATE| CARRIER_ID | FLIGHT_NUM | SCHEDULED_DEPARTURE_TIME | DEPARTURE_TIME | DEPARTURE_DELAY | ARRIVAL_DELAY |

We're populating two input queues. One is just feeded with data from 2008.



Starting ingestion job locally on 4 threads.

```
~/spark-2.1.0-bin-hadoop2.7/bin/spark-submit --master local[4] --conf spark.streaming.backpressure.enabled=true --conf
spark.streaming.receiver.maxRate=4000 ./ingest_files_to_kafka.py input
```

Populating the input folder

```
./move-ontime-perf-to-localfs.sh data/aviation input
```

Sample data in Kafka

```
MCI LAS 2008-04-03 WN 460 1245 1440 115.00 108.00
MCI LAS 2008-04-03 WN 1758 0900 0854 -6.00 -6.00
MCI LAS 2008-04-03 WN 2888 0705 0703 -2.00 -7.00
MCI LAX 2008-04-03 WN 238 1440 1553 73.00 57.00
MCI LAX 2008-04-03 WN 450 1135 1226 51.00 41.00
```

References

- Migration script on GitHub
- Spark Streaming job on GitHub

Question 1.1

Airport from-to information is collected by using flatMap from input stream

```
rows.flatMap(lambda row: [row[0], row[1]])
```

We use the `updateStateByKey` function with Spark checkpoints to count all the occurrences for all airports. The updateFunction is a simple counter function.

```
airports.map(lambda airport: (airport, 1)).updateStateByKey(updateFunction)
```

To reduce the traffic, we cut off the amount of records to just the top 10 most popular in each partition.

```
sorted.transform(lambda rdd: rdd.mapPartitions(cutOffTopTen))
```

```
def cutOffTopTen(iterable):
        topTen = []
        for tupl in iterable:
                if len(topTen) < 10:
                        topTen.append(tupl)
                        topTen.sort(reverse=True)
                elif topTen[9][0] < tupl[0]:
                        topTen[9] = tupl
                        topTen.sort(reverse=True)
        return iter(topTen)
```

Then we sort RDDs by popularity.

```
sorted = sorted.transform(lambda rdd: rdd.sortByKey(False))
```

References

- Spark Streaming job on GitHub

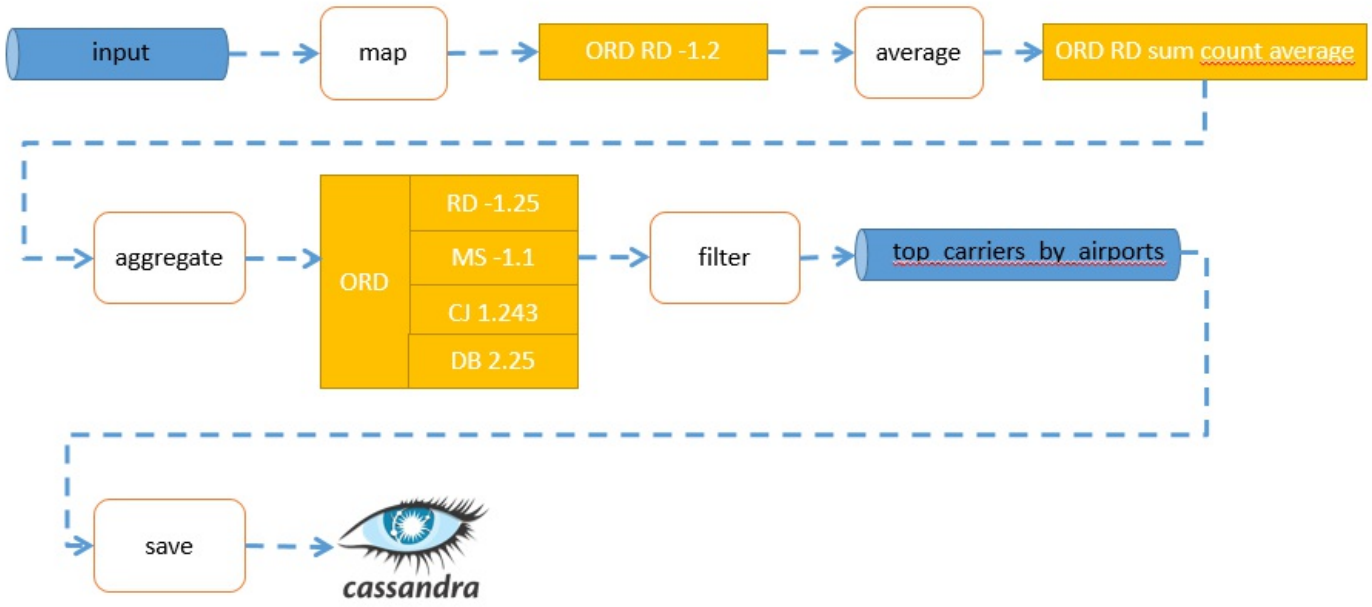
Question 1.2

This is analogue to Question 1.1

References

- Spark Streaming job on GitHub

Question 2.1



First we use the `updateStateByKey` function with Spark checkpoints to count average departure delays for all airport-carrier pairs. The `updateFunction` 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. This is tricky at first, but keeps calculations and data traffic at minimum. Aggregate contains top ten carriers and departure delays. Sample aggregated value for an airport:
`[('TZ',-0.0001), ('AQ',0.025), ('MS',0.3)]`

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

```
def append(aggr, newCarrierAvg):
        """
        Add new element to aggregate. Aggregate contains top ten carriers and departure delays.
        Sample: [('TZ',-0.0001), ('AQ',0.025), ('MS',0.3)]
        """
        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.

References

- Spark Streaming job on GitHub

- Cassandra migration job on GitHub
- Cassandra table definitions

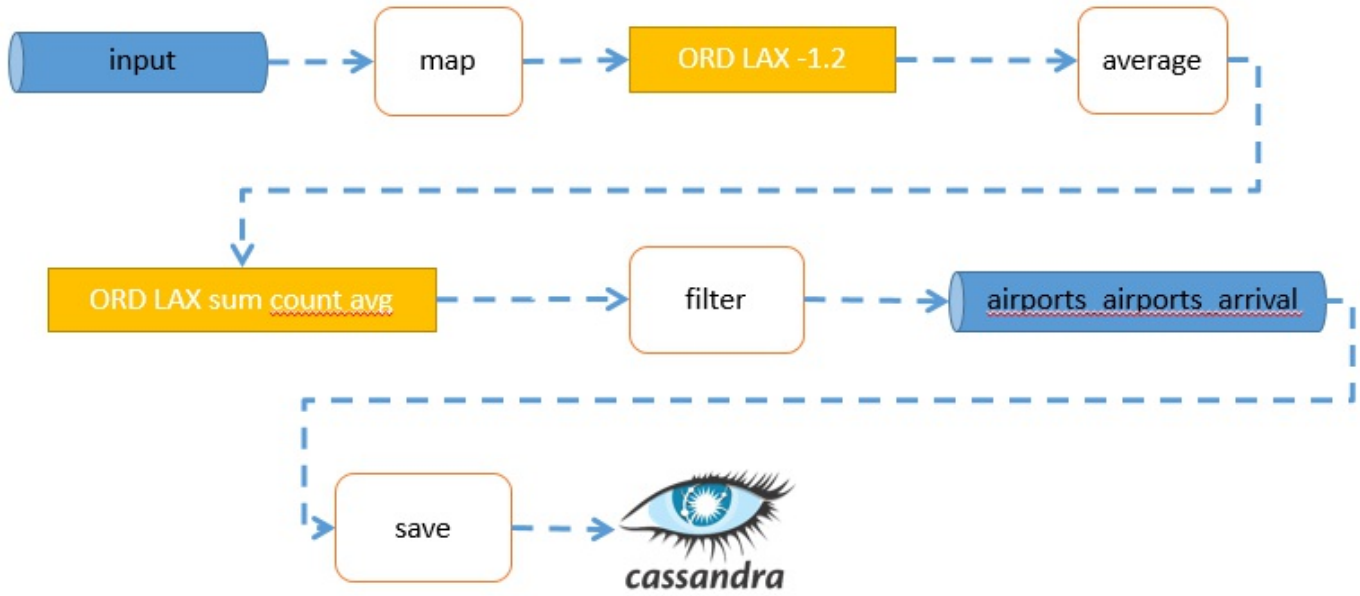
Question 2.2

This is analogue to Question 2.1

References

- Spark Streaming job on GitHub
- Cassandra migration job on GitHub
- Cassandra table definitions

Question 2.4



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.

References

- Spark Streaming job on GitHub
- Cassandra migration job on GitHub
- Cassandra table definitions

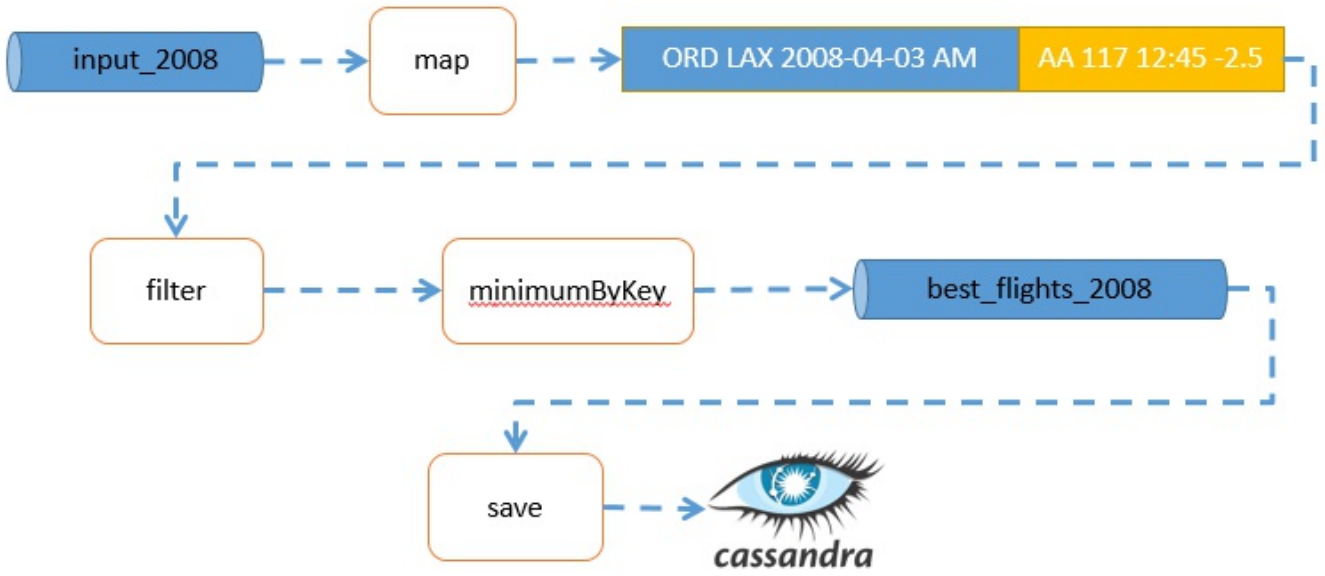
Question 3.1

Question 3.1 is not needed by description of Task 2 Overview. But if you're interested, check out solution in documentation of Task 1.

References

- Documentation of Task 1 in GitHub

Question 3.2



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.

References

- Spark Streaming job on GitHub
- Cassandra migration job on GitHub
- Cassandra table definitions

Results

Question 1.1

```
ORD  12446097
ATL  11537401
DFW  10757743
LAX  7721141
PHX  6582467
DEN  6270420
DTW  5635421
```

```
IAH  5478257
MSP  5197649
SFO  5168898
```

Question 1.2

```
HA -1.01180434575
AQ 1.15692344248
PS 1.45063851278
ML  4.74760919573
PA  5.34822529946
F9  5.46588114882
NW  5.55940916466
WN  5.56149775671
OO  5.73631246366
9E  5.8671846617
```

Question 2.1

```
--- JFK ---

(RU: 5.06730065838)
(UA: 5.96832536487)
(CO: 8.20120808165)
(DH: 8.74298090807)
(AA: 10.0824367451)
(B6: 11.1270962227)
(PA: 11.5265213442)
(NW: 11.6378177165)
(DL: 11.9867913583)
(AL: 12.4135490394)

--- SEA ---

(OO: 2.70581965466)
(PS: 4.72063933287)
(YV: 5.12226277372)
(AL: 6.01471571906)
(TZ: 6.34500393391)
(US: 6.43266197111)
(NW: 6.49876240739)
(DL: 6.53599823636)
(HA: 6.8554526749)
(AA: 6.94025364474)

--- BOS ---

(RU: 2.12059369202)
(TZ: 3.06379208506)
(PA: 4.44716479505)
(ML: 5.73477564103)
(EV: 7.20813771518)
(NW: 7.24518878651)
(DL: 7.44544898813)
(AL: 8.62370894203)
(US: 8.68941529793)
(AA: 8.73400508165)

--- CMH ---

(DH: 3.49111470113)
(AA: 3.51564734686)
(NW: 4.04155500526)
(ML: 4.36645962733)
(DL: 4.71344133974)
(PI: 5.20129487934)
(EA: 5.93738938053)
(US: 5.99299168147)
(AL: 6.02097013345)
```

```
(RU: 6.10234585842)

--- SRQ ---

(TZ: -0.381996974281)
(RU: -0.0880330123796)
(YV: 3.40402193784)
(AA: 3.64747274529)
(UA: 3.95212206243)
(US: 3.96839828967)
(TW: 4.30467606502)
(NW: 4.85635924135)
(DL: 4.86917943416)
(XE: 5.03554868624)
```

Question 2.2

```
--- JFK ---

(SWF: -10.5)
(MYR: 0.0)
(ABQ: 0.0)
(ISP: 0.0)
(ANC: 0.0)
(UCA: 1.91701244813)
(BGR: 3.21028037383)
(BQN: 3.60622761091)
(CHS: 4.40271055179)
(STT: 4.50210682155)

--- SEA ---

(EUG: 0.0)
(PIH: 1.0)
(PSC: 2.65051903114)
(CVG: 3.8787445578)
(MEM: 4.26022369801)
(CLE: 5.17016949153)
(BLI: 5.19824913369)
(YKM: 5.37964774951)
(SNA: 5.40625079405)
(LIH: 5.48108108108)

--- BOS ---

(SWF: -5.0)
(ONT: -3.0)
(GGG: 1.0)
(AUS: 1.20870767104)
(LGA: 3.05401785714)
(MSY: 3.2464678179)
(LGB: 5.13617677287)
(OAK: 5.78321003538)
(MDW: 5.89563753682)
(BDL: 5.98270484831)

--- CMH ---

(SYR: -5.0)
(AUS: -5.0)
(OMA: -5.0)
(MSN: 1.0)
(CLE: 1.10498687664)
(SDF: 1.35294117647)
(CAK: 3.70039421813)
(SLC: 3.93928571429)
(MEM: 4.15202156334)
(IAD: 4.15810344828)

--- SRQ ---

(EYW: 0.0)
(TPA: 1.32885132539)
```

```
(IAH: 1.44455747711)
(MEM: 1.70295983087)
(FLL: 2.0)
(BNA: 2.06231454006)
(MCO: 2.36453769887)
(RDU: 2.53540070988)
(MDW: 2.83812355467)
(CLT: 3.35836354221)
```

Question 2.4

```
LGA -> BOS: 1.48386483871
BOS -> LGA: 3.78411814784
OKC -> DFW: 5.07023374012
MSP -> ATL: 6.73700797367
```

Question 3.1

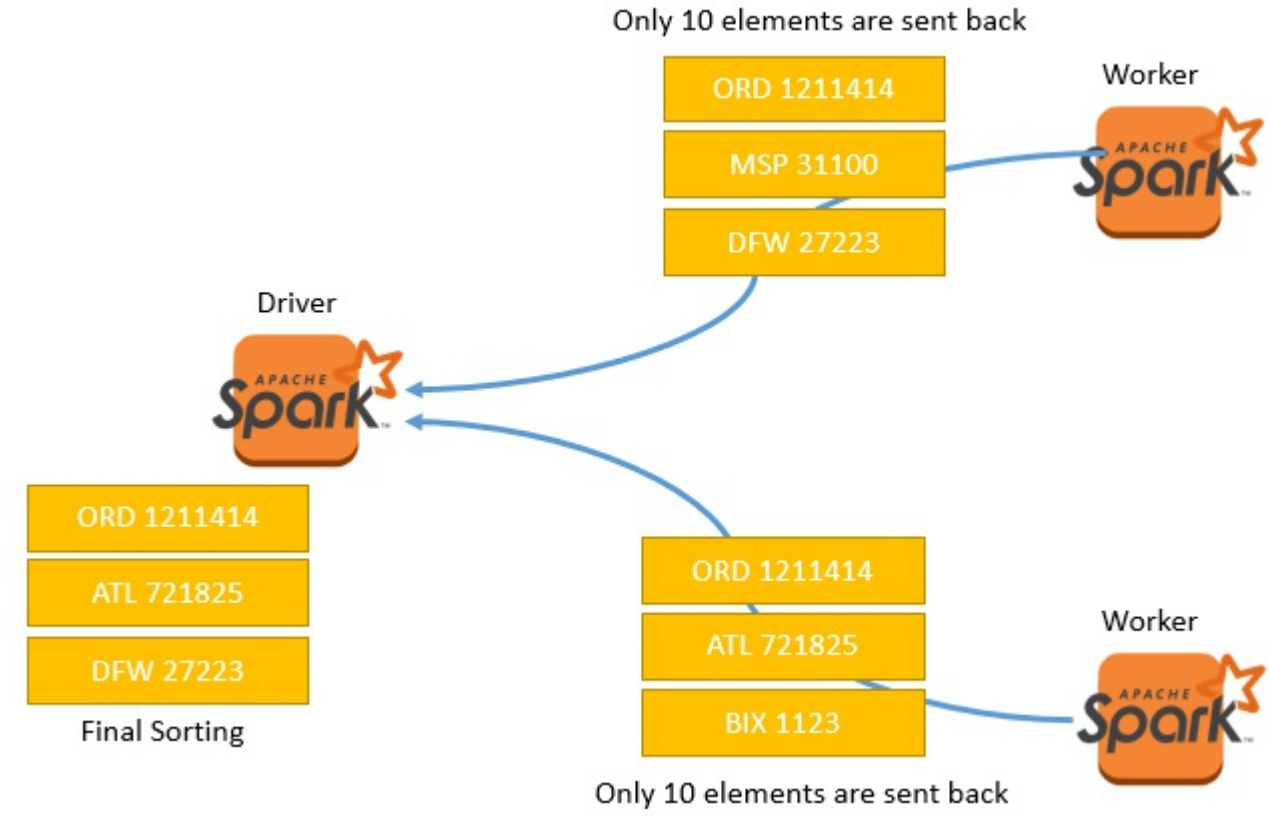
Question 3.1 is not needed by description of Task 2 Overview. But if you're interested, check out solution in documentation of Task 1.

Question 3.2

```
BOS -> ATL on 2008-04-03: Flight: FL 270 at 06:00. Arrival Delay: 7.0
ATL -> LAX on 2008-04-05: Flight: FL 40 at 18:52. Arrival Delay: -2.0

PHX -> JFK on 2008-09-07: Flight: B6 178 at 11:30. Arrival Delay: -25.0
JFK -> MSP on 2008-09-09: Flight: NW 609 at 17:50. Arrival Delay: -17.0

DFW -> STL on 2008-01-24: Flight: AA 1336 at 07:05. Arrival Delay: -14.0
STL -> ORD on 2008-01-26: Flight: AA 2245 at 16:55. Arrival Delay: -5.0

LAX -> MIA on 2008-05-16: Flight: AA 280 at 08:20. Arrival Delay: 10.0
MIA -> LAX on 2008-05-18: Flight: AA 456 at 19:30. Arrival Delay: -19.0
```

Optimizations

Calculating top ten values on each worker and aggregate results on the driver



To reduce the traffic, we cut off the amount of records to just the top 10 most popular in each partition. Workers only have to send 10 records for each RDD to the driver.

- Reduces network traffic since less data has to be sent from worker to driver
- Sort will be faster and less CPU is used. Sort by key function doesn't have to sort all the elements after each iteration.

```
def cutOffTopTen(iterable):
        topTen = []
        for tupl in iterable:
                if len(topTen) < 10:
                        topTen.append(tupl)
                        topTen.sort(reverse=True)
                elif topTen[9][0] < tupl[0]:
                        topTen[9] = tupl
                        topTen.sort(reverse=True)
        return iter(topTen)
```

```
sorted.transform(lambda rdd: rdd.mapPartitions(cutOffTopTen))
```

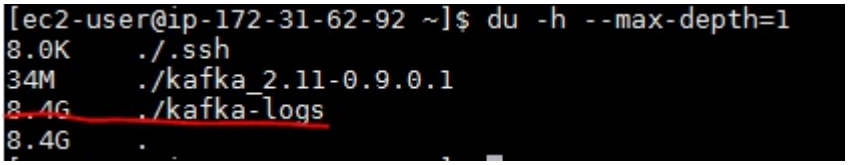
Then we sort RDDs by popularity.

```
sorted = sorted.transform(lambda rdd: rdd.sortByKey(False))
```

Cutting off irrelevant data from 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.



Streaming and Batch Comparison

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.

Considering type of questions & computations

- Hadoop map-reduce 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 iterative micro-batch approach causes more computation overhead. The final results take longer to produce with stream processing, because all the data has to 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.)

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 map-reduce 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.

Command Reference

If the reader is interested, here are the commands that were used to submit Spark jobs.

- GitHub link for terminal commands