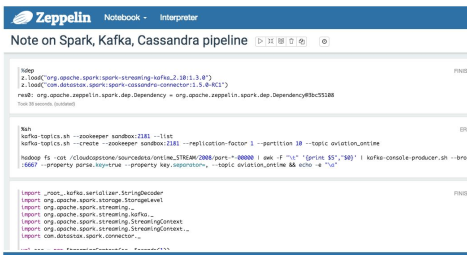
Task 2: Stream Processing with Spark Streaming

Platform setup, Dataset analysis, Cleaning the dataset

To perform Task 2, I set up a real-time data pipeline with Kafka v0.8.2, Spark Streaming v1.3.1 and Cassandra v2.1. Scala platform is v2.10 and Java v1.7. For further details about my clustered platform setup and cleaning the dataset, please refer to my Task 1 report.

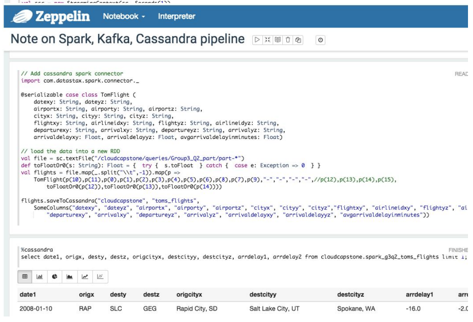


Note on Spark, Kafka, Cassandra pipeline

```
%dep
z.load("org.apache.spark:spark-streaming-kafka_2.10:1.3.0")
z.load("com.datastax.spark:spark-cassandra-connector:1.5.0-RC1")
res0: org.apache.zeppelin.spark.dep.Dependency = org.apache.zeppelin.spark.dep.Dependency@3bc55108
Took 38 seconds. (outdated)
```

```
%sh
kafka-topics.sh --zookeeper sandbox:2181 --list
kafka-topics.sh --create --zookeeper sandbox:2181 --replication-factor 1 --partition 10 --topic aviation_ontime

hadoop fs -cat /cloudcapstone/sourcedata/ontime_STREAM/2008/part-*-00000 | awk -F "\t" '{print $5","$0}' | kafka-console-producer.sh --bro
:6667 --property parse.key=true --property key.separator=, --topic aviation_ontime && echo -e "\a"
```

```
import _root_.kafka.serializer.StringDecoder
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.StreamingContext._
import com.datastax.spark.connector._
```



Note on Spark, Kafka, Cassandra pipeline

```
// Add cassandra spark connector
import com.datastax.spark.connector._

@serializable case class TomFlight (
    datexy: String, dateyz: String,
    airportx: String, airporty: String, airportz: String,
    cityx: String, cityy: String, cityz: String,
    flightxy: String, airlineidxy: String, flightyz: String, airlineidyz: String,
    departurexy: String, arrivalxy: String, departureyz: String, arrivalyz: String,
    arrivaldelayxy: Float, arrivaldelayyz: Float, avgarrivaldelayinminutes: Float)

// load the data into a new RDD
val file = sc.textFile("/cloudcapstone/queries/Group3_Q2_part/part-*")
def toFloatOr0(s: String): Float = {  try {  s.toFloat  } catch {  case e: Exception => 0  } }
val flights = file.map(_.split("\\t",-1)).map(p =>
    TomFlight(p(10),p(11),p(0),p(1),p(2),p(3),p(4),p(5),p(6),p(8),p(7),p(9),"-","-","-","-",//p(12),p(13),p(14),p(15),
        toFloatOr0(p(12)),toFloatOr0(p(13)),toFloatOr0(p(14))))

flights.saveToCassandra("cloudcapstone", "toms_flights",
    SomeColumns("datexy", "dateyz", "airportx", "airporty", "airportz", "cityx", "cityy", "cityz","flightxy", "airlineidxy", "flightyz", "ai
        "departurexy", "arrivalxy", "departureyz", "arrivalyz", "arrivaldelayxy", "arrivaldelayyz", "avgarrivaldelayinminutes"))
```

```
%cassandra
select date1, origx, desty, destz, origcityx, destcityy, destcityz, arrdelay1, arrdelay2 from cloudcapstone.spark_g3q2_toms_flights limit 1;
```

date1	origx	desty	destz	origcityx	destcityy	destcityz	arrdelay1	arr
2008-01-10	RAP	SLC	GEG	Rapid City, SD	Salt Lake City, UT	Spokane, WA	-16.0	-2.0

Exploring Spark and Spark Streaming

Approaching the new Task, I installed and configured Zeppelin tool in Ambari and played with its UI to explore Spark and Spark Streaming API.

Zeppelin integrates all in one Spark console and many other tools, among all, Kafka and Cassandra. I managed to automate some tasks, like loading libraries in the classpath and preparing HDFS filesystem and Cassandra, and run my first Spark scala commands, test the behaviour of some transformations and, more important, testing how Spark Cassandra connector works and effectively stores data in Cassandra!

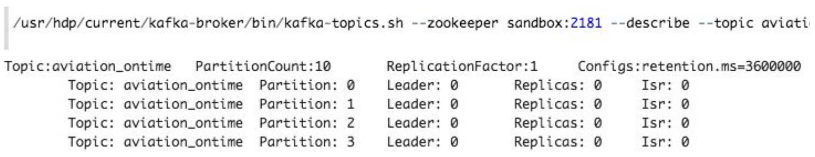
Java Spark

After many attempts, a few successful, I decided that I needed to write more advanced code to make the best possible from Spark streaming transformations and I moved to Spark Java.

Kafka

I used Kafka to stream the data source. I set up one Kafka broker on purpose, local to one of my cluster nodes, and created one topic, *aviation_ontime*.

I used Kafka-console-producer shell script, with awk converting from a simple message to a keyed-message.



```
/usr/hdp/current/kafka-broker/bin/kafka-topics.sh --zookeeper sandbox:2181 --describe --topic aviati

Topic:aviation_ontime   PartitionCount:10       ReplicationFactor:1     Configs:retention.ms=3600000
        Topic: aviation_ontime  Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: aviation_ontime  Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: aviation_ontime  Partition: 2    Leader: 0       Replicas: 0     Isr: 0
        Topic: aviation_ontime  Partition: 3    Leader: 0       Replicas: 0     Isr: 0
```

```
hadoop fs -cat /cloudcapstone/sourcedata/ontime_CLEAN/*/part-* | awk -F "\t" '{print $5","$0}' | kafka-console-producer.sh --broker-list node2:6667 --property parse.key=true --property key.separator=, --topic aviation_ontime && echo -e "\a"
```

Cassandra

Spark Cassandra connector is my choice to process the data stream and send the results to the store. I successfully learned how to configure and use it to update the store with partial results, each streaming round, in a real-time streaming fashion.

Parallelism

Most of my time for Task 2 was spent in exploring Spark API. With all tools in a pipeline, it is very important to set up the parallelism level and be able to scale out the whole process.

Kafka is very flexible in creating several partitions within a topic. The partitions are necessary to allow a consumer to use more threads in reading from a stream. I applied the *directStream* approach, where Spark creates simply as many RDD partitions as Kafka partitions and reads data in parallel. Then, it's just a matter of defining the right key for your RDD transformation and all your data will be processed in parallel.

Dataset preparation for a real-time streaming

After the Task 1 I have in my HDFS filesystem the source data cleaned, which I can use for Task 2.

To solve the problem G3Q2 in the best possible way, I prepared a specific dataset for year 2008 in which I had a stream of events (flights) ordered by date. This is not a constraint to my process, but it makes same optimizations more effective, as I discuss them later on in this document.

Perhaps you would agree that such an assumption of receiving an ordered data stream is appropriate for a streaming process. According to my concept of data streaming pipeline, I strongly wanted to work in a real-time streaming fashion, as much as possible. This is the reason why I stressed my solution to compute and store partial results in real-time.

Solving the queries with Spark Streaming

I developed 2 Spark jobs:

G12StreamingJob. It groups together 5 processors for the 5 queries of groups 1&2 (G1Q1, G1Q3, G2Q1, G2Q2, G2Q4). This job runs the 5 processors in parallel. Each processor is attached to a dedicated stream and receives data from each of Kafka's topics partition.

```
G3StreamingJob Usage: <kafka-broker-host:port> <kafka-topic> <output-path> <cassandra-host> [<stream-freq-in-secs(def:1) <cassandra-concurrent-writes> <cassandra-batch-size>]
```

G3StreamingJob. A "stateful" job, optimized and dedicated to the G3Q2 problem.

G3StreamingJob Usage: <kafka-broker-host:port> <kafka-topic> <output-path> <cassandra-host> [<stream-freq-in-secs(def:1) <purge-days(def:31) run-for-days(def:365) <cassandra-concurrent-writes> <cassandra-batch-size>]

I let some parameters to make the jobs configurable. Among all, please notice the frequency of streaming, which sets the batch duration in Spark Streaming context, and some optional settings to improve the performance of Spark Cassandra connector.

In addition, I developed a *KafkaConsumer* to help with the creation of the stream. The current implementation uses a direct stream (KafkaUtils.createDirectStream). I tried also the approach with receivers, but I noticed no difference and opted for the one described in Spark documentation as a more simple and efficient solution.

The main difference between my two jobs is how they accumulate and store the results.

Accumulators vs. UpdateStateByKey

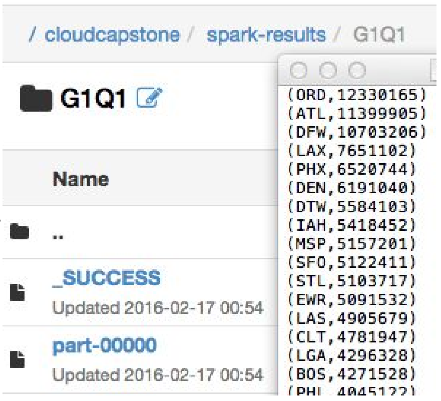
Both jobs process a bit of the stream for each period and compute the partial results. But while the G12 job uses Spark concept of Accumulator, collecting them all and flush them to disk and to datastore at frequent intervals (which I called *savepoints*), the G3 job uses Spark updateStateByKey operation.

This operation allows to transform the stream into a stateful RDD that can be updated by a user defined function. The RDD is of course partitioned and distributed, like Accumulator is, but the latter has got the drawback of collecting all the "local" values into a global one (centralized) before storing them. *UpdateStateByKey* implicitly uses *Checkpointing* to be fail safe.

With a stateful RDD, I was able to combine in memory all the flights and, finally, store the partial results into the datastore as soon as the streaming process was producing them in real-time! I designed some optimizations to keep in memory only the relevant portion of flights. For more details, please see below the description of the G3Q2 problem.

Please refer to my Task 1 report for the definitions of: *On-time arrival/departure performance*, *Average arrival/departure delay*, *Diverted/Cancelled flights*.

Group 1 Q1



```
(ORD,12330165)
(ATL,11399905)
(DFW,10703206)
(LAX,7651102)
(PHX,6520744)
(DEN,6191040)
(DTW,5584103)
(IAH,5418452)
(MSP,5157201)
(SFO,5122411)
(STL,5103717)
(EWR,5091532)
(LAS,4905679)
(CLT,4781947)
(LGA,4296328)
(BOS,4271528)
(PHL,4045122)
```

G12StreamingJob uses *G1Q1Processor* to process the data and accumulate the results into *G1Q1Accumulable*.

This processor uses the following transformations to get the result:

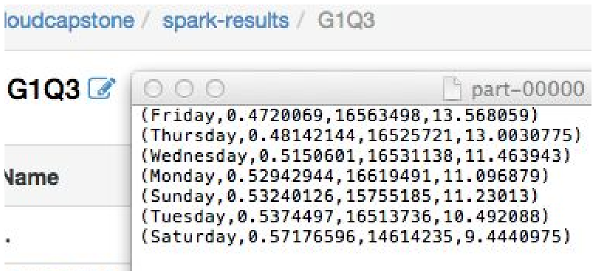
- parses the incoming flights' data, including the cancelled/diverted ones, and for each flight, it applies a mapping to create a pair RDD with *key = origin airport* and a second pair RDD with *key = destination airport*;
- makes a union of the two RDDs and reduces it by key, which gives the count of IN/OUT flight from every airport;

The final RDD *countOfFlightsByAirport* is still partitioned by key, locally to all the executors. Each partition can be processed and the data added to the shared accumulator.

The count of flights is increased in each round.

At the next savepoint, the accumulator value is collected from the master process, and a partial result is ready to be stored in HDFS and Cassandra. In my run, a savepoint usually happens every 30 secs.

Group 1 Q3



```
(Friday,0.4720069,16563498,13.568059)
(Thursday,0.48142144,16525721,13.0030775)
(Wednesday,0.5150601,16531138,11.463943)
(Monday,0.52942944,16619491,11.096879)
(Sunday,0.53240126,15755185,11.23013)
(Tuesday,0.5374497,16513736,10.492088)
(Saturday,0.57176596,14614235,9.4440975)
```

G12StreamingJob uses *G1Q3Processor* and producess results into the shared *G1Q3Accumulable*. See the screenshot of my result (*day, on-time performance, #flights, mean delay*).

This processor uses the following transformations:

- parses the incoming flights' data, skipping the cancelled/diverted ones, and for each flight, it applies a mapping to create a pair RDD by *key = day of the week* and value including the arrival delay, if positive;
- reducing the pair RDD by key, it is possible to count the number of flights, count the flights on-time and compute the on-time arrival performance, finally, summing the arrival delays and computing a partial average delay.

The final RDD *perfByWeekDay* is processed for each partition and the result added to the accumulator. The count of flights and the arrival delay are increased, the mean arrival delay and the on-time arrival performance computed in each round.

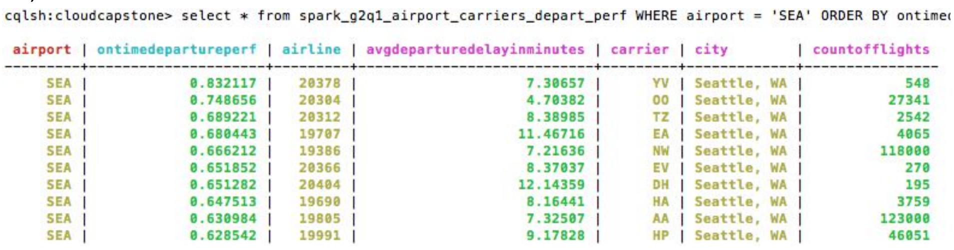
A savepoint occurs after a configured number of periods, then the partial results are all saved to HDFS and stored in Cassandra.

Group 2 Q1

G12StreamingJob uses *G2Q1Processor* and *G2Q1Accumulable* to process the stream of flights, using the following transformations:

- it parses the flights, skipping the cancelled/diverted ones, and mapping a pair RDD with *key = origin airport + "|" + airline* and value including the departure delay, if positive;

- reducing the RDD by key, it aggregates the number of flights and computes the partial on-time departure performance and the partial average departure delay of the carriers by each airport;
- the current RDD is then mapped and grouped by the new key = origin airport, thus collecting the performance by airport, then the values are mapped with sort and limit to 10;



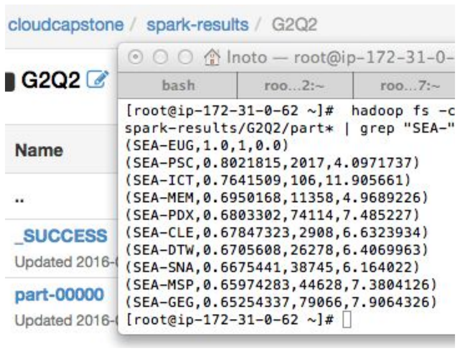
```
cqlsh:cloudcapstone> select * from spark_g2q1_airport_carriers_depart_perf WHERE airport = 'SEA' ORDER BY ontime

 airport | ontimedepartureperf | airline | avgdeparturedelayinminutes | carrier | city         | countofflights
---------+---------------------+---------+----------------------------+---------+--------------+----------------
     SEA |            0.832117 |   20378 |                    7.30657 |      YV | Seattle, WA |            548
     SEA |            0.748656 |   20304 |                    4.70382 |      OO | Seattle, WA |          27341
     SEA |            0.689221 |   20312 |                    8.38985 |      TZ | Seattle, WA |           2542
     SEA |            0.680443 |   19707 |                   11.46716 |      EA | Seattle, WA |           4065
     SEA |            0.666212 |   19386 |                    7.21636 |      NW | Seattle, WA |         118000
     SEA |            0.651852 |   20366 |                    8.37037 |      EV | Seattle, WA |            270
     SEA |            0.651282 |   20404 |                   12.14359 |      DH | Seattle, WA |            195
     SEA |            0.647513 |   19690 |                    8.16441 |      HA | Seattle, WA |           3759
     SEA |            0.630984 |   19805 |                    7.32507 |      AA | Seattle, WA |         123000
     SEA |            0.628542 |   19991 |                    9.17828 |      HP | Seattle, WA |          46051
```

The *performanceRank* RDD gives for every airport, 10 carriers in descending order of on-time departure performance. Each partition is processed and aggregated into the accumulable object, which will be stored in HDFS and Cassandra at the next savepoint.

Group 2 Q2

G12StreamingJob uses *G2Q2Processor* and *G2Q2Accumulable* to process the stream of flights. This process is very similar to the one described in *G2Q1Processor*, with the only difference in the key used to map and reduce the pair RDD.

In this case we use the *key = origin + "|" + destination* and the flights grouped by route (X-Y). In the same way described for G2Q1, I mapped the *performanceRank* RDD and applied the ranking. In the end, the results are aggregated into the accumulator and stored into HDFS and Cassandra, at the next savepoint.

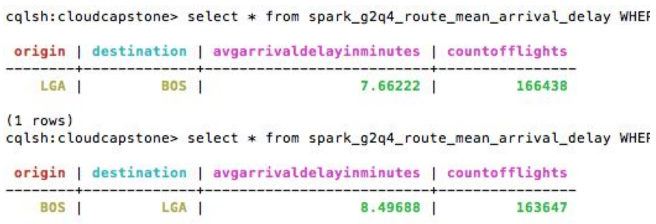


```
[root@ip-172-31-0-62 ~]#  hadoop fs -c
spark-results/G2Q2/part* | grep "SEA-"
(SEA-EUG,1.0,1,0.0)
(SEA-PSC,0.8021815,2017,4.0971737)
(SEA-ICT,0.7641509,106,11.905661)
(SEA-MEM,0.6950168,11358,4.9689226)
(SEA-PDX,0.6803302,74114,7.485227)
(SEA-CLE,0.67847323,2908,6.6323934)
(SEA-DTW,0.6705608,26278,6.4069963)
(SEA-SNA,0.6675441,38745,6.164022)
(SEA-MSP,0.65974283,44628,7.3804126)
(SEA-GEG,0.65254337,79066,7.9064326)
[root@ip-172-31-0-62 ~]#
```

Group 2 Q4

G12StreamingJob uses *G2Q4Processor* and *G2Q4Accumulable* to process the data and accumulate the results, by using the following transformations:

- parses the flights, skipping the cancelled/diverted ones, and mapping a pair RDD with *key = origin + "|" + destination*, including as value the arrival delay, if positive;
- reducing the RDD by key, it aggregates the average arrival delay of every route in the *meanArrivalByRoute* RDD.



```
cqlsh:cloudcapstone> select * from spark_g2q4_route_mean_arrival_delay WHEF

 origin | destination | avgarrivaldelayinminutes | countofflights
--------+-------------+--------------------------+----------------
    LGA |         BOS |                  7.66222 |         166438

(1 rows)
cqlsh:cloudcapstone> select * from spark_g2q4_route_mean_arrival_delay WHEF

 origin | destination | avgarrivaldelayinminutes | countofflights
--------+-------------+--------------------------+----------------
    BOS |         LGA |                  8.49688 |         163647
```

At the savepoint, the accumulated results are stored in HDFS and Cassandra.

Group 3 Q2

G3StreamingJob uses the *G3Q2Processor* to process the data stream. There is no accumulator. Instead, a state RDD is computed every round and produces new combinations and updates, soon saved in the store.

The following transformations are applied to the stream:

- the population of incoming flights, excluding the cancelled/diverted ones, is split into two RDD *flightsLeg1*, *flightsLeg2*;
- the two RDD contains multiple solutions for the same day and route (X-Y), then we can reduce by (*key = date + "|" + orig + "|" + dest)* and keep only the best flights, e.g. the ones with the lower arrival delay;
- at this point, the best flights from both legs can be mapped to a tuple with *(key = date1 + "|" + destY)* and *(value = FlighCombi)* - *FlighCombi* model is a composition of *[<leg1>, <leg2>]* matching the constraints of the same intermediate airport Y and the 2-day stop for sightseeing;

The resulting *bestFlightsByDateAndDestY* RDD contains all partial *FlighCombi*, composed of only one leg, i.e. *[<leg1>, null]* or *[null, <leg2>]*. Now we are ready to process this incoming flights against all the flight combinations from the previous rounds.

From that point, a state RDD is computed every round, by using the operation *updateStateByKey* with the function *mergeState*. The tuples in the state RDD, which are processed by the *mergeState* function are of type *G3Q2State*, which contains a *Map<String, Set<FlightCombi>>*, where the key is always *(date1 + "|" + destY)*. The given function processes the partitions of the state RDD to create new combinations and update the current combinations with better legs, eventually.

The state RDD grows in memory but it remains well distributed locally to the executors. The combinations computed in the state are sent to the data store at the end of every batch period. This real-time flowing of all the updates could result into a dangerous overloading of Cassandra. At this point I needed to design some optimizations to simplify the process and avoid the stress of the resources.

Optimization of G3Q2

First of all, I choosed *key = date1 + "|" + destY*, which allows the best partitioning of the flights. With the strict requirement of combining flights where the leg-2 is exactly 2 day after the leg-1, I was able to transform the *date2* of a leg-2 flight into the *date1* required by the key. This way, all the matching leg-1 and leg-2 are in the same set of *FlightCombi*, partitioned close together! Then, I introduced some useful optimizations, here described.

Flight events ordered, Time-window and Purge

If we assume a **partial/total ordering** of the events (thanks for introducing me to the concept, prof. Indy :-), it would be possible for Spark job to keep for each partition only the last two days of the streamed data, because for that partition (*key = date + "|" + destY)*, there will be no incoming event out the date range *[date, date+2d]*.

But total ordering is hard to achieve with Kafka broker using many partitions, we can instead try to achieve a partial order in an extended **time-window** (i.e. 10 days or a month).

For this reason I made the assumption that the stream should carry all the flights with the same date as much as possible **close together**, or at least **within a time window**. To put it simple, defining in my process a window of 31 days, it would mean that while the flights with date 01/04/2008 are streamed, it is still allowed that a flight with date 01/03/2008 comes late to be processed. But any flight before that date would not be processed correctly!

Then, to put it in practice, I simply ordered the events by date, which is a very plausible assumption for a real-time streaming system.

I designed algorithm to compute, at every round, the date of the less recent incoming event (*dateLB*) and define a configurable number of day, i.e 31. Then I can calculate *purgeDate = (dateLB - 31)* to purge the old combinations older than *purgeDate* (no more useful and already stored in Cassandra). That is! At the end of the round, I simply discard from the memory all the partial combinations out of the time-window *[purgeDate, ∞]*. In the next round, the state will be smaller, but still able to combine the incoming flights, because I assumed that they will have a date not before the purged window lower bound!

As you can see, at this point it's just a matter of ordering the events, configure the right time-window, together with using a limited number of Kafka partition, which should not shuffle the data too much… and we have got a smarter process producing results 100% correct!

Storing updates only

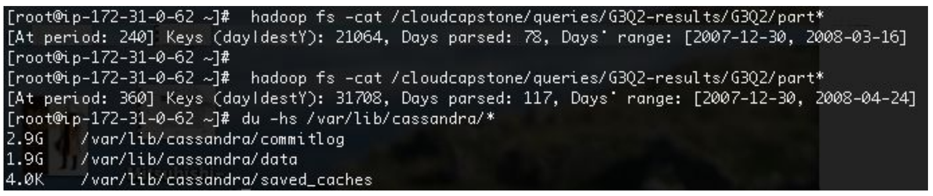
To reduce the load in the store, I optimized the job to flag a *FlightCombi* when it is created or updated, and to store into Cassandra only the flagged ones (check the video to see how low is the resulting write requests rate).

Kafka optimization

I produced keyed messages to make the broker maintain the order of the stream, at least within each partition. With *directStream* approach, there is no need to use groups for the consumers, they all will receive the same messages queued in the one topic.

Cassandra optimization

Storing with savepoints vs. storing in real-time really makes the difference, with the first solution to avoid overloading the store with a lot of partial updates. You can see in this screenshot a early run of G3StreamingJob. Please notice the huge commit log of one of my Cassandra nodes, with my initial unoptimized G3Q2 implementation. At the beginning, I was favouring the first option, until I finally optimized the G3Q2 solution to store only the updates and then the *storeToCassandra* operation attached to the stream RDD started producing a reasonable number of writes (around 100/sec).

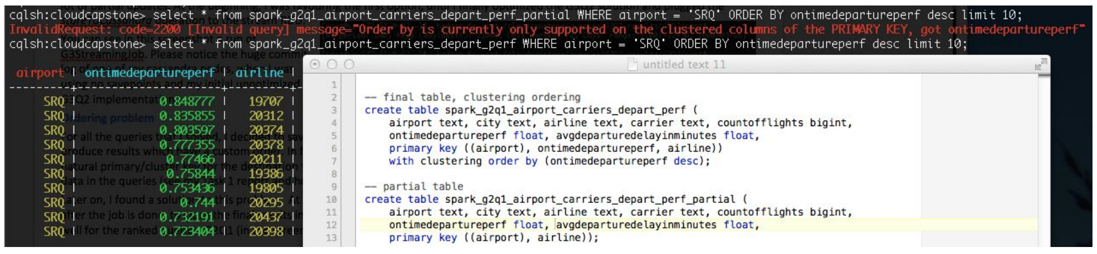


```
[root@ip-172-31-0-62 ~]#  hadoop fs -cat /cloudcapstone/queries/G3Q2-results/G3Q2/part*
[At period: 240] Keys (day|destY): 21064, Days parsed: 78, Days' range: [2007-12-30, 2008-03-16]
[root@ip-172-31-0-62 ~]#
[root@ip-172-31-0-62 ~]#  hadoop fs -cat /cloudcapstone/queries/G3Q2-results/G3Q2/part*
[At period: 360] Keys (day|destY): 31708, Days parsed: 117, Days' range: [2007-12-30, 2008-04-24]
[root@ip-172-31-0-62 ~]# du -hs /var/lib/cassandra/*
2.9G    /var/lib/cassandra/commitlog
1.9G    /var/lib/cassandra/data
4.0K    /var/lib/cassandra/saved_caches
```

Ordering problem

For all the queries that I solved, I decided to save the result into both HDFS and Cassandra. Saving to HDFS, it was easier to produce results which have a custom order. Instead, while streaming updates into Cassandra with the connector, I was forced to use the natural primary/cluster key for the destination tables, and this prevents defining the performance indicator as a key to allow ordering data in the queries (if you don't know how to obtain ordered results into Cassandra, please see my Task 1 report).

Later on, I found a solution to this problem. At savepoints, I used to store the data in a table *_partial*, then having one last savepoint, right after the job is done, to store the final results into the final table, which has a different primary key and allows ordering. It worked well for the ranked queries, G2Q1 (see the two tables' definition and the results in the screenshot) and G2Q2.



```
cqlsh:cloudcapstone> select * from spark_g2q1_airport_carriers_depart_perf_partial WHERE airport = 'SRQ' ORDER BY ontimedepartureperf desc limit 10;
InvalidRequest: code=2200 [Invalid query] message="Order by is currently only supported on the clustered columns of the PRIMARY KEY, got ontimedepartureperf"
cqlsh:cloudcapstone> select * from spark_g2q1_airport_carriers_depart_perf WHERE airport = 'SRQ' ORDER BY ontimedepartureperf desc limit 10;

 airport | ontimedepartureperf | airline |
---------+---------------------+---------+
     SRQ |            0.848777 |   19707 |
     SRQ |            0.835855 |   20312 |
     SRQ |            0.803597 |   20374 |
     SRQ |            0.777355 |   20378 |
     SRQ |             0.77466 |   20211 |
     SRQ |             0.75844 |   19386 |
     SRQ |            0.753436 |   19805 |
     SRQ |               0.744 |   20295 |
     SRQ |            0.732191 |   20437 |
     SRQ |            0.723404 |   20398 |
```

untitled text 11

```

-- final table, clustering ordering
create table spark_g2q1_airport_carriers_depart_perf (
    airport text, city text, airline text, carrier text, countofflights bigint,
    ontimedepartureperf float, avgdeparturedelayinminutes float,
    primary key ((airport), ontimedepartureperf, airline))
    with clustering order by (ontimedepartureperf desc);

-- partial table
create table spark_g2q1_airport_carriers_depart_perf_partial (
    airport text, city text, airline text, carrier text, countofflights bigint,
    ontimedepartureperf float, avgdeparturedelayinminutes float,
    primary key ((airport), airline));
```

Final Considerations

Comparing my batch solution (Task 1) against my streaming pipeline solution (Task 2), one main difference: the pipeline gives early results, even if partial, and produces a final result which is exactly as the batch one, in almost the same processing time.

We can incur in such a situation, when the pipeline final results are not matching the batch ones, i.e. if we missed some messages or interrupted the stream earlier. But the batch still suffers from the pain of waiting till the end to see a result that, maybe, is not significantly different from a partial one produced by the pipeline. Mostly it depends on what is our goal.

For the Tom's problem, both the batch and the pipeline give the best solution at the end of the process. And I could say my Spark streaming solution was agile enough to process the data set with the same performance or even better than my batch solution. Comparing the time, on the same cluster, I've got 1.5h for the batch and 2h for the pipeline but I cannot state which one is faster, because it depends on the efficiency of the solution.

My final opinion in points: (just comparing the two solutions, not Spark vs. Hadoop ;)

- Hadoop MR jobs (via Pig or other tools) over YARN/Tez are easier to design and to code in practice … +1 batch
- Stateful, all-in-memory computations are complex but feasible (thanks to Spark), if well designed … +1 stream
- The pipeline gives (partial) results faster … +1 stream
- I had to put much more effort to setup and maintain the pipeline than maintaining the batch platform … +1 batch

No loser, both win.

Demo video

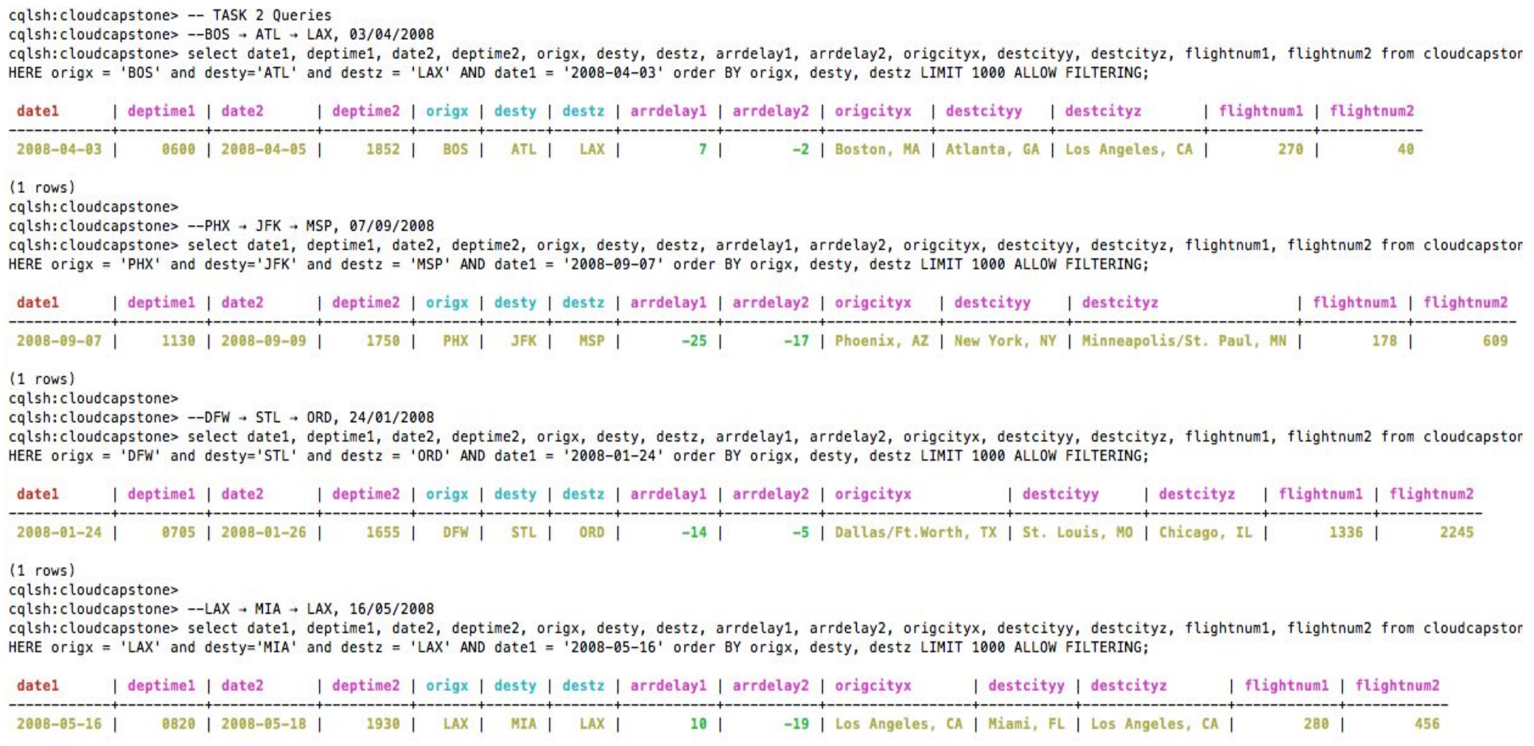
A video showing my work for the Task 2 is on YouTube, https://youtu.be/0bO5Y_G_23w

If you are interested, there is also an extra video of exploring the source code of my solution, https://youtu.be/Xp7kHObxGV4

For any problem, or further question, please email to *luiginoto@gmail.com*.

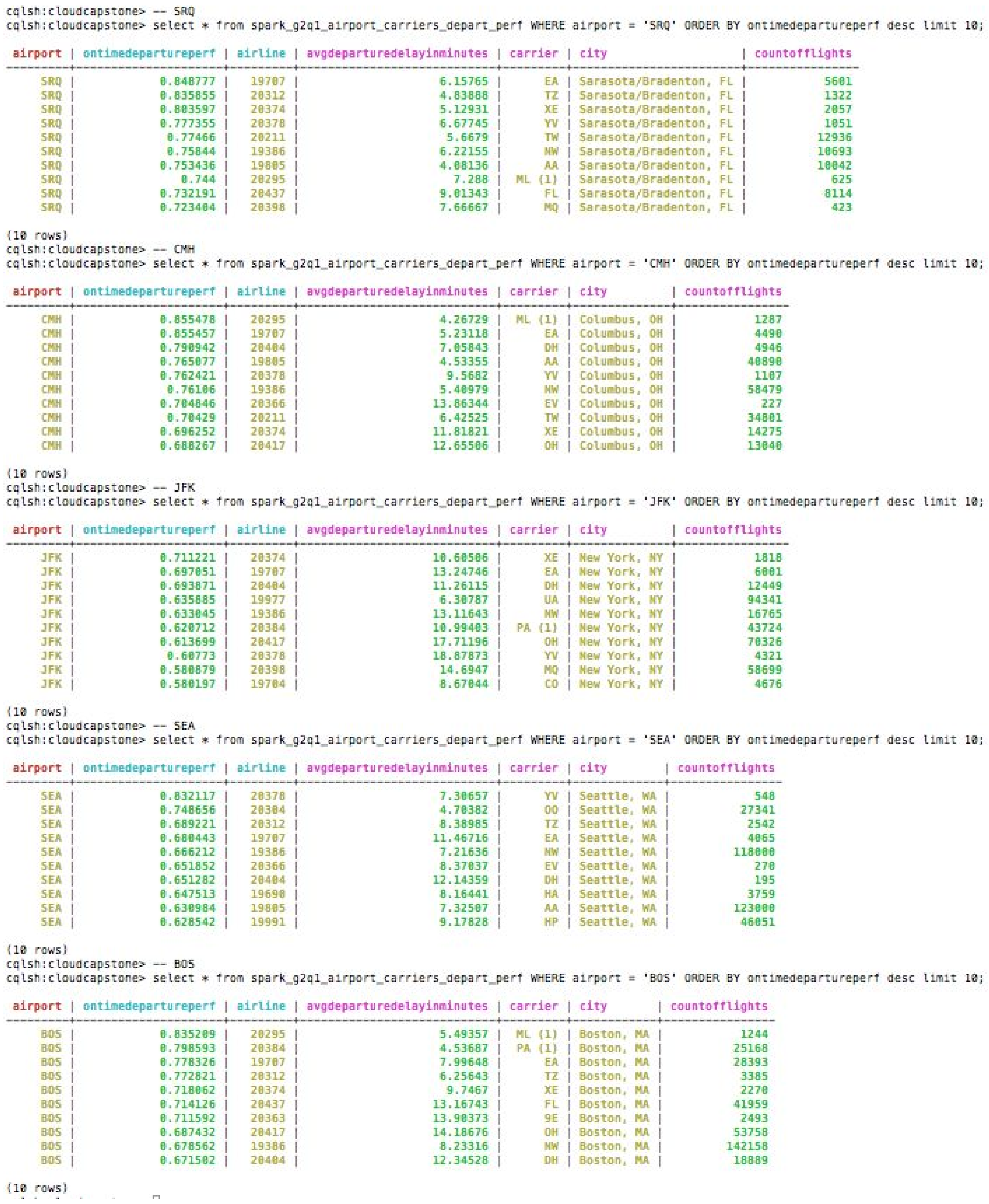
Appendix - Full results

G3Q2



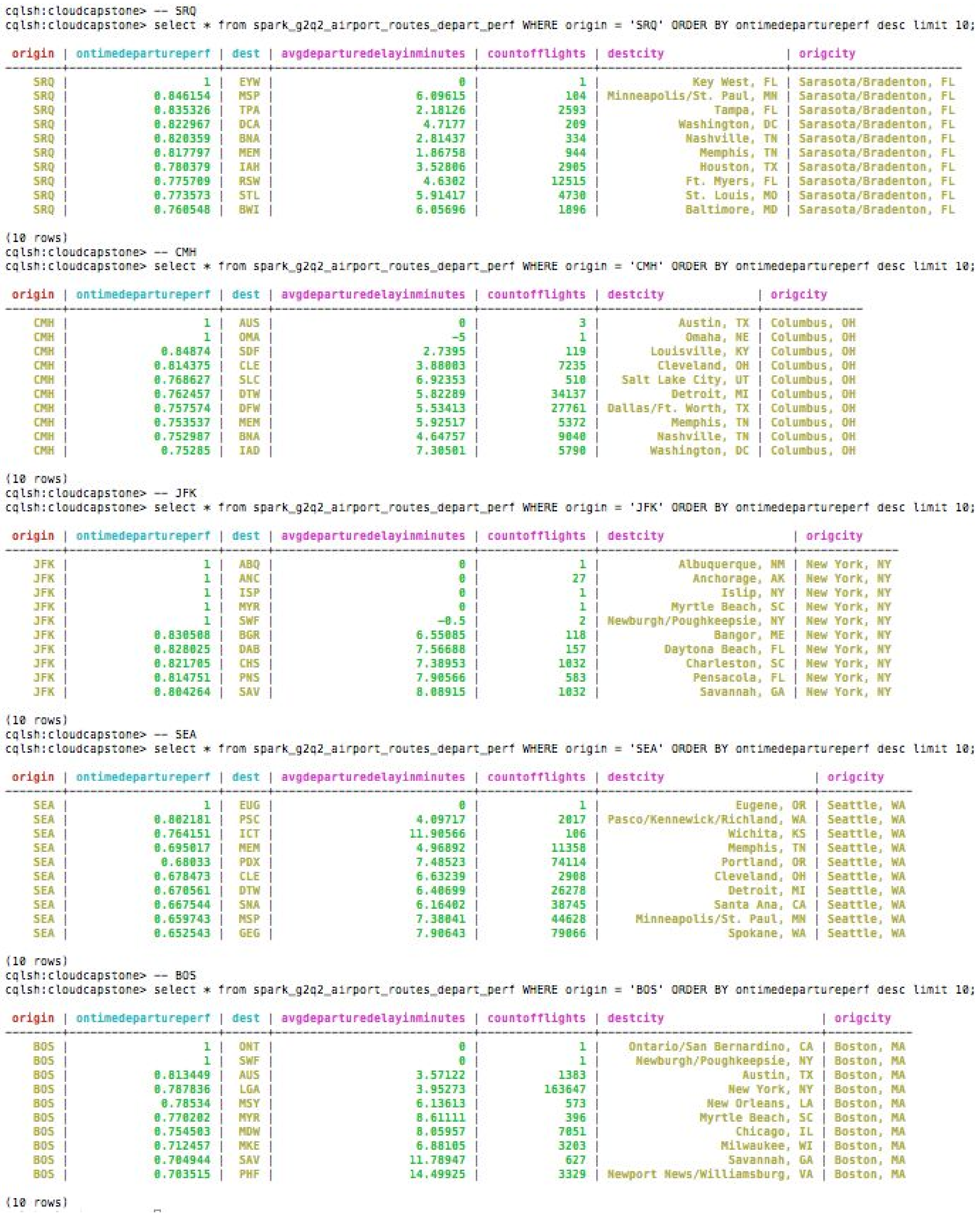
```
cqlsh:cloudcapstone> -- TASK 2 Queries
cqlsh:cloudcapstone> --BOS → ATL → LAX, 03/04/2008
cqlsh:cloudcapstone> select date1, deptime1, date2, deptime2, origx, desty, destz, arrdelay1, arrdelay2, origcityx, destcityy, destcityz, flightnum1, flightnum2 from cloudcapstor
HERE origx = 'BOS' and desty='ATL' and destz = 'LAX' AND date1 = '2008-04-03' order BY origx, desty, destz LIMIT 1000 ALLOW FILTERING;

 date1      | deptime1 | date2      | deptime2 | origx | desty | destz | arrdelay1 | arrdelay2 | origcityx  | destcityy   | destcityz       | flightnum1 | flightnum2
------------+----------+------------+----------+-------+-------+-------+-----------+-----------+------------+-------------+-----------------+------------+------------
 2008-04-03 |     0600 | 2008-04-05 |     1852 |   BOS |   ATL |   LAX |         7 |        -2 | Boston, MA | Atlanta, GA | Los Angeles, CA |        270 |         40

(1 rows)
cqlsh:cloudcapstone>
cqlsh:cloudcapstone> --PHX → JFK → MSP, 07/09/2008
cqlsh:cloudcapstone> select date1, deptime1, date2, deptime2, origx, desty, destz, arrdelay1, arrdelay2, origcityx, destcityy, destcityz, flightnum1, flightnum2 from cloudcapstor
HERE origx = 'PHX' and desty='JFK' and destz = 'MSP' AND date1 = '2008-09-07' order BY origx, desty, destz LIMIT 1000 ALLOW FILTERING;

 date1      | deptime1 | date2      | deptime2 | origx | desty | destz | arrdelay1 | arrdelay2 | origcityx   | destcityy    | destcityz                  | flightnum1 | flightnum2
------------+----------+------------+----------+-------+-------+-------+-----------+-----------+-------------+--------------+----------------------------+------------+------------
 2008-09-07 |     1130 | 2008-09-09 |     1750 |   PHX |   JFK |   MSP |       -25 |       -17 | Phoenix, AZ | New York, NY | Minneapolis/St. Paul, MN |        178 |        609

(1 rows)
cqlsh:cloudcapstone>
cqlsh:cloudcapstone> --DFW → STL → ORD, 24/01/2008
cqlsh:cloudcapstone> select date1, deptime1, date2, deptime2, origx, desty, destz, arrdelay1, arrdelay2, origcityx, destcityy, destcityz, flightnum1, flightnum2 from cloudcapstor
HERE origx = 'DFW' and desty='STL' and destz = 'ORD' AND date1 = '2008-01-24' order BY origx, desty, destz LIMIT 1000 ALLOW FILTERING;

 date1      | deptime1 | date2      | deptime2 | origx | desty | destz | arrdelay1 | arrdelay2 | origcityx           | destcityy     | destcityz   | flightnum1 | flightnum2
------------+----------+------------+----------+-------+-------+-------+-----------+-----------+---------------------+---------------+-------------+------------+------------
 2008-01-24 |     0705 | 2008-01-26 |     1655 |   DFW |   STL |   ORD |       -14 |        -5 | Dallas/Ft.Worth, TX | St. Louis, MO | Chicago, IL |       1336 |       2245

(1 rows)
cqlsh:cloudcapstone>
cqlsh:cloudcapstone> --LAX → MIA → LAX, 16/05/2008
cqlsh:cloudcapstone> select date1, deptime1, date2, deptime2, origx, desty, destz, arrdelay1, arrdelay2, origcityx, destcityy, destcityz, flightnum1, flightnum2 from cloudcapstor
HERE origx = 'LAX' and desty='MIA' and destz = 'LAX' AND date1 = '2008-05-16' order BY origx, desty, destz LIMIT 1000 ALLOW FILTERING;

 date1      | deptime1 | date2      | deptime2 | origx | desty | destz | arrdelay1 | arrdelay2 | origcityx       | destcityy | destcityz       | flightnum1 | flightnum2
------------+----------+------------+----------+-------+-------+-------+-----------+-----------+-----------------+-----------+-----------------+------------+------------
 2008-05-16 |     0820 | 2008-05-18 |     1930 |   LAX |   MIA |   LAX |        10 |       -19 | Los Angeles, CA | Miami, FL | Los Angeles, CA |        280 |        456
```

G2Q1



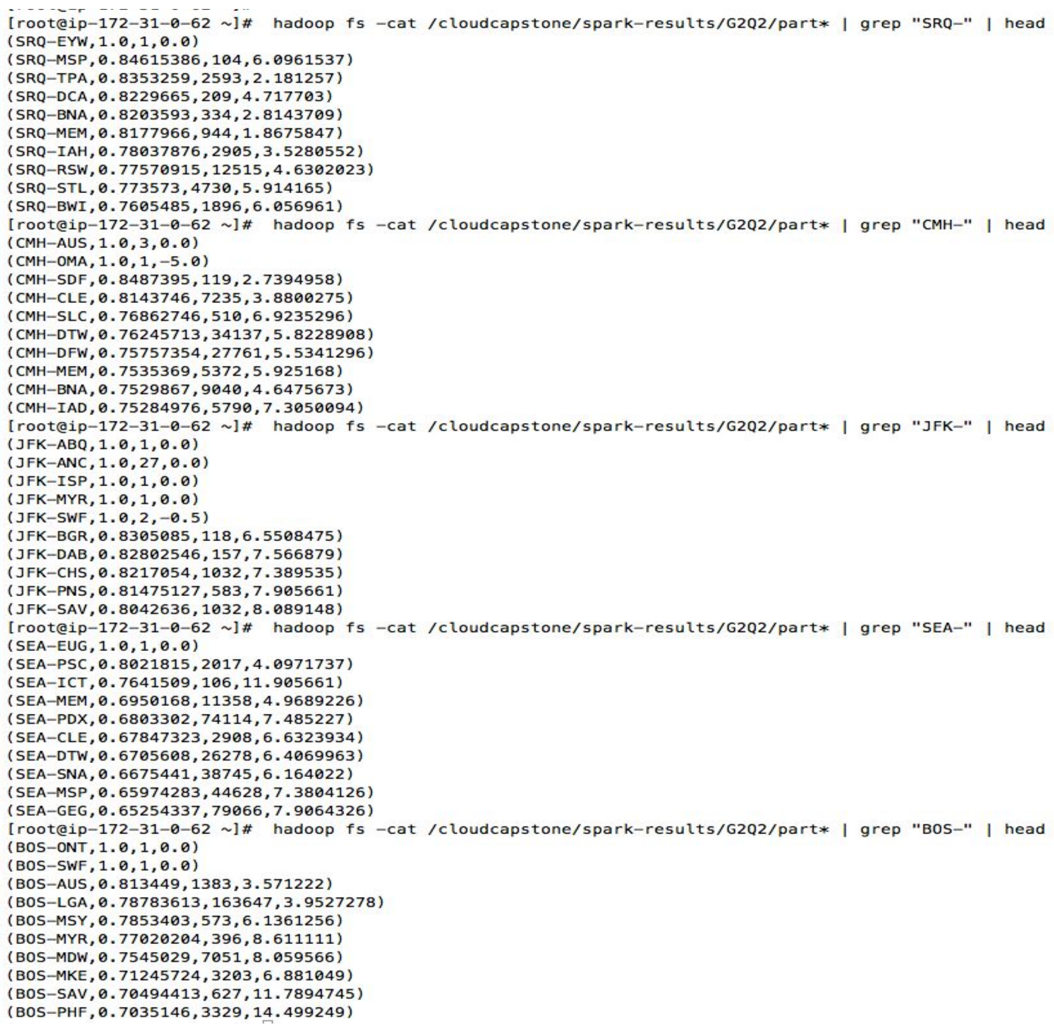
```
cqlsh:cloudcapstone> -- SRQ
cqlsh:cloudcapstone> select * from spark_g2q1_airport_carriers_depart_perf WHERE airport = 'SRQ' ORDER BY ontimedepartureperf desc limit 10;

 airport | ontimedepartureperf | airline | avgdeparturedelayinminutes | carrier | city                   | countofflights
---------+---------------------+---------+----------------------------+---------+------------------------+----------------
     SRQ |            0.848777 |   19707 |                    6.15765 |      EA | Sarasota/Bradenton, FL |           5601
     SRQ |            0.835855 |   20312 |                    4.83888 |      TZ | Sarasota/Bradenton, FL |           1322
     SRQ |            0.803597 |   20374 |                    5.12931 |      XE | Sarasota/Bradenton, FL |           2057
     SRQ |            0.777355 |   20378 |                    6.67745 |      YV | Sarasota/Bradenton, FL |           1051
     SRQ |             0.77466 |   20211 |                     5.6679 |      TW | Sarasota/Bradenton, FL |          12936
     SRQ |             0.75844 |   19386 |                    6.22155 |      NW | Sarasota/Bradenton, FL |          10693
     SRQ |            0.753436 |   19805 |                    4.08136 |      AA | Sarasota/Bradenton, FL |          10042
     SRQ |               0.744 |   20295 |                      7.288 |  ML (1) | Sarasota/Bradenton, FL |            625
     SRQ |            0.732191 |   20437 |                    9.01343 |      FL | Sarasota/Bradenton, FL |           8114
     SRQ |            0.723404 |   20398 |                    7.66667 |      MQ | Sarasota/Bradenton, FL |            423

(10 rows)
cqlsh:cloudcapstone> -- CMH
cqlsh:cloudcapstone> select * from spark_g2q1_airport_carriers_depart_perf WHERE airport = 'CMH' ORDER BY ontimedepartureperf desc limit 10;

 airport | ontimedepartureperf | airline | avgdeparturedelayinminutes | carrier | city         | countofflights
---------+---------------------+---------+----------------------------+---------+--------------+----------------
     CMH |            0.855478 |   20295 |                    4.26729 |  ML (1) | Columbus, OH |           1287
     CMH |            0.855457 |   19707 |                    5.23118 |      EA | Columbus, OH |           4490
     CMH |            0.790942 |   20404 |                    7.05843 |      DH | Columbus, OH |           4946
     CMH |            0.765077 |   19805 |                    4.53355 |      AA | Columbus, OH |          40890
     CMH |            0.762421 |   20378 |                     9.5682 |      YV | Columbus, OH |           1107
     CMH |             0.76106 |   19386 |                    5.40979 |      NW | Columbus, OH |          58479
     CMH |            0.704846 |   20366 |                   13.86344 |      EV | Columbus, OH |            227
     CMH |             0.70429 |   20211 |                    6.42525 |      TW | Columbus, OH |          34801
     CMH |            0.696252 |   20374 |                   11.81821 |      XE | Columbus, OH |          14275
     CMH |            0.688267 |   20417 |                   12.65506 |      OH | Columbus, OH |          13040

(10 rows)
cqlsh:cloudcapstone> -- JFK
cqlsh:cloudcapstone> select * from spark_g2q1_airport_carriers_depart_perf WHERE airport = 'JFK' ORDER BY ontimedepartureperf desc limit 10;

 airport | ontimedepartureperf | airline | avgdeparturedelayinminutes | carrier | city         | countofflights
---------+---------------------+---------+----------------------------+---------+--------------+----------------
     JFK |            0.711221 |   20374 |                   10.60506 |      XE | New York, NY |           1818
     JFK |            0.697051 |   19707 |                   13.24746 |      EA | New York, NY |           6001
     JFK |            0.693871 |   20404 |                   11.26115 |      DH | New York, NY |          12449
     JFK |            0.635885 |   19977 |                    6.30787 |      UA | New York, NY |          94341
     JFK |            0.633045 |   19386 |                   13.11643 |      NW | New York, NY |          16765
     JFK |            0.620712 |   20384 |                   10.99403 |  PA (1) | New York, NY |          43724
     JFK |            0.613699 |   20417 |                   17.71196 |      OH | New York, NY |          70326
     JFK |             0.60773 |   20378 |                   18.87873 |      YV | New York, NY |           4321
     JFK |            0.580879 |   20398 |                    14.6947 |      MQ | New York, NY |          58699
     JFK |            0.580197 |   19704 |                    8.67044 |      CO | New York, NY |           4676

(10 rows)
cqlsh:cloudcapstone> -- SEA
cqlsh:cloudcapstone> select * from spark_g2q1_airport_carriers_depart_perf WHERE airport = 'SEA' ORDER BY ontimedepartureperf desc limit 10;

 airport | ontimedepartureperf | airline | avgdeparturedelayinminutes | carrier | city        | countofflights
---------+---------------------+---------+----------------------------+---------+-------------+----------------
     SEA |            0.832117 |   20378 |                    7.30657 |      YV | Seattle, WA |            548
     SEA |            0.748656 |   20304 |                    4.70382 |      OO | Seattle, WA |          27341
     SEA |            0.689221 |   20312 |                    8.38985 |      TZ | Seattle, WA |           2542
     SEA |            0.680443 |   19707 |                   11.46716 |      EA | Seattle, WA |           4065
     SEA |            0.666212 |   19386 |                    7.21636 |      NW | Seattle, WA |         118000
     SEA |            0.651852 |   20366 |                    8.37037 |      EV | Seattle, WA |            270
     SEA |            0.651282 |   20404 |                   12.14359 |      DH | Seattle, WA |            195
     SEA |            0.647513 |   19690 |                    8.16441 |      HA | Seattle, WA |           3759
     SEA |            0.630984 |   19805 |                    7.32507 |      AA | Seattle, WA |         123000
     SEA |            0.628542 |   19991 |                    9.17828 |      HP | Seattle, WA |          46051

(10 rows)
cqlsh:cloudcapstone> -- BOS
cqlsh:cloudcapstone> select * from spark_g2q1_airport_carriers_depart_perf WHERE airport = 'BOS' ORDER BY ontimedepartureperf desc limit 10;

 airport | ontimedepartureperf | airline | avgdeparturedelayinminutes | carrier | city       | countofflights
---------+---------------------+---------+----------------------------+---------+------------+----------------
     BOS |            0.835209 |   20295 |                    5.49357 |  ML (1) | Boston, MA |           1244
     BOS |            0.798593 |   20384 |                    4.53687 |  PA (1) | Boston, MA |          25168
     BOS |            0.778326 |   19707 |                    7.99648 |      EA | Boston, MA |          28393
     BOS |            0.772821 |   20312 |                    6.25643 |      TZ | Boston, MA |           3385
     BOS |            0.718062 |   20374 |                     9.7467 |      XE | Boston, MA |           2270
     BOS |            0.714126 |   20437 |                   13.16743 |      FL | Boston, MA |          41959
     BOS |            0.711592 |   20363 |                   13.90373 |      9E | Boston, MA |           2493
     BOS |            0.687432 |   20417 |                   14.18676 |      OH | Boston, MA |          53758
     BOS |            0.678562 |   19386 |                    8.23316 |      NW | Boston, MA |         142158
     BOS |            0.671502 |   20404 |                   12.34528 |      DH | Boston, MA |          18889

(10 rows)
```

G2Q2



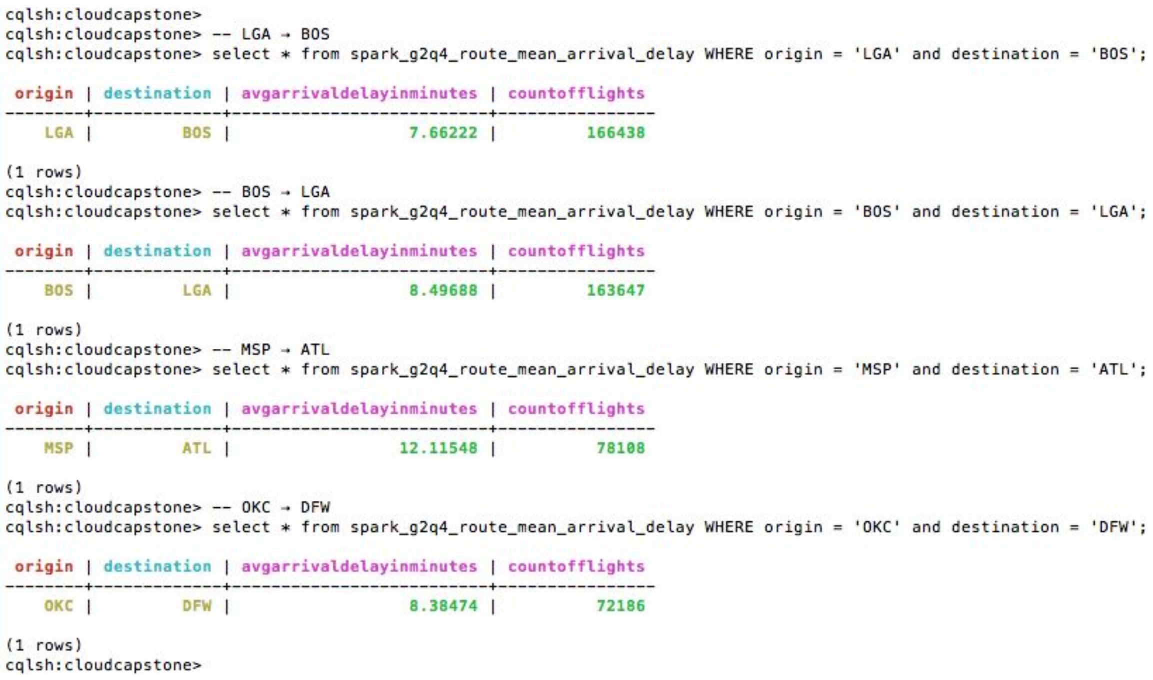
```
cqlsh:cloudcapstone> -- SRQ
cqlsh:cloudcapstone> select * from spark_g2q2_airport_routes_depart_perf WHERE origin = 'SRQ' ORDER BY ontimedepartureperf desc limit 10;

 origin | ontimedepartureperf | dest | avgdeparturedelayinminutes | countofflights | destcity                 | origcity
--------+---------------------+------+----------------------------+----------------+--------------------------+------------------------
    SRQ |                   1 |  EYW |                          0 |              1 |             Key West, FL | Sarasota/Bradenton, FL
    SRQ |            0.846154 |  MSP |                    6.09615 |            104 | Minneapolis/St. Paul, MN | Sarasota/Bradenton, FL
    SRQ |            0.835326 |  TPA |                    2.18126 |           2593 |                Tampa, FL | Sarasota/Bradenton, FL
    SRQ |            0.822967 |  DCA |                     4.7177 |            209 |           Washington, DC | Sarasota/Bradenton, FL
    SRQ |            0.820359 |  BNA |                    2.81437 |            334 |            Nashville, TN | Sarasota/Bradenton, FL
    SRQ |            0.817797 |  MEM |                    1.86758 |            944 |              Memphis, TN | Sarasota/Bradenton, FL
    SRQ |            0.780379 |  IAH |                    3.52806 |           2905 |              Houston, TX | Sarasota/Bradenton, FL
    SRQ |            0.775709 |  RSW |                     4.6302 |          12515 |            Ft. Myers, FL | Sarasota/Bradenton, FL
    SRQ |            0.773573 |  STL |                    5.91417 |           4730 |            St. Louis, MO | Sarasota/Bradenton, FL
    SRQ |            0.760548 |  BWI |                    6.05696 |           1896 |            Baltimore, MD | Sarasota/Bradenton, FL

(10 rows)
cqlsh:cloudcapstone> -- CMH
cqlsh:cloudcapstone> select * from spark_g2q2_airport_routes_depart_perf WHERE origin = 'CMH' ORDER BY ontimedepartureperf desc limit 10;

 origin | ontimedepartureperf | dest | avgdeparturedelayinminutes | countofflights | destcity             | origcity
--------+---------------------+------+----------------------------+----------------+----------------------+-------------
    CMH |                   1 |  AUS |                          0 |              3 |           Austin, TX | Columbus, OH
    CMH |                   1 |  OMA |                         -5 |              1 |            Omaha, NE | Columbus, OH
    CMH |             0.84874 |  SDF |                     2.7395 |            119 |       Louisville, KY | Columbus, OH
    CMH |            0.814375 |  CLE |                    3.88083 |           7235 |        Cleveland, OH | Columbus, OH
    CMH |            0.768627 |  SLC |                    6.92353 |            510 |   Salt Lake City, UT | Columbus, OH
    CMH |            0.762457 |  DTW |                    5.82289 |          34137 |          Detroit, MI | Columbus, OH
    CMH |            0.757574 |  DFW |                    5.53413 |          27761 | Dallas/Ft. Worth, TX | Columbus, OH
    CMH |            0.753537 |  MEM |                    5.92517 |           5372 |          Memphis, TN | Columbus, OH
    CMH |            0.752987 |  BNA |                    4.64757 |           9040 |        Nashville, TN | Columbus, OH
    CMH |             0.75285 |  IAD |                    7.30501 |           5790 |       Washington, DC | Columbus, OH

(10 rows)
cqlsh:cloudcapstone> -- JFK
cqlsh:cloudcapstone> select * from spark_g2q2_airport_routes_depart_perf WHERE origin = 'JFK' ORDER BY ontimedepartureperf desc limit 10;

 origin | ontimedepartureperf | dest | avgdeparturedelayinminutes | countofflights | destcity                   | origcity
--------+---------------------+------+----------------------------+----------------+----------------------------+-------------
    JFK |                   1 |  ABQ |                          0 |              1 |            Albuquerque, NM | New York, NY
    JFK |                   1 |  ANC |                          0 |             27 |              Anchorage, AK | New York, NY
    JFK |                   1 |  ISP |                          0 |              1 |                 Islip, NY | New York, NY
    JFK |                   1 |  MYR |                          0 |              1 |           Myrtle Beach, SC | New York, NY
    JFK |                   1 |  SWF |                       -0.5 |              2 | Newburgh/Poughkeepsie, NY | New York, NY
    JFK |            0.830508 |  BGR |                    6.55085 |            118 |                Bangor, ME | New York, NY
    JFK |            0.828025 |  DAB |                    7.56688 |            157 |          Daytona Beach, FL | New York, NY
    JFK |            0.821705 |  CHS |                    7.38953 |           1032 |             Charleston, SC | New York, NY
    JFK |            0.814751 |  PNS |                    7.90566 |            583 |              Pensacola, FL | New York, NY
    JFK |            0.804264 |  SAV |                    8.08915 |           1032 |               Savannah, GA | New York, NY

(10 rows)
cqlsh:cloudcapstone> -- SEA
cqlsh:cloudcapstone> select * from spark_g2q2_airport_routes_depart_perf WHERE origin = 'SEA' ORDER BY ontimedepartureperf desc limit 10;

 origin | ontimedepartureperf | dest | avgdeparturedelayinminutes | countofflights | destcity                      | origcity
--------+---------------------+------+----------------------------+----------------+-------------------------------+-------------
    SEA |                   1 |  EUG |                          0 |              1 |                    Eugene, OR | Seattle, WA
    SEA |            0.802181 |  PSC |                    4.09717 |           2017 | Pasco/Kennewick/Richland, WA | Seattle, WA
    SEA |            0.764151 |  ICT |                   11.90566 |            106 |                   Wichita, KS | Seattle, WA
    SEA |            0.695017 |  MEM |                    4.96892 |          11358 |                   Memphis, TN | Seattle, WA
    SEA |             0.68033 |  PDX |                    7.48523 |          74114 |                  Portland, OR | Seattle, WA
    SEA |            0.678473 |  CLE |                    6.63239 |           2908 |                 Cleveland, OH | Seattle, WA
    SEA |            0.670561 |  DTW |                    6.40699 |          26278 |                   Detroit, MI | Seattle, WA
    SEA |            0.667544 |  SNA |                    6.16402 |          38745 |                 Santa Ana, CA | Seattle, WA
    SEA |            0.659743 |  MSP |                    7.38041 |          44628 |      Minneapolis/St. Paul, MN | Seattle, WA
    SEA |            0.652543 |  GEG |                    7.90643 |          79066 |                   Spokane, WA | Seattle, WA

(10 rows)
cqlsh:cloudcapstone> -- BOS
cqlsh:cloudcapstone> select * from spark_g2q2_airport_routes_depart_perf WHERE origin = 'BOS' ORDER BY ontimedepartureperf desc limit 10;

 origin | ontimedepartureperf | dest | avgdeparturedelayinminutes | countofflights | destcity                         | origcity
--------+---------------------+------+----------------------------+----------------+----------------------------------+------------
    BOS |                   1 |  ONT |                          0 |              1 |     Ontario/San Bernardino, CA | Boston, MA
    BOS |                   1 |  SWF |                          0 |              1 |      Newburgh/Poughkeepsie, NY | Boston, MA
    BOS |            0.813449 |  AUS |                    3.57122 |           1383 |                     Austin, TX | Boston, MA
    BOS |            0.787836 |  LGA |                    3.95273 |         163647 |                   New York, NY | Boston, MA
    BOS |             0.78534 |  MSY |                    6.13613 |            573 |                New Orleans, LA | Boston, MA
    BOS |            0.770202 |  MYR |                    8.61111 |            396 |               Myrtle Beach, SC | Boston, MA
    BOS |            0.754503 |  MDW |                    8.05957 |           7851 |                    Chicago, IL | Boston, MA
    BOS |            0.712457 |  MKE |                    6.88105 |           3203 |                  Milwaukee, WI | Boston, MA
    BOS |            0.704944 |  SAV |                   11.78947 |            627 |                   Savannah, GA | Boston, MA
    BOS |            0.703515 |  PHF |                   14.49925 |           3329 | Newport News/Williamsburg, VA | Boston, MA

(10 rows)
```

G2Q2 (from HDFS)



```
[root@ip-172-31-0-62 ~]#  hadoop fs -cat /cloudcapstone/spark-results/G2Q2/part* | grep "SRQ-" | head
(SRQ-EYW,1.0,1,0.0)
(SRQ-MSP,0.84615386,104,6.0961537)
(SRQ-TPA,0.8353259,2593,2.181257)
(SRQ-DCA,0.8229665,209,4.717703)
(SRQ-BNA,0.8203593,334,2.8143709)
(SRQ-MEM,0.8177966,944,1.8675847)
(SRQ-IAH,0.78037876,2905,3.5280552)
(SRQ-RSW,0.77570915,12515,4.6302023)
(SRQ-STL,0.773573,4730,5.914165)
(SRQ-BWI,0.7605485,1896,6.056961)
[root@ip-172-31-0-62 ~]#  hadoop fs -cat /cloudcapstone/spark-results/G2Q2/part* | grep "CMH-" | head
(CMH-AUS,1.0,3,0.0)
(CMH-OMA,1.0,1,-5.0)
(CMH-SDF,0.8487395,119,2.7394958)
(CMH-CLE,0.8143746,7235,3.8800275)
(CMH-SLC,0.76862746,510,6.9235296)
(CMH-DTW,0.76245713,34137,5.8228908)
(CMH-DFW,0.75757354,27761,5.5341296)
(CMH-MEM,0.7535369,5372,5.925168)
(CMH-BNA,0.7529867,9040,4.6475673)
(CMH-IAD,0.75284976,5790,7.3050094)
[root@ip-172-31-0-62 ~]#  hadoop fs -cat /cloudcapstone/spark-results/G2Q2/part* | grep "JFK-" | head
(JFK-ABQ,1.0,1,0.0)
(JFK-ANC,1.0,27,0.0)
(JFK-ISP,1.0,1,0.0)
(JFK-MYR,1.0,1,0.0)
(JFK-SWF,1.0,2,-0.5)
(JFK-BGR,0.8305085,118,6.5508475)
(JFK-DAB,0.82802546,157,7.566879)
(JFK-CHS,0.8217054,1032,7.389535)
(JFK-PNS,0.81475127,583,7.905661)
(JFK-SAV,0.8042636,1032,8.089148)
[root@ip-172-31-0-62 ~]#  hadoop fs -cat /cloudcapstone/spark-results/G2Q2/part* | grep "SEA-" | head
(SEA-EUG,1.0,1,0.0)
(SEA-PSC,0.8021815,2017,4.0971737)
(SEA-ICT,0.7641509,106,11.905661)
(SEA-MEM,0.6950168,11358,4.9689226)
(SEA-PDX,0.6803302,74114,7.485227)
(SEA-CLE,0.67847323,2908,6.6323934)
(SEA-DTW,0.6705608,26278,6.4069963)
(SEA-SNA,0.6675441,38745,6.164022)
(SEA-MSP,0.65974283,44628,7.3804126)
(SEA-GEG,0.65254337,79066,7.9064326)
[root@ip-172-31-0-62 ~]#  hadoop fs -cat /cloudcapstone/spark-results/G2Q2/part* | grep "BOS-" | head
(BOS-ONT,1.0,1,0.0)
(BOS-SWF,1.0,1,0.0)
(BOS-AUS,0.813449,1383,3.571222)
(BOS-LGA,0.78783613,163647,3.9527278)
(BOS-MSY,0.7853403,573,6.1361256)
(BOS-MYR,0.77020204,396,8.611111)
(BOS-MDW,0.7545029,7051,8.059566)
(BOS-MKE,0.71245724,3203,6.881049)
(BOS-SAV,0.70494413,627,11.7894745)
(BOS-PHF,0.7035146,3329,14.499249)
```

G2Q3



```
cqlsh:cloudcapstone>
cqlsh:cloudcapstone> -- LGA → BOS
cqlsh:cloudcapstone> select * from spark_g2q4_route_mean_arrival_delay WHERE origin = 'LGA' and destination = 'BOS';

 origin | destination | avgarrivaldelayinminutes | countofflights
--------+-------------+--------------------------+----------------
    LGA |         BOS |                  7.66222 |         166438

(1 rows)
cqlsh:cloudcapstone> -- BOS → LGA
cqlsh:cloudcapstone> select * from spark_g2q4_route_mean_arrival_delay WHERE origin = 'BOS' and destination = 'LGA';

 origin | destination | avgarrivaldelayinminutes | countofflights
--------+-------------+--------------------------+----------------
    BOS |         LGA |                  8.49688 |         163647

(1 rows)
cqlsh:cloudcapstone> -- MSP → ATL
cqlsh:cloudcapstone> select * from spark_g2q4_route_mean_arrival_delay WHERE origin = 'MSP' and destination = 'ATL';

 origin | destination | avgarrivaldelayinminutes | countofflights
--------+-------------+--------------------------+----------------
    MSP |         ATL |                 12.11548 |          78108

(1 rows)
cqlsh:cloudcapstone> -- OKC → DFW
cqlsh:cloudcapstone> select * from spark_g2q4_route_mean_arrival_delay WHERE origin = 'OKC' and destination = 'DFW';

 origin | destination | avgarrivaldelayinminutes | countofflights
--------+-------------+--------------------------+----------------
    OKC |         DFW |                  8.38474 |          72186

(1 rows)
cqlsh:cloudcapstone>
```