A brief overview of system integration

To compute answers, I've used a cluster of Amazon Linux instances with Hadoop 2.7.2, Cassandra 3.2.1, Spark 1.6.0 and Kafka 0.8.2.2 (Scala 2.10). Kafka 0.9 would be better due to simpler, more optimal and more robust new consumer code, but it would require rewriting Kafka support in Spark. The cluster was provisioned on AWS EC2. Datastax Cassandra driver version 3.0.0 was used to connect to Cassandra. To fix library version incompatibilities, Guava library in Hadoop was upgraded to version 16.0.1 required by Datastax Cassandra driver.

The dataset cleaned and loaded in Task1 was streamed into Kafka using kafka-console-producer like this:

```
cat file.tsv | kafka-console-producer.sh --broker-list $localip:9092 --topic topic1
```

then processed in Spark Streaming using Spark-Kafka integration. The topic has number of partitions equal to the number of cluster instances. The topic has one replica. After producing all messages in the dataset each cluster producer sends an "import complete" message into a separate "signal" topic. Spark driver program launches a separate "contextStopper" thread to listen to this topic and stops Spark Context gracefully after all instances of producer are done and received data is processed. This "signal" topic approach is flawed because it has unpredictable delays and there is no guarantee of synchronization between "data" and "signal" (that is, data can arrive much later than signal), but it's easier to implement than special messages in "data" or counting messages, which, together with gracefully stopping Spark contexts work. We also remember that we simulate the stream anyway and this is a workaround.

The results are displayed in console or stored in Cassandra and queried with a program or via cqlsh. The driver application is developed in Java 8 using Maven and packaged with dependencies as an uber-jar using Maven Shade plugin to avoid dependency conflicts. For repeated invocations it is better to setup dependency jars locally.

The cluster was managed with a set of ansible scripts developed by me. The scripts automate the setup of master instance which serves as a launchpad for all EC2 work and as a master for Hadoop, Cassandra, Spark, Kafka and Zookeeper. The scripts also automate cluster instances setup and parallelize the loading of the dataset. To reduce potential attack surface, cluster instances are launched inside VPC, into an isolated security group available only from the launchpad. The cluster contains 1 master instance and 10 worker instances, all of m3.xlarge type.

Approaches and algorithms used to answer questions

Most questions allow some freedom of interpretation in definition of metric as well as its direction. For example, top airports might be counted by outgoing flights, by incoming flights or by a sum of the two. The definition of performance can also vary: total delay, average delay, number of delayed flights, % of delayed flights, etc. Delay might include or exclude early arrivals, etc. Also, top might be taken from ascending or from descending order. Different metrics provide different insights, but as an exercise in cloud computing most are simple aggregate metrics and are not very different from one another in calculation. Thus, the definitions used here are just one option out of many possible.

Most questions allow for very similar algorithms with "map" and "reduce" operations as in Task1 plus some sorting or selection. However, one has to keep in mind that this time we're dealing with streaming small batches of data: Spark stream is made of RDDs. In addition, storing state is expensive in Spark Streaming and Cassandra might be used to mitigate that.

Many questions required keeping global state. For example, "top10" in our streaming context becomes "top10 out of all seen so far". This can be done using "mapWithState" method. It is more efficient than "updateStateByKey". Then the state data can be accessed using "stateSnapshots" and processed with "transform" like an RDD to "sortByKey", select, print, etc.

One has to be careful with global state as keeping it might require lots of Spark memory and traffic for shared state. It might be better to just keep and update the data in Cassandra as we go, lowering the memory requirements at the cost of larger traffic to and from Cassandra.

Group 1

Questions of the first group are very similar to each other. They are simple aggregate queries: to answer them, an aggregate function like SUM can be calculated over a field like Flights and then sorted. This permits a generalized implementation and is used as an optimization to save development effort.

The first mapper outputs key-value pairs according to chosen fields. The first reducer sums values aggregated by key. Then running counts are preserved using “mapWithState”. After that running counts are accessed via “stateSnapshots”, sorted and printed out.

Group 1 Question 1

Required fields for this question are: Origin, Dest and Flights. The aggregation is done using Origin and Flights fields (or Dest and Flights). Flight amounts are summed and the airports are rated by popularity. This generates the following results (equal to Task1, “from” run 2m:13sec, “to” run 1:47):

FROM		TO	
ORD	6205064	ORD	6244290
ATL	5773841	ATL	5766581
DFW	5383016	DFW	5416287
LAX	3862753	LAX	3860843
PHX	3289702	PHX	3295832
DEN	3130001	DEN	3143786
DTW	2810033	DTW	2826589
IAH	2737801	IAH	2742933
MSP	2594956	MSP	2604257
SFO	2589367	SFO	2581656

Group 1 Question 2

Required fields here are UniqueCarrier and ArrDelayMinutes. Aggregating over the first and summing over the second will give us required rating. This generates the following results (equal to Task1, run 1:48):

UniqueCarrier	ArrDelayMinutes
PS	235871
ML (1)	601584
AQ	764277
HA	1037477
TZ	2613299
PA (1)	3050591
F9	3183318
9E	5745865
EA	8402711
PI	9092428

Group 1 Question 3

Required fields here are DayOfWeek and ArrDelayMinutes. Aggregating over the first and summing over the second will give us required rating. This generates the following results (equal to Task1, run 1:49):

DayOfWeek	ArrDelayMinutes
6	139401144
2	174688060
7	178552140
1	186238581
3	191076238
4	216767350
5	227051438

Group 2

In this group the first three questions are very similar and allow for several optimizations of development effort. The first mapper combines a list of specified fields (with a separator) and emits a combined key with a value from a specified field. The intermediate results are mapped with state as in Group 1 and saved into Cassandra. Final results are queried from Cassandra using a small program. As an optimization, prepared queries and connection pools are used for updating Cassandra.

Group 2 Question 1

The key for aggregation here is Origin+UniqueCarrier, the field DepDelayMinutes is summed to determine the result (run 1:57):

SRQ	10	CMH	10	JFK	10	SEA	10	BOS	10
Carrier	**Delay**	**Carrier**	**Delay**	**Carrier**	**Delay**	**Carrier**	**Delay**	**Carrier**	**Delay**
9E	685	EV	4584	XE	21981	OH	1295	EV	5456
MQ	3393	ML (1)	5629	CO	51976	EV	2498	ML (1)	7189
ML (1)	4810	PI	9848	EV	59714	DH	2654	TZ	25234
TZ	7841	YV	11235	EA	85121	YV	4219	XE	26934
YV	7855	EA	26905	YV	86424	PS	7205	9E	38699
XE	13801	DH	38395	PI	97148	PA (1)	7864	AS	59931
B6	26085	B6	38870	DH	151415	PI	8592	YV	96324
UA	29404	OO	40951	NW	247714	XE	18851	PA (1)	119964
EA	40209	9E	43280	US	335196	FL	23551	PI	190998
EV	43910	OH	178488	PA (1)	530873	TZ	24864	DH	247593

Group 2 Question 2

The key for aggregation here is Origin+Dest, the field DepDelayMinutes is summed to determine the result (run 2:02):

SRQ	10	CMH	10	JFK	10	SEA	10	BOS	10
Destination	**Delay**	**Destination**	**Delay**	**Destination**	**Delay**	**Destination**	**Delay**	**Destination**	**Delay**
EYW	0	AUS	0	ABQ	0	EUG	0	ONT	0
FLL	2	OMA	0	ANC	0	PIH	1	SWF	0
CMH	80	SYR	0	ISP	0	MFR	13	GGG	4
BDL	95	MSN	2	MYR	0	TWF	16	ACK	15
MSP	800	ALB	8	SWF	0	MSY	34	OMA	32
DCA	1081	ICT	18	AGS	19	DSM	67	PVD	212
BNA	1202	SBN	28	SAT	80	OGD	206	ACY	1573
IAD	1907	GRR	40	TYS	155	RDM	889	CAE	1918
MEM	2172	ROC	127	BHM	166	ICT	1377	SDF	2773
BOS	4760	GRB	195	LEX	170	LIH	2610	SRQ	2794

Group 2 Question 3

Aggregation key here is Origin+Dest+UniqueCarrier, the field DepDelayMinutes is summed to determine the result (run 5:27):

LGA BOS	9	BOS LGA	10	OKC DFW	7	MSP ATL	8
Carrier	**Delay**	**Carrier**	**Delay**	**Carrier**	**Delay**	**Carrier**	**Delay**
TW	3	TW	0	OH	60	9E	9
AA	50	TZ	44	TW	1716	EA	3126
OH	1114	AA	161	OO	9377	OH	5639
NW	9535	OH	499	EV	14144	OO	12146
EA	31839	NW	7657	DL	91392	EV	60687
PA (1)	41453	EA	33315	MQ	97559	FL	95276
US	130610	PA (1)	83500	AA	225413	DL	240242
DL	184434	US	176997			NW	295902
MQ	237250	DL	191920				
		MQ	254267				

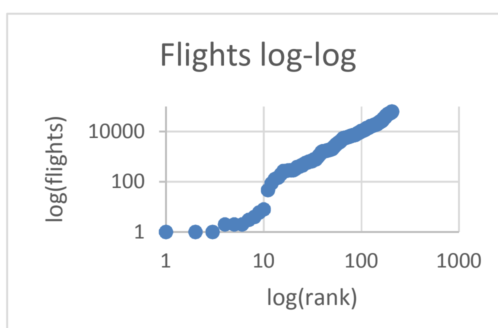
Group 2 Question 4

Skipped to favour Question 3.

Group 3

Group 3 Question 1

To answer this question, we use code from Group 1 Question 1 and Excel. We rerun the job, but instead of limiting to top10, we can use one of “saveAs*” functions to export running counts, for example into Hadoop. According to [Zipf’s Law article in Wikipedia](), it is useful to build log-log plots in an attempt to understand the distribution. While a more robust approach would try to fit the distribution in a tool like R, we use a simple approach and try to guess here. However, it is hard to say if it’s Zipf.



Group 3 Question 2

To answer this question first we filter flights by route, selecting single flight with minimal delay from a route. There may be more than one such flight and that might introduce variability in results. Showing all fitting flights might be an option. For sake of brevity, we show only one. The minimal delays flights are preserved using “mapWithState” and stored in Cassandra. The Cassandra is then queried for a combination of flights for a trip on a given date. This generates the following results of the queries. They might differ depending on which flight was selected as minimal delay flight. This run for about 2 minutes, Cassandra check showed it stopped producing new results and I stopped it after 20 minutes.

x	y	z	flight_date	flight1_num	flight2_num
BOS	ATL	LAX	2008-04-03	270	75
PHX	JFK	MSP	2008-09-07	118	609
DFW	STL	ORD	2008-01-24	1136	2245
LAX	MIA	LAX	2008-05-16	280	972

Results of each question

The results of each questions are provided in the respective sections above.

System and application-level optimizations

Several optimizations are used at each level. At the system level, the replication level in Kafka is set to 1 to lower the costs of storage. Additionally, topic is partitioned to allow parallel processing. Data for Question 3.2 is stored and streamed separately to lower storage requirements. Loading of data is parallelized with Ansible to the whole cluster. Kafka and Cassandra were deployed with Spark on the same instances to localize traffic and reduce costs.

My opinion if the results make sense and are useful

In my opinion the results make sense. The results allow one to choose better travel options.

Differences between stacks in Task1 and Task2

It seems that most questions were more appropriate for batch processing. This made Spark Streaming stack less convenient for this particular set and introduced a series of “artificial” obstacles, in addition to causing confusion. However, Kafka and Spark Streaming is a very interesting combination of stream processing systems: replication and partitioning options in Kafka make it resilient and parallelizable, and RDD is a very interesting abstraction to work with. Spark also leans on functional programming, which is interesting. Also, if one uses Java 8 and lambdas, the code is quite compact.

Video report

[Data loading and results](https://youtu.be/Sg0awqDpV1k) (https://youtu.be/Sg0awqDpV1k)
[Bonus video: Cluster setup](https://youtu.be/kPPO53MIgCo) (https://youtu.be/kPPO53MIgCo)