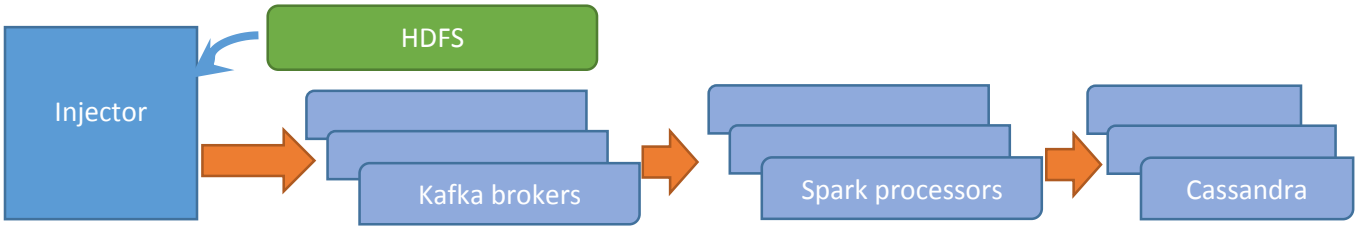
Cloud Computing Capstone Project Report. Part 2

1. System integration

1.1. System structure

Cleaned data has been read from HDFS file and injected to Kafka brokers. System structure is presented on the picture:



I used cluster from 4 c4.xlarge instances (4 cores. 8 Gb RAM) from the previous task. One node has been reserved for HDFS Name Node.

Other 3 nodes contain Kafka brokers and YARN clients. Cassandra has been installed onto other 3 nodes. Such amount of nodes was used to reduce writing time when solving Task 3.2.

Injector is a Java application that reads data from HDFS file and post them as a Kafka messages.

1.2. Data processing pipeline.

Because we must use streaming for this task, I added processing modules for each task to one pipeline.

a) Spark streaming connector retrieves from Kafka messages which contains information about flights.
b) Each task module extracts needed fields from flight data and pass them to its own processing.
c) Each module stores aggregated results to the storage.
d) There are new results for all tasks after each stream batch has been processed. And user can see them with the minimal delay.

1.3. Performance metrics

Please note that I used ***only positive delay times*** as a performance metrics, so my results may vary from example results.

2. Analysis tasks.

2.1. Group 1, task 2. Rank the top 10 airlines by on-time arrival performance.

Approach: For each streaming batch (Sparks DStream RDD) extract airline ID as a key and <arrival performance, 1> as a value. Then sum values (arrival delays and number of flights) for each airline. Then – aggregate each batch results using updateStateByKey and flush current results to the Cassandra table.

Optimization: Use repartition(1) to collect all results to the one RDD to write to Cassandra in one batch insert.

Results:

Airline name	Airline ID	Average arrival delay
Hawaiian Airlines Inc.: HA	19690	3.97
Aloha Airlines Inc.: AQ	19678	5.05
Pacific Southwest Airlines: PS (1)	19391	5.67
Midway Airlines Inc. (1): ML (1)	20295	8.70
Southwest Airlines Co.: WN	19393	9.13
Frontier Airlines Inc.: F9	20436	9.93
Pan American World Airways (1): PA (1)	20384	10.35
US Airways Inc.: US	20355	10.53
Northwest Airlines Inc.: NW	19386	10.58
SkyWest Airlines Inc.: OO	20304	10.66

2.2. Group 1, task 3. Rank the days of week by on-time arrival performance.

Approach: The same as in previous task, except day_of_week was extracted as a key.

Processing time: 2 minutes.

№ day of week	Day of week	Average arrival delay
6	Saturday	9.45
2	Tuesday	10.48
1	Monday	11.10
7	Sunday	11.22
3	Wednesday	11.44
4	Thursday	12.98
5	Friday	13.55

2.3. Group 2, task 1. For each airport X, rank the top-10 carriers in decreasing order of on-time departure performance from X.

Approach: Two staged aggregation is used. At the first stage <airport_code, airline_Id> is extracted and passed as a key and <depDelay,1> - as a value. Then data is aggregated by reduceByKey across one DStream batch and by updateStateByKey across all processed batches.

At the second stage keys are splitted: only airport code is left as a key, airline_id is moved to the value tuple. Then all tuples for the same key are collected and 10 with top departure performance were selected and written to the Cassandra database.

Cassandra schema: I used airport code as a primary (partition key), carrier rank as a secondary (clustering key), carrier code and average delay as a regular columns.

Results:

SRQ, delay		CMH, delay		JFK, delay		SEA, delay		BOS, delay	
19805, AA	4,56	20295, ML 1	4,37	19391, PS 1	4,98	19977, UA	7,64	20384, PA 1	4,75
19977, UA	5,03	19805, AA	5,24	20304, OO	5,09	19704, CO	10,79	20295, ML 1	5,71
20355, US	5,82	19707, EA	5,33	19790, DL	7,49	19805, AA	11,21	20312, TZ	7,45
20312, TZ	5,93	19822, PI	5,78	20378, YV	7,70	20384, PA 1	11,82	19707, EA	8,22
20211, TW	6,11	19386, NW	6,13	20355, US	7,81	20374, XE	12,06	19790, DL	8,39
19707, EA	6,23	19790, DL	6,22	19386, NW	8,16	20404, DH	12,16	20366, EV	8,60
19790, DL	6,25	20211, TW	6,91	19805, AA	8,23	19790, DL	13,18	19386, NW	9,17
20374, XE	6,57	20355, US	7,14	19393, WN	8,70	20409, B6	13,37	20355, US	10,05
19386, NW	7,12	20404, DH	7,76	19704, CO	8,95	20211, TW	13,44	19805, AA	10,21
20378, YV	7,18	19393, WN	8,41	20366, EV	9,25	19707, EA	13,62	20211, TW	11,07

2.4. Group 2, task 2. For each airport X, rank the top-10 airports in decreasing order of on-time departure performance from X.

Approach: Fully the same as for the previous task except departure delay has been averaged for pairs of origin and destination airports.

Cassandra schema: The same as for the previous task. Destination airport is a primary (partition key), carrier rank is a secondary (clustering key), carrier code and average delay are regular columns.

Results.

SRQ		CMH		JFK		SEA		BOS	
Dest	Delay	Dest	Delay	Dest	Delay	Dest	Delay	Dest	Delay
EYW	0	AUS	0	ABQ	0	EUG	0	ONT	0
FLL	2,00	OMA	0	ANC	0	PSC	4,26	SWF	0
MEM	2,28	SYR	0	ISP	0	CVG	5,01	LGA	4,49
TPA	2,83	SDF	2,90	MYR	0	BLI	5,79	AUS	4,88
MCO	2,91	CLE	4,60	SWF	0	MEM	5,92	BDL	6,31
RDU	3,49	IND	4,79	UCA	2,61	YKM	6,62	MSY	7,39
BNA	3,57	CAK	4,80	AGS	4,75	SNA	6,95	LGB	7,48
IAH	4,56	BNA	5,00	STT	5,77	MKE	7,05	DCA	7,60
DCA	4,88	DAY	5,92	BQN	5,97	LIH	7,05	MKE	7,92
RSW	4,95	CLT	6,03	STX	6,22	PIT	7,41	SJC	8,30

2.5. Group 2, task 4. For each source-destination pair X-Y, determine the mean arrival delay (in minutes) for a flight from X to Y.

Approach: Pairs <src_airport, dest_airport> are extracted and passed as a key and <arrDelay,1> - as a value. Then data is aggregated by reduceByKey across one DStream batch and by updateStateByKey across all processed batches. Each current aggregated data is stored to Cassandra table.

Optimizations. Data is written to Cassandra using batch inserts.

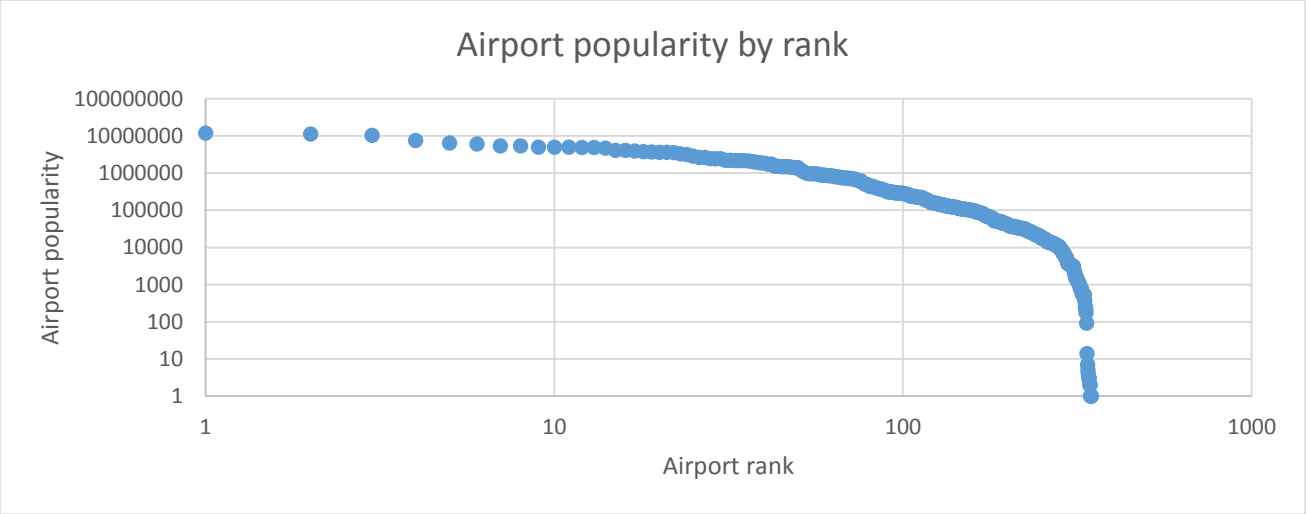
Cassandra schema: I used origin airport name as a partitioning key and destination airport name as a clustering key, mean delay stored as a value.

Results:

LGA->BOS	BOS->LGA	OKC->DFW	MSP->ATL
7,66	8,5	8,38	12.07

2.6. Group 3, task 1

Approach: Extract Sequence of source port and destination port using flatMap, port code is a key, 1 – is a value. And then just aggregate data by key. Results are stored to Cassandra table.



You can see from this graph that popularity distribution of airports does not follow ZIpf distribution and it follows some logarithmic distribution.

2.7. Group 3, task 2

Approach: This task is difficult to be solved via streaming approach. The main problem is that in streaming you need to prepare result for each batch, each streaming step. And because of relatively slow Cassandra performance I can't just write all collected data after every batch – it takes a huge amount of time. So here I used special aggregation that allows me to store only those records which are new for this stream chunk.

Lets renew general consideration about this task. Tom wants to depart first airport ***before*** 12 p.m., and wants to depart second one – ***after*** 12 p.m. It means that possible ranges of departure time for flights for both legs of his journey ***do not*** overlapped and each flight might be eligible to only first or second journey leg. There are no flights that could be used for both legs of journey in different trips.

First, I extracted flight data and prepare the flight key. It is a pair <mid-trip airport Y, journey start date>. The value is a tuple <arrival delay, airline ID, departure time>. For trips Y->Z as a journey start date I used Y->Z flight date/time shifted back by two days.

Each this data aggregated on the updateByKey stage after DStream chunk has been processed. I developed special class – TomAggregation that collect in two collections flights for the first and second journey legs. When new flight is added to the appropriate collection – it is performed if flight is new or has a better arrival performance – all possible flights are stored to the Cassandra.

Results:

Route	Start date	First leg	Second leg	Delay, min
BOS → ATL → LAX	03/04/2008	20437 (FL) at 6:00	20437 (FL) at 18:52	7
PHX → JFK → MSP	07/09/2008	20409 (B6) at 11:30	19386 (NW) at 17:50	0
DFW → STL → ORD	24/01/2008	19805 (AA) at 7:05	19805 (AA) at 13:35	0
LAX → MIA → LAX	16/05/2008	19805 (AA) at 8:20	19805 (AA) at 19:30	0

3. Stack comparison.

Actually the stacks are developed for the different tasks, so it is slightly difficult to compare them. Hadoop allows to process a huge amount of data but it is restricted to only map-reduce operations. Spark has more useful operators and used in-memory storage to hold RDD – partially or whole ones.

To develop Hadoop applications using Java is the best way, for Spark – Java is a worst one. Scala or Python would be much more better.

Almost all tasks from this Capstone are good to be solved using batch processing. There is no actual needs for streaming because there is no real-time analytic such as logs or sensor events processing to use Spark for these tasks in real project. But it was interesting to solve the same task using both approaches.

4. Performance.

All data processing except writing Task 3.2 results to Cassandra DB took 35 minutes.

Writing Task 3.2 results took about 4 hours because of several data updates.