Christian Tucker
CS598 Cloud Computing Capstone
GitHub: https://github.com/Tucker459/conair
Video Link: https://mediaspace.illinois.edu/media/t/1_qo3gygt6
June 24, 2019

Conair Capstone Report

Starting with my extraction and cleaning process. I created some go programs that would take the data in its zip format unzip them, delete the header, and then append them into one file. In the end I had each year in one file and then I exported that data into aws s3 into separate folders based off of the year. These files sizes were around 1.6 - 2.2 GBs all of files together was around 33 GBs representing 1988 through 2008. Even with these small files working through each of them with OpenRefine to do some in-depth data cleaning proved to be very difficult even on expensive hardware. My next decision was to do some more research of the actual data and the particular columns I needed to answer the questions. With that knowledge I was able to create another go program that subsetted the data with only the columns I needed this would decrease the file size. This was super fast solution as well even with running this on the file that contained all of years of data taking only minutes. Once this was completed I was able to use OpenRefine to further clean my data and make sure that we didn't have any corrupt data. I was able to find that November and December of 2008 data was corrupt and could be thrown out.

After cleaning the data and making sure we had adequate data quality. I had to figure out how I was going to integrate the services that I was using for this project. I architected out two different pipelines that would be provide me with the flexibility and robustness of a production system. My first pipeline option was to use AWS Datapipeline to integrate S3, Hadoop, EMR, EMRFS, Hive, and DynamoDB together. Utilizing this first option would allow me to focus on optimizations, core logic, and relieve me of writing a ton of glue code to stitch services together. I wouldn't have to handle failure logic, retries, alerting, timeouts, or inter-dependencies between all of these services. My second option was create an event-driven micro service fault-tolerant architecture (rdd.io). The pros of this pipeline was that everything was going to be decoupled, you had the ability to swap different services in/out, you had the option of a wider range of services the could be used and you also could have a more accurate optimization control over each step within the pipeline. I ended up going with using the AWS Datapipeline option because it allowed me to focus on vital optimizations, core logic instead of glue code, and it gave the more time to dive deeper within each application to really learn their pros and cons. I wouldn't have the time to do that if I went with the second option based off the time I have to complete this first task.

I choose Hive as my go to application to use because I'm more comfortable with sql over python and the mapreduce paradigm. Using Hive gives the ability to abstract away the map-reduce paradigm in favor of a more familiar programming language. For all questions I was able to use common-table-expressions (CTEs), aggregate, and group-by functions. In particular, for group two and three I also used window functions to partition data based off certain criteria. For question 3.2 I also used one optimization that hive can implement when joining tables. Hive assumes the last table in the join statement is the biggest so it buffers all of the other tables and streams the last table into the join. So if you had put your biggest table first in a join statement Hive would have to buffer that entire table and then do the join causing your query to run long. So a good rule of thumb is to always put your biggest table last in a join statement.

From a system and application level I applied a number of optimizations to both Hive and DynamoDB. Starting with hive I will discuss the four most important optimizations I made. I changed the execution engine to tez from mapreduce because the tez engine using directed acyclic graphs (DAGs) that greatly improve the speed of hive queries. I also implemented intermediate compression between map and reduce jobs using the snappy compression algorithm. Doing this compresses the outputted data between the jobs decreasing the disk input/output, increasing the throughput, and performance necessary to move the data. Another optimization was experimenting with the different available file formats. You had options to choose from sequence (row-based), parquet (column-based), and orc (column-based). The differences being that each had their own attributes and what makes them unique. When doing some testing of these formats I ended up choosing parquet file format which is a column based format. This format had the biggest leap in performance when testing on my queries. When it came to moving 3.2 table of data to dynamodb I switched to the orc format and decreased the map split size. Doing these two things I was able increase the number of mappers which in effect increased parallelism for that transfer of 62 millions rows of data between the two applications. From a dynamodb standpoint the main point I wanted to focus on was the data access patterns of my data. What was the inputs going to be? So I designed some tables with composite keys (partition value and sort key) this allows the data to physically close on the machine which allows for faster querying and avoids "hot" (heavily requested) partition key values. I also designed some of tables partition keys to include include multiple key columns this further distributed the partition workload evenly allowing for more efficient performance of the tables and this also increases the querying performance. Instead of having to scan the whole table you can look up key data in the partition key itself. I believe this was a great project that I got to build end-to-end. I was really pushed in my thinking and I got a chance to learn a lot about the various ways to optimize Hive and DynamoDB.

Group 1: 1.1 Top10 Most Popular Airports by Number of Flights In/Out	
Airports	**Number of Flights**
ORD	12449351
ATL	11540420
DFW	10799303
LAX	7723593
PHX	6585530
DEN	6273785
DTW	5636622
IAH	5480734
MSP	5199213
SFO	5171023

Group 1: 1.2 Top10 Airlines by On-Time-Arrival-Perfomance	
Airlines	**Number of Flights**
HA	-0.697854
AQ	1.599318
PS	4.622253
TZ	5.554235
PA	5.564994
F9	5.885831
NW	6.086370
RU	6.170409
ML	6.229677
OO	7.082626

Group 2: 2.1 - Top 10 Carriers in Decreasing Order of On-Time-Departure-Performance		
Origin	**Carrier**	**Avg. Depart Delay**
CMI	OH	0.611626
	US	2.033047
	TW	4.120615
	PI	4.455628
	DH	6.027888
	EV	6.665138
	MQ	8.016005
BWI	F9	0.756244
	PA	4.761905
	CO	5.179341
	YV	5.496503
	NW	5.705573
	AL	5.751642
	AA	6.002852
	9E	7.239806
	US	7.504232
	DL	7.676822
MIA	9E	-3
	EV	1.202643
	RU	1.302166
	TZ	1.782244
	XE	2.745645
	PA	4.200004
	NW	4.501666
	US	6.061162
	UA	6.869732
	ML	7.50455

Group 2: 2.2 - Top 10 Destination Airports in Decreasing Order of On-Time-Departure-Performance		
Origin	**Destination**	**Avg. Depart Delay**
CMI	ABI	-7
	PIT	1.102431
	CVG	1.894762
	DAY	3.116235
	STL	3.981673
	PIA	4.591892
	DFW	5.944143
	ATL	6.665138
	ORD	8.194098
BWI	SAV	-7
	MLB	1.155367
	DAB	1.469595
	SRQ	1.588484
	IAD	1.790941
	UCA	3.65417
	CHO	3.744928
	GSP	4.197687
	SJU	4.444658
	OAJ	4.471111
MIA	SHV	0
	BUF	1
	SAN	1.710383
	SLC	2.53719
	HOU	2.912199
	ISP	3.647399
	MEM	3.745107
	PSE	3.975845
	TLH	4.261484
	MCI	4.612245

Group 2: 2.1 - Top 10 Carriers in Decreasing Order of On-Time-Departure-Performance

Origin	Carrier	Avg. Depart Delay
LAX	RU	1.948387
	MQ	2.407222
	OO	4.221959
	FL	4.725127
	TZ	4.763941
	PS	4.860337
	NW	5.119551
	F9	5.729155
	HA	5.813646
	YV	6.024156
IAH	NW	3.563711
	PA	3.984727
	PI	3.988667
	RU	4.798696
	US	5.059231
	AL	5.09683
	F9	5.545244
	AA	5.703959
	TW	6.048777
	WN	6.231133
SFO	TZ	3.952416
	MQ	4.853924
	F9	5.162445
	PA	5.287612
	NW	5.757806
	PS	6.303519
	DL	6.56273
	CO	7.083049
	US	7.396203
	TW	7.794883

Group 2: 2.2 - Top 10 Destination Airports in Decreasing Order of On-Time-Departure-Performance

Origin	Destination	Avg. Depart Delay
LAX	BZN	-0.727273
	SDF	-16
	LAX	-2
	RSW	-3
	DRO	-6
	IDA	-7
	MAF	0
	PHI	0
	IKW	1.269825
	MFE	1.376471
IAH	MLI	-0.5
	AGS	-0.61879
	MSN	-2
	EFD	1.887708
	HOU	2.172037
	JAC	2.570588
	MTJ	2.950157
	RNO	3.221584
	BPT	3.599533
	VCT	3.611909
SFO	OAK	-0.8132
	PIE	-1.34104
	LGA	-1.757576
	SDF	-10
	PIH	-4
	MSO	-4
	FAR	0
	BNA	2.425966
	MEM	3.302482
	SCK	4

Group 2: 2.3 - Each Source-Destination Pair, Rank Top 10 Carriers in Decreasing Order of On-Time-Arrival-Performance

Origin_Dest	Carrier	Avg. Arrival Delay
CMI_ORD	MQ	10.143663
IND_CMH	CO	-2.545855
	AA	5.5
	HP	5.697255
	NW	5.761538
	US	6.470749
	AL	8.402795
	DL	10.6875
	EA	10.813084

Group 2: 2.3 - Each Source-Destination Pair, Rank Top 10 Carriers in Decreasing Order of On-Time-Arrival-Performance		
Origin_Dest	**Carrier**	**Avg. Arrival Delay**
DWF_IAH	PA	-1.596491
	EV	5.092513
	UA	5.414201
	CO	6.493732
	OO	7.564007
	RU	7.791492
	AA	8.381228
	XE	8.442866
	DL	8.598509
	MQ	9.103211
LAX_SFO	AL	-1.965245
	F9	-2.028686
	PS	-2.146341
	TZ	-7.619048
	UA	10.129421
	DL	11.027245
	TW	11.196664
	PA	12.29052
	AS	13.518272
	XE	13.6
JFK_LAX	UA	3.313874
	HP	6.680599
	AA	6.903725
	DL	7.93446
	TW	11.702008
	PA	11.019444
ATL_PHX	FL	4.552632
	US	6.288115
	HP	8.481436
	EA	8.953571
	DL	9.808275

Group 3: 3.2 - Best Flights based on Origin, Destination, & Flight Date										
		First Leg					**Second Leg**			
Origin	**Dest**	**Airline/Flight Num**	**Sched Depart**	**Arrival Delay**	**Origin**	**Dest**	**Airline/Flight Num**	**Sched Depart**	**Arrival Delay**	**Total Overall Delay**
CMI	ORD	MQ 607	04/03/2008	-14	ORD	LAX	AA 607	04/05/2008	-24	-38
JAX	DFW	AA 854	09/09/2008	1	DFW	CRP	MQ 3627	09/11/2008	-7	-6
SLC	BFL	OO 3755	04/01/2008	12	BFL	LAX	OO 5429	04/03/2008	6	18
LAX	SFO	WN 3534	07/12/2008	-13	SFO	PHX	US 412	07/14/2008	-19	-32
DFW	ORD	UA 1104	06/10/2008	-21	ORD	DFW	AA 2341	06/12/2008	-10	-31
LAX	ORD	UA 944	01/01/2008	1	ORD	JFK	B6 918	01/03/2008	-7	-6