ETL Report

Storm Troopers

Weather

Airport Codes were formatted like KJFK with a K at the start, so removed the K so it matched the flights dataset JFK

```
weather = weather.withColumn("AirportCode",
substring(weather["AirportCode"], 2, 3))
```

Removed several columns: LocationLat, LocationLng, City, County, State, Timezone, ZipCode
Turned the StartTime(UTC) and EndTime(UTC) columns into datetime columns

```
weatherDF = weather.toPandas()

weatherDF['StartTime(UTC)'] = pd.to_datetime(weatherDF['StartTime(UTC)'])
weatherDF['EndTime(UTC)'] = pd.to_datetime(weatherDF['EndTime(UTC)'])
```

Exported the cleaned Weather dataset

Timezone

Removed all columns except time_zone_id and code (i.e. JFK)

Flights

12 months of 2021 flight data was downloaded from the Bureau of Transportation Statistics and loaded as separate blobs into the cohort40storage Azure container.
Using unions, the monthly flight data was joined together to create a 2021 flight dataset. This dataset has a row count of 6,311,671.
Timezone data was then merged to the 2021 dataset on the origin and destination columns. It was then filtered using the code below to create a canceled_2021dataset and a departed_2021 dataset:

```
canceled_2021 = final.filter((final.CANCELLED == "1") | (final.CANCELLED ==
"1.00") | (final.DIVERTED == "1") | (final.DIVERTED == "1.00"))

departed_2021 = final.filter(((final.CANCELLED == "0") | (final.CANCELLED
== "0.00")) & ((final.DIVERTED == "0") | (final.DIVERTED == "0.00")))
```

Once these combined datasets were created, the below transformation steps were taken.

Made all column names lowercase and then changed them to be more understandable (i.e. fl_date -> flight_date)
Joined the timezone data and flight data on if flights[origin]==timezone[airportcode], join with dep_+column_name and if flights[destination]==timezone[airportcode], join with arr_+column_name

dep_time_zone_id	dep_code	arr_time_zone_id	arr_code
America/Denver	ABQ	America/Chicago	DFW
America/New_York	ATL	America/New_York	CLT
America/New_York	ATL	America/New_York	CLT
America/New_York	ATL	America/Chicago	DFW
America/New_York	ATL	America/Chicago	DFW

Datetime Issue

Wanted to join Flights and Weather data on “if flights[AirportCode] == weather[AirportCode] and weather starts before takeoff and ends after takeoff:

append weather[Row] to row in flights”

The weather dataset had times in this format “4/15/2021 13:35” and it was in UTC time, so we had to format it that way and convert all timezones to UTC

Times were in military time (1335 for 1:35 PM) so changed 2400 to 0 as the computer can read 0 as a time and not 2400, this pushes it into the next day, which will be addressed later

```
for col_name in cols:
    df = df.withColumn(col_name, when(col(col_name) == 2400,
0).otherwise(col(col_name)))
```

Created a function to turn all of those values into hh:mm format (35 -> 0:35, 1926->19:26)

```
def format_time(value):
    value = str(value)
    if len(value) == 1:
        return "0:0" + value
    elif len(value) == 2:
        return "0:" + value
    elif len(value) == 3:
        return value[0] + ":" + value[1:]
    elif len(value) == 4:
        return value[:2] + ":" + value[2:]
```

Wanted to join flight date to the times (departure time from 6:35 -> 4/1/2021 6:35) but flight date included a 0:00 time, so got rid of that

```
df.createOrReplaceTempView('vw_df')
sql = "SELECT SUBSTRING(flight_date,0,CHARINDEX(' ',flight_date)) AS
FL_DATE_2, * FROM vw_df"
df =
spark.sql(sql).withColumn('flight_date',f.col('FL_DATE_2')).drop('FL_DATE_2
')
df.display()
```

Then concatenated flight date to the time columns
Then turned those datetimes into actual timestamps

```
spark.conf.set("spark.sql.legacy.timeParserPolicy","LEGACY")

for c in ['scheduled_departure_time','scheduled_arrival_time',
"departure_time", "takeoff_time", "arrival_time", "landing_time"]:
    df = df.withColumn(c,f.to_timestamp(c, "MM/dd/yyyy HH:mm"))
```

How it was looking (remember these started in military time ie “615” or “27” or “2322”)

scheduled_departure_time	departure_time
2021-04-01T06:15:00.000+0000	2021-04-01T06:09:00.000+0000
2021-04-01T06:43:00.000+0000	2021-04-01T06:39:00.000+0000
2021-04-01T12:20:00.000+0000	2021-04-01T12:16:00.000+0000
2021-04-01T08:00:00.000+0000	2021-04-01T07:58:00.000+0000
2021-04-01T09:30:00.000+0000	2021-04-01T09:27:00.000+0000

Now that we have them in datetimes, we can now add a day to all the times that were 0:00 (there were no 0’s before we changed the 2400s to 0s so all 0s we found were ones we changed from 2400 of the previous day)

```
for q in ['scheduled_departure_time','scheduled_arrival_time',
"departure_time", "takeoff_time", "arrival_time", "landing_time"]:
    df = df.withColumn(q, when(hour(col(q)) == 0, date_add(col(q),
1)).otherwise(col(q)))
```

Then changed all the date times to UTC time based on their timezone

```
from datetime import timedelta
from pyspark.sql.functions import when, col

for c in ['scheduled_departure_time',"departure_time", "takeoff_time"]:
    df = df.withColumn(c, when(col("dep_time_zone_id") ==
"America/New_York", col(c) + timedelta(hours=5)).otherwise(col(c)))
    df = df.withColumn(c, when(col("dep_time_zone_id") ==
```

```
"America/Chicago", col(c) + timedelta(hours=6)).otherwise(col(c)))
    df = df.withColumn(c, when(col("dep_time_zone_id") == "America/Denver",
col(c) + timedelta(hours=7)).otherwise(col(c)))
    df = df.withColumn(c, when(col("dep_time_zone_id") ==
"America/Los_Angeles", col(c) + timedelta(hours=8)).otherwise(col(c)))
    df = df.withColumn(c, when(col("dep_time_zone_id") ==
"America/Anchorage", col(c) + timedelta(hours=9)).otherwise(col(c)))
    df = df.withColumn(c, when(col("dep_time_zone_id") ==
"Pacific/Honolulu", col(c) + timedelta(hours=10)).otherwise(col(c)))

for c in ['scheduled_arrival_time', "arrival_time", "landing_time"]:
    df = df.withColumn(c, when(col("arr_time_zone_id") ==
"America/New_York", col(c) + timedelta(hours=5)).otherwise(col(c)))
    df = df.withColumn(c, when(col("arr_time_zone_id") ==
"America/Chicago", col(c) + timedelta(hours=6)).otherwise(col(c)))
    df = df.withColumn(c, when(col("arr_time_zone_id") == "America/Denver",
col(c) + timedelta(hours=7)).otherwise(col(c)))
    df = df.withColumn(c, when(col("arr_time_zone_id") ==
"America/Los_Angeles", col(c) + timedelta(hours=8)).otherwise(col(c)))
    df = df.withColumn(c, when(col("arr_time_zone_id") ==
"America/Anchorage", col(c) + timedelta(hours=9)).otherwise(col(c)))
    df = df.withColumn(c, when(col("arr_time_zone_id") ==
"Pacific/Honolulu", col(c) + timedelta(hours=10)).otherwise(col(c)))
```

Had to add a day to all overnight flights. There was only one flight date column, so overnight flights would say they took off at, for instance, 4/1/2021 22:30 and landed at 4/1/2021 4:25. It is impossible to land before you takeoff, unless you go over a timezone, but now since they were all in the same timezone we could check for certain about flight lengths, so any flights that landed before they took off we could be sure were an overnight flight, and add a day to.

```
df = df.withColumn("scheduled_arrival_time",
when(col("scheduled_arrival_time") < col("scheduled_departure_time"),
date_add(col("scheduled_arrival_time"),
1)).otherwise(col("scheduled_arrival_time")))
df = df.withColumn("arrival_time", when(col("arrival_time") <
col("departure_time"), date_add(col("arrival_time"),
1)).otherwise(col("arrival_time")))
df = df.withColumn("landing_time", when(col("landing_time") <
col("takeoff_time"), date_add(col("landing_time"),
1)).otherwise(col("landing_time")))
```

Joining Flights with Weather Dataset

Had to reimport weather as CSV, so had to turn the weather datetime columns back into datetimes

```
for c in ['StartTime(UTC)','EndTime(UTC)']:
    weather = weather.withColumn(c,f.to_timestamp(c, "yyyy-MM-dd HH:mm"))
```

Had to join it 3 ways:

1. The weather of the origin at scheduled takeoff
2. The weather at the destination at scheduled takeoff
3. The weather at the destination at scheduled arrival

So first had to create three new datasets with a way to differentiate the column names

```
weather1 = weather
columns = ["EventId", "Type", "Severity", "StartTime(UTC)", "EndTime(UTC)",
"Precipitation(in)", "AirportCode"]
new_columns = ["origin_takeoff_weather " + c for c in columns]
weather1 = weather1.select([col(c).alias(nc) for c, nc in zip(columns,
new_columns)])

weather2 = weather
columns = ["EventId", "Type", "Severity", "StartTime(UTC)", "EndTime(UTC)",
"Precipitation(in)", "AirportCode"]
new_columns = ["dest_takeoff_weather " + c for c in columns]
weather2 = weather2.select([col(c).alias(nc) for c, nc in zip(columns,
new_columns)])

weather3 = weather
columns = ["EventId", "Type", "Severity", "StartTime(UTC)", "EndTime(UTC)",
"Precipitation(in)", "AirportCode"]
new_columns = ["dest_arrival_weather " + c for c in columns]
weather3 = weather3.select([col(c).alias(nc) for c, nc in zip(columns,
new_columns)])
```

Now that we could differentiate the different weather events and locations, we could join in the 3 ways we wanted to!

```
df = df.join(weather1, (col("origin") == col("origin_takeoff_weather
AirportCode")) & (col("origin_takeoff_weather StartTime(UTC)") <=
col("scheduled_departure_time")) & (col("origin_takeoff_weather
EndTime(UTC)") >= col("scheduled_departure_time")), "left")
```

```
df = df.join(weather2, (col("dest") == col("dest_takeoff_weather
AirportCode")) & (col("dest_takeoff_weather StartTime(UTC)") <=
col("scheduled_departure_time")) & (col("dest_takeoff_weather
EndTime(UTC)") >= col("scheduled_departure_time")), "left")

df = df.join(weather3, (col("dest") == col("dest_arrival_weather
AirportCode")) & (col("dest_arrival_weather StartTime(UTC)") <=
col("scheduled_arrival_time")) & (col("dest_arrival_weather EndTime(UTC)")
>= col("scheduled_arrival_time")), "left")
```

Finally, we could save the resulting massive database as a CSV.

To Predictive

First, got rid of all columns for things we wouldn't know before takeoff except for what we wanted to predict (which was whether it was canceled or not or length of delay {David's only included all flights, Meghan's only included departed flights}), i.e. destination weather at arrival, what time the weather event ended, actual time in flight, actual time of arrival (we only know the scheduled arrival time before takeoff)
Got rid of columns that repeated information, i.e. destination_airport_code was the same as dest, city_market_ID was just the city name but with a number code
Got rid of all location information except the airport codes, if the airport codes are significant, we can gather something about the area from that rather than have more columns about the same thing.
Got rid of columns that wouldn't be predictive, i.e. Weather_Event_ID - the fact the weather ID is "W-5518129" doesn't tell me anything. Year - the entire column is 2021 so it's not very helpful

Turned the columns back into the appropriate types, whether a timestamp or int or float

The flight time didn't tell us much as it was in UTC time, which isn't very helpful for gathering information about the flight at the place it takes place, and we already had a flight time block (hour of the day takeoff took place i.e. 700-759). Decided to make a new column that was how long the weather at the origin and destination had been going on before scheduled takeoff.

```
from pyspark.sql.functions import *

og_seconds = col("scheduled_departure_time") - col("origin_takeoff_weather
StartTime(UTC)")
dest_seconds = col("scheduled_departure_time") - col("dest_takeoff_weather
StartTime(UTC)")

df = df.withColumn("origin_weather_length_preflight",
(((og_seconds)/60)).cast(IntegerType()))
df = df.withColumn("dest_weather_length_preflight",
```

```
(((dest_seconds)/60)).cast(IntegerType()))
```

Then dropped the time columns, which we didn't need anymore

```
cols4 = ["scheduled_departure_time", "scheduled_arrival_time",
"origin_takeoff_weather StartTime(UTC)", "dest_takeoff_weather
StartTime(UTC)"]

df = df.drop(*cols4)
```

Grouped the time blocks from hourly to morning, afternoon, evening, and night so we have fewer categories in our model

```
df = df.withColumn('dep_time_blk',
when(col("dep_time_blk").isin("0600-0659", "0700-0759", "0800-0859",
"0900-0959", "1000-1059", "1100-1159"),
"morning").otherwise(col("dep_time_blk")))
df = df.withColumn('dep_time_blk',
when(col("dep_time_blk").isin("1200-1259", "1300-1359", "1400-1459",
"1500-1559", "1600-1659", "1700-1759"),
"afternoon").otherwise(col("dep_time_blk")))
df = df.withColumn('dep_time_blk',
when(col("dep_time_blk").isin("1800-1859", "1900-1959", "2000-2059",
"2100-2159"), "evening").otherwise(col("dep_time_blk")))
df = df.withColumn('dep_time_blk',
when(col("dep_time_blk").isin("2200-2259", "2300-2359", "0001-0559"),
"evening").otherwise(col("dep_time_blk")))
```

Changed the weather severity scale from Unknown/Other, Light, Moderate, Heavy and Severe to an scale from 0 to 4 and cast it as an integer

```
df = df.withColumn('origin_takeoff_weather Severity',
when(col("origin_takeoff_weather Severity").isin("UNK", "Other"),
"0").otherwise(col("origin_takeoff_weather Severity")))
df = df.withColumn('origin_takeoff_weather Severity',
when(col("origin_takeoff_weather Severity").isin("Light"),
"1").otherwise(col("origin_takeoff_weather Severity")))
df = df.withColumn('origin_takeoff_weather Severity',
when(col("origin_takeoff_weather Severity").isin("Moderate"),
"2").otherwise(col("origin_takeoff_weather Severity")))
df = df.withColumn('origin_takeoff_weather Severity',
when(col("origin_takeoff_weather Severity").isin("Heavy"),
"3").otherwise(col("origin_takeoff_weather Severity")))
df = df.withColumn('origin_takeoff_weather Severity',
when(col("origin_takeoff_weather Severity").isin("Severe"),
```

```
"4").otherwise(col("origin_takeoff_weather Severity")))

df = df.withColumn("origin_takeoff_weather Severity",
col("origin_takeoff_weather Severity").cast(IntegerType()))
```

How it was looking

	dep_time_blk	arr_time_blk	cancelled	scheduled_elapsed_time	distance	origin_takeoff_weather Type	origin_takeoff_weather Severity
1	afternoon	afternoon	0	90	378	Fog	4
2	afternoon	afternoon	0	92	378	Fog	4
3	afternoon	afternoon	0	82	378	Fog	4
4	afternoon	afternoon	0	82	378	Fog	4
5	afternoon	afternoon	0	82	378	Fog	4

Our resulting databases were still too large and took too long to do any modeling in.

So found out the top ten airports had the most flights and dropped all rows that weren't flying from or to one of those airports.

```
airports_to_keep =
["ATL","ORD","DFW","DEN","CLT","LAX","SEA","PHX","IAH","LAS"]
df =
df.where(col("origin").isin(airports_to_keep)|col("dest").isin(airports_to_
keep))
```

We also found the four biggest marketing airlines and dropped all that weren't the top 4.

```
airlines_to_keep = ["AA", "DL", "WN", "UA"]
df = df.where(col("airline_code").isin(airlines_to_keep))
```

The resulting dataset was *still* too big so we dropped any flights to or from any airports that made less than 10,000 flights a year. Had to create a new column/dataframe that counted how many rows each airport was in and keep only those that were in 10,000+ rows.

```
from pyspark.sql.functions import count

counts = df.groupBy("origin").agg(count("*").alias("count"))
df = df.join(counts, "origin", "inner").filter(col("count") >
9999).drop(col("count"))

counts = df.groupBy("dest").agg(count("*").alias("count"))
df = df.join(counts, "dest", "inner").filter(col("count") >
9999).drop(col("count"))
```

Due to not being able to use null values in machine learning, had to create 4 datasets, one that had no weather at origin or destination, one that had weather at origin but not destination, one that had weather at destination but not origin, and one that had weather at both. Example below for the df that had weather at the destination and not at the origin.

```
#creating the new df
df_dest_weather = df

#dropping the columns that we don't need (since this is for weather at the
destination at scheduled takeoff time, we drop the origin columns)
drop_df_dest_weather_cols = ["origin_takeoff_weather Type",
"origin_takeoff_weather Severity", "origin_takeoff_weather_Precipitation",
"origin_weather_length_preflight"]
df_dest_weather = df_dest_weather.drop(*drop_df_dest_weather_cols)

#now we drop all rows that include null values in the remaining columns,
which would be rows that have no weather at the destination
df_dest_weather = df_dest_weather.dropna()
```

Do this for all 4, and turn the resulting datasets to CSVs and then we can get into our Jupyter Notebooks!

Calls to Flight Radar API

To make requests to FlightRadar24's API, we first generated a key. With the key generated, we created a query to return the current flight status of major US airlines.

```
carriers = ['AAL', 'DAL', 'AAY', 'JBU', 'FFT', 'NKS', 'SWA', 'UAL']
for carrier in carriers:
    url = "https://flight-radar1.p.rapidapi.com/flights/list-by-airline"
    #create parameters to list flights by airline
    querystring = {"airline":carrier}
    #get headers
    headers = {
        "X-RapidAPI-Key":
"c539fb4f1emsh1635d803b09ad55p151757jsn0277163eae2b",
        "X-RapidAPI-Host": "flight-radar1.p.rapidapi.com"
    }
    #make request
    response = requests.request("GET", url, headers=headers,
params=querystring)
    #convert to json format
    data = json.loads(response.text)
```

```
    data_list.append(data)
print(data_list)
```

The resulting list contains unique flight identification numbers that can be used to find more in-depth information on flights.

```
import requests
flight_list = []
for i in flights[0:3]:
    url = "https://flight-radar1.p.rapidapi.com/flights/detail"
    querystring = {"flight":'2efc5e63'}
    headers = {
      "X-RapidAPI-Key":
"c539fb4f1emsh1635d803b09ad55p151757jsn0277163eae2b",
      "X-RapidAPI-Host": "flight-radar1.p.rapidapi.com"
    }
    response1 = requests.request("GET", url, headers=headers,
params=querystring)
    data1 = json.loads(response1.text)
    data1
    flight_list.append(data1)
```

The desired information is then extracted from the json and stored in a list of rows, from which the data is transformed into a new spark dataframe. The time information is stored as unixtime and was transformed to UTF time. To calculate the delay time, two columns were created to find the minutes and seconds relative to the scheduled arrival time of each flight.

Kafka

1. Import the required packages.
2. Create an admin client, producer, and topics.

```
admin_client = AdminClient({
    'sasl.mechanism': 'PLAIN',
    'security.protocol': 'SASL_SSL',
    'auto.offset.reset': 'earliest',
    'bootstrap.servers': confluentBootstrapServers,
    'sasl.username': confluentApiKey,
    'sasl.password': confluentSecret,
    'group.id': str(uuid.uuid1()),
    'error_cb': error_cb,
```

```
})
producer = kafka.Producer({
    'bootstrap.servers': confluentBootstrapServers,
    'sasl.mechanism': 'PLAIN',
    'security.protocol': 'SASL_SSL',
    'sasl.username': confluentApiKey,
    'sasl.password': confluentSecret,
    'group.id': '44444',
    'auto.offset.reset': 'earliest',
    'error_cb': error_cb,
})

topic = 'group-4'
topics = [NewTopic(topic, 1, 3)]
admin_client.create_topics(topics)
```

3. Convert the cleaned FlightRadar data into a Pandas dataframe, then convert to a dictionary.

```
Out[30]: dict_items([('aircraft_id', '2ef7074c'), ('aircraft', 'Boeing 787-8
Dreamliner'), ('airline', 'American Airlines'), ('origin', 'ICN'), ('destination',
'DFW'), ('scheduled_departure', '2023-01-25 09:45:00'), ('scheduled_arrival',
'2023-01-25 22:20:00'), ('actual_departure', '2023-01-25 10:19:38'),
('actual_arrival', None), ('delay-minutes', nan), ('delay-seconds', nan)])
```

4. Iterate through the flight dictionary to produce json messages for each flight in the dictionary.
5. Append successful messages to a json output file.
6. Create a kafka consumer and set a topic name for the consumer to subscribe to.
7. Check for messages. If messages are present, create a df containing the data in the messages.

```
messages = []
try:
    while len(messages) < 5:
        msg = consumer.poll(10.0)
        if msg is None:
            raise Exception('Message is None')
        elif msg.error() is not None:
            raise Exception(msg.error())
        else:
            messages.append(msg)
except Exception as e:
    print(f'Exception: {str(e)[:100]}')
```

```
finally:
    consumer.close()
```

8. Output the resulting df to a storage container in json format.