Distributed Stream Processing, Systems Integration, and Cloud Services Applied to Flight Analytics

Ivan A. Gomes
Department of Computer Science
University of Illinois Urbana-Champaign
Champaign, IL
iagomes2@illinois.edu

***Abstract*—Conducting business and providing services generates a large volume of data, and the airline industry is no exception. While recording, storing, and processing this data can be challenging significant value can be captured for both the business and its customers by doing so. With the use of distributed computing, systems integration, and cloud services this can be achieved with relative ease and modest budget. This paper presents such an approach that leverages Apache Spark, an open-source distributed general-purpose cluster-computing framework, and the commercially available cloud computing service Amazon Web Services (AWS) to analyze a dataset from the Bureau of Transportation Statistics of all domestic flights in the US between 1988 and 2008. This paper will specifically use Spark Streaming, the stream processing capability of the framework, in order to continuously query a dataset that grows over time and compare the experience to traditional batch processing. A wide range of analytics can be conducted using these commodity tools and this paper provides a number of examples including popularity of airports, on-time departure performance of airlines and airports, and optimal flight routes given arbitrary constraints.**

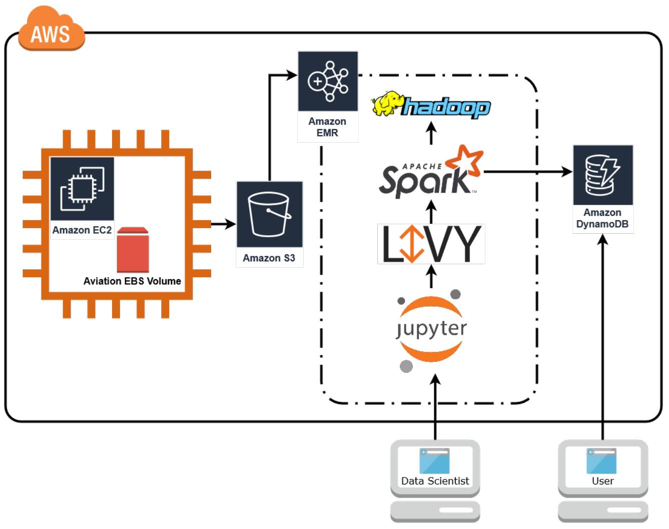

Fig. 1. Software, services, and their integrations.

I. Introduction

The amount of data being generated is continually increasing and analyzing it can provide valuable insights to businesses. Due to its volume and structural complexity, many traditional tools of the trade are not capable of producing results in a timely manner. Apache Spark and its constituent libraries address this need by distributing both the data and work to a cluster of computers that can use commodity hardware, i.e. no specialized hardware requirements. By leveraging this framework, much of the complexity of coordinating a distributed infrastructure is buried under a higher level abstraction for developers and data scientists. While knowledge of the underpinnings can be utilized to improve performance, as shown in subsequent sections, it is easier to get started than its predecessors with related functionality like Hadoop. This paper focuses on Spark's stream processing capability, which builds on its success in batch processing, enables continuous querying on streams of data, and manages to do so using a similar API.

The dataset used to demonstrate this process is titled ”Reporting Carrier On-Time Performance” [1] and is provided by the Bureau of Transportation Services. For convenience a superset including it has been rehosted on AWS as a snapshot [2], which is used for the purposes of this paper.

A set of queries that represent actionable insights are processed against the streamed dataset using Apache Spark Streaming and their results are stored in AWS DynamoDB, a fully managed key-value database service. This enables real-time access for consumers of the processed results and serve as an example of how cloud services and systems integration can be used in this process to leverage the best qualities of different systems and minimize operational overhead.

The remainder of the paper describes the procedure for processing and querying the dataset, discusses optimizations that were applied, and presents the results gathered. An accompanying video walkthrough can be found at https://mediaspace.illinois.edu/media/t/0_f1wj3euj [3].

II. Procedure

A number of services provided by AWS and open-source software packages were used to analyze the dataset whose integrations are outlined in Figure 1.

Amazon Web Services:

- Elastic Compute Cloud (EC2): general purpose virtual machines
- Simple Storage Service (S3): object storage

- Elastic Map Reduce (EMR): managed machine clusters configured with distributed frameworks like Hadoop and Spark
- DynamoDB: fully managed, serverless key-value and document database

Software:

- Apache Hadoop: MapReduce was not used as the programming framework, but the Hadoop Distributed File System (HDFS) was used to distribute the dataset to nodes
- Apache Spark: distributed framework for big data processing
- Apache Livy: a REST interface for Apache Spark
- Jupyter: interactive editor for notebooks, which contain human-readable narrative and executable code

The first step was to extract the dataset as provided in an Amazon Elastic Block Store (EBS) volume snapshot so that the dataset could be easily accessed by Spark, which has out of the box support for S3. The snapshot was instantiated to an EBS volume and an EC2 instance was created using an Ubuntu 18.04 LTS Amazon Machine Image (AMI) and the t2.micro instance size. The EBS volume was attached to this instance and started. The following Shell script was used to mount the attached volume, extract the contained zip archives, and upload to S3. Once completed, this machine could be stopped and terminated as S3 would be used moving forward.

```
#!/bin/bash

S3_CONTAINER=ccc1-aviation-data

sudo apt-get install awscli unzip
aws configure # enter AWS credentials that
    have access to S3 bucket
sudo fdisk -l # show available disks to
    identify aviation data disk
sudo mkdir /mnt/data
sudo mount /dev/xvdf /mnt/data # mount volume
    with aviation data
mkdir -p ~/aviation/airline_ontime
cd ~/aviation/airline_ontime
cp -r /mnt/data/aviation/airline_ontime/* . #
    copying to local disk since mounted volume
     is read-only
find . -name '*.zip' -exec sh -c 'unzip -o -d
    `dirname {}` {}' ';' # requires ~38 GB of
    disk space
aws s3 sync . "s3://$S3_CONTAINER/aviation/
    airline_ontime" # uploading to S3 bucket
```

Next an EMR cluster was created with the following software configuration: EMR Release 5.21.0, Hadoop 2.8.5, JupyterHub 0.9.4, Hive 2.3.4, Hue 4.3.0, Spark 2.4.0, Livy 0.5.0, and Pig 0.17.0. The hardware configuration consisted of one master instance with size m5.xlarge and 32GB of EBS storage and core instances with size m5.2xlarge and 256 GB of EBS storage each. A clone of this cluster can be created using the emr-create-cluster.sh script on GitHub [4].

The JupyterHub server running in the cluster was configured to be able to execute Spark jobs through the Livy REST API. This configuration is set up by AWS during cluster provisioning and was not needed to be modified for the purposes of this project. A Pyspark (Python Spark API) notebook named DataScientistNotebookStreaming was created on the JupyterHub server in order to stream the dataset from S3 and execute the Spark jobs necessary to generate the desired analytics. The Spark jobs were defined primarily through Spark SQL, a higher level API using a familiar language, to define multi-stage queries on the structured data. The results for each stream batch were then written to DynamoDB using AWS's Python library for its services called boto3.

Lastly a local notebook named UserNotebookStreaming was created as a stand-in for a web application (e.g. distributed with AWS S3) that calls a web service (e.g. AWS API Gateway + Lambda), which queries DynamoDB for the data. Response is in real-time because the pre-computed results are stored in DynamoDB and the clients only have to present it. The service component is exemplified by boto3 calling DynamoDB and pandas being used to format it. The web application component is exemplified by Jupyter's built-in rendering capabilities for tables, matplotlib for plot generation, and ipywidgets for user interaction.

All source code and related documentation is published on GitHub with the Apache 2.0 license at https://github.com/ivan-gomes/flight-analytics-stream-processing [4]. The code snippets provided in the remainder of the paper are excerpts from the Jupyter notebooks.

III. Optimizations

The dataset was provided as a set of files with comma-separated values (CSV). While it is a ubiquitous file format, it does not offer much in the way of query optimization or compression. Therefore, the first step was to use Spark to convert the CSV files to Apache Parquet, which is a columnar storage format, and then persisted across the cluster in HDFS. Its columnar structure reduces the amount of data read from disk as columns that aren't queried for an analysis can be skipped. In addition the compression it applies greatly shrinks the total file size, in our case from 37 GB to 2 GB solely from the conversion. After this step, the Parquet files are used exclusively which served as a checkpoint instead of needing to potentially stream from S3 multiple times during prototyping. This reduces both the query time and AWS costs.

The columns in the dataset were also evaluated for relevance to the desired analytics and those not needed were dropped. Also columns were parsed, cast, and split into appropriate data types. These further minimized disk usage, increased performance, and avoided needing to cast during query time for comparison enabling more readable queries.

Another type of optimization utilized applies to certain analytics that had results whose count was bounded. For example ranking carriers by on-time performance per airport would result in a maximum of number of airports times number of carriers rows. Considering these numbers are less than one thousand for US domestic flights it is safe to stream the results entirely into memory. Doing so results in faster execution and fewer coordinated Spark stages.

When this wasn't possible due to the size of the results, the writes to DynamoDB were parallelized in the cluster. Additionally, the provisioned write capacity of the table was set to allow 20,000 writes per second and writes are batched in groups of 10,000 to minimize request overhead. These optimizations were critical to completing the upload in a timely manner. After completion of the historical data, the provisioned write capacity was decreased to reduce AWS costs.

IV. Comparison to Batch Processing

Previous work [5] computed the same analytics using Spark's traditional batch processing functionality and provides a baseline for comparison. One of the valuable aspects of Spark Streaming is that much of the code used for its batch processing can be adapted for stream processing as the higher level APIs, including Spark SQL, are the same resulting in less rework when the scope increases to process continuous data. That being said, query coverage is not identical and streaming does require modifications in such cases. For example Spark Streaming support for join operations are limited at the time of writing (Spark 2.4.0) and window operations require watermarking [6], which are considerations not necessary with batch processing. Given these characteristics it is arguably easier to begin with batch processing for learning or prototyping and adapt it for streaming if the use case demands it. All other things being equal, the performances of batch and stream processing using Spark for a fixed dataset are similar and certainly in the same order of magnitude. This observation is in line with expectation as both paradigms use the same mechanism for defining and distributing tasks under the hood. However, processing new data requires a full recalculation using batch processing while stream processing can handle it incrementally making it a better choice for growing datasets.

V. Results

For each desired analytics the description, technical decomposition, Spark SQL query, and results (sample, when applicable) will be presented. Boilerplate and extended metrics are omitted for brevity but can be found in the open source publication in UserNotebookStreaming [4].

A. (1-1) Rank the top 10 most popular airports by numbers of flights to/from the airport.

Popularity is ranked by the count of flights for which an airport is either the origin or the destination.

```
airports_df = cleaned_df.select(cleaned_df.
    Origin).withColumnRenamed("Origin", "
    Airport").union(cleaned_df.select(
    cleaned_df.Dest).withColumnRenamed("Dest",
     "Airport"))
```

```
SELECT `Airport`, COUNT(1) as `Count` FROM `
    airports` GROUP BY `Airport` ORDER BY `
    Count` DESC LIMIT 10
```

TABLE I
1-1 Results

Airport	Count
ORD	12,449,354
ATL	11,540,422
DFW	10,799,303
LAX	7,723,596
PHX	6,585,534
DEN	6,273,787
DTW	5,636,622
IAH	5,480,734
MSP	5,199,213
SFO	5,171,023

TABLE II
1-2 Results

Carrier	Delay	Count
HA	-1.011804	261,175
AQ	1.156923	151,507
PS	1.450639	41,581
ML (1)	4.747609	69,119
PA (1)	5.322431	294,891
F9	5.465881	320,468
NW	5.557783	9,473,613
WN	5.560774	14,818,642
OO	5.736312	2,936,047
9E	5.867185	462,424

B. (1-2) Rank the top 10 airlines by on-time arrival performance.

On-time performance is defined as lowest mean delay including negative delay. This definition is used for both arrival and departure performance for all queries.

```
SELECT `UniqueCarrier` AS `Airline`, MEAN(`
    ArrDelay`) AS `Delay`, COUNT(1) AS `Count`
     FROM `cleaned` WHERE `ArrDelay` IS NOT
    NULL GROUP BY `UniqueCarrier` ORDER BY `
    Delay` ASC LIMIT 10
```

C. (1-3) Rank the days of the week by on-time arrival performance.

DayOfWeek value of 1 is Monday, 2 is Tuesday, 7 is Sunday, etc. [1]

```
SELECT `DayOfWeek`, MEAN(`ArrDelay`) AS `Delay
    `, COUNT(1) AS `Count` FROM `cleaned`
    WHERE `ArrDelay` IS NOT NULL GROUP BY `
    DayOfWeek` ORDER BY `Delay` ASC
```

TABLE III
1-3 Results

Day Of Week	Delay	Count
6	4.301670	14,756,761
2	5.990459	16,663,909
7	6.613280	15,917,782
1	6.716103	16,774,403
3	7.203656	16,702,628
4	9.094441	16,700,097
5	9.721032	16,758,394

TABLE IV
2-1 RESULTS

Rank	SRQ	CMH	JFK	SEA	BOS
1	TZ	DH	UA	OO	TZ
2	XE	AA	XE	PS	PA (1)
3	YV	NW	CO	YV	ML (1)
4	AA	ML (1)	DH	TZ	EV
5	UA	DL	AA	US	NW
6	US	PI	B6	NW	DL
7	TW	EA	PA (1)	DL	XE
8	NW	US	NW	HA	US
9	DL	TW	DL	AA	AA
10	MQ	YV	TW	CO	EA

TABLE V
2-2 RESULTS

Rank	SRQ	CMH	JFK	SEA	BOS
1	EYW	AUS	SWF	EUG	SWF
2	TPA	OMA	ANC	PIH	ONT
3	IAH	SYR	MYR	PSC	GGG
4	MEM	MSN	ISP	CVG	AUS
5	FLL	CLE	ABQ	MEM	LGA
6	BNA	SDF	UCA	CLE	MSY
7	MCO	CAK	BGR	BLI	LGB
8	RDU	SLC	BQN	YKM	OAK
9	MDW	MEM	CHS	SNA	MDW
10	CLT	IAD	STT	LIH	BDL

D. (2-1) For each airport X, rank the top-10 carriers in decreasing order of on-time departure performance from X.

```
SELECT `Origin`, `UniqueCarrier`, MEAN(`
      DepDelay`) AS `Delay`, COUNT(1) AS `Count`
       FROM `cleaned` WHERE `DepDelay` IS NOT
      NULL GROUP BY `Origin`, `UniqueCarrier`
      ORDER BY `Origin`, `Delay` ASC
```

E. (2-2) For each airport X, rank the top-10 airports in decreasing order of on-time departure performance from X.

This can be interpreted to mean that the ranked airports are the flight destinations, and for each origin airport find the top ten destinations by on-time departure performance from the origin.

```
SELECT `Origin`, `Dest`, MEAN(`DepDelay`) AS `
      Delay`, COUNT(1) AS `Count` FROM `cleaned`
       WHERE `DepDelay` IS NOT NULL GROUP BY `
      Origin`, `Dest` ORDER BY `Origin`, `Delay`
       ASC
```

F. (2-3) For each source-destination pair X-Y, rank the top-10 carriers in decreasing order of on-time arrival performance at Y from X.

```
SELECT `Origin`, `Dest`, `UniqueCarrier`, MEAN
      (`ArrDelay`) AS `Delay`, COUNT(1) AS `
      Count` FROM `cleaned` WHERE `ArrDelay` IS
      NOT NULL GROUP BY `Origin`, `Dest`, `
      UniqueCarrier` ORDER BY `Origin`, `Dest`,
      `Delay` ASC
```

TABLE VI
2-3 RESULTS

Rank	LGA-BOS	BOS-LGA	OKC-DFW	MSP-ATL
1	TW	TW	TW	EA
2	US	US	EV	OO
3	PA (1)	DL	AA	FL
4	DL	PA (1)	MQ	DL
5	EA	EA	DL	NW
6	MQ	MQ	OO	OH
7	NW	NW	OH	EV
8	OH	AA		
9	AA	OH		
10		TZ		

TABLE VII
2-4 RESULTS

X	Y	Delay	Count
LGA	BOS	1.483865	168,421
BOS	LGA	3.784118	165,623
OKC	DFW	4.969055	72,678
MSP	ATL	6.737008	79,010

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

Includes negative values for arrival delay in mean.

```
SELECT `Origin`, `Dest`, MEAN(`ArrDelay`) AS `
      Delay`, COUNT(1) AS `Count` FROM `cleaned`
       WHERE `ArrDelay` IS NOT NULL GROUP BY `
      Origin`, `Dest` ORDER BY `Origin`, `Delay`
       ASC
```

H. (3-2 - Hypothetical flight routes given arbitrary constraints)

Tom wants to travel from airport X to airport Z. However, Tom also wants to stop at airport Y for some sightseeing on the way. More concretely, Tom has the following requirements (for specific queries, see the Task 1 Queries and Task 2 Queries)

a) The second leg of the journey (flight Y-Z) must depart two days after the first leg (flight X-Y). For example, if X-Y departs on January 5, 2008, Y-Z must depart on January 7, 2008.

b) Tom wants his flights scheduled to depart airport X before 12:00 PM local time and to depart airport Y after 12:00 PM local time.

c) Tom wants to arrive at each destination with as little delay as possible. You can assume you know the actual delay of each flight.

Find, for each X-Y-Z and day/month (dd/mm) combination in the year 2008, the two flights (X-Y and Y-Z) that satisfy constraints (a) and (b) and have the best individual performance with respect to constraint (c), if such flights exist.

```
three_two_query1 =
SELECT CONCAT(`x`.`Origin`, '-', `y`.`Origin`,
        '-', `y`.`Dest`, '-', `x`.`FlightDate`)
     AS `Key`, `x`.`Origin` AS `X`, `x`.`
      FlightDate` AS `FlightDate1`, `x`.`
      UniqueCarrier` AS `UniqueCarrier1`, `x`.`
      FlightNum` AS `FlightNum1`, `x`.`
```

TABLE VIII
3-2 RESULTS

Date	X	Flight 1	Y	Flight 2	Z
4/3/2008	BOS	FL 270 - 6:00	ATL	FL 40 - 18:52	LAX
9/7/2008	PHX	B6 178 - 11:30	JFK	NW 609 - 17:50	MSP
1/24/2008	DFW	AA 1336 - 7:05	STL	AA 2245 - 16:55	ORD
5/16/2008	LAX	AA 280 - 8:20	MIA	AA 456 - 19:30	LAX

[a]For full flight details see UserNotebookStreaming on GitHub [4].

```
    CRSDepTimeHour` AS `CRSDepTimeHour1`, `x
    `.`CRSDepTimeMinute` AS `CRSDepTimeMinute1
    `, `x`.`ArrDelay` AS `ArrDelay1`, `y`.`
    Origin` AS `Y`, `y`.`FlightDate` AS `
    FlightDate2`, `y`.`UniqueCarrier` AS `
    UniqueCarrier2`, `y`.`FlightNum` AS `
    FlightNum2`, `y`.`CRSDepTimeHour` AS `
    CRSDepTimeHour2`, `y`.`CRSDepTimeMinute`
    AS `CRSDepTimeMinute2`, `y`.`ArrDelay` AS
    `ArrDelay2`, `y`.`Dest` AS `Z`, `x`.`
    ArrDelay` + `y`.`ArrDelay` AS `
    TotalArrDelay` \
FROM `cleaned` AS `x` \
INNER JOIN `cleaned` AS `y` ON (`x`.`Dest` = `
    y`.`Origin` AND `y`.`FlightDate` = `x`.`
    FlightDate` + INTERVAL 2 days) \
WHERE `x`.`ArrDelay` IS NOT NULL AND `y`.`
    ArrDelay` IS NOT NULL AND `x`.`
    CRSDepTimeHour` < 12 AND (`y`.`
    CRSDepTimeHour` > 12 OR (`y`.`
    CRSDepTimeHour` = 12 AND `y`.`
    CRSDepTimeMinute` > 0)) AND YEAR(`x`.`
    FlightDate`) = 2008
```

```
three_two_query2 = three_two_query1_df.groupBy
    ("Key").agg(F.min(F.struct("TotalArrDelay"
    , "Key", "X", "FlightDate1", "
    UniqueCarrier1", "FlightNum1", "
    CRSDepTimeHour1", "CRSDepTimeMinute1", "
    ArrDelay1", "Y", "FlightDate2", "
    UniqueCarrier2", "FlightNum2", "
    CRSDepTimeHour2", "CRSDepTimeMinute2", "
    ArrDelay2", "Z")).alias("best")) \
.select("Key", "best.X", "best.FlightDate1", "
    best.UniqueCarrier1", "best.FlightNum1", "
    best.CRSDepTimeHour1", "best.
    CRSDepTimeMinute1", "best.ArrDelay1", "
    best.Y", "best.FlightDate2", "best.
    UniqueCarrier2", "best.FlightNum2", "best.
    CRSDepTimeHour2", "best.CRSDepTimeMinute2"
    , "best.ArrDelay2", "best.Z", "best.
    TotalArrDelay")
```

VI. CONCLUSION

With the convenient abstractions provided by the Apache Spark Streaming stack and the breadth of managed cloud services offered by AWS, applying distributed stream processing and integrating various systems to store and process large datasets can be done with a modest amount of time and resources. The approach employed to generate these flight analytics can be extended to larger datasets and a wide range of queries. Due to its distributed nature performance is effectively horizontally scalable, so more data or complexity can be handled by provisioning a cluster with more commodity hardware. In addition to the value that can be captured from the analytics, the ability to scale to handle the ever growing amount of data generated makes this approach durable for tomorrow's data needs.

REFERENCES

[1] OST_R - BTS - Transtats. "https://www.transtats.bts.gov/Tables.asp?DB_ID=120&DB_Name=Airline%20On-Time%20Performance%20Data&DB_Short_Name=On-Time"

[2] Transportation Databases - AWS Public Data Set. "https://aws.amazon.com/datasets/transportation-databases/"

[3] Distributed Stream Processing, Systems Integration, and Cloud Services Applied to Flight Analytics - Illinois Media Space. "https://mediaspace.illinois.edu/media/t/0_f1wj3euj"

[4] ivan-gomes/flight-analytics-stream-processing - GitHub. "https://github.com/ivan-gomes/flight-analytics-stream-processing"

[5] I. Gomes. Distributed Computing, Systems Integration, and Cloud Services Applied to Flight Analytics.

[6] Structured Streaming Programming Guide - Spark 2.4.0 Documentation. "https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#support-matrix-for-joins-in-streaming-queries"