NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

USING APACHE SPARK TO SPEED ANALYSIS OF ADS-B AIRCRAFT-TRACKING DATA TECHNIQUES

by

Jim Z. Zhou

June 2018

Thesis Advisor: Neil C. Rowe
Second Reader: Arijit Das

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2018	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE USING APACHE SPARK TO SPEED ANALYSIS OF ADS-B AIRCRAFT-TRACKING DATA TECHNIQUES	**5. FUNDING NUMBERS** W8A99
6. AUTHOR(S) Jim Z. Zhou	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

The U.S. Navy is exploring the feasibility of using a big-data platform and machine-learning algorithms to analyze combat-identification data. Combat identification involves a large number of remote sensors that report back data for aggregation and analysis. In this thesis, we used a sample of ADS-B aircraft-tracking data to test big-data methods for machine-learning methods developed previously. We showed large speed improvements in the analysis setup over the previous single-processor methods, and a lesser speed improvement for machine-learning based anomaly analysis.

14. SUBJECT TERMS big data, machine learning, Apache Spark, ADS-B	**15. NUMBER OF PAGES** 63
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

USING APACHE SPARK TO SPEED ANALYSIS OF ADS-B AIRCRAFT-TRACKING DATA TECHNIQUES

Jim Z. Zhou
Civilian, Naval Postgraduate School
BS, Naval Postgraduate School, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2018

Approved by: Neil C. Rowe
Advisor

Arijit Das
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The U.S. Navy is exploring the feasibility of using a big-data platform and machine-learning algorithms to analyze combat-identification data. Combat identification involves a large number of remote sensors that report back data for aggregation and analysis. In this thesis, we used a sample of ADS-B aircraft-tracking data to test big-data methods for machine-learning methods developed previously. We showed large speed improvements in the analysis setup over the previous single-processor methods, and a lesser speed improvement for machine-learning based anomaly analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ADS-B	Automatic Dependent Surveillance – Broadcast
DoD	Department of Defense
HDFS	Hadoop Distributed File System
HPC	High-Performance Computing
JSON	JavaScript Object Notation
LLA	Lexical Link Analysis
MR	MapReduce

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Neil Rowe, who believed in me while completing this thesis and gave guidance, direction and suggestions throughout the process. I would like to thank Professor Arijit Das, who provided the hardware for this thesis and guided and helped me while writing this thesis. I would also like to thank the Graduate Writing Center coaches and thesis processors for their help and greatly appreciated assistance.

Finally, to my wife and daughter, who were patient and supportive all throughout this effort.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Decision-making is critical to Navy missions to maximize effectiveness and to minimize casualties during the mission (Joint Chiefs of Staff, 1998). A combat-identification system identifies friendly, hostile, and neutral entities in a battlespace. During Operation Desert Storm, two friendly Black Hawk helicopters shot down, which suggested improvement in the combat-identification system (Government Accountability Office, 1995). This thesis looks at using a big-data platform to process aircraft data for combat-identification purposes.

Accurately and quickly identifying aircraft is difficult because they move very fast compared to other targets. In combat, assuming that a vehicle will be cooperative does not hold true always, which makes the task of combat identification a challenge. In addition, networks used in combat-identification systems can be overwhelmed with data from sensors and can malfunction. Much aircraft data collects via satellite and radar, which monitor (via sensors) the complete flight path. This leads to an overabundance of data that handles by combat-identification platforms operating in time-critical scenarios. Large amounts of data could be sent to systems designed for big-data analysis, but transmitting that data would require a large bandwidth which is difficult to obtain in the Navy. Inadequate data in combat identification can cause bad decisions, as it is hard to judge the value of data out of context.

Figure 1 shows a typical combat-identification scenario with a centralized combat-identification server which communicates via a limited link bandwidth to aircraft, tanks, and radar equipment. The figure shows how combat-identification data transfers from variety of combat-identification sensors to a server. Link-16 is a tactical communication link that is often used for these data transfers. However, combat-identification functioning in a purely centralized fashion would be a single point of failure, so an adversary could attack it to disrupt the entire process of combat identification.

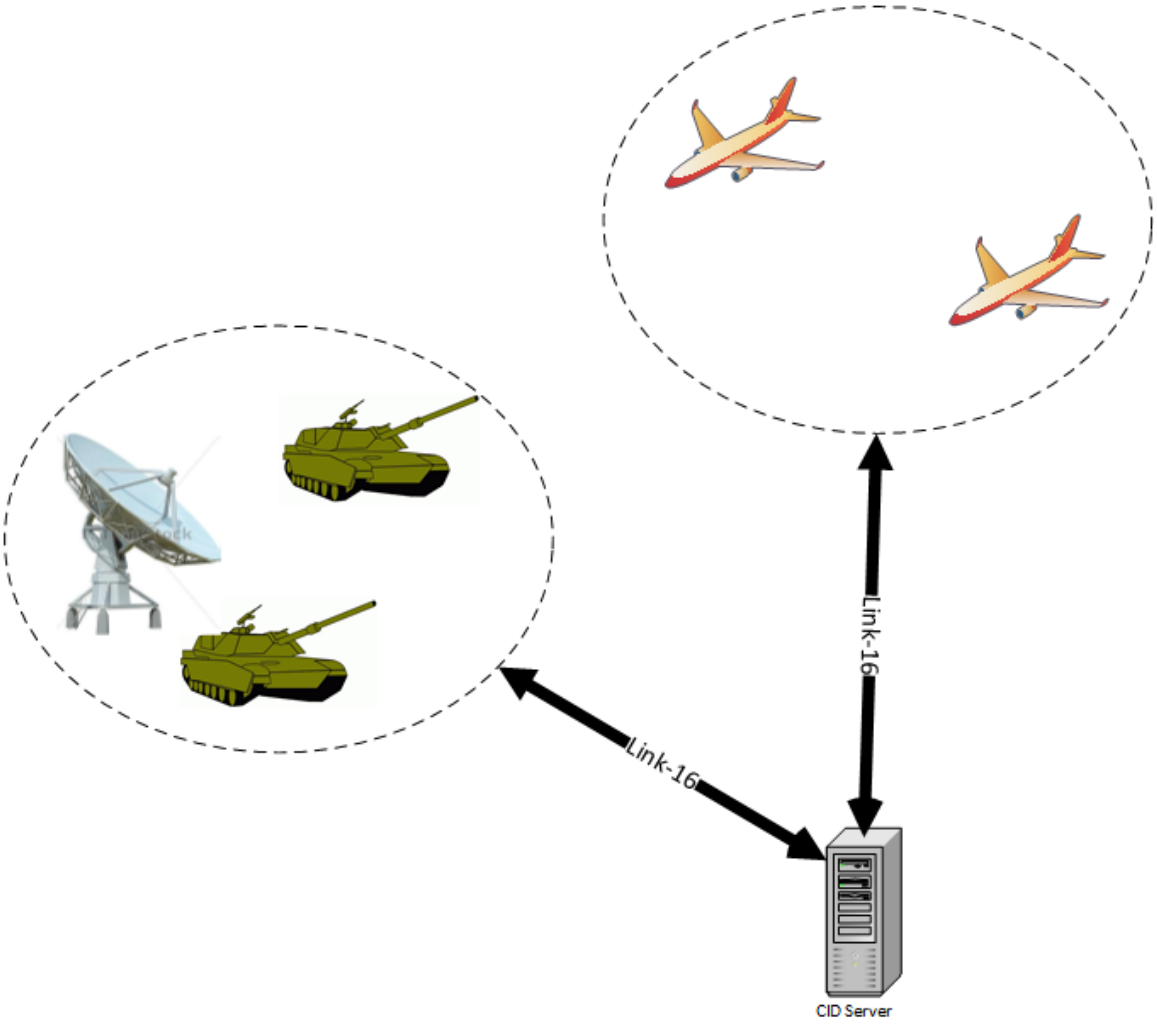

Figure 1. Information exchange between combat-identification server and aircraft. Adapted from Hinson, Summitt, and Shepko (2009).

Today the U.S. military has many combat-identification sensors around the world, and their total data volume often overwhelms combat-identification systems (Hinson, Summitt, & Shepko, 2009). To process information more rapidly, a distributed-processing solution (Gebara, Hofstee, & Nowka, 2015) might be a good choice. Distributed processing uses high-performance computing (HPC) hardware, but this is not very cost-effective because it requires custom hardware, software, and programming techniques. A commodity solution for parallel processing with much of community support is the Hadoop Distributed File System (HDFS). It comes from the Apache Software Foundation, a non-

profit organization that promotes open-source software. Its big-data ecosystem consists of a distributed file system HDFS, software applications, and a programming environment for software developers. It uses a divide-and-conquer model wherein data is distributed across nodes of a cluster, processing is done at each node, and the data from each node is aggregated for the result.

Machine-learning algorithms do many repetitive calculations in which intermediate calculations written to memory or storage, and read back many times. This kind of analysis happens in big-data contexts with the standard programming paradigm known as MapReduce (MR), in which data is mapped to nodes, shuffled based on the key values, and aggregated in the reduce phase. This whole process creates many reads and writes using disk input/output (disk I/O), which can slow down the machine learning. Gollapudi (2016) provides some further background about machine-learning algorithms.

Most commercial and non-commercial aircraft now have transponders for Automatic Dependent Surveillance – Broadcast (ADS-B) data format (Federal Aviation Administration, 2010). A typical active aircraft produces approximately 40 GB of data every day (ADS-B Exchange, n.d.-a). Anomalous behavior occurs when aircraft fly outside of the normal altitude, speed, location, and turn radius values. Automated detection of anomalies uses various techniques including machine learning. In recent years with advancement of computing technologies, machine learning becomes increasingly feasible (Rowe, Das, & Zhou, 2018).

The research questions we will address are:

- How much faster can Apache Spark perform in a cluster of a few nodes compared to a single-node server?
- How well can the anomaly-detection techniques apply to the ADS-B data set?

This thesis used a sample of the publicly available ADS-B data which is similar to the military combat-identification aircraft data that is the ultimate subject of this work. Both have latitude, longitude, altitude, timestamp, speed, etc. We compared the time spent

on a machine running a program in the Python programming language on a single processor with programs to do the same things using Pyspark, a Python-compatible Spark distributed-programming language.

Chapter II discusses related work about ADS-B and other attempts to solve the combat-identification problem. Chapter III introduces ADS-B, the data studied for this research, and Apache Spark, the system used to process the ADS-B data. It also introduces machine learning. Chapter IV describes our methods, and Chapter V discusses findings. Lastly, Chapter VI presents conclusions and suggestions for further study.

II. RELATED WORK

This chapter will explore previous work on the topics of combat identification, ADS-B, and big-data clusters.

A. COMBAT IDENTIFICATION

Ten to twenty percent of casualties in military operations reported, caused by friendly fire (Ministry of Defence, 2002; Defense Science Board, 1996). During Operation Desert Storm the rate was 24 percent (United States General Accounting Office, 2001). To better identify targets (friend, foe, and neutral), the General Accounting Office recommended the Department of Defense (DoD) develop an enterprise architecture for combat identification. However, there are many challenges in developing and implementing these systems.

Much analysis can be done if relevant data can be collected in a centralized locations or set of locations. Van Gosliga and Jansen (2003) used a machine-learning technique as known as a Bayesian network to analyze combat-identification data. A Bayesian network uses probabilities based on prior knowledge of factors. Combat-identification data includes factors such as altitude, speed, location, and state information. If some data is missing, it can be estimated by prediction. However, collecting large quantities of data on a single site can be a challenge.

B. ADS-B

A sample of ADS-B data tested our machine-learning techniques and processes. ADS-B gives aircraft data similar to military-aircraft data and is open-source. Sun, Ellerbroek and Hoekstra (2016) noted a number of errors in ADS-B data which make it troublesome to process. They extracted the ADS-B data for individual flights, and then used fuzzy logic to analyze it. They used thefields of aircraft identification number, timestamp, latitude, and speed. They used MongoDB (NoSQL database) for the storage of data, run on a solid-state disk (SSD).

Tabassum, Allen and Semke (2017) showed that, more than errors, missing values will highly affect the data accuracy. Removing them will improve performance of separation-detection and collision-detection algorithms for manned and unmanned aircraft. There is a discussion of NextGen, which is a newer system that assures safety.

The data we are using is large since it contains a record for every major aircraft in the air every second. Gebara et al. (2015) showed that to do successful big-data processing, scalability, resilience, and ease of programming are needed. The Apache HDFS operates nodes in a failsafe mode to meet the resiliency requirement. A Hadoop cluster can scale up to a few hundred nodes. Hadoop also provides the MapReduce toolset for programmers to build applications. Apache Spark uses in-memory processing techniques to achieve even faster speed with big data. The speed of the network interconnection between Hadoop nodes improves with faster interconnect.

Salcido, Kendall and Zhao (2017) used lexical link analysis (LLA) to analyze ADS-B data. LLA is a text mining method that finds relationships and associations within the data. They used a Hadoop cluster for parallel processing due to the large data sizes, and got encouraging results. This inspired us to try a similar approach with a different set of analysis algorithms for aircraft data.

III. METHODS

This chapter will introduce big-data technology and explain its application to the current problem. Using any new technology comes with the price of new hardware and software, so costs must be kept manageable. One also must use the right tools to solve the problem at hand.

A. BIG-DATA METHODS

The legacy architecture to process large amounts of data is high-performance computing (HPC) which involves specialized hardware and custom software. Programming algorithms in HPC involve management of the data flow by manual program writing. Overall, HPC is difficult for teams with limited budgets wanting to solve large data problems. In contrast, "big-data" computing uses commodity hardware and automated data movement.

One big-data approach, Google File System, originated from Google's proprietary technology to process large amounts of data that could not be handled by a traditional hardware/software architecture (Ghemawat, Gobioff, & Leung, 2003). Google File System uses the MapReduce idea discussed in section II.B. It works well for tasks where the processing is alike for all the data and not context-dependent, so data can be partitioned arbitrarily for processing. In addition to "divide and conquer," the end user of Google File System does not have to manage the nodes or data movement, which is all handled by the system. Automatic workload distribution is also done so faster nodes are tasked with more work. The cluster of nodes can be easily scaled (data growth) without installation of any new software. This technology is proprietary, and involves expensive licensing costs (Shen, Wu, Li, & Liu, 2016).

Yahoo created an open-source version of Google File System called the Hadoop Distributed File System (HDFS) (Shvachko, Kuang, Radia, & Chansler, 2010). Open-source projects provide technology at a lower cost, and provide a large pool of developers who can write, test, and deliver a finished product at minimal cost. Due to rising costs of

in-house and proprietary software development, the DoD is increasingly using open-source licensing.

The early version of HDFS, known as Map-Reduce, was very intensive in file input/output, meaning data read from disk, processed in memory, and then written back. This meant that program ran slowly because a disk is significantly slower than memory. The community improved on HDFS with Apache Spark (Ganelin et al., 2016). Spark uses intelligent data management to store as much as possible in memory for as long as possible. Figure 2 summarizes HDFS, Map-Reduce, and Spark as used in our research. Yet Another Resource Manager (YARN) is part of HDFS and manages the load distribution across nodes.

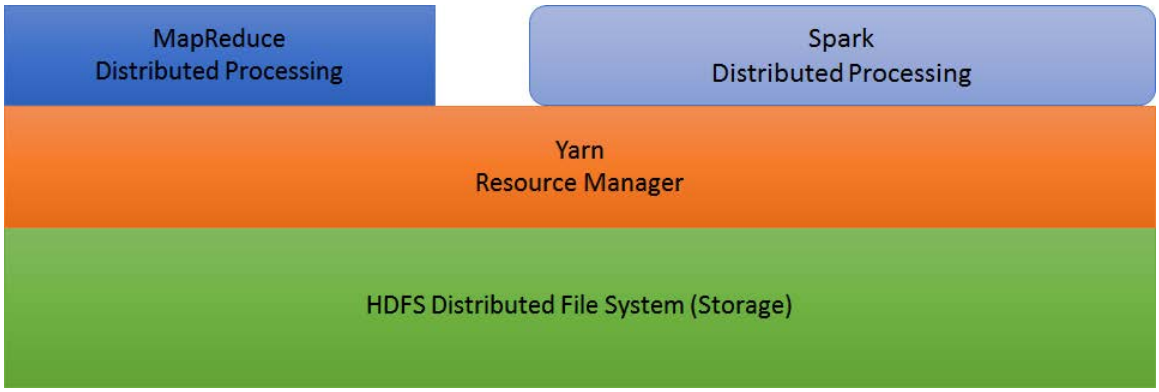

Figure 2. The Hadoop system. Adapted from Achari (2015).

HDFS and Spark coexist in the same platform, so a programmer who implements the algorithms has to decide which to pick. HDFS gives access to files on disk, read and processed, while Spark takes over management of files instead giving a dataframe (tabular view of the data) to the programmer. This means a significant change of paradigm when it comes to implementing algorithms.

B. LINEAR MODELING

This research used a linear model, mathematically represented as:

$$y = w1 * k1 + w2 * k2 + \ldots\ldots + w11 * k11$$

The w's represent the weights applied on each factor and the k's represent the factors. This work implemented a model with seven factors related to aircraft speed, altitude, commerciality, and operator (Rowe et al., 2018).

The z-transform was used to normalize factors that had varying ranges, and involved subtracting the average from each data point and dividing by its standard deviation (Sun et al., 2016). A sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ was then applied to the factors to put them in a range of 0 to 1.

C. ADS-B DATA

A sample from the Automatic Dependent Surveillance – Broadcast (ADS-B) database was used in this research. A full combat-identification system would use additional sources of data. ADS-B is publicly available on a server (data exchange), and can be retrieved in the open data standard JavaScript Object Notation (JSON). We used ADS-B data in our work, as it is similar to Navy combat-identification aircraft data which contains latitude, longitude, altitude, and other fields that define an aircraft track. Figure 3 shows ADS-B and how its data is available. The ADS-B exchange is a comprehensive (worldwide) repository.

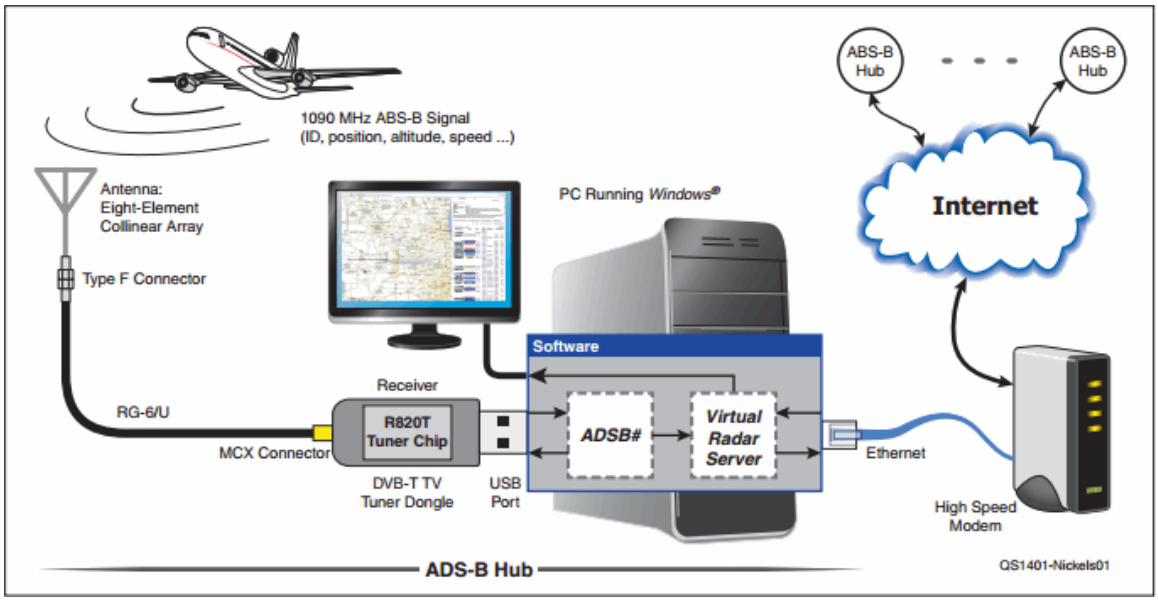

Figure 3. ADS-B receiving station block diagram. Source: Nickels (2014).

ADS-B data is not always consistent. Data points were missing, some tracks were not contiguous, and some tracks only appeared for a short period (Tabassum et al., 2017). We chose the following fields for our work.

Table 1. ADS-B data field descriptions. Source: ADS-B Exchange (n.d.-b).

Data Field	Data Type	Description
ICAO	six-digit hex	The code that uniquely identifies each aircraft.
Alt	integer	The altitude in feet at standard pressure.
Lat	float	The aircraft's latitude.
Long	float	The aircraft's longitude.
PosTime	units: epoch milliseconds	The time of reporting of the data.
Spd	knots, float	The ground speed in knots.
Trak	degrees, float	Aircraft's track angle across the ground clockwise from 0° north.
Op	string	The name of the aircraft operator (owner).
Cou	string	The aircraft country registration.
Type	string	Aircraft type, e.g., B777.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. EXPERIMENTS

The experiments in this research used a cluster of three Dell servers. They were configured to run Cloudera Hadoop Distribution upgraded with Spark software. The ADS-B data downloaded from the exchange and moved to the cluster. Spark organized the ADS-B files as a dataframe (equivalent to a database table) for running machine-learning algorithms in the PySpark package in the Python programming language.

A. CLUSTER HARDWARE AND SOFTWARE

The three Dell servers had different memory, CPU, and disk configurations, and each was configured and mounted on a single rack. The operating systems for these servers were Red Hat Enterprise Linux 7. The servers networked via network interface cards (1 GB speed) through an Ethernet switch. The Hadoop Distributed File System (HDFS) installed on top of the Linux file system (Das & Plunket, 2015). We used a vendor-supported version of HDFS (Cloudera). YARN software distributed the processing load and available disk space so that less powerful servers in the cluster were tasked less compared to the more powerful servers (Murthy, 2012).

Table 2. Cloudera cluster servers' hardware configurations

	CPU	Memory	Hard Drive
Dell PowerEdge R630	Intel® Xeon ® 1.6 GHz processor speed with 8 cores	384GB	3TB (6 x 500GB) Samsung Solid State Drive
Dell PowerEdge R230	Intel® Xeon ® 3.5 GHz processor speed with 12 cores	64GB	2 TB (4 x 500GB) Samsung Solid State Drive (SSD)
Dell PowerEdge R230	Intel® Xeon ® 3.5 GHz processor speed with 12 cores	64GB	2 TB (4 x 500GB) Samsung Solid State Drive (SSD)

The cluster we tested used solid-state drives rather than the slower mechanical spinning disks. Previous work at the Naval Postgraduate School has shown that the biggest processing bottleneck was the sequence of data transfers to the disk. HDFS is very disk-intensive so this helped speed up processing (Das & Annamalai, 2015).

Memory size affected processing speed considerably for our tasks because data used repeatedly is better to keep in memory than on disk. Memory is expensive, so as a tradeoff one server configured with 384 GB of memory compared to 64GB for the others. HDFS YARN recognized this and allocated more tasks to the large-memory server. After the hardware was configured and tested with Cloudera HDFS, Apache Spark was installed (Cloudera, n.d.), followed by the Python package PySpark2.

B. LOADING THE ADS-B DATA

Data was downloaded from the website ADS-B Exchange (https://adsbexchange.com/data). Data from the website is available in zip-compressed JSON format. This exchange records around 11 million records per day, about every second for major aircraft and less often for non-major aircraft. For this research, Prof. Rowe chose a single day of the month from each of the 12 months. The files loaded to HDFS were replicated three times (the default redundancy of HDFS). 1TB (948GB) was used to store the data.

Files in HDFS can be processed with programming languages like Java and Python. Spark is the next generation of HDFS processing that uses memory store, and can be programmed with PySpark. Spark does not give the user direct access to files, but manages it internally using a combination of available memory and disk, and then provides a dataframe for the user to work with. A dataframe is like a database table and requires Structured Query Language (SQL) to access it (Xin, Armbrust, & Liu, 2015). To load data into Spark, the data format is changed from JSON to Parquet. Parquet is a tabular storage system (Apache, n.d.-b; Chen, 2016). It has advantages, for example, when finding the average over a single column since it only reads data for one column. Our experiments required roughly 24 minutes to load 762 GB of data.

C. PREPROCESSING THE DATA

Not all fields of ADS-B data were used in our work, so they were preprocessed to extract the data shown in Table 1. There were still some faulty data like blanks, and invalid data values (Tabassum et al., 2017) which needed to be excluded. The following PySpark command extracts these fields from a dataframe df to create a new dataframe df1:

```
df1 = df.select(‘Icao', ‘Alt', ‘Lat’, Long’, ‘Spd', ‘Trak', ‘PosTime', ‘Op', ‘Cou')
.na.drop(subset=["Icao"])
```

The na.drop command drops any row with a null ICAO value.

For anomaly detection, more dataframes must be created. The data was averaged in bins based on latitude and longitude. The original prototype processing in Python read and did calculations row by row for each file. With Spark, we used SQL commands on a dataframe. The code first adds a bin identifier to each row of data and then uses the GROUP BY construct to aggregate properties for that bin. Spark knows where it has stored the data in memory or disk, and on which server, and how to run the logic in parallel across each server, so the end user does not need to specify it.

Another programming feature used was user-defined functions (UDFs) (Apache, n.d.-a). Below an UDF is defined to square the data in a dataframe and display it. SQL cannot use complex data manipulation constructs, so UDFs are necessary for them.

Step 1 (Register an UDF):

```
Def squared(s)
        Return s * s
sqlContext.udf.register(“squareWithPython”, squared)
```

Step 2 (Use the UDF with a dataframe):

```
From pySpark.sql.functions import udf
squared_udf = udf(squared, LongType())
display(df.select(“id”, squared_udf(“id”.alias(“id_squared)))
```

The following example shows how to create a column of latitude bins. First, we create a user-defined function that rounds the latitudes.

```
udf('int')
def round_lat(lat):
    return round(lat)
```

Second, we add the latitude as a column in the data frame.

```
df = df.withColumn('lat_bin', round_lat(df.Lat))
```

Then we use the SQL “groupby” command to aggregate the latitudes.

```
aircraft_avg_df = df.groupby('lat_bin')
```

A separate process collected data for each track and summarized it. The ICAO number (unique aircraft identifier) was used to sort the dataframe by the “orderby” SQL command. Aggregate calculations including the mean, average, maximum, and minimum were done by the “window.partitionby” with SQL. This is a step-by-step process, and a new column was added each time to the dataframe using the “withcolumn” SQL command. The “withcolumn” combined with a user-defined function to do computations on columns, like frequency calculation on airline operator. We had nine user-defined functions.

In more detail, we first sorted the data by ICAO to separate out each aircraft:

```
my_window = Window.partitionBy('Icao').orderBy("PosTime")
```

Next, for each ICAO we segmented tracks if the time between two consecutive records was more than one hour:

```
df2 = df1.withColumn("trek", new_trek(df1.diff, df1.Icao))
```

Next, we separated each track using “window.partitionby”.

```
trek_window = Window.partitionBy('trek'). orderBy('PosTime')
```

Finally, for each track, we computed the key statistics of the starting, ending, and peak altitude, and the associated locations and times. This then permitted us to calculate the

whole trip duration, average latitude-longitude deviation from a straight line, and average altitude deviation. The full code is in the Appendix.

D. ANOMALY DETECTION

Unusual flight paths indicate anomalous behavior for aircraft, and these could represent military threats. There are twelve anomaly factors that we have identified (Rowe et al., 2018). At the time of completion of our work, we had used seven of those, all weighted equally in a sum. Although the factors are separately summed for each aircraft record, some data comes from aggregation tables discussed earlier for latitude-longitude bins, speeds, times, and operators.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESULTS

The original single-threaded Python code ran on a Linux sever with 512 GB of memory. Our experiments tested the degree to which using the HDFS/SPARK cluster resulted in a speedup of processing. Figure 4 shows the relative processing times of five key parts of the analysis of the aircraft data.

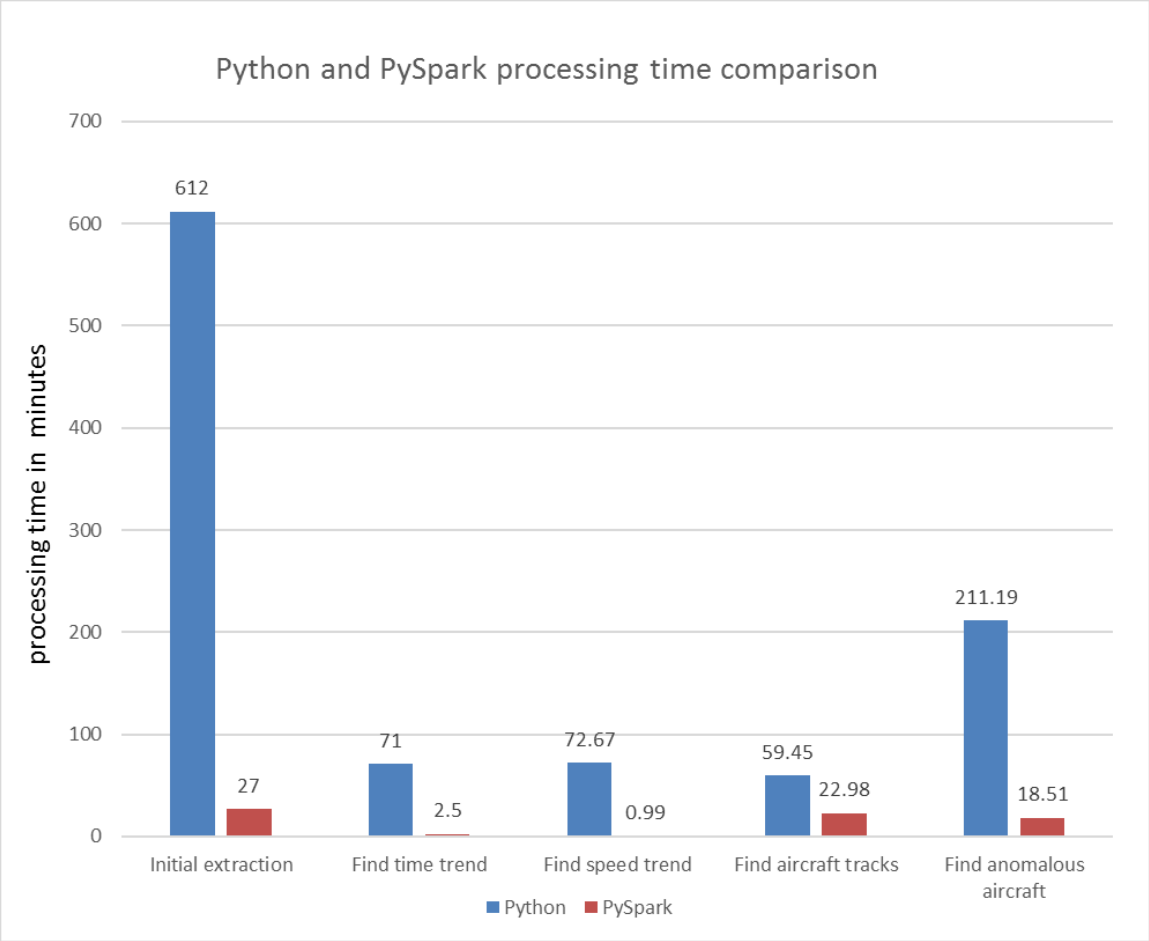

Figure 4. Python and PySpark processing-time comparison on the ADS-B sample

In the first step, aircraft data (for twelve days) in JSON format was loaded into Spark and the relevant fields were extracted. The next two calculations calculated averages as a function of time and speed. The aircraft track calculation was done next, followed by anomaly detection on each aircraft record. All the calculations compared traditional Python programs versus PySpark Python programs with database support. The PySpark code showed speedups ranging from one hundred to two.

The Cloudera HDFS provides a visualization tool to view resource usage, disk input/output, CPU usage, and network traffic between server nodes. To provide a comparison, four different size data files for one day, three days, seven days, and twelve days were tested. The process time, CPU load, disk input/output, and network traffic were all proportional to the amount of data, which is encouraging for this approach. Table 3 shows statistics where IO (input/output) is measured in megabytes per second.

Table 3. Comparison of different data sizes on CPU load, Disk IO and Network IO

Data size	Process time (minute)	CPU load	Disk IO (M/S)	Network IO (M/S)
One day (38GB)	9.75	50%	525	7
Three days (110GB)	32.45	50%	572	28
Seven days (260GB)	86	50%	572	47
Twelve days (316GB)	143	30%	572	75

Figure 5 is a sample of Cloudera Cluster CPU monitoring chart; the horizontal axis is time and the vertical axis is percentage of CPU usage. The dotted line is the peak value and the solid blue line is the mean average value. This plot shows the reading of the JSON data, extraction, and loading into a dataframe.

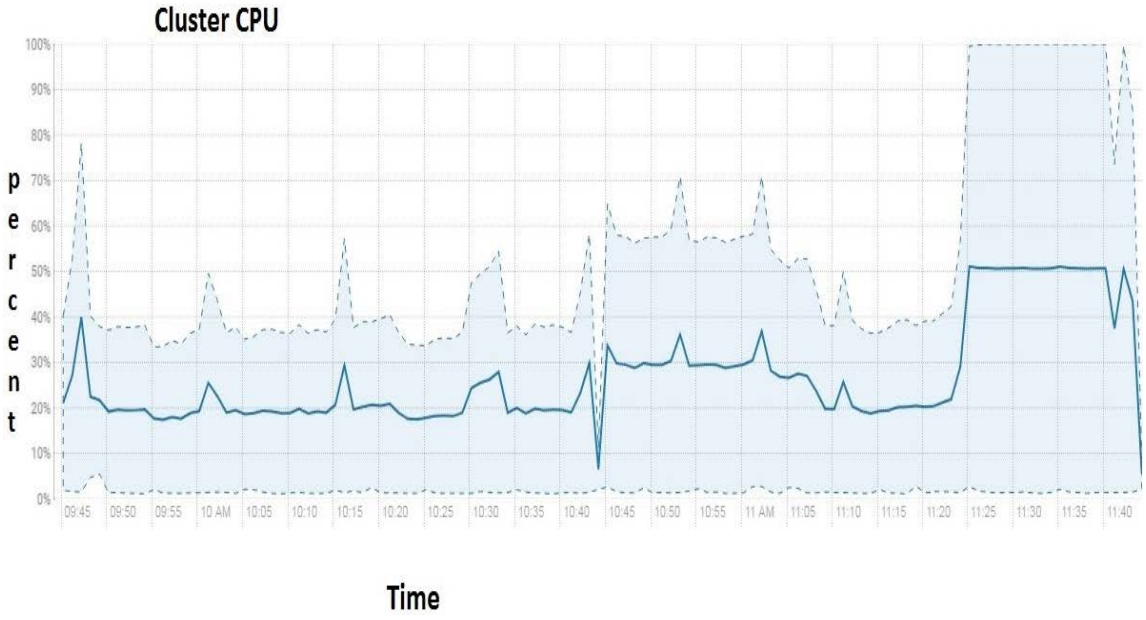

Figure 5. Cloudera monitoring chart of cluster CPU utilization for processing of twelve days of data

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION AND RECOMMENDATIONS

This research compared traditional computing to a distributed big-data implementation using HDFS/Spark for a combat-identification problem. Experiments showed that the HDFS/Spark architecture does provide a significant speedup in computation time. For initial data extraction, HDFS/Spark was 22 times faster, and for aggregations over columns, HDFS/Spark was 35 and 71 times faster. For simplified anomaly calculation, HDFS/Spark was 11 times faster, but for track computations, HDFS/Spark was only 2.7 times faster. The last two reflect the added computational complexity of their tasks.

The next step is to study increasing data sizes and numbers of processors to see how the speedup will scale up. All programming in these experiments was Python, and it could improve speed to recode all the algorithms in Scala, which natively uses the Java Virtual Machine (JVM). The speed could also improve by use of the Spark machine-learning library.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX

A. MAKING ADS-B BINS

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, avg, desc, col, expr, when
import time, math
import pyspark.sql.functions as func
# UDFs' are pyspark user defined functions
# Use udf to define a round-latitudes
@udf('int')
def round_lat(lat):
        return round(lat)
# Use udf to define a round-longitudes udf
@udf('int')
def round_lon(lon):
        return round(lon)
####### Use udf to define x velocity ##############
@udf('float')
def vx(spd, trak):
        return (spd*math.cos(math.radians(trak)))
####### Use udf to define y velocity ##############
@udf('float')
def vy(spd, trak):
        return  (spd*math.sin(math.radians(trak)))
########### udf alt_adj_spd  ##########################
@udf
def alt_adj_spd(alt, spd):
        if (alt < 50000) and (alt >= 0.0):
                adjustment = 450*(1-math.exp(-0.000092*alt))
        else:
                adjustment = 450
        if (adjustment > 0.00001) and (alt >= 0.0) and (spd >= 0.0):
                adjustedspd = spd/(450*(1-math.exp(-0.000092*alt)))
        else:
                adjustedspd = 0.0
        return 10*math.log(adjustedspd+1)
####### add airlinenames ######
airlinenames = ['airline',' air','air ','airways','express','virgin', \
      'ryanair','easyjet','lufthansa','alitalia', 'qantas', \
      'american eagle','aer lingus','finnair','cityhopper', \
      'tap portugal','eurowings','regional','aeroméxico', \
```

```
        'icelandair','egyptair','transavia','cityjet', \
        'connection','tunisair','citilink']

######## udf for airline ####################
@udf('int')
def airline_operator(opname):
  for airlinename in airlinenames:
    if (opname.find(airlinename) > -1):
      return 1
  return 0
######## udf for non_airline ####################
@udf('int')
def non_airline_operator(opname):
  for airlinename in airlinenames:
    if (opname.find(airlinename) > -1):
      return 0
  return 1
### udf for oplogfreqs ####################
@udf('float')
def oplogfreqs(cnt):
          global totalcnt
          if (cnt == 0 or totalcnt == 0):
                    return 0.0
          else:
                    return (-math.log(cnt/totalcnt))

### Start the Spark session
spark = SparkSession.builder.appName("Find Anomalous
 Aircrafts"). getOrCreate()
spark.sparkContext.setLogLevel("ERROR")
start_time = time.time()
# Reading one year(one day a month for 12 months) JSON data
df = spark.read.json("/user/jim/adsb/2017")
## select the fields of interest and drop nulls
df = df.select('Icao', 'Alt', 'Lat', 'Long', 'Spd', 'Trak', 'PosTime', 'Op', 'Cou').\
          na.drop(subset=["Icao"]).na.drop(subset=["Lat"]).\
          filter((df.Lat > -90.1) & (df.Lat < 90.1) & (df.Long > -180.1) &
          (df.Long < 180.1 ) \
          & (df.Spd > -1.0) & (df.Spd < 4000) & (df.Alt > -1.0) & (df.Alt < 100000)
\
           & (df.Trak >= 0.0) & (df.Trak < 360.0001 ) & (df.PosTime != 0)).\
 withColumn('PosTime', (df.PosTime/1000).cast('timestamp')).
          orderBy("Icao", "PosTime")
############### add lat_bin and lon_bin columns ###############
```

```
        df = df.withColumn('lat_bin', round_lat(df.Lat)).\
                withColumn('lon_bin', round_lon(df.Long))
        df = df.withColumn('lat_bin', round_lat(df.Lat)).withColumn('lon_bin',
round_lon(df.Long))
        ########## add columns airlines and non_airlines ############
        df = df.withColumn('airlines', airline_operator(df.Op)).withColumn('non_airlines',
non_airline_operator(df.Op))
        ############# add airline_vx non_airline_vx airline_vy non_airline_vy
columns ############################
        airline_vx = when(df.airlines == 1, vx(df.Spd, df.Trak)).otherwise(0.0)
        non_airline_vx = when(df.airlines == 0, vx(df.Spd, df.Trak)).otherwise(0.0)
        airline_vy = when(df.airlines == 1, vy(df.Spd, df.Trak)).otherwise(0.0)
        non_airline_vy = when(df.airlines == 0, vy(df.Spd, df.Trak)).otherwise(0.0)
        df = df.withColumn('airline_vx', airline_vx).withColumn('non_airline_vx',
non_airline_vx).\
                withColumn('airline_vy', airline_vy).withColumn('non_airline_vy',
non_airline_vy)
        ############## add altitude for airline and altitude for non_airline
############################
        airline_alt = when(df.airlines == 1, df.Alt).otherwise(0)
        non_airline_alt = when(df.airlines == 0, df.Alt).otherwise(0)
        df = df.withColumn('airline_alt', airline_alt).withColumn('non_airline_alt',
non_airline_alt)
        ############## add columns for airline_alt_adj_spd and
non_airline_alt_adj_spd ###################
        airline_alt_adj_spd = when(df.airlines == 1, alt_adj_spd(df.Alt,
df.Spd)).otherwise(0)
        non_airline_alt_adj_spd = when(df.airlines == 0, alt_adj_spd(df.Alt,
df.Spd)).otherwise(0)
        df = df.withColumn('airline_alt_adj_spd',
airline_alt_adj_spd).withColumn('non_airline_alt_adj_spd', non_airline_alt_adj_spd)

        totalcnt = float(df.count())
        ### create a dataframe for groupby op and counts
        oplogfreq_df = df.groupby('Op').agg(func.count('Op').cast('float').alias('cnt'))
        ## add another column oplogfreq
        oplogfreq_df = oplogfreq_df.withColumn('oplogfreq',
oplogfreqs(oplogfreq_df.cnt))
        #oplogfreq_df.write.save('/user/jim/adsb/oplogfreqs', format='parquet',
mode='append')
        ### join two dataframes
        df = df.join(oplogfreq_df, ["Op"])
        #### add airline_oplogfreqs and non_airline_oplogfreqs columns
################
```

```
        airline_oplogfreqs = when(df.airlines == 1, oplogfreqs(df.cnt)).otherwise(0.0)
        non_airline_oplogfreqs= when(df.airlines == 0, oplogfreqs(df.cnt)).otherwise(0.0)
        df = df.withColumn('airline_oplogfreqs',
airline_oplogfreqs).withColumn('non_airline_oplogfreqs', non_airline_oplogfreqs)
        # average airline_oplogfreqs, non_airline_oplogfreqs ##
        aircraft_avg_df = df.groupby('lat_bin', 'lon_bin').agg(func.sum('airlines').\
                alias('airline_count'), func.sum('non_airlines').alias('non_airline_count'), \
                func.avg('airline_vx').alias('avg_airline_vx'), func.avg('non_airline_vx').\
                alias('avg_non_airline_vx'), func.avg('airline_vy').alias('avg_airline_vy'),\
                func.avg('non_airline_vy').alias('avg_non_airline_vy'), \
                func.avg('airline_alt').alias('avg_airline_alt'), \
                func.avg('non_airline_alt').alias('avg_non_airline_alt'),\
                func.avg('airline_alt_adj_spd').alias('avg_airline_alt_adj_spd'),\
                func.avg('non_airline_alt_adj_spd').alias('avg_non_airline_alt_adj_spd'),\
                func.avg('airline_oplogfreqs').alias('avg_airline_oplogfreqs'),\
                func.avg('non_airline_oplogfreqs').alias('avg_non_airline_oplogfreqs'))

        ###################### round floating points to 3 decimal place
###################
        aircraft_avg_df = aircraft_avg_df.select('lat_bin', 'lon_bin', 'airline_count',
        'non_airline_count',\
                func.round('avg_airline_vx', 3).alias('avg_airline_vx'),
        func.round('avg_non_airline_vx', 3).\
                alias('avg_non_airline_vx'), func.round('avg_airline_vy',
3).alias('avg_airline_vy'),\
                func.round('avg_non_airline_vy', 3).alias('avg_non_airline_vy'),\
                func.round('avg_airline_alt', 3).alias('avg_airline_alt'),\
                func.round('avg_non_airline_alt', 3).alias('avg_non_airline_alt'),\
                func.round('avg_airline_alt_adj_spd', 3).alias('avg_airline_alt_adj_spd'),\
                func.round('avg_non_airline_alt_adj_spd',
3).alias('avg_non_airline_alt_adj_spd'),\
                func.round('avg_airline_oplogfreqs', 3).alias('avg_airline_oplogfreqs'),\
                func.round('avg_non_airline_oplogfreqs',
3).alias('avg_non_airline_oplogfreqs'))

        aircraft_avg_df.write.save('/user/jim/adsb/adsb_bins_df', format='parquet',
mode='append')
        elapsed_time = time.time() - start_time
        total_time = round(elapsed_time/60, 2)
        print("Time : ", total_time, " minutes")
```

B. B. FIND SPEED TREND

```
########## Speed trend ##################
# this small file is to read the ADSB dataframe
# that was filtered out the important columns
# ''Icao', 'Alt', 'Lat', 'Long', 'Spd', 'Trak', 'PosTime', 'Op', 'Cou'
# then to use the 'Alt' and 'Spd' information to find out
# the speed trend with average speed and standard deviation speed output
from pyspark.sql import SparkSession
import pyspark.sql.functions as func
from pyspark.sql.functions import udf, avg, desc, col, expr, when, round, year, hour
import time, math, sys
spark = SparkSession.builder.appName("Speed Trend Aircrafts").getOrCreate()
spark.sparkContext.setLogLevel("ERROR")
## create a speed_trend dataframe
# by reading one year filtered file in parquet format
df = spark.read.parquet("/user/jim/adsb/df/*.parquet")
start_time = time.time()
granularity = 100
# udf to defind binindex
@udf('int')
def binindex(alt):
        return int((alt//granularity)*granularity+(0.5*granularity))

# create a speed_trend dataframe with only 'alt' and 'spd'
speed_trend_df = df.select('Alt', 'Spd')

# add another column as 'binindex'
speed_trend_df = speed_trend_df.withColumn('binindex', binindex(speed_trend_df.Alt))

# aggregated function on binindex and added two more columns 'avspeed' and 'sdspeed'
speed_trend_total_df = speed_trend_df.groupby('binindex').agg(func.avg('Spd').\
        alias('avspeed'), func.stddev('Spd').alias('sdspeed')).sort('binindex')

speed_trend_out = speed_trend_total_df.withColumn('avgspeed',
func.round(speed_trend_total_df['avspeed'], 2)).\
        withColumn('stdspeed', func.round(speed_trend_total_df['sdspeed'], 2)).\
        drop('sdspeed', 'avspeed')
# save speed trend for later usage (find_anomalous_data)
speed_trend_out.write.save('/user/jim/adsb/speed_trend_df', format='parquet',
mode='append')
speed_trend_out.show()
elapsed_time = time.time() - start_time
#total_time = round(elapsed_time/60, 2)
```

```
print("Time : ", elapsed_time/60, " minutes")
```

B. FIND TIME TREND

```
########## Time trend ##################
# this small file is to read the ADSB dataframe
# that was filtered out the important columns
# "Icao', 'Alt', 'Lat', 'Long', 'Spd', 'Trak', 'PosTime', 'Op', 'Cou'
# then to use the 'PosTime' information to find out
# the time trend with hourly how many events output
from pyspark.sql import SparkSession
import pyspark.sql.functions as func
from pyspark.sql.functions import udf, avg, desc, col, expr, when, round, year, hour
import time, math, sys
spark = SparkSession.builder.appName("Time Trend Aircrafts").getOrCreate()
spark.sparkContext.setLogLevel("ERROR")
start_time = time.time()

# create a dataframe by reading saved one month ADSB filterd file
# in hdfs on parquet format
df = spark.read.parquet("/user/jim/adsb/df/*.parquet")
time_trend_df = df.select('PosTime', 'Long')
@udf('int')
def longcorr(lon):
        return int((lon/360.0)*24)

#time_trend_df.withColumn("Year", year(time_trend_df.PosTime)).show()
time_trend_df = time_trend_df.withColumn("Hour", hour(time_trend_df.PosTime))
#should not need this information, will delete it later
time_trend_df = time_trend_df.withColumn("longcorr", longcorr(time_trend_df.Long))
time_trend_df =
time_trend_df.groupby('Hour').agg(func.count('longcorr').alias('hour_lon_count')).sort(de
sc("Hour"))
time_trend_df.write.save('/user/jim/adsb/time_trend_df', format='parquet',
mode='append')
time_trend_df.show(24)
elapsed_time = time.time() - start_time
#total_time = round(elapsed_time/60, 2)
print("Time : ", elapsed_time/60, " minutes")
```

C. FIND AIRCRAFT TRACKS

```
sqlContext.clearCache()
from pyspark.sql import SparkSession
import pyspark.sql.functions as func
from pyspark.sql.functions import udf, avg, desc, col, expr, when, first, last
import time, math, sys
from pyspark.sql.window import Window
from math import sin, cos, atan2, sqrt, degrees, radians, pi
from geopy.distance import great_circle as distance
from geopy.point import Point
from pyspark.sql.types import FloatType

#### udf to make new trek if
# icao changed or time diff more than an hour
@udf
def new_trek(diff, df_icao):
        global trek_num, icao_g, num
        if (df_icao != icao_g):
                num = num + 1
                trek_num = str(df_icao) + str(num)
                icao_g = df_icao
                return trek_num
        elif(df_icao == icao_g):
                if (diff > timethresh):
                        num = num + 1
                        trek_num = str(df_icao) + str(num)
                        return trek_num
                else:
                        trek_num = str(df_icao) + str(num)
                        return trek_num
        else:
                trek_num = str(df_icao) + str(num)
                return trek_num

@udf ('double')
def midpoint(a, b):
  a_lat, a_lon = radians(a.latitude), radians(a.longitude)
  b_lat, b_lon = radians(b.latitude), radians(b.longitude)
  delta_lon = b_lon - a_lon
  B_x = cos(b_lat) * cos(delta_lon)
  B_y = cos(b_lat) * sin(delta_lon)
  mid_lat = atan2(
    sin(a_lat) + sin(b_lat),
```

```
    sqrt(((cos(a_lat) + B_x)**2 + B_y**2))
  )
  mid_lon = a_lon + atan2(B_y, cos(a_lat) + B_x)
  # Normalise
  mid_lon = (mid_lon + 3*pi) % (2*pi) - pi
  return Point(latitude=degrees(mid_lat), longitude=degrees(mid_lon))
@udf('double')
def dist(x1,y1, x2,y2, x3,y3): # x3,y3 is the point
        px = x2-x1
        py = y2-y1
        something = px*px + py*py
        if (something == 0):
                something = 0.000000001
        u = ((x3 - x1) * px + (y3 - y1) * py) / float(something)
        if u > 1:
                u = 1
        elif u < 0:
                u = 0
        x = x1 + u * px
        y = y1 + u * py
        dx = x - x3
        dy = y - y3
        # Note: If the actual distance does not matter,
        # if you only want to compare what this function
        # returns to other results of this function, you
        # can just return the squared distance instead
        # (i.e. remove the sqrt) to gain a little performance
        dist = math.sqrt(dx*dx + dy*dy)
        return dist

spark = SparkSession.builder.appName("Find Anomalous Aircrafts").getOrCreate()
spark.sparkContext.setLogLevel("ERROR")
df = spark.read.parquet("/user/jim/adsb/df/*.parquet")
start_time = time.time()
my_window = Window.partitionBy('Icao').orderBy("PosTime")
## convert postime back to unix time for calculation
unix_time = func.unix_timestamp(df.PosTime)
df = df.withColumn('PosTime', unix_time)
df1 = df.withColumn("prev_postime", func.lag(df.PosTime).over(my_window))
df1 = df1.withColumn("diff", func.when(func.isnull(df1.PosTime - df1.prev_postime),
0).otherwise(df1.PosTime - df1.prev_postime))
df1 = df1.withColumn("prev_icao", func.lag(df.Icao).over(my_window))
df1 = df1.withColumn("new_icao", func.when((df1.Icao == df1.prev_icao),
0).otherwise(1))
```

```
timethresh = 3600000
trek_num = ""
icao_g = ""
num = 0
df2 = df1.withColumn("trek", new_trek(df1.diff, df1.Icao))
trek_window = Window.partitionBy('trek'). orderBy('PosTime')
trek_df = df2.groupby('Icao', 'trek').agg(func.max('Alt').\
          alias('max-alt'), func.min('Alt').alias('min-alt'),\
          func.max('Spd').alias('max-spd'),\
          func.min('Spd').alias('min-spd')).sort('Icao')
df3 = trek_df.withColumn('con_icao_trek', func.concat(trek_df.Icao, trek_df.trek))
df4 = df2.select('Icao', 'trek', 'PosTime', 'Lat', 'Long', 'Alt', 'Spd').sort('trek', 'PosTime')
w1 = Window.partitionBy('trek'). orderBy('PosTime')
#this will get the last Postime
w2 = Window.partitionBy("trek").orderBy(func.col('PosTime').desc())
w3 = Window.partitionBy("trek").orderBy(func.col('Alt').desc())
df5 = df4.select("trek",*([first(c).over(w1).alias("first_" + c) for c in df4.columns if c !=
"trek"] +\
          [first(c).over(w2).alias("last_" + c) for c in df4.columns if c != "trek"] +\
          [first(c).over(w3).alias("peak_" + c) for c in df4.columns if c != "trek"]
)).distinct().sort('trek')
df5 = df5.withColumn('duration', (df5.last_PosTime -
df5.first_PosTime)).sort(desc('duration'))
avg_spd = df4.groupby('trek').agg(func.avg(df4.Spd).alias('avg_spd'))
#df6.join(avg_spd ,func.col('df6.trek') == func.col('avg_spd.trek')).show()
df6 = df5.filter(df5.duration > 3600000)
df6 = df6.drop('peak_Icao', 'last_Icao')
df7 = df6.join(avg_spd, ["trek"])
df8 = df4.select("trek", "Lat", "Long", *([first(c).over(w1).alias("first_" + c) for c in
df4.columns if c != "trek"] +\
          [first(c).over(w2).alias("last_" + c) for c in df4.columns if c != "trek"] +\
          [first(c).over(w3).alias("peak_" + c) for c in df4.columns if c != "trek"]
)).sort('trek', 'Postime')
df9 = df8.withColumn('dist', dist(df8.first_Lat, df8.first_Long, df8.last_Lat,
df8.last_Long, df8.Lat, df8.Long))
####average deviation of lat and long
avdev = df9.groupby('trek').agg(func.avg(df9.dist).alias('avdev'))
df10 = df7.join(avdev, ["trek"])
#df6 = df6.distinct().sort('trek').show()

####average deviation of altitude
df10.write.save('/user/jim/adsb/trek_df', format='parquet', mode='append')
#speed_trend_out.show()
elapsed_time = time.time() - start_time
```

```
total_time = round(elapsed_time/60, 2)
print("Time : ", total_time, " minutes")
```

D. FIND ANOMALOUS AIRCRAFT

```
sqlContext.clearCache()
from pyspark.sql import SparkSession
import pyspark.sql.functions as func
from pyspark.sql.functions import udf, avg, desc, col, expr, when, first, last, lit,
year, hour
import time, math, sys
from pyspark.sql.window import Window
from math import sin, cos, atan2, sqrt, degrees, radians, pi
from geopy.distance import great_circle as distance
from geopy.point import Point
from pyspark.sql.types import FloatType, DoubleType

@udf('double')
def sigmoid(X,AV,SD):
  if (SD > 0):
    D = abs(X-AV)/SD
    Y = D/(D+1)
    return Y
  elif (X > AV):
    return 1.0
  else:
    return 0.0

@udf('double')
def ztrans(X,AV,SD):
  if (SD > 0):
    Y = abs(X-AV)/SD
    return Y
  else:
    return 0.0

@udf('int')
def binindex(alt):
        return int((alt//granularity)*granularity+(0.5*granularity))

@udf('int')
def round_lat(lat):
        return round(lat)
```

```
# Use udf to define a round-longitudes udf
@udf('int')
def round_lon(lon):
        return round(lon)

airlinenames = ['airline',' air','air ','airways','express','virgin', \
      'ryanair','easyjet','lufthansa','alitalia', 'qantas', \
      'american eagle','aer lingus','finnair','cityhopper', \
      'tap portugal','eurowings','regional','aeroméxico', \
      'icelandair','egyptair','transavia','cityjet', \
      'connection','tunisair','citilink']
@udf('int')
def airline_operator(opname):
  for airlinename in airlinenames:
    if (opname.find(airlinename) > -1):
      return 1
  return 0
@udf('int')
def non_airline_operator(opname):
  for airlinename in airlinenames:
    if (opname.find(airlinename) > -1):
      return 0
  return 1
@udf('int')
def longcorr(lon):
  return int((lon/360.0)*24)

@udf('float')
def norm1(avg_air_vx, avg_air_vy):
        result = (avg_air_vx * avg_air_vx) + (avg_air_vy * avg_air_vx)
        if (result > 0):
                return math.sqrt(result)
        else:
                return 0.0
####### Use udf to define x velocity ##############
@udf('float')
def vx(spd, trak):
  return (spd*math.cos(math.radians(trak)))
####### Use udf to define y velocity ##############
@udf('float')
def vy(spd, trak):
  return (spd*math.sin(math.radians(trak)))
@udf('float')
```

```
	def log_hour_count(count, total):
		result = (count + 0.5)/total
		if (result > 0):
			return -math.log(result)
		else:
			return 0.0
	# add diff1
	@udf('float')
	def countfrac_non_airline(airline, non_airline):
	  if (airline == 0):
	    return 0.0
	  else:
	    return (airline/(airline + non_airline))
	@udf('float')
	def countfrac_airline(airline, non_airline):
	  if (airline == 0):
	    return 0.0
	  else:
	    return (1-(airline/(airline + non_airline)))
	@udf('float')
	def cosinedist(avg_air_vx, avg_air_vy, vx, vy, norm1, spd):
	  if (spd == 0 or norm1 == 0):
	    return 0.0
	  else:
	    return (1-(abs((avg_air_vx * vx) + (avg_air_vy * vy)) / (norm1 * spd)))
	#udf for mathLog
	@udf('float')
	def math_log(a):
		if (a > 0):
			return math.log(a)
		else:
			return 0.0
	@udf('float')
	def math_sqrt(a):
	  return math.sqrt(a)
	# udf for abs
	@udf('float')
	def ab_s(a):
	  return abs(a)
	@udf('float')
	def zdiff(wc, zdiff1, wv, zdiff2, wa, zdiff3, ws, zdiff4, wo, zdiff5, wt, zdiff6, wn,
zdiff7):
	  return (wc*zdiff1 + wv*zdiff2 + wa*zdiff3 + ws*zdiff4 + wo*zdiff5 +
wt*zdiff6 + wn*zdiff7)
```

```
        spark = SparkSession.builder.appName("Find Anomalous Aircrafts").
getOrCreate()
        spark.sparkContext.setLogLevel("ERROR")
        start_time = time.time()
        df = spark.read.parquet("/user/jim/adsb/df/*.parquet")
        granularity = 100
        wc = 0.2
        wv = 0.2
        wa = 0.2
        ws = 0.2
        wo = 0.2
        wt = 0.2
        wn = 0.2
        wdl = 0.2
        wda = 0.2
        wbd = 0.2
        # add another column as 'binindex'
        df = df.withColumn('binindex', binindex(df.Alt))
        speed_df = spark.read.parquet("/user/jim/adsb/speed_trend_df/*.parquet")
        oplogfreq_df = spark.read.parquet("/user/jim/adsb/oplogfreqs/*.parquet")
        ### load the lat and long bins
        bins_df = spark.read.parquet("/user/jim/adsb/adsb_bins_df/*.parquet")
        time_df = spark.read.parquet("/user/jim/adsb/time_trend_df/*.parquet")
        # add a column diff4 adjusted-speed average difference
        df = df.join(speed_df, ["binindex"])
        df = df.withColumn('diff4', sigmoid(df.Spd, df.avgspeed, df.stdspeed))
        ############### add lat_bin and lon_bin columns ###############
        df = df.withColumn('lat_bin', round_lat(df.Lat)).\
                withColumn('lon_bin', round_lon(df.Long))
        df = df.join(bins_df, ["lat_bin", "lon_bin"])
        ### add zdiff4 column
        df = df.withColumn('zdiff4', ztrans(df.diff4,lit(0.358),lit(0.217)))
        ## add airline
        df = df.withColumn('airlines', airline_operator(df.Op)).withColumn('non_airlines',
non_airline_operator(df.Op))
        ### add diff3 column
        diff3 = when(df.airlines == 1, sigmoid(df.Alt,df.avg_airline_alt, lit(10000
))).otherwise(sigmoid(df.Alt, df.avg_non_airline_alt, lit(10000)))
        df = df.withColumn('diff3', diff3)
        ### add zdiff4 column
        df = df.withColumn('zdiff3', ztrans(df.diff4,lit(0.364),lit(0.195)))
        diff1 = when(df.airlines == 1, countfrac_airline(df.airline_count,
df.non_airline_count)).otherwise(countfrac_non_airline(df.airline_count,
df.non_airline_count))
```

```
        df = df.withColumn('diff1', diff1)
        df = df.join(oplogfreq_df, "Op")
        c = bins_df.groupBy().sum().rdd.map(lambda x: x[2]).collect()
        totalcountc = int(c[0])
        n = bins_df.groupBy().sum().rdd.map(lambda x: x[3]).collect()
        totalcountn = int(n[0])
        num_nunempty_bins = bins_df.where((bins_df.airline_count) +
(bins_df.non_airline_count) > 0 ).count()
        log_total_count_commercial = math.log(totalcountc/num_nunempty_bins)
        log_total_count_noncommercial = math.log(totalcountn/num_nunempty_bins)
        diff7 = when(df.airlines == 1, sigmoid(lit(log_total_count_commercial),
math_log(1+df.airline_count), lit(1))).\
          otherwise(sigmoid(lit(log_total_count_noncommercial),
math_log(1+df.non_airline_count), lit(1)))
        df = df.withColumn('diff7', diff7)
        diff5 = when(df.airlines == 1, sigmoid(df.oplogfreq, df.avg_airline_oplogfreqs,
lit(1))).\
          otherwise(sigmoid(df.oplogfreq, df.avg_non_airline_oplogfreqs, lit(1)))
        df = df.withColumn('diff5', diff5)
        time_c = time_df.groupBy().sum().rdd.map(lambda x: x[1]).collect()
        total_time_count = float(time_c[0])
        time_df = time_df.withColumn('log_hour_count',
log_hour_count(time_df.hour_lon_count, lit(total_time_count)))
        maxhourlog = time_df.groupBy().max().rdd.map(lambda x: x[2]).collect()
        maxhourlog = float(maxhourlog[0])
        diff6 = sigmoid(log_hour_count(time_df.hour_lon_count, lit(total_time_count)),
lit(maxhourlog), lit(3.0))
        time_df = time_df.withColumn('diff6', diff6)
        df = df.withColumn('norm1', norm1(df.avg_airline_vx, df.avg_airline_vy))
        # add vx and vy
        ############## add airline_vx non_airline_vx airline_vy non_airline_vy
columns ############################
        airline_vx = when(df.airlines == 1, vx(df.Spd, df.Trak)).otherwise(0.0)
        airline_vy = when(df.airlines == 1, vy(df.Spd, df.Trak)).otherwise(0.0)
        df = df.withColumn('airline_vx', airline_vx).withColumn('airline_vy', airline_vy)
        # add diff2
        diff2 = when(((df.norm1 > 0.0 ) & (df.Spd > 0.0)), cosinedist(df.avg_airline_vx,\
          df.avg_airline_vy, df.airline_vx, df.airline_vy, df.norm1, df.Spd)).otherwise(0.0)
        df = df.withColumn('diff2', diff2)
        # add hour column to datafram then join time_df
        df = df.withColumn("Hour", hour(df.PosTime))
        df = df.join(time_df, ["Hour"])
        df = df.withColumn('zdiff1', ztrans(df.diff1,lit(0.347),lit(0.217)))
        df = df.withColumn('zdiff2', ztrans(df.diff2,lit(0.238),lit(0.285)))
```

```
        df = df.withColumn('zdiff5', ztrans(df.diff5,lit(0.522),lit(0.215)))
        df = df.withColumn('zdiff6', ztrans(df.diff6,lit(0.203),lit(0.093)))
        df = df.withColumn('zdiff7', ztrans(df.diff7,lit(0.589),lit(0.178)))
        df = df.withColumn('zdiff', zdiff(lit(wc), df.zdiff1, lit(wv), df.zdiff2,\
         lit(wa), df.zdiff3, lit(ws), df.zdiff4, lit(wo), df.zdiff5, lit(wt), df.zdiff6, lit(wn),
df.zdiff7))
        df = df.select('Icao', 'Alt', 'Lat', 'Long', 'Spd', 'Trak', 'PosTime', 'Op', 'Cou',
func.round('zdiff',3).alias('zdiff'))
        #only get the data above the 2.5 threshold
        df = df.filter("zdiff > 2.5")
        df.write.save('/user/jim/adsb/anomalous_df', format='parquet', mode='append')
        elapsed_time = time.time() - start_time
        total_time = round(elapsed_time/60, 2)
        print("Time : ", total_time, " minutes")
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Achari, S. (2015). *Hadoop essentials*. Birmingham, UK: Packt Publishing.

ADS-B Exchange. (n.d.-a). Accessing data collected by ADS-B Exchange. Retrieved May 23, 2018, from https://www.adsbexchange.com/data/#

ADS-B Exchange. (n.d.-b). Data field descriptions (JSON and database). Retrieved May 23, 2018, from https://adsbexchange.com/datafields/

Apache. (n.d.-a). User defined function – Python. Retrieved March 22, 2018, from https://docs.databricks.com/spark/latest/spark-sql/udf-in-python.html

Apache. (n.d.-b). Apache Parquet. Retrieved from http://parquet.apache.org/.

Chen, J. (2016, January 14). 5 Reasons to choose Parquet for Spark SQL. Retrieved from https://developer.ibm.com/hadoop/2016/01/14/5-reasons-to-choose-parquet-for-spark-sql/

Cloudera. (n.d.). Spark Installation. Retrieved June 4, 2018, from https://www.cloudera.com/documentation/enterprise/5-4-x/topics/cdh_ig_spark_installation.html

Das, A., & Annamalai, M. (2015, October). Naval Postgraduate School report on Oracle big data connectors and Oracle Database 12c. Paper presented at Oracle Openworld 2015, San Francisco, CA.

Das, A., Plunkett, T. (2015). Naval Postgraduate School Report on the Oracle Cloudera Solution for Big Data. Paper presented at BIWA Summit 20115, Redwood Shores, CA.

Defense Science Board. (1996). *Report of the Defense Science Board Task Force on Combat Identification*. Washington, DC: Office of the Under Secretary of Defense for Acquisition & Technology.

Federal Aviation Administration. (2010, May 28). Automatic Dependent Surveillance—Broadcast (ADS-B) out performance requirements to support Air Traffic Control (ATC) service. 14 CFR § 91 (2010). Retrieved from https://www.gpo.gov/fdsys/pkg/FR-2010-05-28/pdf/2010-12645.pdf

Ganelin, I., Orhian, E., Sasaki, K. & York, B. (2016). *Spark: Big data cluster computing in production*. Indianapolis, IN: Wiley.

Ghemawat, S., Gobioff, H., & Leung, S. (2003). The Google file system. *ACM SIGOPS Operating Systems Review*, *37*(5), 29. doi:10.1145/1165389.945450

Gebara, F., Hofstee, H., & Nowka, K. (2015). Second-generation big data systems. *Computer*, *48*(1), 36-41. doi:10.1109/MC.2015.25

Gollapudi, S. (2016). Practical machine learning. Birmingham, UK: Packt Publishing.

Hinson, J., Summitt, B., & Shepko, M. (2009, May). Combat identification server: Blue force tracker in the cockpit. *Combat Identification Newsletter*, (09-31), 11–16. Retrieved from https://www.globalsecurity.org/military/library/report/call/call_09-31.pdf

Ismail, H. (2017, January 6). Learning data science: Day 9 – Linear regression on Boston Housing dataset. Retrieved from https://medium.com/@haydar_ai/learning-data-science-day-9-linear-regression-on-boston-housing-dataset-cd62a80775ef

Joint Chiefs of Staff. (1998, February). *Joint warfighter science & technology plan*. Washington, DC: Department of Defense.

Ministry of Defence [United Kingdom]. (2002). Combat identification. London, UK: The Stationery Office. Retrieved from https://www.nao.org.uk/wp-content/uploads/2002/03/0102661.pdf

Murthy, A. (2012, August 15). Apache Hadoop YARN – concepts and applications. Retrieved from https://hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/

Nickels, R. (2014). Virtual radar from a digital TV dongle. Retrieved from http://www.arrl.org/files/file/QST/This%20Month%20in%20QST/January%202014/VirtualRadarJan2013QST.pdf

Rowe, N.C., Das, A., Zhou, J. (2018). Distributed combat identification of interesting aircraft. Unpublished paper.

Salcido, R., Kendall, A., Zhao, Y., (2017). *Analysis of Automatic Dependent Surveillance-Broadcast data* (AAAI Technical Report FS-17-03). Retrieved from https://aaai.org/ocs/index.php/FSS/FSS17/paper/viewFile/15996/15318

Shen, Y., Li, Y., Wu, L., Liu, S., & Wen, Q. (2016). Big data overview. In I. Management Association (Ed.), *Big data: Concepts, methodologies, tools, and applications* (pp. 1–29). Hershey, PA: IGI Global. doi:10.4018/978-1-4666-9840-6.ch001

Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The Hadoop distributed file system. *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies* (pp. 1–10). doi:10.1109/MSST.2010.5496972

Sun, J., Ellerbroek, J., & Hoekstra, J. (2016). Large-scale flight phase identification from ADS-B data using machine learning methods. Paper presented at the 7th International Conference on Research in Air Transportation.

Tabassum, A., Allen, N., & Semke, W. (2017, September). ADS-B message contents evaluation and breakdown of anomalies. Paper presented at the Digital Avionics Systems Conference (DASC), 2017 IEEE/AIAA 36th, St. Petersburg, FL, USA.

United States General Accounting Office (2001). *Combat identification systems: Strengthened management efforts needed to ensure required capabilities*. Retrieved from https://www.gao.gov/assets/240/231698.pdf

U.S. Government Accountability Office (1995). *Combat identification systems: Changes needed in management plans and structure* (GAO-NSIAD-95-153). Retrieved from https://www.gao.gov/products/155084

Van Gosliga, S., & Jansen, H. (2003). A Bayesian network for combat identification. *Proc. NATO RTO IST Symposium on Military Data and Information Fusion*, Prague, CZ, October, NATO RTO-MP-IST-040

Witten, L., Frank, E. & Hall, L. (2010). *Data mining practical machine learning tools and techniques*. Burlington, MA: Morgan Kaufmann.

Xin, R., Armbrust, M. & Liu, D. (2015, February 17). Introducing DataFrames in Apache Spark for large scale data science [Blog post]. Retrieved from https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California