**DSC650 Final Project: Titanic Data Pipeline – Nick Blackford**

# Introduction

This report walks through the development of a distributed data pipeline leveraging four core technologies from the Hadoop ecosystem: Apache NiFi, HDFS, Apache Hive, and Apache Spark. The goal was to showcase how these tools integrate in a real-world scenario to move and transform data effectively. Starting with ingestion and ending in structured query access, this project was designed to reflect practical skills in managing the lifecycle of data across a modern big data architecture.

# Dataset

The dataset used is the Titanic passenger manifest, a well-known dataset rich with fields like age, fare, class, gender, and survival status. It was sourced from a public GitHub repo and not one of the examples provided in course materials. That ensured compliance with the uniqueness requirement. The structure made it ideal for pipeline work: it's clean, it covers multiple data types, and it’s small enough to manage in a constrained environment without losing the need for transformation.  
  
Dataset URL: https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv

# Pipeline Overview

To build this pipeline, I used the following sequence of technologies, each with a defined role:

1. NiFi pulled the Titanic CSV from a public URL using the InvokeHTTP processor.
2. The file was written to HDFS using PutHDFS, ending up in /user/root/nifi\_input/.
3. I used PySpark to read that raw file directly from HDFS.
4. Two transformations were applied: a new survived\_flag column and an age\_group classification.
5. Finally, the transformed DataFrame was written to Hive as default.titanic\_cleaned.

Each step was executed inside a containerized environment on a VPN-accessed VM with --master yarn, using my core-site.xml config path to properly link Spark with HDFS and Hive.

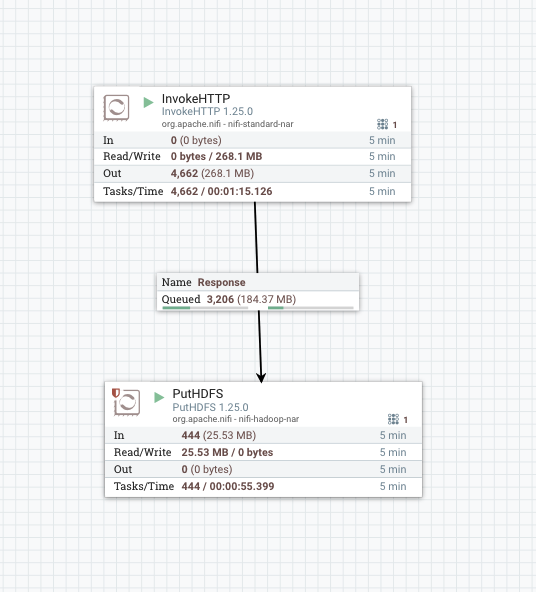
# Issues Encountered and Resolved

This pipeline hit a couple of real-world snags. The biggest issue was NiFi writing 2,800+ duplicate files into HDFS due to how the flow was configured. This completely choked my PySpark session every time I tried to launch it. I ended up pausing the cleanup after deleting a few hundred files, which reduced enough load on the NameNode to stabilize the system. It was a good reminder that file count—not just file size—matters in HDFS performance.

Another minor issue was that NiFi named the files using UUIDs, which made automation downstream more tedious. I resolved it by renaming the correct file to titanic.csv after confirming its contents, keeping the rest of the pipeline simpler.

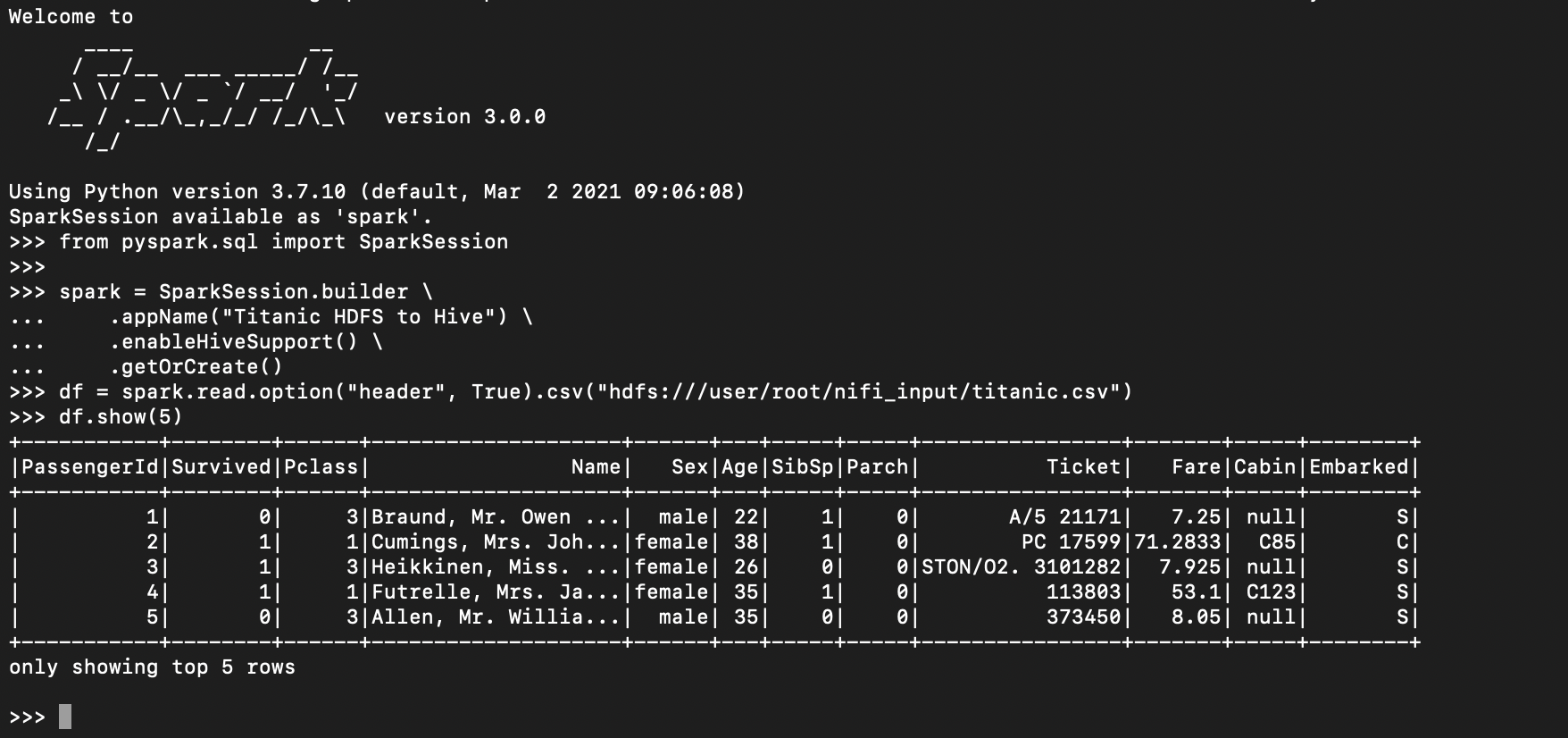
# Screenshots

## Nifi Flow

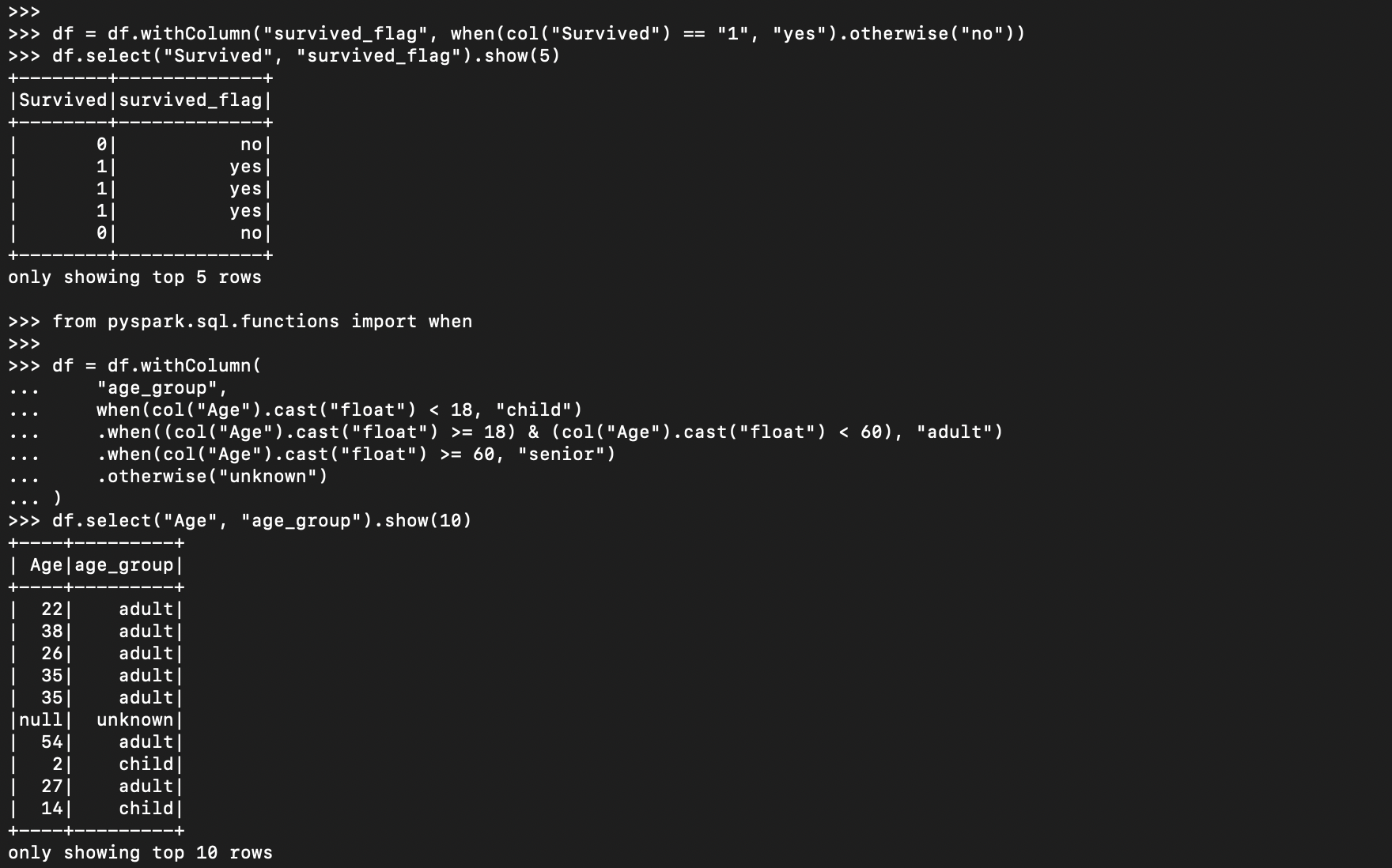


## HDFS File

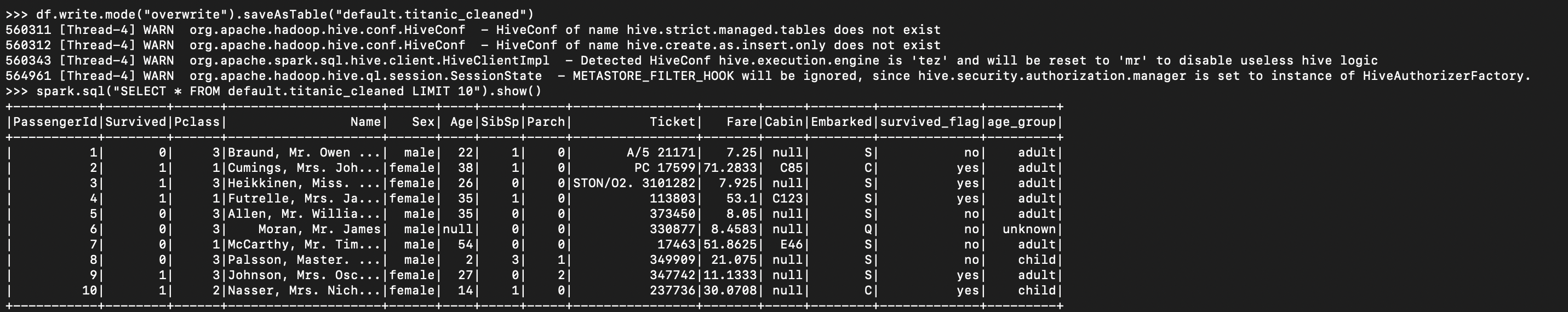
## Spark Read from HDFS



## Spark Transformations



## Hive Query



# Code

# Launch PySpark

pyspark --master yarn --conf spark.sql.catalogImplementation=hive

# Create Spark session

from pyspark.sql import SparkSession

from pyspark.sql.functions import when, col

spark = SparkSession.builder \

.appName("Titanic HDFS to Hive") \

.enableHiveSupport() \

.getOrCreate()

# Read data from HDFS

df = spark.read.option("header", True).csv("hdfs:///user/root/nifi\_input/titanic.csv")

df.show(5)

# Transformation 1: Binary to label

df = df.withColumn("survived\_flag", when(col("Survived") == "1", "yes").otherwise("no"))

# Transformation 2: Age grouping

df = df.withColumn(

"age\_group",

when(col("Age").cast("float") < 18, "child")

.when((col("Age").cast("float") >= 18) & (col("Age").cast("float") < 60), "adult")

.when(col("Age").cast("float") >= 60, "senior")

.otherwise("unknown")

)

# Write to Hive

df.write.mode("overwrite").saveAsTable("default.titanic\_cleaned")

# Verify

spark.sql("SELECT \* FROM default.titanic\_cleaned LIMIT 10").show()

# Stop session

spark.stop()

# Conclusion

This project put all the major components of the Hadoop ecosystem to work in a tight, purpose-driven pipeline. NiFi handled ingestion cleanly, HDFS provided durable storage, PySpark made transformations intuitive, and Hive gave us SQL-level access to structured output. The technical hurdles around file count and configuration helped me better understand HDFS’s operational behavior. Ultimately, this project reflected not only technical implementation but also realistic debugging and system understanding—skills that matter in a production setting.