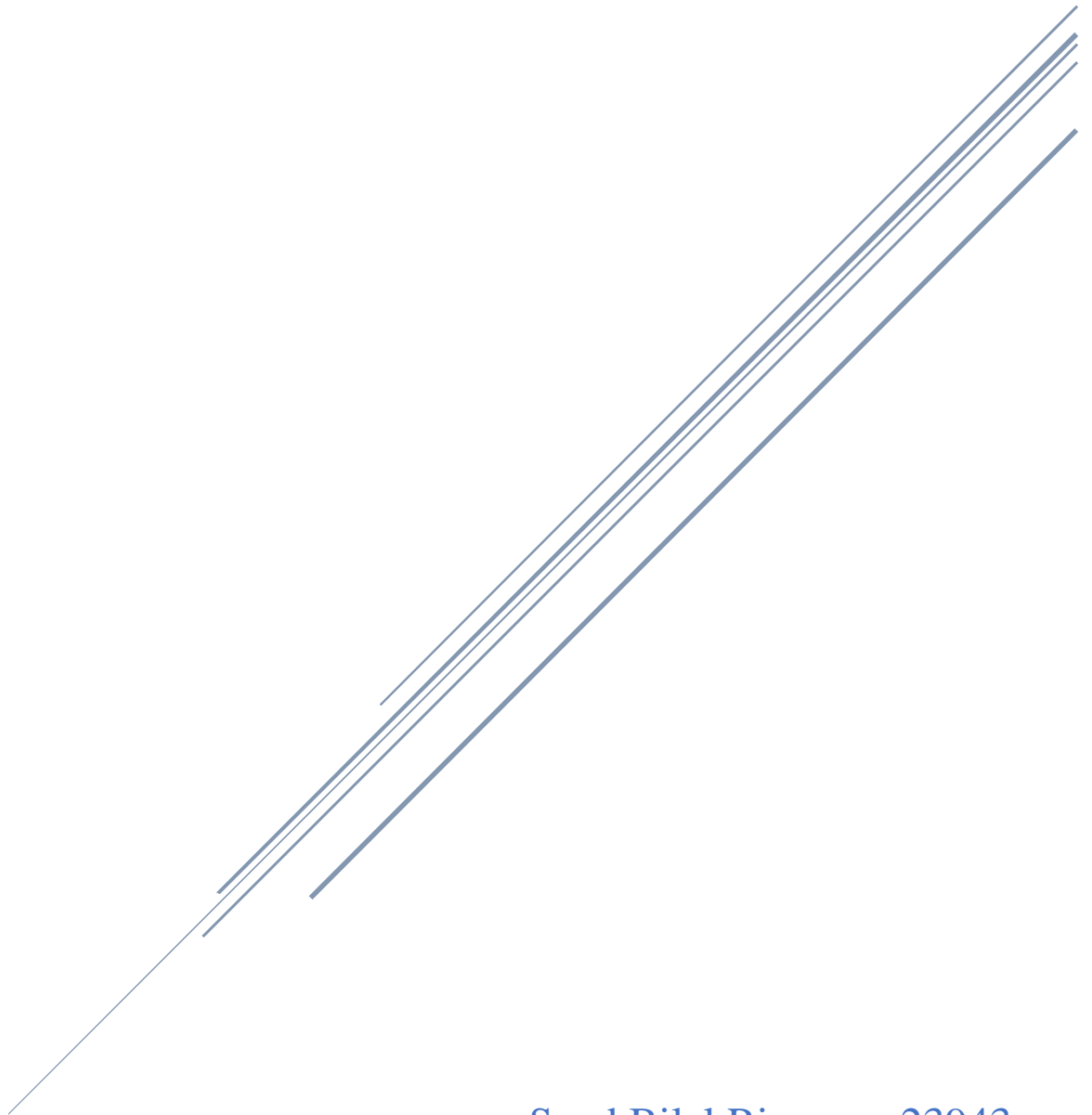


REAL-TIME SPARK PROCESSING (LAMBDA ARCHITECTURE)

Big Data Analytics – Final Project



Syed Bilal Rizwan - 23943

Table of Contents

1. Introduction	2
Dataset:	2
2. Project Architecture (Flow-Chart).....	3
3. Step-by-step Code Walkthrough	4
3.1 Setting up Environment Variables:	4
3.2 Setting up Docker Containers:	4
3.3 Creating Data Feed and Streaming:	6
3.4 PostgreSQL:	8
3.5 Real-Time Streaming Pipeline:.....	9
3.6 Dashboard (Front-end using Django):	14
3.7 Demo Walkthrough Link:	15
4. Challenges and Conclusion	16

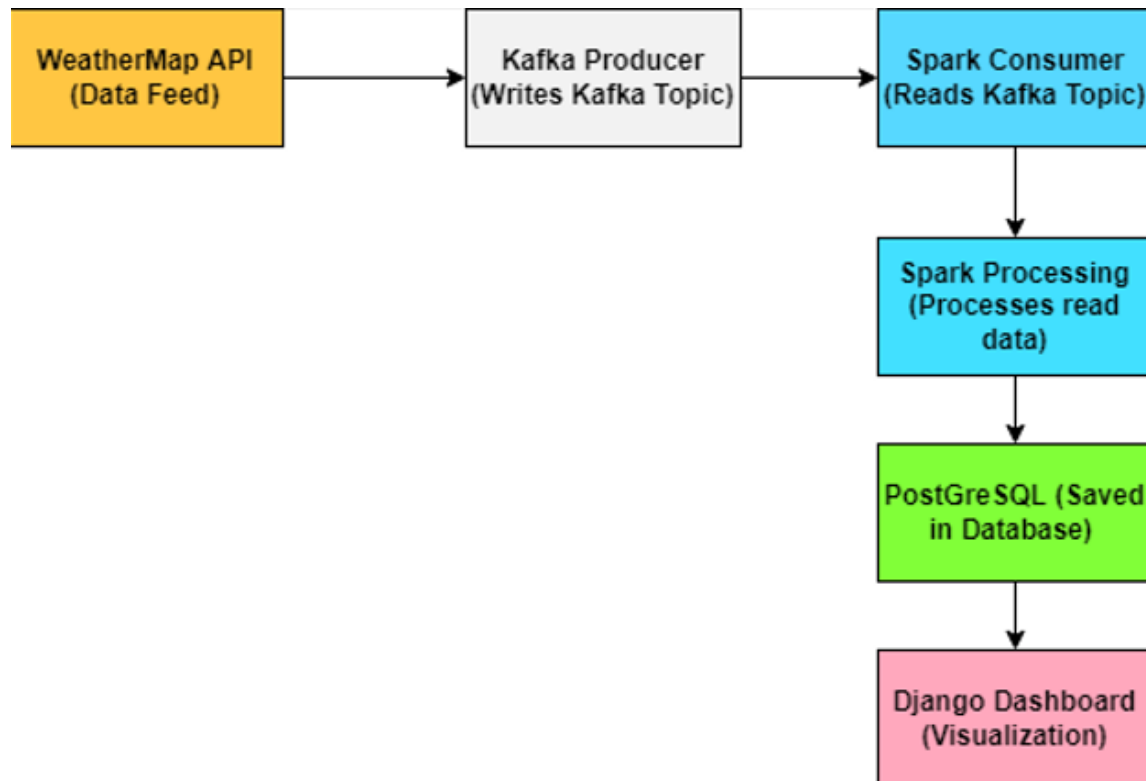
1. Introduction

Weather monitoring predates back to 1849 where weather maps were plotted based on telegraphic reports. Since then, with the advancements in technology, it is now possible to measure within seconds important metrics of weather including but not limited to Temperature, Humidity etc. Furthermore, it has become even more important to accurately monitor and forecast weather as it is now used to predict future changes in our climate. With the advancement in IOT, a person sitting at home can keep a temperature sensor to measure temperature and hygrometer to measure humidity each second creating a high velocity data stream. Similarly, several weather APIs are now available which create data streams. However, it has become really challenging to manage streaming data with high velocity and volume. It has become quite important to deal with streaming data efficiently and gain insights from it.

Dataset:

For this project, OpenWeatherMap API was used to get temperature and humidity of 3 different cities of Pakistan. The API was hit after every 5 seconds to create a stream.

2. Project Architecture (Flow-Chart)



3. Step-by-step Code Walkthrough

This project was created to create a streaming pipeline analyzed by Spark and show its results on a dashboard. First, data feed was created which was used by Kafka producer to create a stream. Stream was consumed by spark and written into a temporary data frame. Data frame was analyzed, and processed data is stored into POSTGRESQL. A Django based dashboard is used to read processed data and show into an interactive dashboard.

3.1 Setting up Environment Variables:

- Hadoop 3.2.2 was used and added in environment variables as:

User variables for bilal	
Variable	Value
HADOOP_HOME	C:\hadoop

- WinUtils directory was also added in environment variables using the same name as Hadoop_home:

HADOOP_HOME	C:\Winutils\hadoop-3.2.2
-------------	--------------------------

- Several versions of Java were tried and finally Java 11.0.15 was installed and setup in environment variables:

JAVA_HOME	C:\Program Files\Java\jdk-11.0.15
-----------	-----------------------------------

- Spark was also installed with the name of Spark 3.2.1 bin Hadoop 3.2.2 and a Spark_home env variable was created:

SPARK_HOME	C:\Program Files\spark-3.2.1-bin-hadoop3.2
------------	--

- Lastly, all these variables were added to path system variable as shown below:

%JAVA_HOME%\bin
C:\Program Files\Docker\Docker\resources\bin
C:\ProgramData\DockerDesktop\version-bin
%SCALA_HOME%\bin
%SPARK_HOME%\bin
%HADOOP_HOME%\bin
%HADOOP_HOME%\sbin

3.2 Setting up Docker Containers:

The container architecture was replicated by watching tutorials from DataMaking, their examples helped a lot to setup the entire thing. Their script was used to build images for containers and run them in a network

At first a docker network was created using the following command:

```
1) docker network create --subnet=172.20.0.0/16 datamakingnet
```

Then, a zookeeper container was created by simultaneously running the following commands:

```
2) docker pull zookeeper:3.4
3) docker run -d --hostname zookeepernode --net datamakingnet --ip
   172.20.1.3 --name datamaking_zookeeper --publish 2181:2181
   zookeeper:3.4
```

Similarly, docker kafka container was created using the following commands:

```
4) docker pull ches/kafka
5) docker run -d --hostname kafkanode --net datamakingnet --ip
   172.20.1.4 --name datamaking_kafka --publish 9092:9092 --publish
   7203:7203 --env KAFKA_ADVERTISED_HOST_NAME=172.26.48.1 --env
   ZOOKEEPER_IP=172.26.48.1 ches/kafka
```

It is important to note that the hostname and IP of zookeeper must be your own WSL IP address of your machine.

Then onwards, two shell scripts were run using the following command by going inside the directory:

```
6) ./1_create_hadoop_spark_image.sh
```

. The following shell commands were run to create images first:

```
#!/bin/bash

# generate ssh key
echo "Y" | ssh-keygen -t rsa -P "" -f configs/id_rsa

# Building Hadoop Docker Image
docker build -f ./datamaking_hadoop/Dockerfile . -t
hadoop_spark_cluster:datamaking_hadoop

# Building Spark Docker Image
docker build -f ./datamaking_spark/Dockerfile . -t
hadoop_spark_cluster:datamaking_spark

# Building PostgreSQL Docker Image for Hive Metastore Server
docker build -f ./datamaking_postgresql/Dockerfile . -t
hadoop_spark_cluster:datamaking_postgresql

# Building Hive Docker Image
docker build -f ./datamaking_hive/Dockerfile . -t
hadoop_spark_cluster:datamaking_hive
```





These commands created images for each Hadoop, Spark, PostgreSQL, and Hive. The commands can be found in 1_create_hadoop_spark.sh file. It must be run in Linux, so I used Git Bash, to run

them by running `./1_create_hadoop_spark_image.sh` command. After command was run, it ran the executable shell script file with contents as above and created images.

The images were used to create containers by using the 2nd shell script file named `2_create_hadoop_spark_cluster.sh`. This file created all containers needed for the project. The file is quite lengthy so it can be viewed from project files.

```
7) ./2_create_hadoop_spark_cluster.sh create
```

The following containers were created:

	datamaking_kafka <small>ches/kafka</small>
	<small>RUNNING PORT: 7203</small>
	masternode <small>hadoop_spark_c...</small>
	<small>RUNNING PORT: 10000</small>
	postgresqlnode <small>hadoop_spark_c...</small>
	<small>RUNNING PORT: 5432</small>
	datamaking_zookeeper <small>zookeeper:3.4</small>
	<small>RUNNING PORT: 2181</small>

Now, once containers are created and running, they are used to run python scripts created below which contains actual data feed and PySpark code.

3.3 Creating Data Feed and Streaming:

Since OpenWeatherMap API's sensor only reads data every hour, it did not qualify as a source for high-speed stream. To combat that, after API was hit every 5 seconds, received data was slightly manipulated randomly to change values. A script was created named `weatherdata_producer.py` which called weather API every 5 seconds and retrieved data. Data was written into Kafka Topic using a Kafka Producer. This created a stream of data feed. The code is shown below. At first libraries were imported:

```
import time
import json
from kafka import KafkaProducer
import requests
import random
```

Then, container details were provided, and Kafka Producer was initialized, it is important to note that the IP address is my localhost one with ports defined when containers were created:

```

BOOTSTRAP_SERVERS_CONS = '172.26.48.1:9092'
kafka_topic_name = 'sampletopic1'

json_message = None
json_message1, json_message2, json_message3 = None, None, None
city_name = None
temperature = None
humidity = None
openweathermap_api_endpoint = None
appid = None

if __name__ == "__main__":
    print("Weather Data | Kafka Producer Application Started ... ")

    producer = None
    producer = KafkaProducer(bootstrap_servers=BOOTSTRAP_SERVERS_CONS,
                             value_serializer=lambda x:
    json.dumps(x).encode('utf-8'), api_version=(0,10,0,1))

```

Then, a while loop was used to create streaming data and writing it into Kafka Topic, only “Karachi” code is shown as below:

```

while True:
    city_name = "Karachi"
    appid = '7e8331a91d3f6b9908ee2e589dc19acf'
    openweathermap_api_endpoint =
"http://api.openweathermap.org/data/2.5/weather?appid=" + appid + "&q=" +
city_name
    api_response = requests.get(openweathermap_api_endpoint)
    json_data1 = api_response.json()
    city_name = json_data1['name']
    humidity = json_data1['main']['humidity'] + random.randint(1,2)
    temperature = json_data1['main']['temp'] + random.randint(1,5)
    json_message1 = {"CityName": city_name,
                     "Temperature": temperature,
                     "Humidity": humidity,
                     'CreationTime': time.strftime("%Y-%m-%d %H:%M:%S")}
    producer.send(kafka_topic_name, json_message1)
    print("Published message 1: " + json.dumps(json_message1))

```

The file must be run only after Kafka container is running as it will only publish data if there is a kafka broker available. The code run can be seen and tested by using PyCharm terminal and running the code directly:


```

Weather Data | Kafka Producer Application Started ...
Published message 1: {"CityName": "Karachi", "Temperature": 304.05, "Humidity": 80, "CreationTime": "2022-06-04 01:37:49"}
Published message 2: {"CityName": "Lahore", "Temperature": 307.14, "Humidity": 34, "CreationTime": "2022-06-04 01:37:49"}
Published message 3: {"CityName": "Islamabad", "Temperature": 303.5, "Humidity": 28, "CreationTime": "2022-06-04 01:37:50"}
Published message 1: {"CityName": "Karachi", "Temperature": 302.05, "Humidity": 80, "CreationTime": "2022-06-04 01:37:55"}
Published message 2: {"CityName": "Lahore", "Temperature": 306.14, "Humidity": 34, "CreationTime": "2022-06-04 01:37:55"}
Published message 3: {"CityName": "Islamabad", "Temperature": 307.5, "Humidity": 28, "CreationTime": "2022-06-04 01:37:55"}
Published message 1: {"CityName": "Karachi", "Temperature": 304.05, "Humidity": 79, "CreationTime": "2022-06-04 01:38:01"}

```

As can be seen, the data is publishing on Kafka Topic.

3.4 PostgreSQL:

Data stored in PostgreSQL table can be accessed by first bashing into the PostgreSQL container as shown below:

```

1) docker exec -it 05f80dbc073e bash
2) psql -U postgres
3) CREATE USER demouser WITH PASSWORD 'demouser';
4) ALTER USER demouser WITH SUPERUSER;
5) CREATE DATABASE weather_db;
6) GRANT ALL PRIVILEGES ON DATABASE weather_db TO demouser;
7) \c weather_db;

```

```

C:\Users\bilal>docker exec -it 05f80dbc073e bash
root@postgresnode:/# psql -U postgres
psql (14.3 (Debian 14.3-1.pgdg110+1))
Type "help" for help.

postgres=# \d
Did not find any relations.
postgres=# \c weather_db
You are now connected to database "weather_db" as user "postgres".
weather_db=# \d

```

Schema	Name	Type	Owner
public	dashboard_weather_detail_agg_tbl	table	demouser
public	dashboard_weather_detail_agg_tbl_id_seq	sequence	demouser
public	django_migrations	table	demouser
public	django_migrations_id_seq	sequence	demouser

```

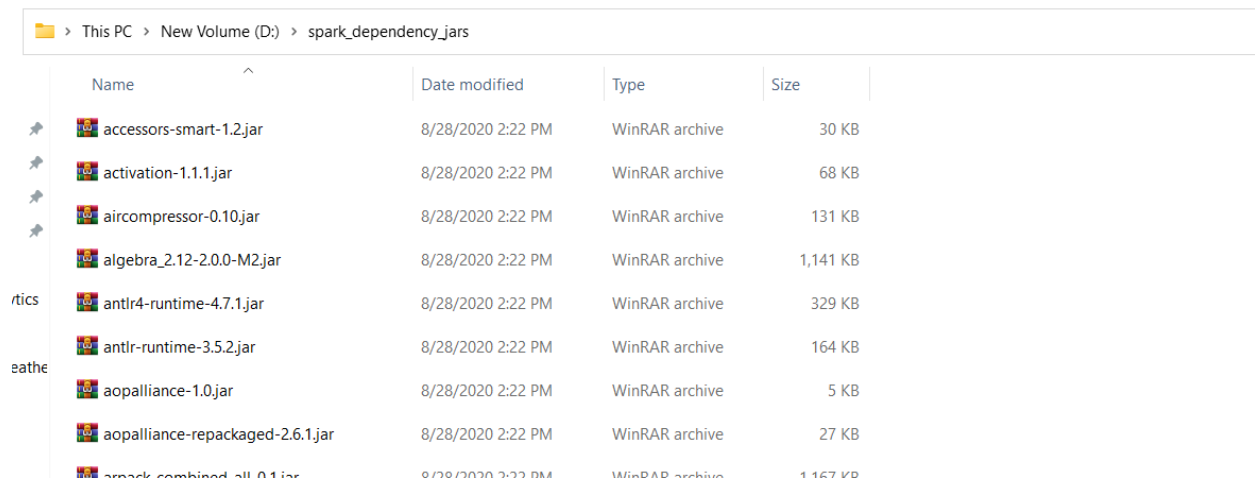
(4 rows)

```

This will create a DB instance and user privileges.

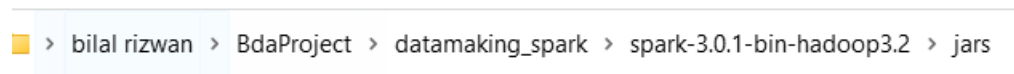
3.5 Real-Time Streaming Pipeline:

Once a datafeed is created using weatherapi and Kafka, the code stream had to be processed using Spark and a separate script named `real_time_streaming_data_pipeline.py` was created for this purpose. The script read streaming data from Kafka into a data frame using Spark. The data frame is manipulated using Spark to calculate average temperature and humidity within that batch stream, then the processed data was pushed into PostgreSQL. As a pre-requisite, a directory must be created as follows:



Name	Date modified	Type	Size
accessors-smart-1.2.jar	8/28/2020 2:22 PM	WinRAR archive	30 KB
activation-1.1.1.jar	8/28/2020 2:22 PM	WinRAR archive	68 KB
aircompressor-0.10.jar	8/28/2020 2:22 PM	WinRAR archive	131 KB
algebra_2.12-2.0.0-M2.jar	8/28/2020 2:22 PM	WinRAR archive	1,141 KB
antlr4-runtime-4.7.1.jar	8/28/2020 2:22 PM	WinRAR archive	329 KB
antlr-runtime-3.5.2.jar	8/28/2020 2:22 PM	WinRAR archive	164 KB
aopalliance-1.0.jar	8/28/2020 2:22 PM	WinRAR archive	5 KB
aopalliance-repackaged-2.6.1.jar	8/28/2020 2:22 PM	WinRAR archive	27 KB
spark-combined-all-0.1.jar	8/28/2020 2:22 PM	WinRAR archive	1,167 KB

Where, all spark jars must be extracted which are in the following directory:



>	bilal rizwan	>	BdaProject	>	datamaking_spark	>	spark-3.0.1-bin-hadoop3.2	>	jars
---	--------------	---	------------	---	------------------	---	---------------------------	---	------

And additionally, 5 more spark jars must be downloaded and kept in this directory before running the pipeline script, the additional jar files are named below:

- commons-pool2-2.8.1.jar
- postgresql-42.2.16.jar
- spark-sql-kafka-0-10_2.12-3.0.1.jar
- kafka-clients-2.6.0.jar
- spark-streaming-kafka-0-10-assembly_2.12-3.0.1.jar

These jars are necessary to run the script. At the first step, the libraries are imported:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
import pyspark.sql.functions as fn
```

Then, kafka topic and server details along with postgres configuration is added:

```
# Kafka Broker/Cluster Details
KAFKA_TOPIC_NAME_CONS = "sampletopic1"
KAFKA_BOOTSTRAP_SERVERS_CONS = '172.26.48.1:9092'

# PostgreSQL Database Server Details
postgres_host_name = "172.26.48.1"
```

```

postgresql_port_no = "5432"
postgresql_user_name = "demouser"
postgresql_password = "demouser"
postgresql_database_name = "weather_db"
postgresql_driver = "org.postgresql.Driver"

```

Then onwards, database properties were created:

```

db_properties = {}
db_properties['user'] = postgresql_user_name
db_properties['password'] = postgresql_password
db_properties['driver'] = postgresql_driver

```

Then a function was created which saves data into postgres table:

```

def save_to_postgresql_table(current_df, epoc_id, postgresql_table_name):
    print("Inside save_to_postgresql_table function")
    print("Printing epoc_id: ")
    print(epoc_id)
    print("Printing postgresql_table_name: " + postgresql_table_name)

    postgresql_jdbc_url = "jdbc:postgresql://" + postgresql_host_name + ":" +
    str(postgresql_port_no) + "/" + postgresql_database_name

    #Save the dataframe to the table.
    current_df.write.jdbc(url = postgresql_jdbc_url,
                          table = postgresql_table_name,
                          mode = 'append',
                          properties = db_properties)

    print("Exit out of save_to_postgresql_table function")

```

Then onwards, a spark session was created with jar files configurations as shown below:

```

spark = SparkSession \
    .builder \
    .appName("Real-Time Streaming Data Pipeline") \
    .master("local[*]") \
    .config("spark.jars", "file:///D://spark_dependency_jars//commons-pool2-2.8.1.jar,file:///D://spark_dependency_jars//postgresql-42.2.16.jar,file:///D://spark_dependency_jars//spark-sql-kafka-0-10_2.12-3.0.1.jar,file:///D://spark_dependency_jars//kafka-clients-2.6.0.jar,file:///D://spark_dependency_jars//spark-streaming-kafka-0-10-assembly_2.12-3.0.1.jar") \
    .config("spark.executor.extraClassPath", "file:///D://spark_dependency_jars//commons-pool2-2.8.1.jar;file:///D://spark_dependency_jars//postgresql-42.2.16.jar;file:///D://spark_dependency_jars//spark-sql-kafka-0-10_2.12-3.0.1.jar;file:///D://spark_dependency_jars//kafka-clients-2.6.0.jar;file:///D://spark_dependency_jars//spark-streaming-kafka-0-10-assembly_2.12-3.0.1.jar") \
    .config("spark.executor.extraLibrary", "file:///D://spark_dependency_jars//commons-pool2-2.8.1.jar;file:///D://spark_dependency_jars//postgresql-

```

```
42.2.16.jar:file:///D://spark_dependency_jars//spark-sql-kafka-0-10_2.12-3.0.1.jar:file:///D://spark_dependency_jars//kafka-clients-2.6.0.jar:file:///D://spark_dependency_jars//spark-streaming-kafka-0-10-assembly_2.12-3.0.1.jar") \
    .config("spark.driver.extraClassPath",
"file:///D://spark_dependency_jars//commons-pool2-2.8.1.jar:file:///D://spark_dependency_jars//postgresql-42.2.16.jar:file:///D://spark_dependency_jars//spark-sql-kafka-0-10_2.12-3.0.1.jar:file:///D://spark_dependency_jars//kafka-clients-2.6.0.jar:file:///D://spark_dependency_jars//spark-streaming-kafka-0-10-assembly_2.12-3.0.1.jar") \
    .getOrCreate()
```

After, a sparksession was created, the data was read from kafka topic and saved into database using spark's readStream:

```
weather_detail_df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", KAFKA_BOOTSTRAP_SERVERS_CONS) \
    .option("subscribe", KAFKA_TOPIC_NAME_CONS) \
    .option("startingOffsets", "latest") \
    .load()

print("Printing Schema of weather_detail_df: ")
weather_detail_df.printSchema()
```

The dataframe has the following schema:

```
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

To convert this schema into a structured one, the following code was written:

```
# Schema Code Block Starts Here

weather_details_schema = StructType([
    StructField("event_id", StringType()),
    StructField("CityName", StringType()),
    StructField("Temperature", DoubleType()),
    StructField("Humidity", IntegerType()),
    StructField("CreationTime", StringType())
])

# Schema Code Block Ends Here

weather_detail_df_1 = weather_detail_df.selectExpr("CAST(value AS STRING)")
weather_detail_df_2 = weather_detail_df_1.select(from_json(col("value"),
```

```
weather_details_schema).alias("weather_message_detail"))

weather_detail_df_3 = weather_detail_df_2.select("weather_message_detail.*")

print("Printing Schema of weather_detail_df_3: ")
weather_detail_df_3.printSchema()
```

Till this point, dataframe transformations were going on using spark and the output of weather_detail_df_3 schema is shown below:

```
Printing Schema of weather_detail_df_3:
root
 |-- event_id: string (nullable = true)
 |-- CityName: string (nullable = true)
 |-- Temperature: double (nullable = true)
 |-- Humidity: integer (nullable = true)
 |-- CreationTime: string (nullable = true)
```

Then onwards, spark sql was used to first group by temperature and humidity values using CityName within that batch and calculate average values as shown below:

```
weather_detail_df_4 = weather_detail_df_3.select(["CreationTime", \
    "CityName", \
    "Temperature", \
    "Humidity", ])
weather_detail_df_4.printSchema()
# Data Processing/Data Transformation
weather_detail_agg_df = weather_detail_df_4\
    .groupby("CityName", "CreationTime")\
    .agg(fn.avg('Temperature').alias('AverageTemperature'),
        fn.avg('Humidity').alias('AverageHumidity'))
weather_detail_agg_df.printSchema()
```

The schema of the new dataframe is:

```
root
 |-- CityName: string (nullable = true)
 |-- CreationTime: string (nullable = true)
 |-- AverageTemperature: double (nullable = true)
 |-- AverageHumidity: double (nullable = true)
```

Then onwards, the data was saved into postgresql table as shown below by using spark and calling save to postgresql table function defined above:

```

postgresql_table_name = "dashboard_weather_detail_agg_tbl"

weather_detail_agg_df \
.writeStream \
.trigger(processingTime='2 seconds') \
.outputMode("update") \
.foreachBatch(lambda current_df, epoc_id:
save_to_postgresql_table(current_df, epoc_id, postgresql_table_name)) \
.start()

```

Finally, for monitoring purpose, the raw dataframe and dataframe with averages was printed out using the code below:

```

weather_detail_write_stream = weather_detail_df_4 \
.writeStream \
.trigger(processingTime='2 seconds') \
.outputMode("update") \
.option("truncate", "false") \
.format("console") \
.start()
print("Hello1")
weather_detail_agg_write_stream = weather_detail_agg_df \
.writeStream \
.trigger(processingTime='2 seconds') \
.outputMode("update") \
.option("truncate", "false") \
.format("console") \
.start()

weather_detail_write_stream.awaitTermination()

print("Real-Time Streaming Data Pipeline Completed.")

```

The results can be seen below:

```

Batch: 3
-----
+-----+-----+-----+-----+
|CreationTime      |CityName |Temperature|Humidity|
+-----+-----+-----+-----+
|2022-06-04 02:02:21|Karachi  |302.05     |79      |
|2022-06-04 02:02:21|Lahore   |309.14     |35      |
|2022-06-04 02:02:21|Islamabad|306.95     |30      |
|2022-06-04 02:02:27|Karachi  |306.05     |80      |
|2022-06-04 02:02:27|Lahore   |307.14     |35      |
|2022-06-04 02:02:27|Islamabad|306.95     |30      |
|2022-06-04 02:02:33|Karachi  |303.05     |79      |

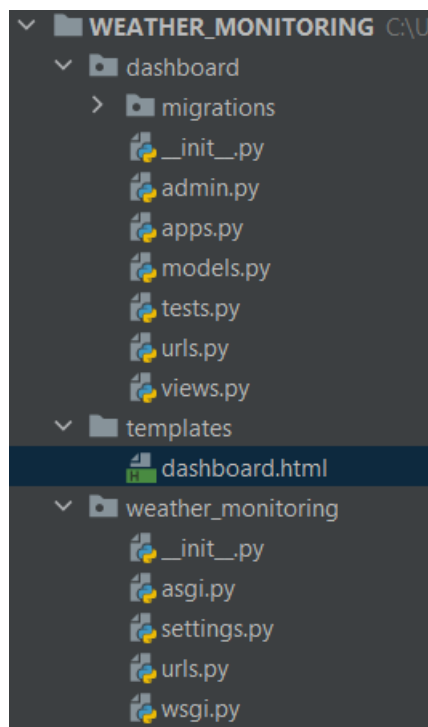
```

CityName	CreationTime	AverageTemperature	AverageHumidity
Karachi	2022-06-04 02:02:15	303.05	80.0
Lahore	2022-06-04 02:02:15	309.14	34.0

Now our streaming data pipeline is complete.

3.6 Dashboard (Front-end using Django):

Data has been processed and saved into DB. Now, a dashboard will be created using Django to show our results. Since there was no expertise in Django, I took a lot of help online and tried to patch up an extremely basic dashboard. The files are provided in project directory under the name WEATHER_MONITORING. Since, Django is a powerful framework, it requires many dependencies as shown below:



Manage.py was my server script which uses all other scripts as dependency. dashboard.html used JQuery with AJAX and flex monster to create pivot tables and charts. At first, it was important to migrate our data from postgresql table to dashboard front-end, so the following command was run:

- 1) python manage.py migrate dashboard

Then, the kafka producer script, along with real-time streaming pipeline scripts were started for data ingestion into DB. Then the server was started using the following command:

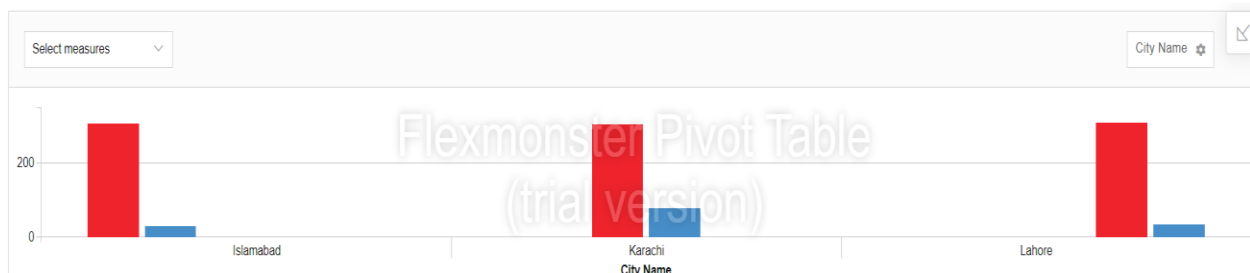
- 2) python manage.py runserver

The server was started with the host link exposed as shown below:

```
Django version 4.0.5, using settings 'weather_monitoring.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[03/Jun/2022 21:38:14] "GET / HTTP/1.1" 200 4539
PS C:\Users\bilal\BdaProject\WEATHER_MONITORING> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
```

Upon clicking this link, the following front-end can be seen:



Average of Average Temperature/Humidity means that the average computed using Spark is only for a batch, so all the averages inside that POSTGRESQL were averaged to get a full complete average value of temperature and humidity of each city.

3.7 Demo Walkthrough Link:

- YouTube: <https://www.youtube.com/watch?v=zdMtojwQm2M>
- Google Drive: https://drive.google.com/drive/folders/1-uNunyUKERd_NLayVymxI0vLh3g04mjw?usp=sharing

4. Challenges and Conclusion

The major challenge came in setting up dependencies and dealing with a plethora of java errors. In 80+ hours spent on this project; I would say at least 20 hours of those were spent resolving java errors. The major problem was finding a compatible java version which would work with Hadoop and spark. After several tries, it was realized that Java 17 or later does not support a lot of pyspark functions, so I had to use Java 11 version. After that, it was smooth sailing. It was quite difficult to connect with a PostgreSQL database. After trying a lot on the zeppelin example, I finally gave up on it and decided to replicate the project from scratch using DataMaking videos. I would like to thank them in helping me to reach this point.

The front-end is basic and there was truly little time to play around with. so, in the future I would like to use HDFS with the project and query it is using Hive, then later, I would like to create more meaningful dashboard. There was a lot of learning in this project since the technology stack is new and continuously evolving, it was difficult to find concrete examples to help with this project.

Overall, it was quite a meaningful project which included a lot of learning related to technical understanding. I hope to take this project further and make it even better than it is now.