Alexis Balayre

Cloud Computing Assignment

School of Aerospace, Transport and Manufacturing
Computational Software of Techniques Engineering

MSc
Academic Year: 2023 - 2024

Supervisor: Dr Stuart Barnes
2nd January 2024

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In an increasingly connected world, cloud computing and the Internet of Things (IoT) are revolutionising many fields, including environmental monitoring. This technological development offers unprecedented possibilities for managing and analysing air quality, a major public health issue. This report, drawn up as part of my Master's degree in Cloud and Embedded Systems Science and Technology (CSTE), focuses on the use of these technologies to collect, process and distribute environmental data.

The main aim of this assignment is to store and make accessible the latest air quality data, captured by a network of small environmental IoT sensors. The project aims to provide a reliable platform for real-time consultation of environmental data, a crucial tool for researchers, decision-makers and the general public.

We face a number of technical challenges in achieving this objective. Firstly, managing the large quantities of data generated by IoT sensors requires a robust and adaptable cloud infrastructure. Secondly, calculating the Air Quality Index (AQI) from this data in real time requires considerable processing power and accuracy. Finally, the need to keep the system adaptable and responsive to varying workloads presents an additional challenge.

To address these challenges, our approach is to use a database located in the cloud, specifically designed to manage and process large volumes of IoT data. This database will be regularly updated with new data, while allowing quick and easy access for end users. In addition, we will be implementing advanced algorithms for calculating the AQI, guaranteeing the accuracy and reliability of the information provided.

The importance of this system is not limited to environmental monitoring; it also has a significant impact on public health, urban planning and environmental awareness. By providing accurate and up-to-date data, we contribute to a better understanding and management of air quality.

In conclusion, this report will detail our methodology, the architecture of the system, the challenges encountered and the solutions adopted. We will also discuss the implications of our work, not only in technical terms but also in terms of its practical applications and impact on different stakeholders.

Chapter 2

Methodologies

2.1 Data Collecting, Processing & Storing

2.1.1 Overview of the Pipeline Architecture

The initial pipeline in this project consists of three primary components: **Data Collecting**, **Data Processing**, and **Data Storing**.

During the **Data Collecting** phase, the most recent version of the dataset is acquired from its source. This is followed by the **Data Processing** phase, where the data is formatted, and the Air Quality Index (AQI) is calculated for each particulate matter sensor. Lastly, in the **Data Storing** phase, the data from each sensor is methodically stored in a time-series database.

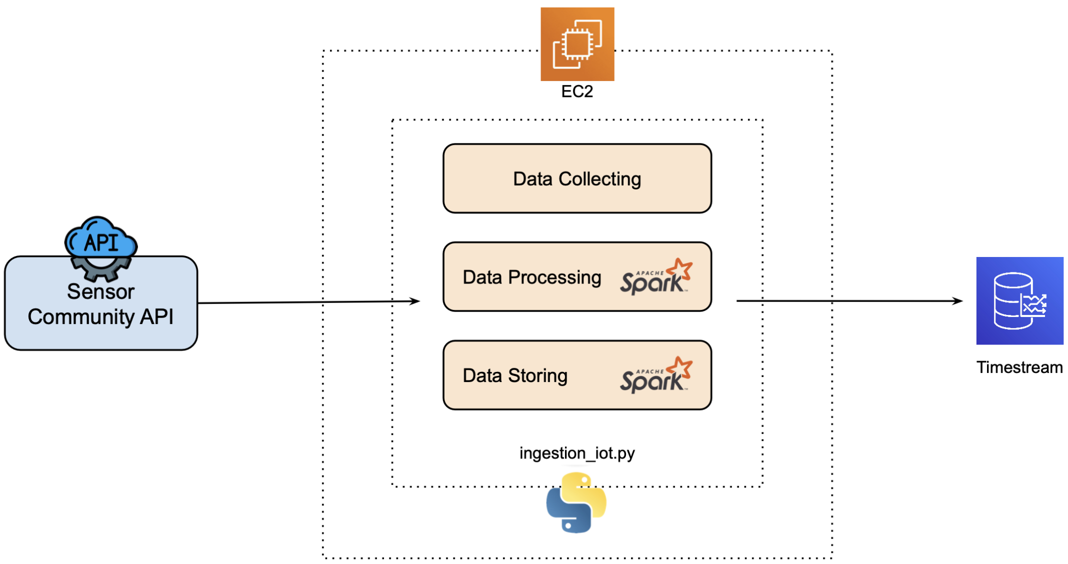


Figure 2.1: Data Collecting, Processing & Storing Pipeline Diagram

2.1.2 Data Collecting

2.1.2.1 Data Source

The Sensor Community network is a global, contributor-driven initiative that collects open environmental data through a vast network of sensors. These sensors, deployed in over 70 countries, collect real-time data on air quality, temperature, humidity and pressure. On average, the sensors send new data every 145 seconds (1). The Sensor Community network offers two main API endpoints for accessing their environmental data:

1. **5-Minute Averaged Data API:** This API provides data averaged over the last 5 minutes for each sensor. This is useful for near real-time analysis or immediate air quality assessments, particularly in test or active monitoring contexts.

2. **24 Hour Averaged Data API:** This API provides data averaged over the last 24 hours for each sensor. It is particularly suited to analysing daily trends and understanding environmental changes over a longer period.

2.1.2.2 Python Script

In this project, both APIs were used to collect data from the Sensor Community. The data was collected using the `requests` library in Python and stored in the cache of the local machine in a Spark DataFrame. This step is performed every 10 seconds to ensure that the data is up to date.

2.1.3 Choice of Database

This project uses Amazon Timestream, which is a cloud-native time series database. It is a wise choice because of its superior time series management capabilities, which are particularly well suited to data from IoT sensors. This database is distinguished by its speed of ingestion and efficiency in processing vast volumes of data, facilitating regular updates and in-depth analyses of the Air Quality Index. Its ability to adjust to variations in workload is a major asset, ensuring consistent performance. What's more, Timestream's advanced security features meet strict standards of confidentiality and data sovereignty, an essential criterion for the secure management of sensitive information.

2.1.4 Query 3

2.2 Queries Optimisation

2.3 Pipeline of the Project

2.3.1 Pipeline Overview

The pipeline of this project is composed of four main components: **data ingestion**, **query 1**, **query 2** and **query 3**.

The **ingestion** task retrieves the last version of the data set from the source and stores it in the data lake (CSV file stored in local storage). Then, the **queries 1, 2 & 3** tasks retrieves the data from the data lake and performs the queries on the data.

2.3.2 Pipeline Orchestration

In order to orchestrate and automate the pipeline, a scheduled task must be run every day to retrieve the latest version of the dataset and run the tasks when a new daily row is added at 23:59 UTC to the dataset.

To perform this task, a DAG (Directed Acyclic Graph) was created using Apache Airflow. The DAG is scheduled to run every day at 00:00 UTC and is composed of four tasks: **ingestion**, **query 1**, **query 2** and **query 3**. The screenshot below 2.3 shows the DAG graph of the pipeline in the web interface of Apache Airflow.

The benefits of using a workflow platform such as Apache Airflow are its ability to schedule and automate the pipeline, as well as its ability to monitor the pipeline and send alerts if a task fails.

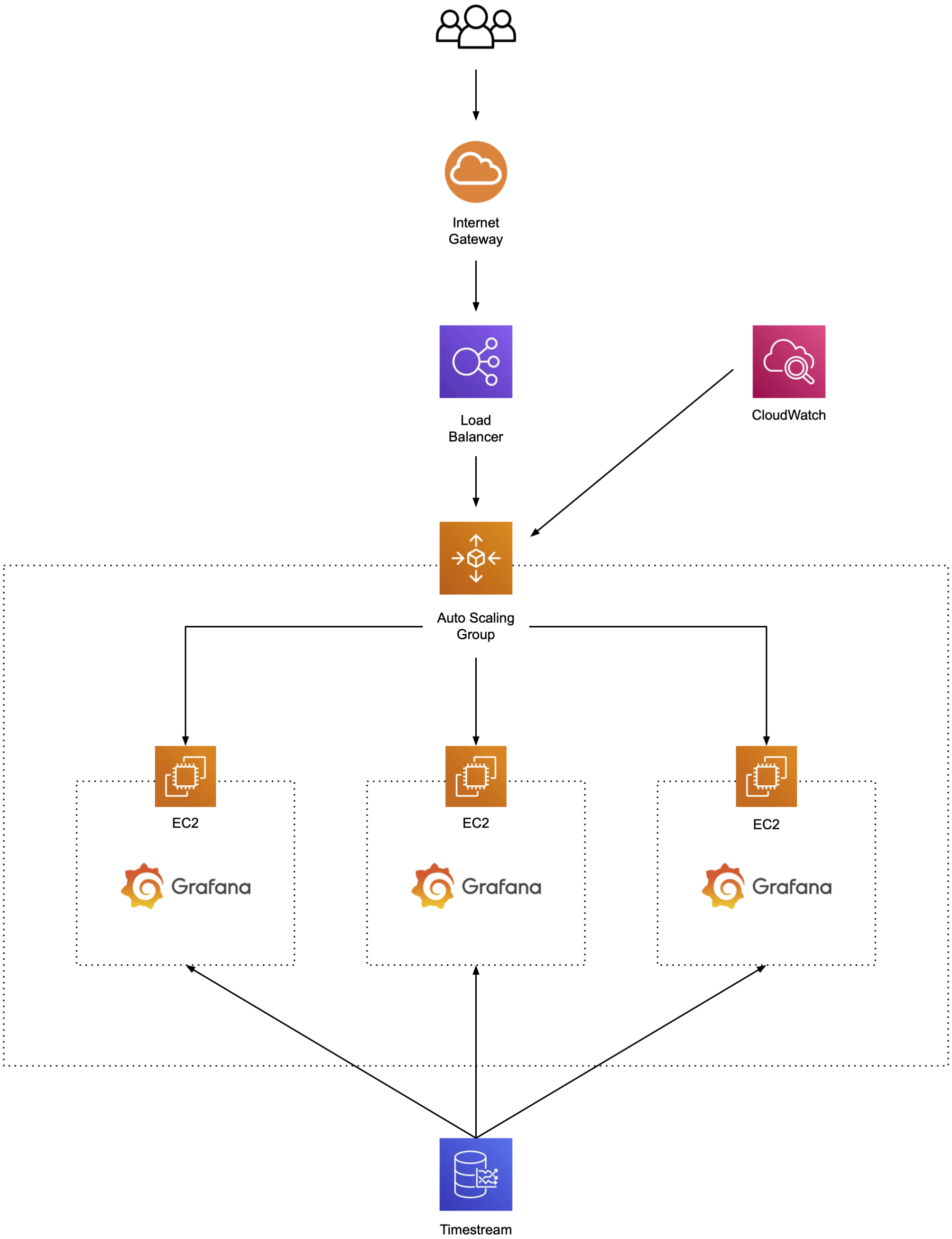


Figure 2.2: Data Distributing Pipeline Diagram

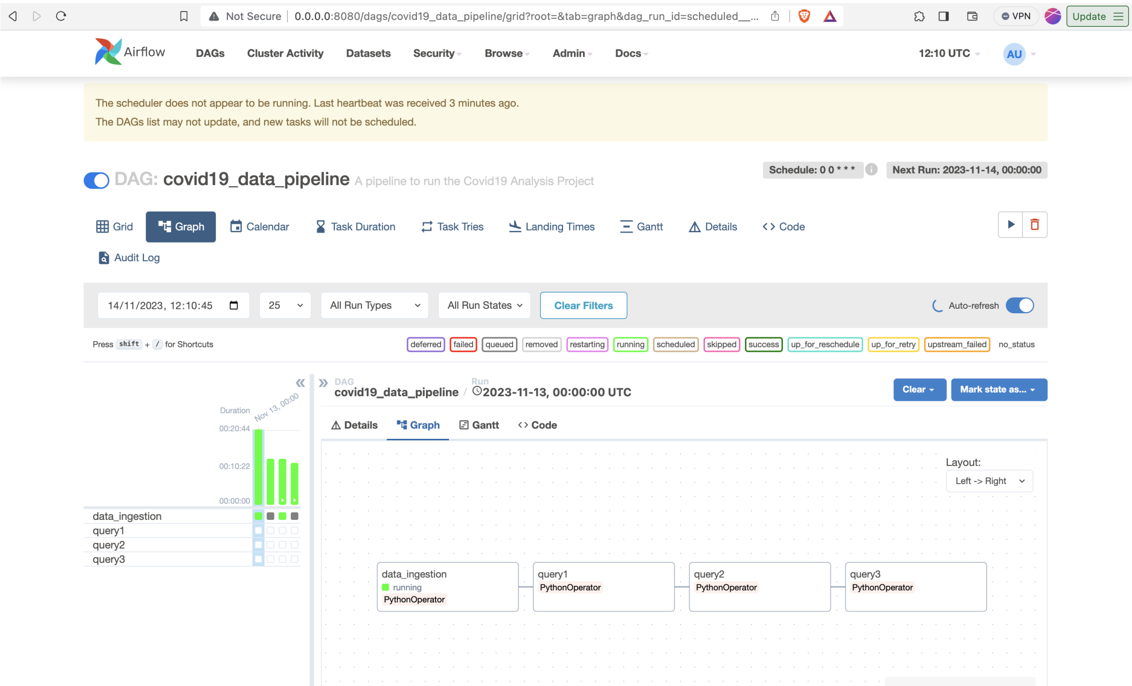


Figure 2.3: Apache Airflow DAG Graph

Chapter 3

Results & Discussion

3.1 Queries Results

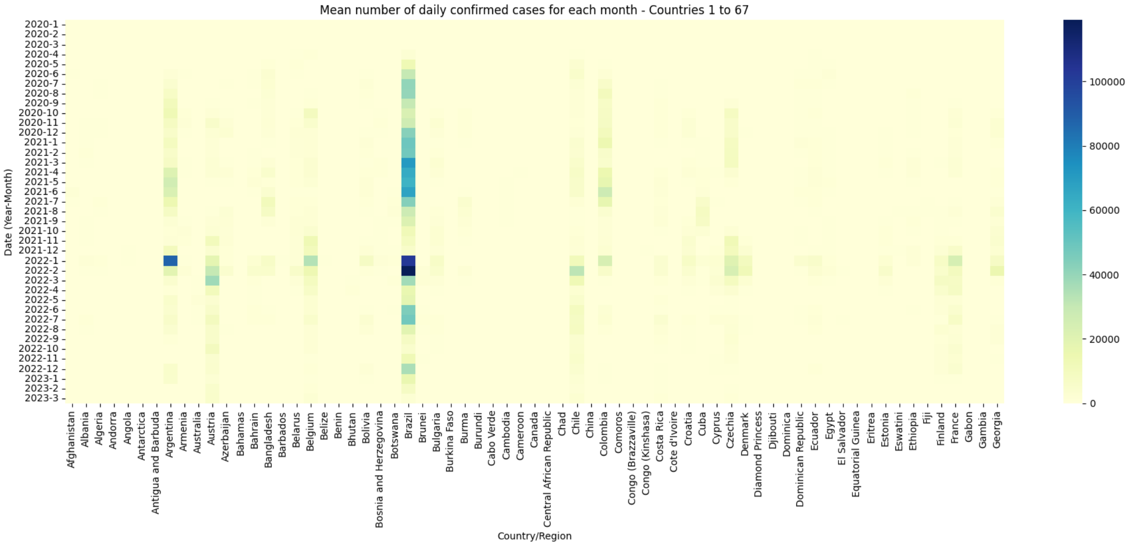
The programme was last run on 18 November 2023. The appendix **??** shows the output of the program on the terminal.

3.1.1 Query 1

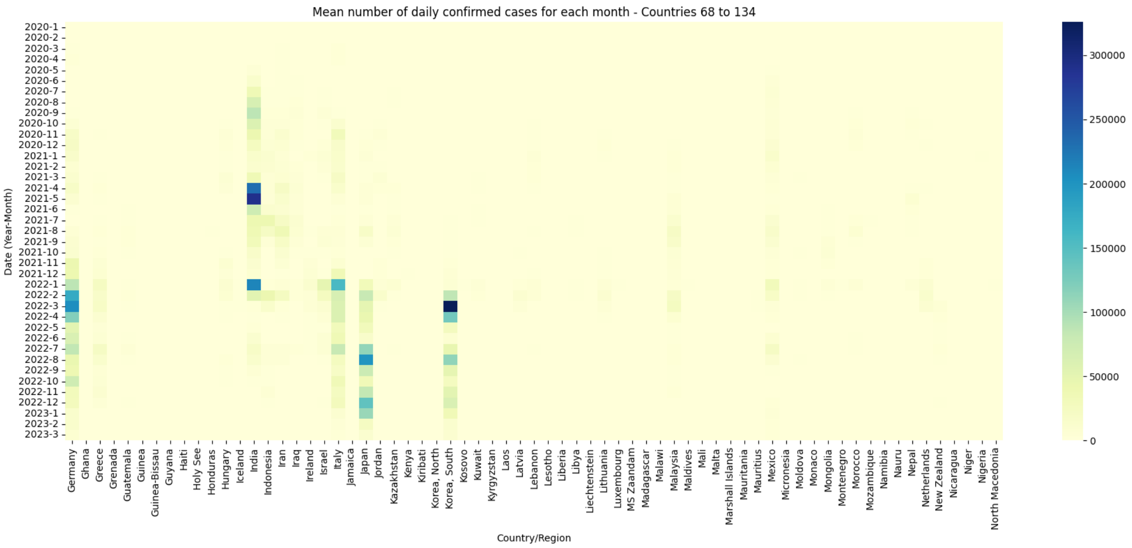
The first query takes around 3 seconds, and the table 3.1 shows a sample of the data calculated during the task. In order to evaluate performance, an equivalent script not using Spark was run. Execution time was 0.5 sec. The 3.2 section will cover this point. The results are consistent with what was expected (**?**). For example, the figure 3.1a shows that Brazil was heavily impacted by the pandemic, reaching an average peak in February 2022. The same is true for Korea in figure 3.1b and the United States in figure 3.1c, which were heavily impacted.

Table 3.1: Query 1 Results Sample

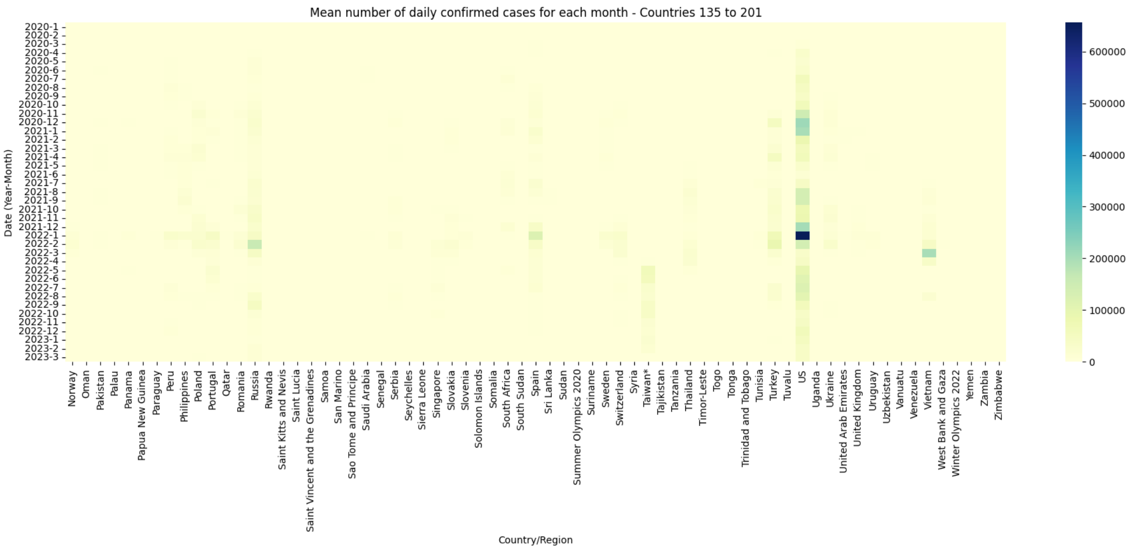
	Country/Region	Year	Month	Average
1	Afghanistan	2020	1	0.0
2	Afghanistan	2020	2	0.1724137931034483
3	Afghanistan	2020	3	5.193548387096774
4	Afghanistan	2020	4	55.36666666666667
5	Afghanistan	2020	5	430.741935483871



(a) Countries 1 to 67



(b) Countries 68 to 134



(c) Countries 135 to 201

Figure 3.1: Mean Daily Confirmed Cases Per Month

3.1.2 Query 2

The second query takes around 15 seconds, and the table 3.2 shows a sample of the data calculated during the task. In order to evaluate performance, an equivalent script not using Spark was run. Execution time was 6 sec. The 3.2 section will cover this point. Locations used to compute the statistics are shown on the map of figure 3.2. The area of the circles is proportional to how the location has been affected by the pandemic.

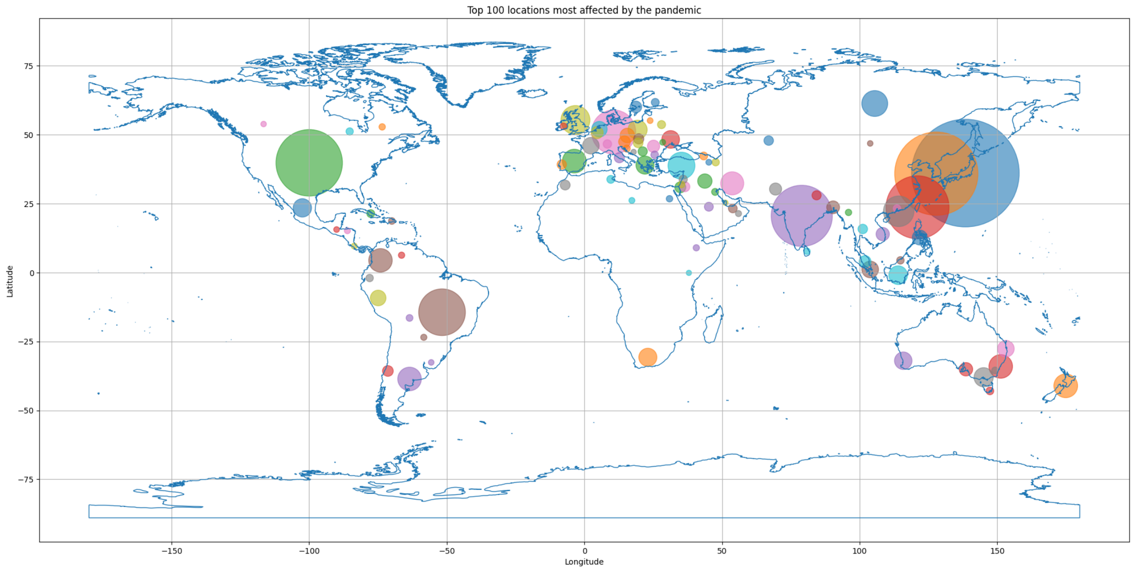


Figure 3.2: Top 100 Locations most affected by the pandemic

Table 3.2: Query 2 Results Sample

	Continent	WeekRange	Mean	Std	Min	Max
1	Africa	19/01/2020-25/01/2020	0.0	0.0	0	0
2	Africa	26/01/2020-01/02/2020	0.0	0.0	0	0
3	Africa	02/02/2020-08/02/2020	0.0	0.0	0	0
4	Africa	09/02/2020-15/02/2020	0.0	0.0	0	0
5	Africa	16/02/2020-22/02/2020	0.0	0.0	0	0
6	Africa	23/02/2020-29/02/2020	0.0	0.0	0	0
7	Africa	01/03/2020-07/03/2020	0.02857	0.16903	0	1
8	Africa	08/03/2020-14/03/2020	0.65714	1.73108	0	9
9	Africa	15/03/2020-21/03/2020	2.74285	4.53964	0	17
10	Africa	22/03/2020-28/03/2020	12.62857	15.33754	0	59
11	Africa	29/03/2020-04/04/2020	14.97142	19.31242	0	82

The figures below 3.3 and 3.4 show the mean and standard deviation of the number of confirmed cases by week and continent. The results are consistent with expectations: the continents most affected are America and Europe (**?**).

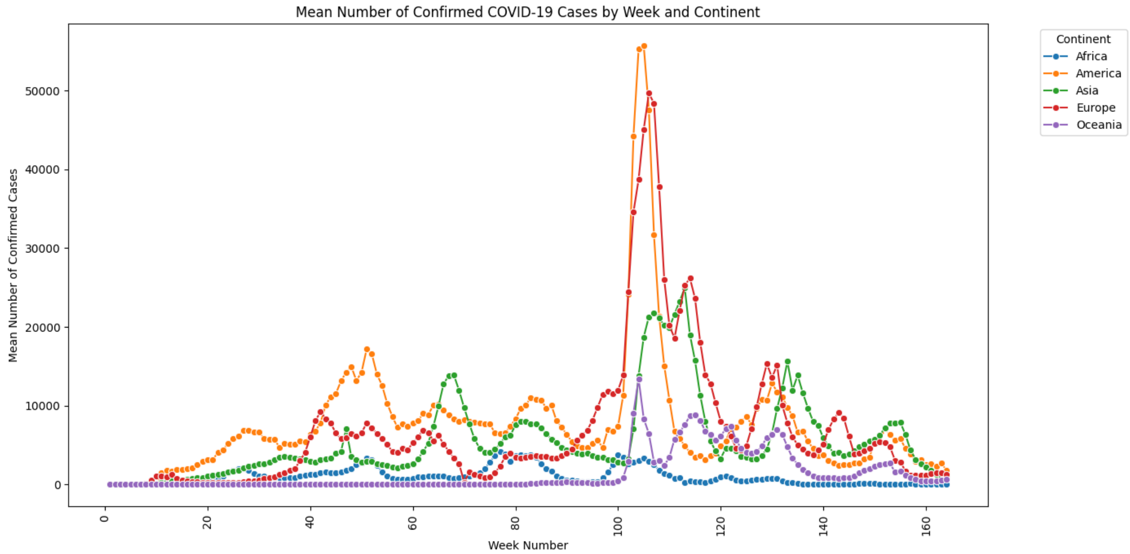


Figure 3.3: Mean Confimed Cases By Week and Continent

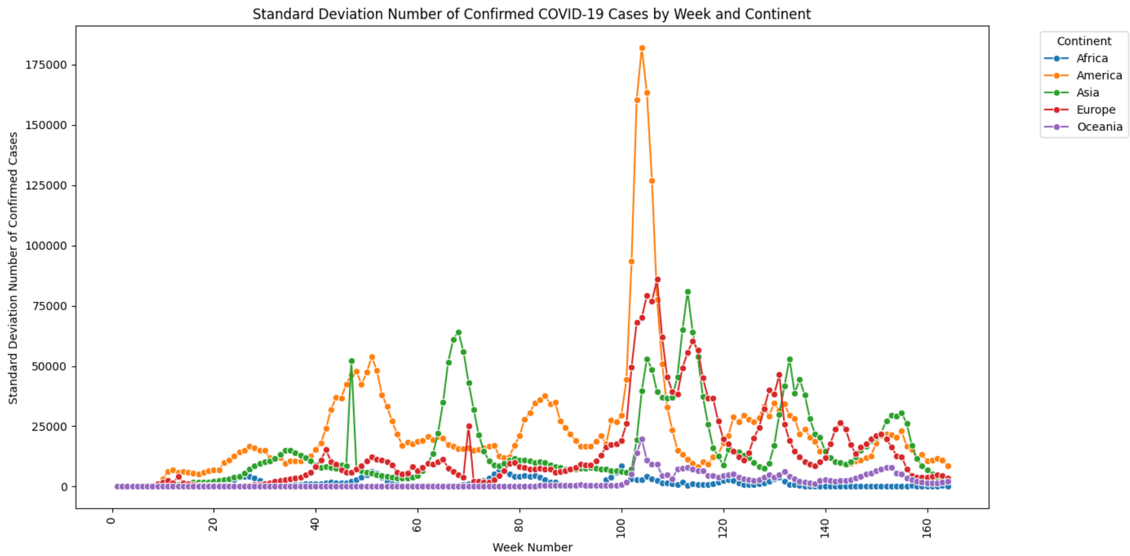


Figure 3.4: Standard Deviation Confimed Cases By Week and Continent

The figures 3.5 and 3.6 show the maximum and minimum of the number of confirmed cases by week and continent.

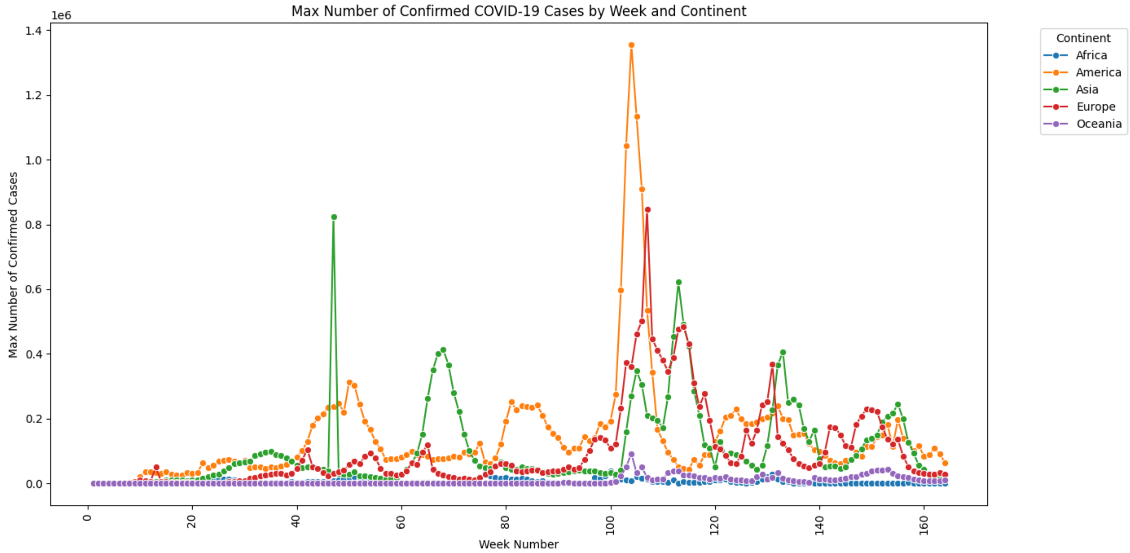


Figure 3.5: Maximum Confimed Cases By Week and Continent

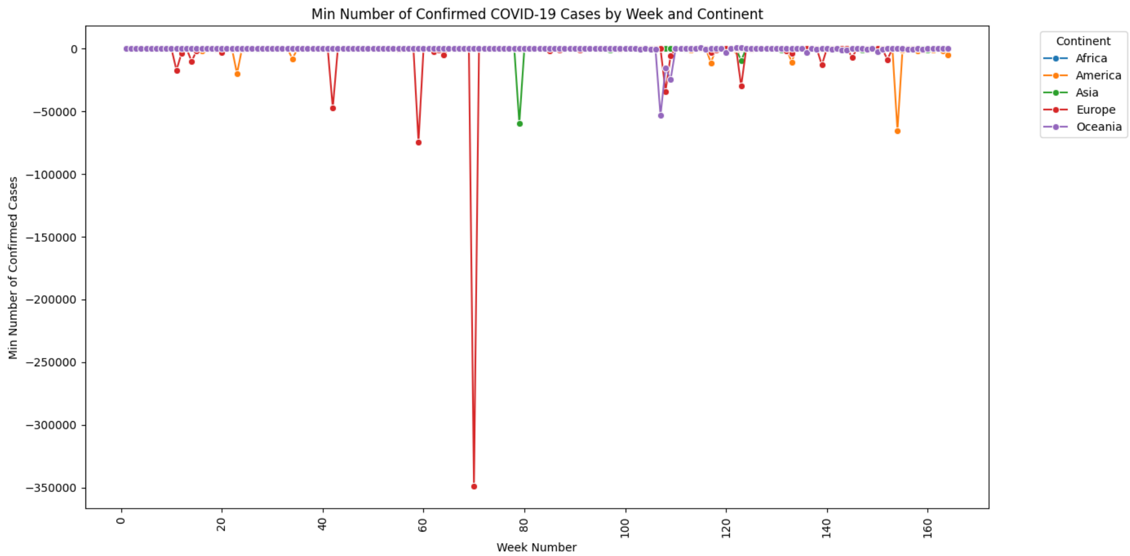


Figure 3.6: Minimum Confimed Cases By Week and Continent

3.1.3 Query 3

The third query takes around 3 minutes as clustering with the custom implementation takes 60 seconds and clustering with the Spark MLlib implementation takes 110 seconds. The table 3.3 shows a sample of the data calculated during the task with the custom implementation and the table 3.4 with the Spark MLlib implementation. Locations used to compute the statistics are shown on the map of figure 3.7. The area of the circles is proportional to how the location has been affected by the pandemic.

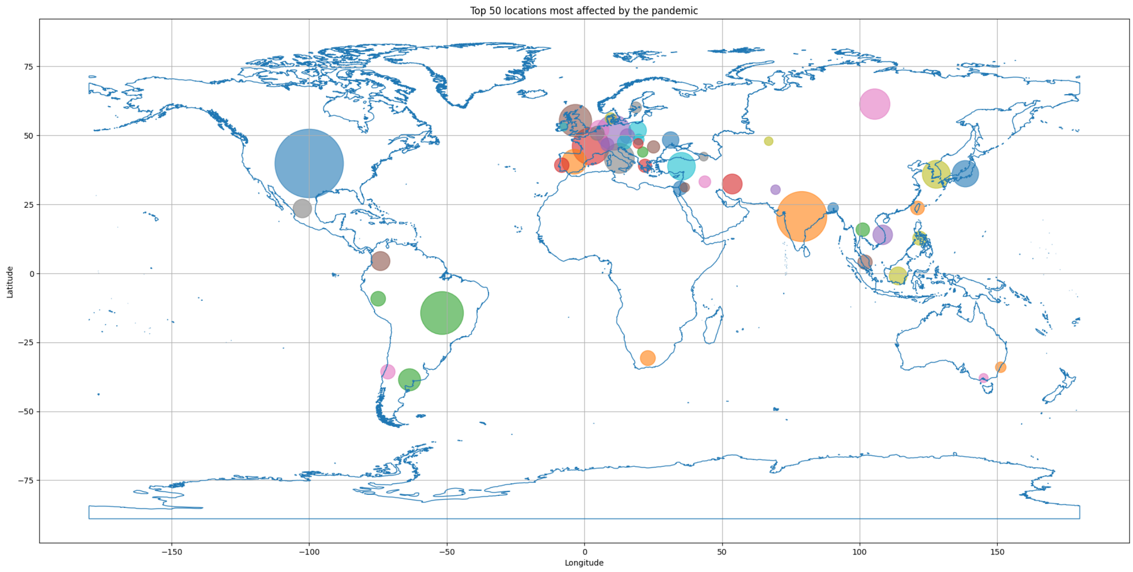


Figure 3.7: Top 50 Locations most affected by the pandemic

Table 3.3: Query 3 Results Sample - Custom KMeans Clustering

	Location	Month	Cluster
1	Argentina	2020-01	2
2	Austria	2020-01	2
3	Brazil	2020-01	2
4	Czechia	2020-01	2
5	France	2020-01	1

Table 3.4: Query 3 Results Sample - Spark MLlib KMeans Clustering

	Location	Month	Cluster
1	Argentina	2020-01	2
2	Austria	2020-01	0
3	Brazil	2020-01	2
4	Czechia	2020-01	2
5	France	2020-01	1

The figures below 3.8 and 3.9 show the clusters of the top 50 locations most affected by the pandemic in March 2020. The clusters are represented by different colours.

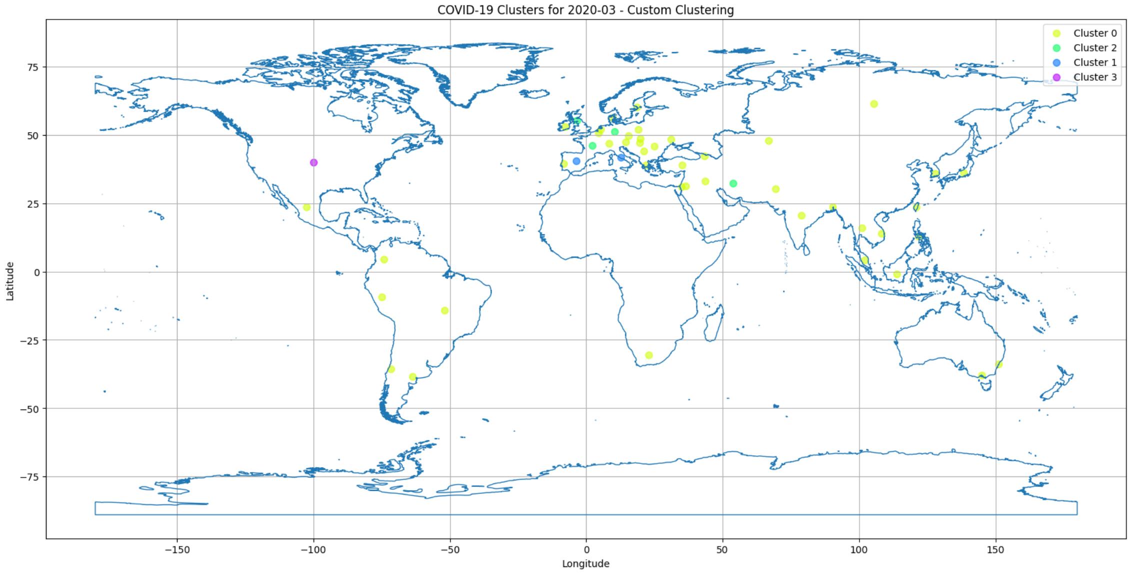


Figure 3.8: Custom KMeans Clustering on 03/2020

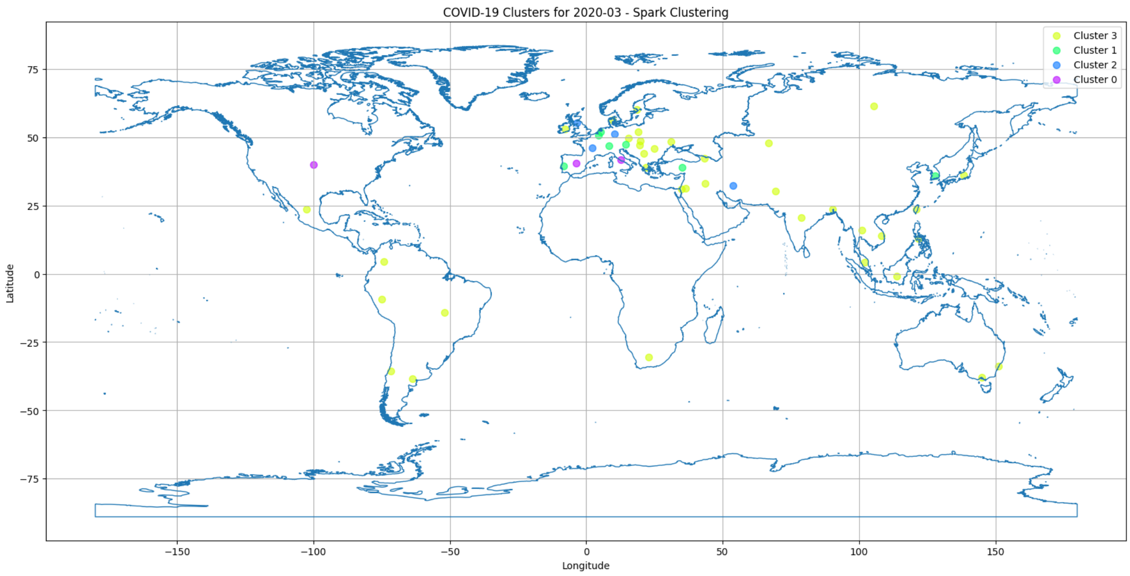


Figure 3.9: Spark MLlib KMeans Clustering on 03/2020

3.2 Discussion of Results

As the previous results show, it seems that scripts not using Spark are faster. While these differences in execution time may seem surprising at first glance, they can in fact be attributed to several factors:

1. **Data size and structure:** The DataFrame, with its 289 entries and 1147 columns occupying 2.5 MB of memory, is relatively modest in size. Pandas is particularly efficient at processing such large amounts of data in memory on a single node. Spark, on the other hand, is designed for distributed processing of large datasets. In this case, the overheads of distributing the data and managing the Spark environment may outweigh the benefits of using it for smaller datasets.

2. **Operational efficiency:** Pandas performs vectorised operations that are optimised for speed, especially with datasets that fit easily into memory on a single computer. Spark, while powerful for processing large volumes of data, introduces an initial overhead for distributing data and configuring the distributed environment, which can slow down processing for smaller datasets.

3. **Complexity of the environment:** Complexity of the environment: Running operations in a Spark environment involves initializing a cluster (even in local mode), distributing tasks, and managing distributed memory, which adds extra processing time compared to running in-memory Pandas directly.

Thus in this project run locally, Pandas is faster than Spark, due to its efficient management of in-memory operations on a single node. Spark's distributed processing overhead makes it less efficient for such tasks.

3.3 Ethical Considerations and Challenges

Chapter 4

Conclusion

References

1. peoter. Are all Sensor Community sensors synchronized?. Sensor Community Forum; 2023. Available at: `https://forum.sensor.community/t/are-all-sensor-community-sensors-synchronized/2479/2`. (Accessed: December 18, 2023).

Appendices

Appendix A

Documentation

Appendix A.A Project tree

```
lib/
collecting.py
processing.py
storing.py
scripts/
    get_iam_credentials.sh
    start_spark_job.sh
services/
    get_iam_credentials.service
    spark_python_job.service
    grafana_server.service
artillery_load_test.yml
ingestion_iot_data_flatten.py
main.py
README.md
requirements.txt
```

Appendix A.B Getting Started

To run the program, follow these steps:

1. Create a virtual environment using `python3 -m venv venv`.
2. Activate the virtual environment using `source venv/bin/activate`.
3. Install the required dependencies using `pip3 install -r requirements.txt`.
4. Run the program using `python3 main.py`.
5. Visualise the results using `visualisation.ipynb` (Jupyter Notebook).

Appendix A.C Detailed Features of Functions

`collecting.py`

- `fetch_sensors_data(sparkSession)`: Function to ingest the latest data from the sensors and returns it as a Spark DataFrame.

`processing.py`

- `get_aqi_value_p25(value)`: Function for calculating the AQI value for PM2.5.
- `get_aqi_value_p10(value)`: Function for calculating the AQI value for PM10.
- `computeAQI(df)`: Function for calculating the AQI value for each particulate matter sensor and returning the DataFrame with the AQI column.

`storing.py`

- `keepOnlyUpdatedRows(database_name, table_name, df)`: Function for keeping only the rows that have been updated in the DataFrame.
- `_print_rejected_records_exceptions(err)`: Internal function for printing the rejected records exceptions.
- `write_records(database_name, table_name, client, records)`: Internal function for writing a batch of records to the Timestream database.
- `writeToTimestream(database_name, table_name, partionned_df)`: Function for writing the DataFrame to the Timestream database.

Appendix B

Source Codes

Appendix B.A Ingestion, Processing & Storing Pipeline Source Code

```python
import findspark

findspark.init()  # Initializing Spark

from pyspark.sql import SparkSession

import datetime as dt
import time

from lib.collecting import fetch_sensors_data
from lib.processing import computeAQI
from lib.storing import keepOnlyUpdatedRows, writeToTimestream

if __name__ == "__main__":
    # Define the Timestream database and table names
    DATABASE_NAME = "iot_project"
    TABLE_NAME = "iot_table"

    # Initializing Spark Session
    sparkSession = (
        SparkSession.builder.appName("Cloud Computing Project")
        .master("local[*]")
        .config("spark.sql.inMemoryColumnarStorage.compressed", "
true")
        .config("spark.sql.inMemoryColumnarStorage.batchSize", "
10000")
        .config("spark.serializer", "org.apache.spark.serializer.
KryoSerializer")
        .config("spark.ui.enabled", "true")
        .config("spark.io.compression.codec", "snappy")
        .config("spark.rdd.compress", "true")
        .getOrCreate()
    )

    while True:
```

```
        try:
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " Starting the pipeline..."
            )
            # Fetch the data from the sensors
            iotDfRaw = fetch_sensors_data(sparkSession)

            # Compute the AQI for each sensor
            iotDfFormatted = computeAQI(iotDfRaw)

            # Filter the data to keep only the updated rows
            dataFiltered = keepOnlyUpdatedRows(
                DATABASE_NAME, TABLE_NAME, iotDfFormatted
            )

            # Write the data to Timestream
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " 4. Writing the data to Timestream..."
            )
            dataFiltered.foreachPartition(
                lambda partition: writeToTimestream(
                    DATABASE_NAME, TABLE_NAME, partition
                )
            )
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " Done writing the data to Timestream.\n"
            )

            # Sleep for 10 seconds
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " Done with the pipeline. Waiting for 2 minutes.\n"
            )
            time.sleep(10)
        except Exception as e:
            print(f"Exception: {e}")
```

Appendix B.B Data Collecting Source Code

```
# collecting.py
# The first step of the pipeline

from requests import Session
import datetime as dt


def fetch_sensors_data(sparkSession):
    """
```

```
    Fetches the latest data from the sensors and returns it as a Spark DataFrame

    Args:
        sparkSession (SparkSession): The SparkSession instance

    Returns:
        df (DataFrame): The DataFrame containing the last data from the sensors
    """

    # Fetches the latest data from the data.sensor.community API
    url = "https://data.sensor.community/static/v2/data.json"
    # Use a session to avoid creating a new connection for each request
    session = Session()
    try:
        print(
            dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            + " 1. Fetching the latest data..."
        )
        response = session.get(url)
        # If the response was successful, no Exception will be raised
        if response.status_code == 200 and response.content:
            # Convert the response to a Spark DataFrame
            df = sparkSession.read.option("multiline", "true").json(
                sparkSession.sparkContext.parallelize([response.text])
            )
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " Done fetching the latest data.\n"
            )
            return df
    except Exception as e:
        print(f"Request failed with exception {e}")
    finally:
        session.close()
    return None
```

Appendix B.C Data Processing Source Code

```
# collecting.py
# The second step of the pipeline

from pyspark.sql.types import FloatType, IntegerType
import pyspark.sql.functions as F
import datetime as dt


```

```python
# Defining a UDF to compute the AQI value for PM2.5
@F.udf(returnType=IntegerType())
def get_aqi_value_p25(value):
    """
    Computes the AQI value for PM2.5

    Args:
        value (float): The value of PM2.5
    Returns:
        aqi (int): The AQI value
    """

    if value is None:
        return None
    if 0 <= value <= 11:
        return 1
    elif 12 <= value <= 23:
        return 2
    elif 24 <= value <= 35:
        return 3
    elif 36 <= value <= 41:
        return 4
    elif 42 <= value <= 47:
        return 5
    elif 48 <= value <= 53:
        return 6
    elif 54 <= value <= 58:
        return 7
    elif 59 <= value <= 64:
        return 8
    elif 65 <= value <= 70:
        return 9
    return 10


# Defining a UDF to compute the AQI value for PM10
@F.udf(returnType=IntegerType())
def get_aqi_value_p10(value):
    """
    Computes the AQI value for PM10

    Args:
        value (float): The value of PM10

    Returns:
        aqi (int): The AQI value
    """

    if value is None:
        return None
    if 0 <= value <= 16:
        return 1
    elif 17 <= value <= 33:
        return 2
    elif 34 <= value <= 50:
        return 3
```

```
    elif 51 <= value <= 58:
        return 4
    elif 59 <= value <= 66:
        return 5
    elif 67 <= value <= 75:
        return 6
    elif 76 <= value <= 83:
        return 7
    elif 84 <= value <= 91:
        return 8
    elif 92 <= value <= 99:
        return 9
    return 10


def computeAQI(df):
    """
    Computes the AQI for each particulate matter sensor

    Args:
        df (DataFrame): The DataFrame containing the data from the
    sensors

    Returns:
        df_grouped (DataFrame): The DataFrame containing the AQI
    for each sensor
    """

    print(dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S") + " 2.
    Computing the AQI...")
    df_exploded = df.withColumn(
        "sensordatavalue",
        F.explode("sensordatavalues"),  # Explode the
    sensordatavalues column
    ).withColumn(
        "aqi",
        F.when(
            F.col("sensordatavalue.value_type") == "P1",
            get_aqi_value_p25(
                F.col("sensordatavalue.value").cast(FloatType())
            ),  # Cast the value to float and compute the AQI of
    PM2.5
        ).when(
            F.col("sensordatavalue.value_type") == "P2",
            get_aqi_value_p10(
                F.col("sensordatavalue.value").cast(FloatType())
            ),  # Cast the value to float and compute the AQI of
    PM10
        ),
    )
    df_exploded.cache()  # Cache the DataFrame to avoid recomputing
     it
    df_grouped = (
        df_exploded.groupBy("sensor.id", "timestamp")  # Group by
    sensor and timestamp
        .agg(
```

```
            F.first("id").alias("id"),
            F.first("location").alias("location"),
            F.first("sensor").alias("sensor"),
            F.max("aqi").alias("aqi"),  # Compute the maximum AQI
    between PM2.5 and PM10
            F.collect_list("sensordatavalue").alias("
    sensordatavalues"),
        )  # Aggregate the AQI and the sensordatavalues
        .selectExpr(
            "sensor.id as sensor_id",
            "sensor.pin as sensor_pin",
            "sensor.sensor_type.id as sensor_type_id",
            "sensor.sensor_type.manufacturer as
    sensor_type_manufacturer",
            "sensor.sensor_type.name as sensor_type_name",
            "location.country as country",
            "location.latitude as latitude",
            "location.longitude as longitude",
            "location.altitude as altitude",
            "location.id as location_id",
            "aqi",
            "sensordatavalues",
            "timestamp",
        )  # Select the columns to keep
    )
    df_exploded.unpersist()  # Unpersist the DataFrame to free
    memory
    print(
        dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S") + " Done
    computing the AQI.\n"
    )
    return df_grouped
```

Appendix B.D Data Storing Source Code

```
# storing.py
# The last step of the pipeline

from pyspark.sql.types import BooleanType
import pyspark.sql.functions as F
from pyspark.sql import Row
from botocore.config import Config
import boto3
import time
import datetime as dt


def keepOnlyUpdatedRows(database_name, table_name, df):
    """
    Verifies if the data is already stored in Timestream and keeps
    only the updated values

```

```
    Args:
        database_name (string): The name of the database
        table_name (string): The name of the table
        df (DataFrame): The DataFrame containing the data to be stored

    Returns:
        df_updated (DataFrame): The DataFrame containing only the updated rows
    """

    print(
        dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        + " 3. Filtering the data to keep only the updated rows..."
    )
    query = """
        SELECT sensor_id, MAX(time) as last_timestamp
        FROM {}.{}
        GROUP BY sensor_id
    """.format(
        database_name, table_name
    )

    # Initialize the boto3 client
    session = boto3.Session()  # Create a boto3 session
    query_client = session.client(
        "timestream-query", config=Config(region_name="us-east-1")
    )  # Create a boto3 client
    paginator = query_client.get_paginator("query")  # Create a paginator

    # Get the last timestamp for each sensor
    last_timestamps = (
        {}
    )  # Initialize a dictionary to store the last timestamp for each sensor
    response_iterator = paginator.paginate(QueryString=query)  # Paginate the query
    for response in response_iterator:
        for row in response["Rows"]:
            sensor_id = row["Data"][0]["ScalarValue"]
            last_timestamps[sensor_id] = row["Data"][1]["ScalarValue"]

    # If there is no data in Timestream, return the DataFrame as is
    if len(last_timestamps) == 0:
        print("No data in Timestream")
        return df

    # Define an UDF to check if the row is updated
    @F.udf(returnType=BooleanType())
    def isUpdated(sensor_id, timestamp):
        """
        Checks if the row is updated

        Args:
```

```
            sensor_id (string): The sensor ID
            timestamp (string): The timestamp of the row

        Returns:
            isUpdated (boolean): True if the row is updated, False
    otherwise
        """

        if str(sensor_id) not in last_timestamps:
            return True
        current_timestamp = dt.datetime.strptime(timestamp, "%Y-%m
    -%d %H:%M:%S")
        last_timestamp_micro = last_timestamps[str(sensor_id)][
            :26
        ]  # Keep only up to microseconds
        last_sensor_timestamp = dt.datetime.strptime(
            last_timestamp_micro, "%Y-%m-%d %H:%M:%S.%f"
        )
        return (
            current_timestamp > last_sensor_timestamp
        )  # Return True if the row is updated

    df_updated = df.filter(
        isUpdated("sensor_id", "timestamp")
    )  # Filter the DataFrame to keep only the updated rows
    print(
        dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        + " Done filtering the data to keep only the updated rows.\
    n"
    )
    return df_updated


def _print_rejected_records_exceptions(err):
    """
    Prints the rejected records exceptions

    Args:
        err (RejectedRecordsException): The
    RejectedRecordsException
    """

    print("RejectedRecords: ", err)
    for rr in err.response["RejectedRecords"]:
        print("Rejected Index " + str(rr["RecordIndex"]) + ": " +
    rr["Reason"])
        if "ExistingVersion" in rr:
            print("Rejected record existing version: ", rr["
    ExistingVersion"])


def write_records(database_name, table_name, client, records):
    """
    Helper function to write records to Timestream

    Args:
```

```
        database_name (string): The name of the database
        table_name (string): The name of the table
        client (TimestreamWriteClient): The TimestreamWriteClient
        records (list): The list of records to write
    """
    try:
        result = client.write_records(
            DatabaseName=database_name,
            TableName=table_name,
            CommonAttributes={},
            Records=records,
        )
        print(
            "WriteRecords Status: [%s]" % result["ResponseMetadata"]["HTTPStatusCode"]
        )
    except client.exceptions.RejectedRecordsException as err:
        _print_rejected_records_exceptions(err)
    except Exception as err:
        print("Error:", err)


def writeToTimestream(database_name, table_name, partionned_df):
    """
    Writes the data to Timestream

    Args:
        database_name (string): The name of the database
        table_name (string): The name of the table
        partionned_df (DataFrame): The DataFrame containing the data to be stored
    """

    # Initialize the boto3 client for each partition
    session = boto3.Session()
    write_client = session.client(
        "timestream-write",
        config=Config(
            read_timeout=20, max_pool_connections=5000, retries={"max_attempts": 10}
        ),
    )

    # Create a list of records
    records = []
    for row in partionned_df:
        try:
            # Skip rows that are not of type Row
            if not isinstance(row, Row):
                continue

            # Convert timestamp to Unix epoch time in milliseconds
            timestamp_datetime = dt.datetime.strptime(
                row.timestamp, "%Y-%m-%d %H:%M:%S"
            )
```

```
            row_timestamp = str(int(timestamp_datetime.timestamp()
    * 1000))

            # altitude
            altitude = row.altitude if row.altitude != "" else 0

            # Create dimensions list
            dimensions = [
                {"Name": "country", "Value": str(row.country)},
                {"Name": "latitude", "Value": str(row.latitude)},
                {"Name": "longitude", "Value": str(row.longitude)},
                {"Name": "altitude", "Value": str(altitude)},
                {"Name": "location_id", "Value": str(row.
    location_id)},
                {"Name": "sensor_id", "Value": str(row.sensor_id)},
                {"Name": "sensor_pin", "Value": str(row.sensor_pin)
    },
                {
                    "Name": "sensor_type_manufacturer",
                    "Value": str(row.sensor_type_manufacturer),
                },
                {"Name": "sensor_type_name", "Value": str(row.
    sensor_type_name)},
                {"Name": "sensor_type_id", "Value": str(row.
    sensor_type_id)},
            ]

            # Create a record for each measurement
            measuresValues = []
            for measure in row.sensordatavalues:
                measureValue = {
                    "Name": measure.value_type,
                    "Value": str(measure.value),
                    "Type": "DOUBLE",
                }
                measuresValues.append(measureValue)

                if measure.value_type == "P2" and row.aqi is not
    None:
                    aqi_measureValue = {
                        "Name": "aqi",
                        "Value": str(row.aqi),
                        "Type": "BIGINT",
                    }
                    measuresValues.append(aqi_measureValue)

            # Create a record for each sensor
            record = {
                "Dimensions": dimensions,
                "Time": row_timestamp,
                "TimeUnit": "MILLISECONDS",
                "MeasureName": "air_quality",
                "MeasureValueType": "MULTI",
                "MeasureValues": measuresValues,
            }
            records.append(record)
```

```

            # Write records to Timestream if there are 98 records
            if len(records) >= 98:
                write_records(
                    database_name, table_name, write_client, records
                )  # Write records to Timestream
                records = []  # Reset the records list
                time.sleep(1)  # Sleep for 1 second

        except Exception as e:
            print(f"Error processing row: {row}")
            print(f"Exception: {e}")

    # Write records to Timestream if there are any remaining records
    if len(records) > 100:
        while len(records) > 100:
            write_records(
                database_name, table_name, write_client, records[:99]
            )  # Write records to Timestream
            records = records[99:]  # Keep the remaining records
            time.sleep(1)  # Sleep for 1 second
    elif len(records) > 0:
        write_records(database_name, table_name, write_client, records)
```

Appendix B.E Scripts & Services Source Code

B.E.1 Scripts

Script used by the `get_iam_credentials` service to retrieve the IAM credentials from the metadata server.

```
#!/bin/bash

# Get the authentication token from the EC2 metadata service
TOKEN=$(curl -X PUT "http://169.254.169.254/latest/api/token" -H "X-aws-ec2-metadata-token-ttl-seconds: 21600" -s)

# Name of the IAM role to assume
ROLE_NAME="LabRole"

# Get temporary credentials using the IAM role
IAM_ROLE_CREDENTIALS=$(curl -H "X-aws-ec2-metadata-token: $TOKEN" -s http://169.254.169.254/latest/meta-data/iam/security-credentials/$ROLE_NAME)

# Extract the credentials and session token
AWS_ACCESS_KEY_ID=$(echo $IAM_ROLE_CREDENTIALS | jq -r .AccessKeyId)
```

```
AWS_SECRET_ACCESS_KEY=$(echo $IAM_ROLE_CREDENTIALS | jq -r .
    SecretAccessKey)
AWS_SESSION_TOKEN=$(echo $IAM_ROLE_CREDENTIALS | jq -r .Token)
AWS_DEFAULT_REGION="us-east-1"

# Export the credentials and session token
export AWS_ACCESS_KEY_ID
export AWS_SECRET_ACCESS_KEY
export AWS_SESSION_TOKEN
export AWS_DEFAULT_REGION
```

Script used by the `spark_python_job` service to run the Python Spark job.

```
#!/bin/bash

# Run the spark job in the background and log output to output.log
    file
nohup python3 /home/ubuntu/iot_project/ingestion_iot.py >/home/
    ubuntu/iot_project/output.log 2>&1 &
```

B.E.2 Services

Service used by the Ubuntu EC2 instance to retrieve the IAM credentials from the metadata server.

```
[Unit]
Description=Script to setup AWS cli thanks to the attached IAM
    Profile

[Service]
ExecStart=/usr/local/bin/get_iam_credentials.sh

[Install]
WantedBy=multi-user.target
```

Service used by the Ubuntu EC2 instance to run the Python Spark job (Data Collecting, Processing and Storing).

```
[Unit]
Description=Script to run the ingestion python script

[Service]
ExecStart=/usr/local/bin/start_spark_job.sh

[Install]
WantedBy=multi-user.target
```

Service used by the Linux EC2 instances to run the Grafana server (Data Distributing).

```
[Unit]
Description=Grafana instance
```

```
Documentation=http://docs.grafana.org
Wants=network-online.target
After=network-online.target
After=postgresql.service mariadb.service mysqld.service influxdb.service

[Service]
EnvironmentFile=/etc/sysconfig/grafana-server
User=grafana
Group=grafana
Type=notify
Restart=on-failure
WorkingDirectory=/usr/share/grafana
RuntimeDirectory=grafana
RuntimeDirectoryMode=0750
ExecStart=/usr/share/grafana/bin/grafana server \
                            --config=${CONF_FILE} \
                            --pidfile=${PID_FILE_DIR}/grafana-server.pid \
                            --packaging=rpm \
                            cfg:default.paths.logs=${LOG_DIR} \
                            cfg:default.paths.data=${DATA_DIR} \
                            cfg:default.paths.plugins=${PLUGINS_DIR} \
                            cfg:default.paths.provisioning=${PROVISIONING_CFG_DIR}
LimitNOFILE=10000
TimeoutStopSec=20
CapabilityBoundingSet=
DeviceAllow=
LockPersonality=true
MemoryDenyWriteExecute=false
NoNewPrivileges=true
PrivateDevices=true
PrivateTmp=true
ProtectClock=true
ProtectControlGroups=true
ProtectHome=true
ProtectHostname=true
ProtectKernelLogs=true
ProtectKernelModules=true
ProtectKernelTunables=true
ProtectProc=invisible
ProtectSystem=full
RemoveIPC=true
RestrictAddressFamilies=AF_INET AF_INET6 AF_UNIX
RestrictNamespaces=true
RestrictRealtime=true
RestrictSUIDSGID=true
SystemCallArchitectures=native
UMask=0027
```

```

[Install]
WantedBy=multi-user.target
```