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.

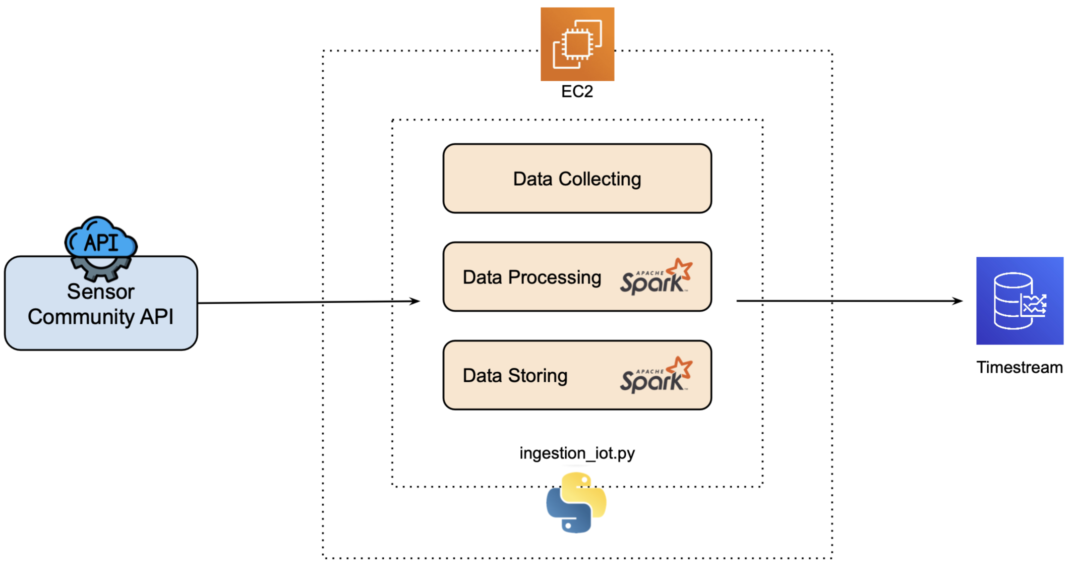


Figure 2.1: Data Collecting, Processing & Storing Pipeline Diagram

2.1.2 Choice of Database

This project uses a Timestream database. Time series databases are designed specifically for time-ordered data, and are ideal for processing information from sensors recorded at regular intervals. These systems offer remarkable efficiency in ingesting, storing and querying large quantities of time series data, facilitating complex analyses and rapid decision-making based on constantly updated data. Their optimised architecture for accessing and analysing data over different time periods is crucial for applications requiring real-time monitoring.

Timestream was chosen for our project because of its ability to process environmental data from IoT sensors quickly and accurately. Its ability to process large volumes of data means that the air quality index can be constantly updated and analysed. Its scalability guarantees flexible adaptation to variations in workload. What's more, its advanced security mechanisms ensure the protection of sensitive data, meeting the confidentiality and data sovereignty requirements of our project.

2.1.3 Query 3

The last task is to perfom KMeans clustering on the top 50 affected locations (depending on the slope of monthly confirmed cases) for each month. This task is implemented by the class **Query3** in Appendix D. Here are the steps performed by the task:

1. **Data Preparation (`__prepare_data method`):** The COVID data is prepared for analysis. A "Location" column is created by combining the values "Province/State" and "Country/Region" to consider the country if the state is not indicated.

2. **Pivoting the DataFrame (`__pivot_table method`):** The "covidDataDf" DataFrame initially contains dates in columns. This step reorganises the DataFrame so that the dates are row entries. It uses the stack function to transform each date column into a row, associated with its corresponding value. The result is a table with the following columns: "Location", "Date" and "Value"

3. **Computing Daily Confirmed Cases (`__compute_daily_confirmed_cases method`):** Daily confirmed cases are computed using the formula:

$$\text{DailyCases} = \begin{cases} 0, & \text{if PrevValue is null} \\ \text{Value} - \text{PrevValue}, & \text{otherwise} \end{cases} \tag{2.1}$$

   where `PrevValue` is the number of cases from the previous day.

4. **Computing Monthly Slopes (`__compute_monthly_slopes method`):** The slopes of monthly confirmed cases for each location are computed using linear regression. The slope formula used is:

$$\text{MonthlySlope} = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2} \tag{2.2}$$

   where:

- x is the number of months since the start of the dataset.
- y is the number of monthly confirmed cases.
- n is the number of observations.

5. **Filtering Top Affected Locations (`__filter_top_affected method`):** The top 50 affected locations are identified based on the slope of monthly confirmed cases. To do this, the data is grouped by location and the mean slope is calculated for each group. The top 50 locations are then selected.

6. **Applying Custom Clustering (`__apply_custom_clustering method`):** A custom KMeans algorithm is applied to cluster the locations based on their slopes. This source code was adapted from the following Medium article (1).

7. **Applying Clustering (`__apply_clustering method`):** The standard KMeans algorithm from Spark MLlib is used for clustering the locations.

2.2 Queries Optimisation

Once the 3 queries had been built, an optimisation step was carried out to improve task execution time. This section will look at the various difficulties encountered and how to overcome them.

2.2.1 Common Performance Problems in Spark

The five most common performance issues encountered in Apache Spark, known as the 5 Ss (2) are:

1. **Spill:** When there is not enough RAM memory to process the current data, Spark is forced to move some data to the hard disk, a process known as "spilling".

2. **Skew:** The "skew" problem arises when data is not distributed evenly across partitions in Spark. Some nodes end up with a much heavier workload than others, creating bottlenecks and slowing down overall processing.

3. **Shuffle:** During complex operations such as joins or clusters, Spark redistributes the data so that the corresponding elements are on the same node. Shuffling is costly in terms of performance because it involves intensive data transfer over the network.

4. **Storage:** Inefficient storage management, such as processing many small files or using non-optimised file formats, can lead to a high number of I/O operations, which slows down performance. In addition, a poor storage strategy can also lead to shuffling and asymmetry problems.

5. **Serialization:** "Serialization" i in Spark refers to the conversion of objects into a format that can be easily transmitted over the network or stored on disk. Inefficient serialization and deserialization processes can significantly slow down data transfer between cluster nodes and increase overall processing time.

2.2.2 Optimisation Solutions

Here is a list of techniques for optimising Apache Spark processes and overcoming performance problems (3):

1. **Efficient partitioning:** Efficient partitioning in Apache Spark is essential for the balanced distribution of data across cluster nodes. This distribution plays a crucial role in optimising performance, particularly for shuffle operations. By adjusting the number of partitions with repartition() or coalesce(), it is possible to balance the workload between nodes, which is effective for join and aggregation operations, for example.

2. **Persist/Cache and Early Filtering:** The judicious use of caching and early filtering goes hand in hand in optimising Spark processes. By using persist() or cache(), frequently accessed data is retained in memory, reducing the time required

for repeated operations. At the same time, early data filtering, before resource-intensive operations such as joins or shuffles, can significantly reduce the amount of data processed, speeding up the whole process.

3. **Data Format Choice:** The choice of data format is an often underestimated but crucial aspect of Spark optimisation. Columnar formats such as Parquet or ORC are preferable for efficient read and write operations. By choosing the right data format, you can significantly improve read performance and reduce the amount of storage space required.

4. **Minimising Shuffle and Broadcast Operations:** Minimising shuffle operations is crucial to improving performance in Spark. Shuffles, which are necessary for operations such as joins and groupings, can be costly in terms of performance because they involve moving large amounts of data around the network. In addition, broadcasting is a powerful technique for joins involving a large table and a smaller table.

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.

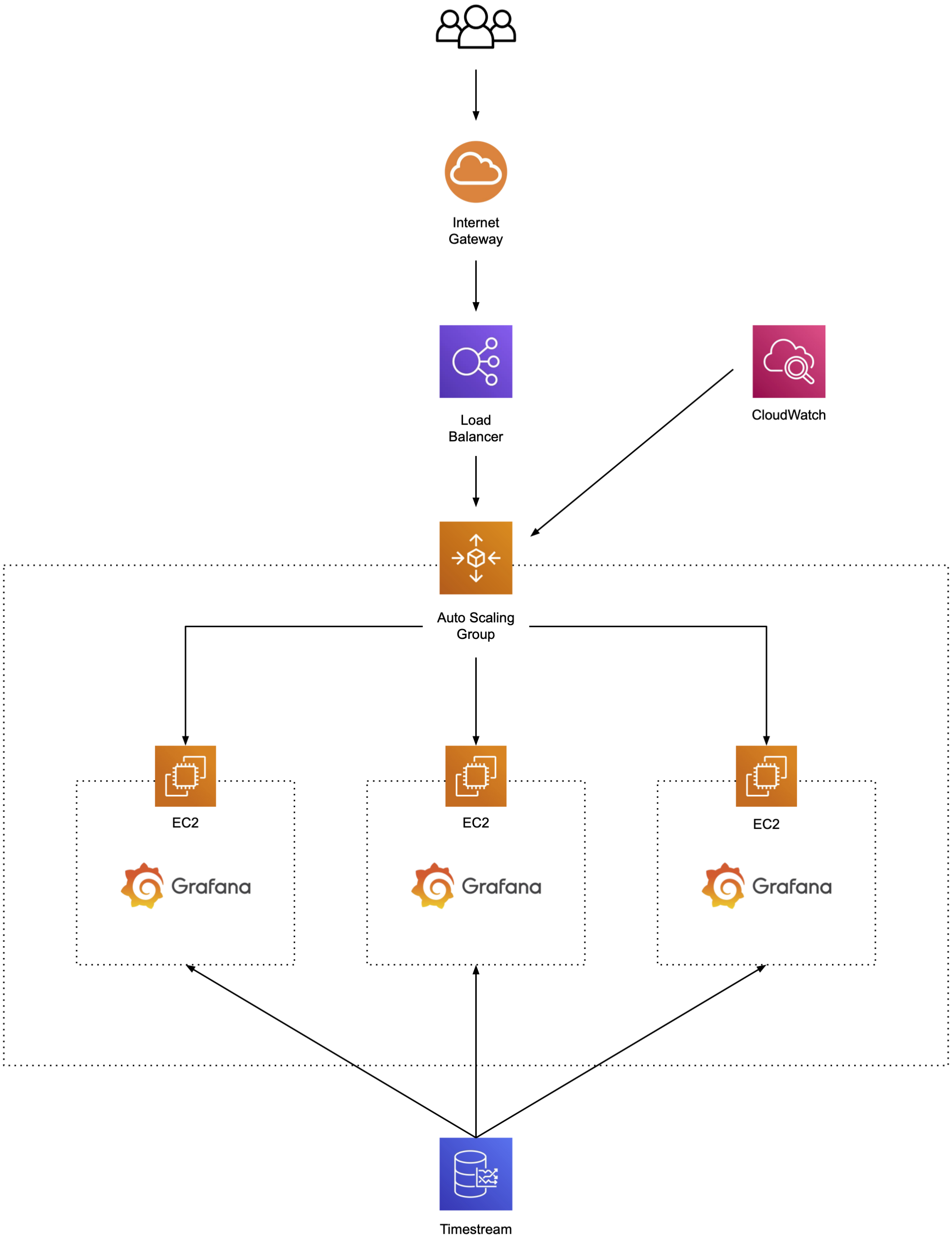


Figure 2.2: Data Distributing Pipeline Diagram

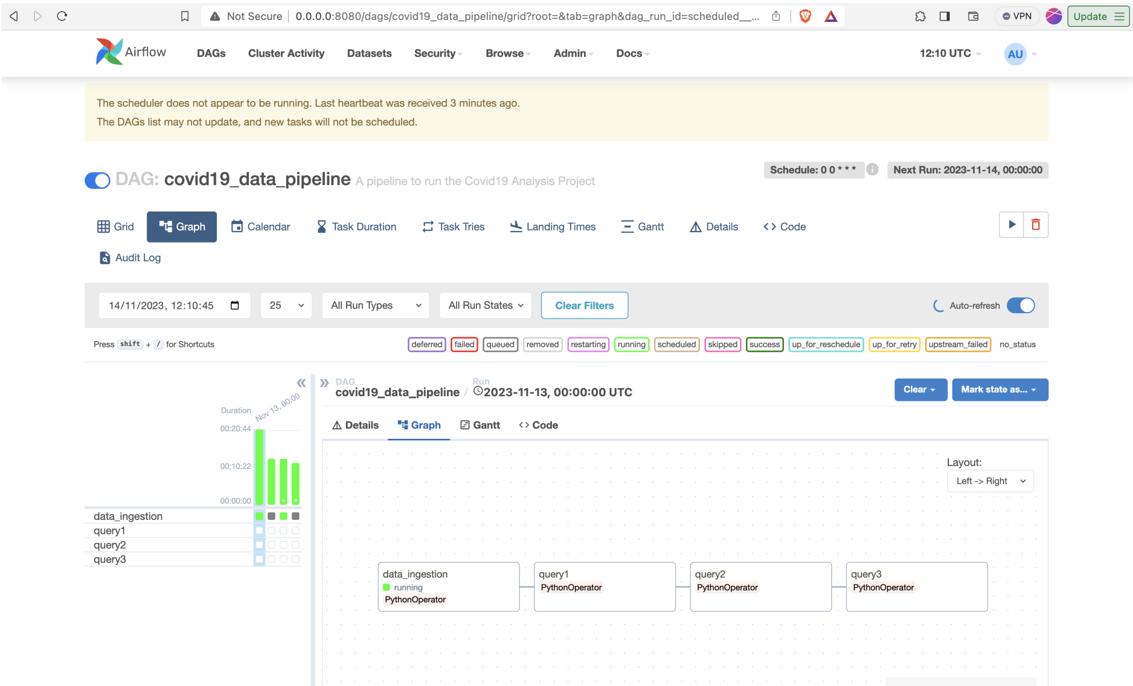


Figure 2.3: Apache Airflow DAG Graph

Chapter 3

Results & Discussion

3.1 Queries Results

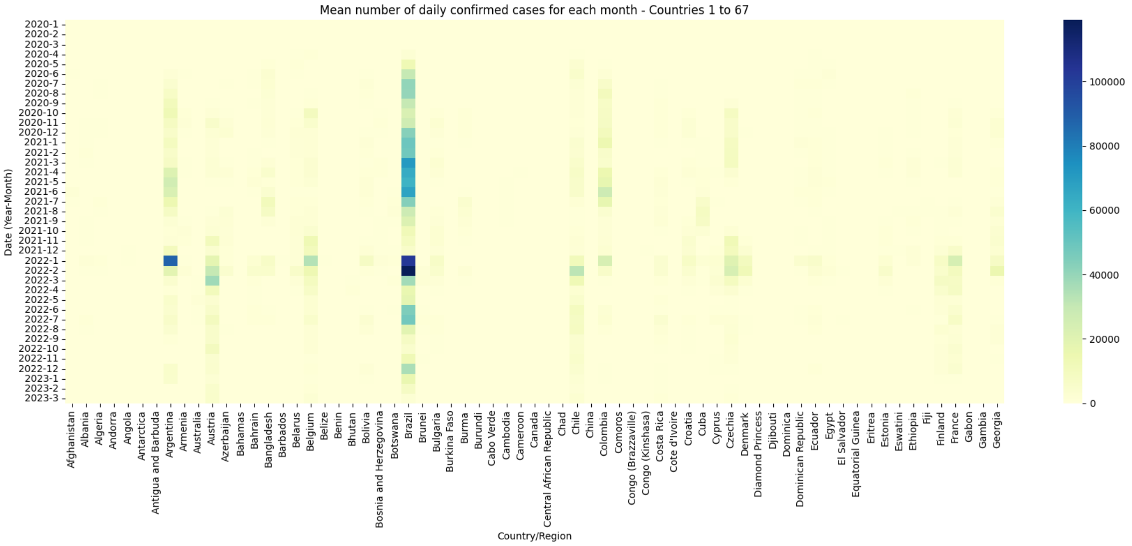
The programme was last run on 18 November 2023. The appendix E shows the output of the program on the terminal.

3.1.1 Query 1

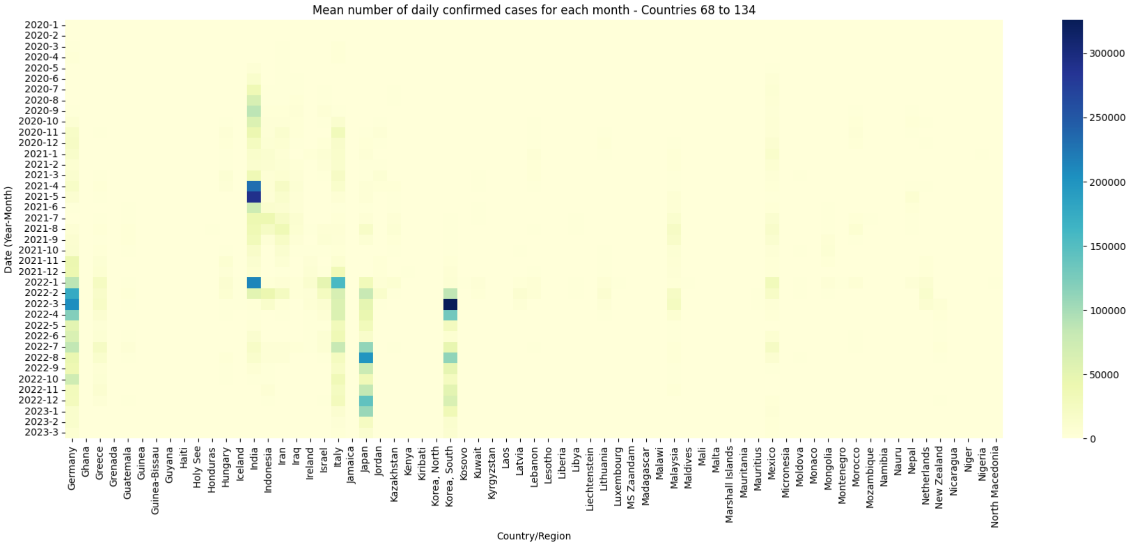
The first query takes around 3 seconds, and the table 3.1 shows a sample of the data calculated during the task. In order to evaluate performance, an equivalent script not using Spark was run. Execution time was 0.5 sec. The 3.2 section will cover this point. The results are consistent with what was expected (4). 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

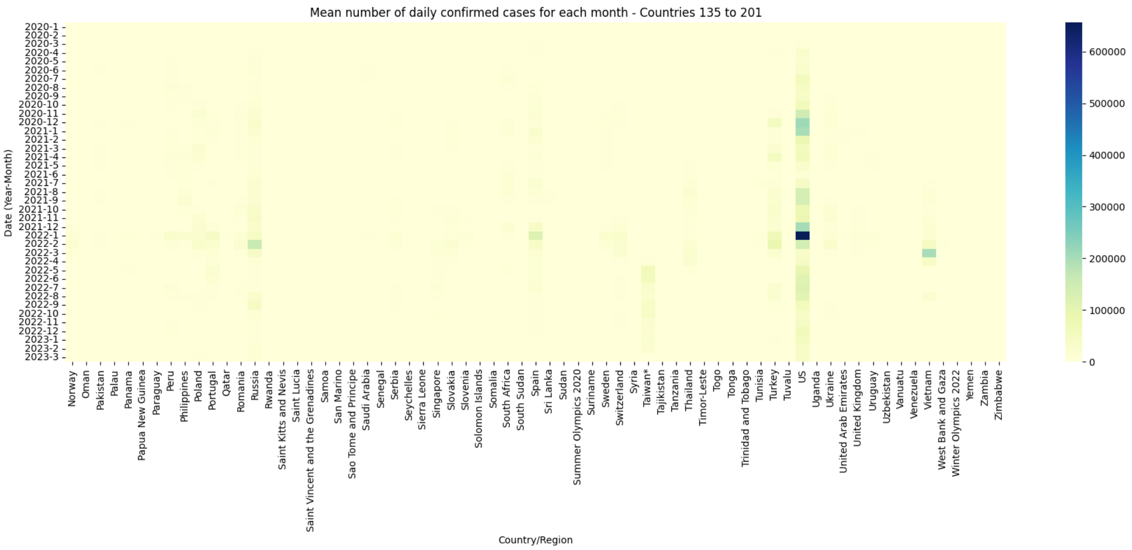
	Country/Region	Year	Month	Average
1	Afghanistan	2020	1	0.0
2	Afghanistan	2020	2	0.1724137931034483
3	Afghanistan	2020	3	5.193548387096774
4	Afghanistan	2020	4	55.36666666666667
5	Afghanistan	2020	5	430.741935483871



(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.

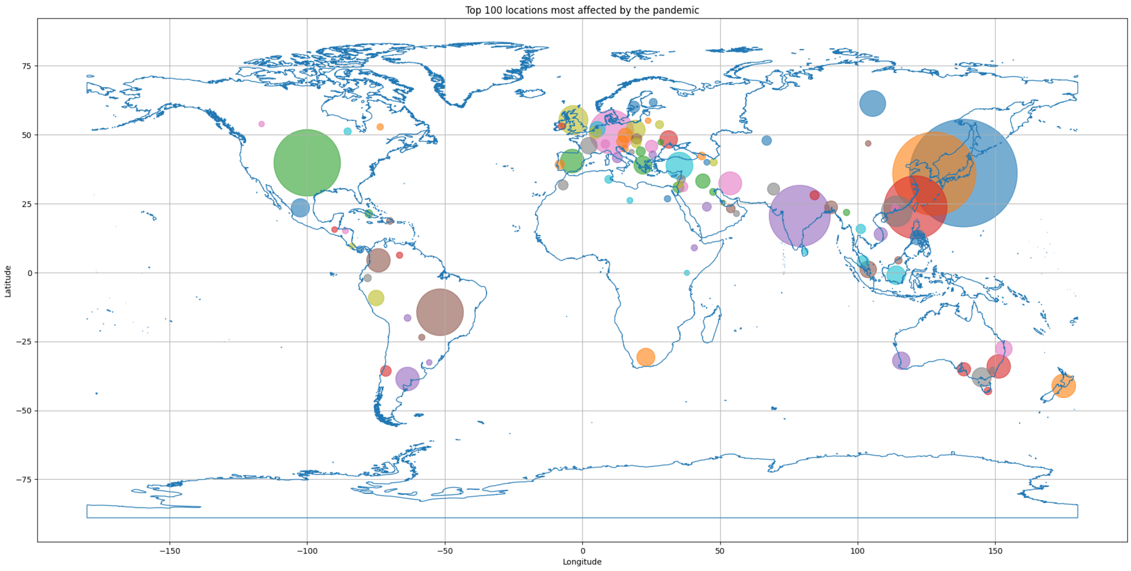


Figure 3.2: Top 100 Locations most affected by the pandemic

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 (4).

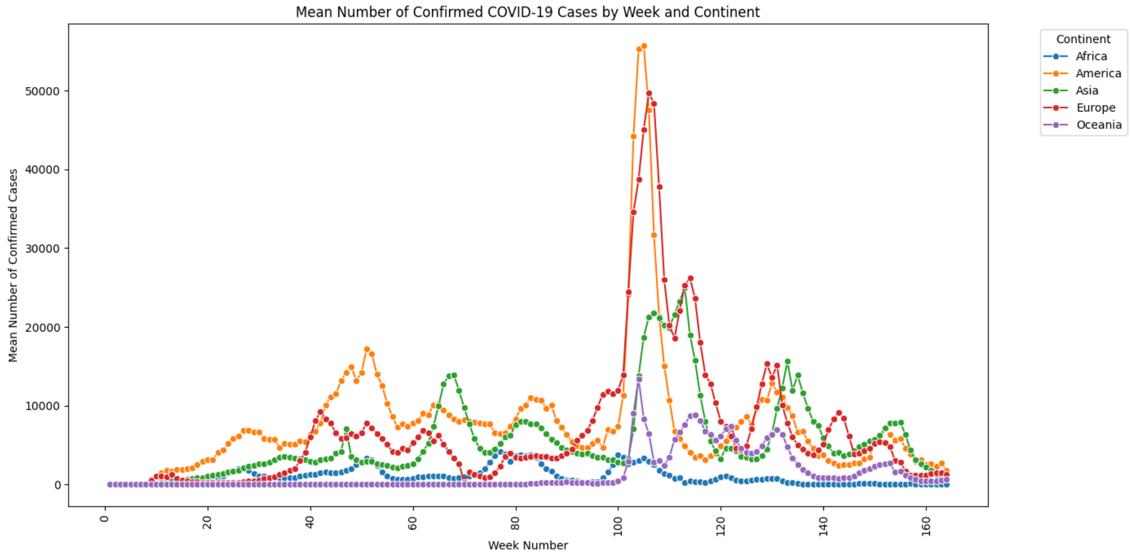


Figure 3.3: Mean Confimed Cases By Week and Continent

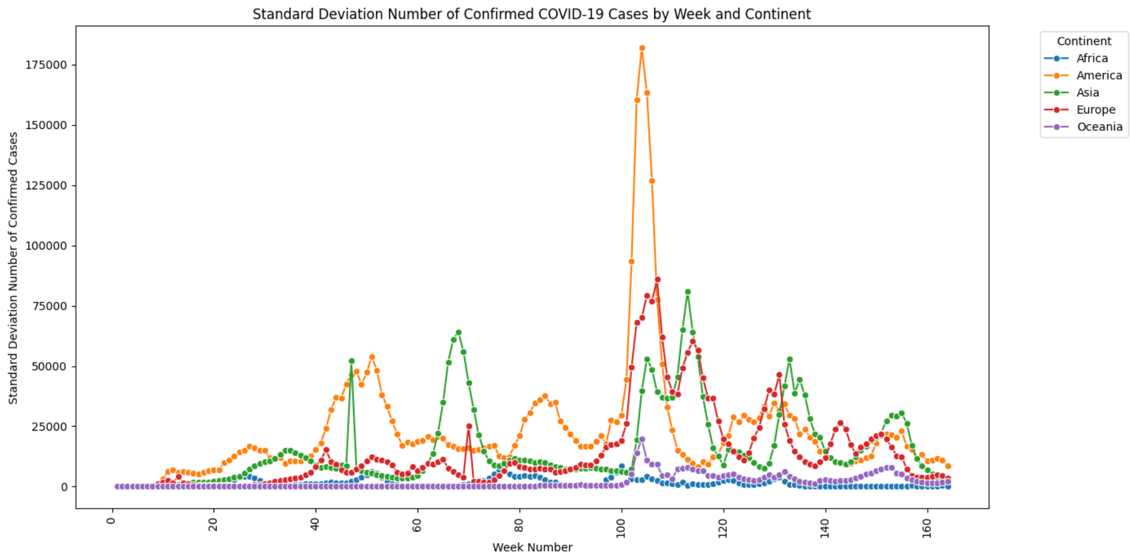


Figure 3.4: Standard Deviation Confimed Cases By Week and Continent

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

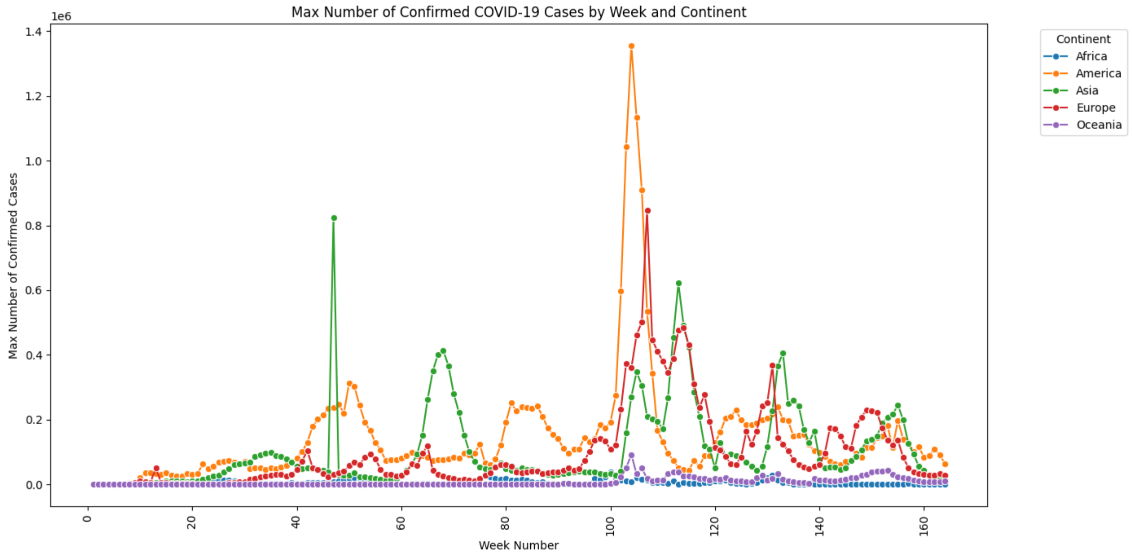


Figure 3.5: Maximum Confimed Cases By Week and Continent

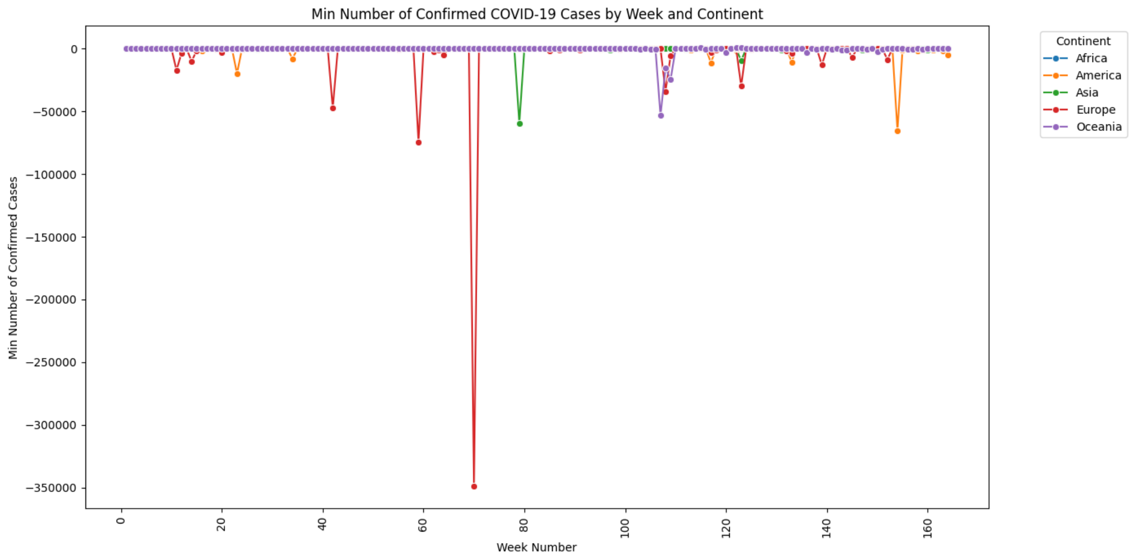


Figure 3.6: Minimum Confimed Cases By Week and Continent

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.

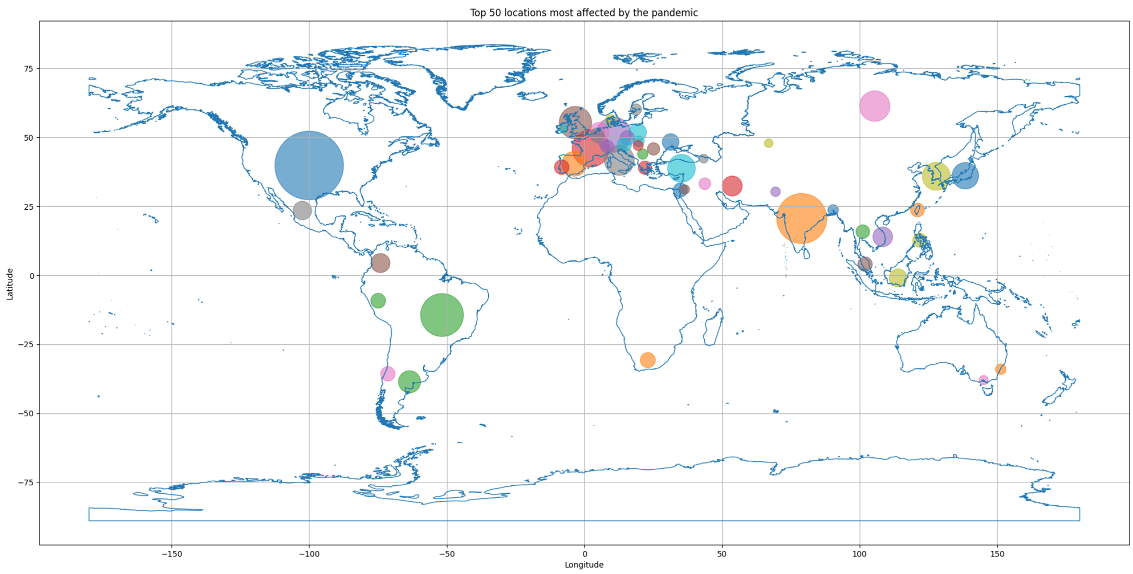


Figure 3.7: Top 50 Locations most affected by the pandemic

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.

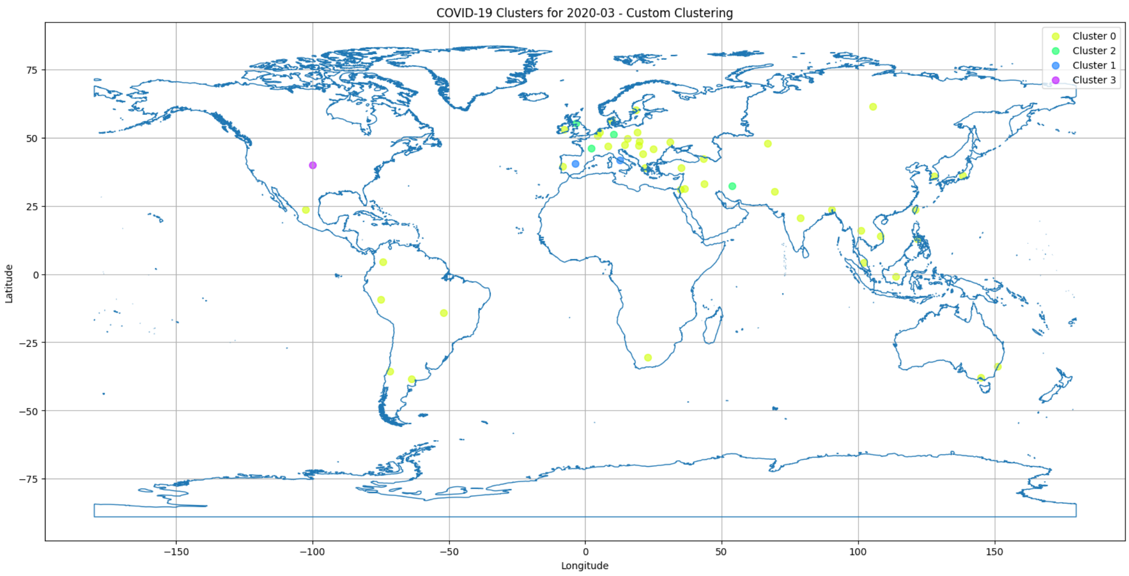


Figure 3.8: Custom KMeans Clustering on 03/2020

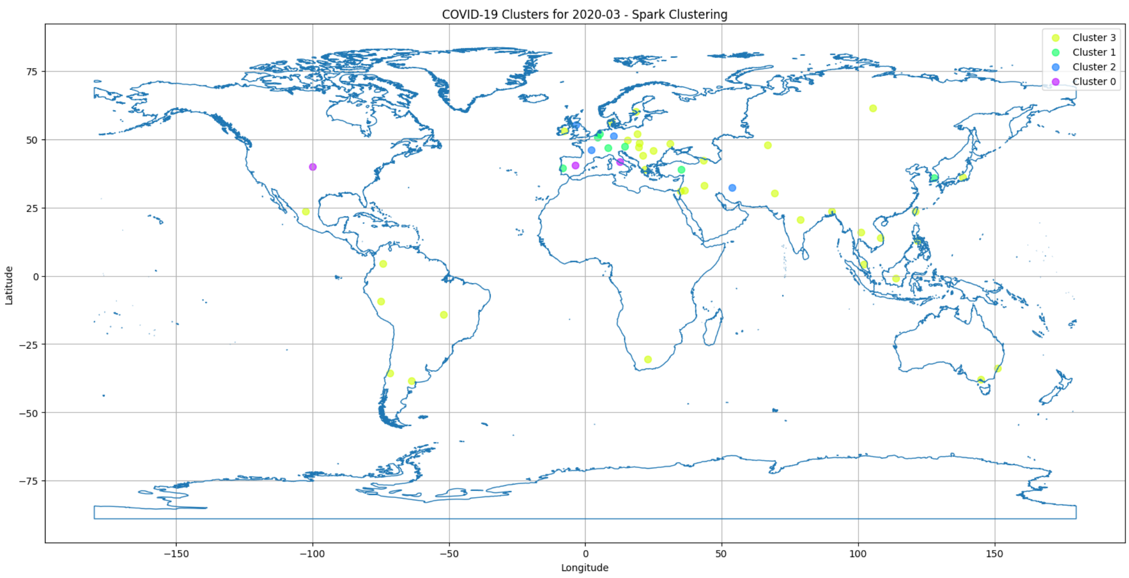


Figure 3.9: Spark MLlib KMeans Clustering on 03/2020

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

In the context of the COVID-19 pandemic, the use of Machine Learning and Big Data, while beneficial for crisis management, raises complex ethical issues and challenges.

Firstly, the management of health data, which is essential for tracking and preventing the spread of the virus, creates major risks for privacy. preventing the spread of the virus, creates major privacy risks. Massive data collection and analysis can lead to intrusive surveillance, where the boundaries between public security and individual privacy become blurred. This can constitute an infringement of the right to privacy. In addition, some countries have taken the opportunity to monitor their populations. For example, China has used facial recognition technology to track people's movements and apply quarantine measures.

Secondly, algorithmic biases represent a major challenge. Machine learning systems, although powerful, can incorporate and amplify existing biases in the data. In the context of the pandemic, this could result in inequitable distribution of medical resources, biased diagnoses or discriminatory public health policies, disproportionately affecting certain social or ethnic groups.

Thirdly, transparency and accountability are crucial issues. Complex and often opaque algorithms make it difficult to understand and challenge AI-based decisions. This raises questions of governance and regulation: who is responsible for errors or harm caused by automated decisions? How can these technologies be effectively controlled and regulated?

Finally, there is the overall challenge of striking a balance between the benefits of using these technologies and respect for fundamental rights. Striking the right balance between effective management of the pandemic and protection of individual freedoms is a delicate exercise, requiring careful ethical reflection and judicious regulation.

Chapter 4

Conclusion

This study demonstrated the effectiveness of Big Data and Machine Learning technologies in the analysis and management of the COVID-19 pandemic. By exploiting the potential of Apache Spark to process large datasets, the analysis revealed significant trends in the spread of the virus, providing essential information for public health decisions. The methodologies adopted provided an in-depth understanding of the dynamics of the pandemic, highlighting geographical and temporal variations in confirmed cases. At the same time, the ethical challenges raised, such as privacy protection and algorithmic fairness, underline the need for a balanced and regulated approach to the use of information technologies. In short, this project illustrates how the judicious integration of AI and Big Data can transform our response to health emergencies, while reminding us of the importance of ethical responsibility in the application of technological advances.

References

1. Dabbura I. K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks. Towards Data Science. 2018 Sep. Available at: https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a. (Accessed: November 8, 2023).

2. Ippolito PP. Apache Spark Optimization Techniques. Towards Data Science. 2023 Jan. Available at: https://towardsdatascience.com/apache-spark-optimization-techniques-fa7f20a9a2cf. (Accessed: November 8, 2023).

3. (NNK) N. Spark Performance Tuning & Best Practices; 2023. Available at: https://sparkbyexamples.com/spark/spark-performance-tuning/. Last modified: February 7, 2023, (Accessed: November 8, 2023).

4. The New York Times. Coronavirus World Map: Tracking the Global Outbreak. The New York Times; 2021. Available at: https://www.nytimes.com/interactive/2021/world/covid-cases.html. Last modified: March 10, 2023, (Accessed: November 10, 2023).

Appendix A

Documentation

1. 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
```

2. 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).

3. Detailed Features of Classes and Functions

Classes

`Spark`: Class to create a Spark session and load the data set into a DataFrame. This class has the following methods:

- `__init__(self, appName, master)`: Constructor to initialise the class with an application name and a master URL.
- `getSpark(self)`: Method to return the Spark session.
- `getSparkDf(self, path)`: Method to load the data set into a DataFrame and return it, given a path.
- `stopSpark(self)`: Method to stop the Spark session.

`Query1`: Class to perform Query 1, which is to calculate the average number of confirmed daily cases of people infected with COVID-19 for each country in the dataset, and save the results to a CSV file, using Spark. This class has the following methods:

- `__init__(self, sparkSession, covidDataDf)`: Constructor to initialise the class with a Spark session and a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

`Query2`: Class to perform Query 2, which is to calculate the mean, standard deviation, minimum and maximum of the number of confirmed cases daily for each week and continent, and save the results to a CSV file, using Spark. This class has the following methods:

- `__init__(self, sparkSession, covidDataDf)`: Constructor to initialise the class with a Spark session and a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

`Query3`: Class to perform Query 3, which is to perform clustering on the top 50 affected locations based on the maximum slope of monthly confirmed cases, and save the results to a CSV file, using Spark. This class has the following methods:

- `__init__(self, sparkSession, covidDataDf)`: Constructor to initialise the class with a Spark session and a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

`CustomKMeans`: Class to perform a custom KMeans clustering algorithm. This class has the following methods:

- `__init__(self, k, max_iter)`: Constructor to initialise the class with the number of clusters and the maximum number of iterations.
- `fit(self, X)`: Method to fit the model to the data.
- `predict(self, X)`: Method to predict the cluster of each data point.

`CustomQuery1`: Class to perform Query1, using Pandas without Spark. This class has the following methods:

- `__init__(self, covidDataDf)`: Constructor to initialise the class with a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

`CustomQuery2`: Class to perform Query2, using Pandas without Spark. This class has the following methods:

- `__init__(self, covidDataDf)`: Constructor to initialise the class with a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

Functions

`ingestion.py`

- `fetch_covid_data()`: Function to ingest the data set from the source and store it in the data lake.

`utils.py`

- `getContinentByCoordinates(longitude, latitude)`: Function used by CustomQuery2 class to assign a continent to a location based on its coordinates.
- `compute_slope(row)`: Function used by CustomQuery2 class to compute the slope of a location.
- `format_date_range(date)`: Function used by CustomQuery2 class to format a date range.

Appendix B

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 C

Data Collecting Source Code

```python
# 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 D

Data Processing Source Code

```python
# 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


# 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 E

Data Storing Source Code

```python
# 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
    )

```

```python
    # 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 F

Scripts & Services

Appendix F.A 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 &
```

Appendix F.B 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
```