Alexis Balayre

Machine learning & Big Data Assignment

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

MSc
Academic Year: 2023 - 2024

Supervisor: Dr Jun Li
20th November 2023

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the global COVID-19 pandemic, nations took advantage of Big Data Analytics and Artificial Intelligence technologies to understand and combat the spread of the virus. This report focuses on the detailed analysis of a comprehensive global dataset of confirmed cases of COVID-19, leveraging Apache Spark, an advanced data processing framework. The main objective is to provide a clear picture of the evolution of the pandemic on a global scale. The aim is to understand not only the speed and patterns of spread of the virus. It also aims to demonstrate how Big Data tools can be used to transform, analyse and interpret massive volumes of data, offering a unique and valuable perspective in the management of global health crises.

Chapter 2

Dataset Description

COVID-19 Dataset

As shown in table 2.1, the "time_series_covid19_confirmed_global" dataset presents each row with key identifiers such as "province/state" and "country/region", as well as geographical coordinates (latitude and longitude). Importantly, it traces the progression of the pandemic since its inception, with each column dated in mm/dd/yy format, revealing the total number of confirmed cases at that date. This cumulative data structure enables a dynamic and detailed analysis of the spread of the virus from 22 January 2020 to 9 March 2023. Moreover, the dataset contains 201 distinct Countries/Regions, 92 distinct Provinces/States, and 198 null values for this last as shown in tables 2.2 and 2.3.

Table 2.1: COVID-19 Dataset Sample

	Province/State	Country/Region	Lat	Long	1/22/20	...	3/9/23
1	NaN	Afghanistan	33.93911	67.709953	0		209451
2	NaN	Albania	41.15330	20.168300	0		334457
3	NaN	Algeria	28.03390	1.659600	0		271496
4	NaN	Andorra	42.50630	1.521800	0		47890
5	NaN	Angola	-11.20270	17.873900	0		105288

Table 2.2: COVID-19 Dataset Statistics

Column	Number of entries
Distinct Province/State	92
Distinct Country/Region	201

Table 2.3: COVID-19 Dataset Null Values

Column	Number of entries
Province/State	198
Lat	2
Long	2

Chapter 3

Methodologies

3.1 Queries Tasks

3.1.1 Query 1

The first task is to calculate the average number of confirmed daily cases of people infected with COVID-19 for each country in the dataset. This task is implemented by the class **Query1** in Appendix B. The steps of the task are as follows:

1. **Pivoting the DataFrame:** 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: "Country/Region", "Date" and "Value"

2. **Formatting Date Column:** The "Date" column is converted from a character string to an appropriate date format, in this case "M/d/yy".

3. **Calculating Daily Cases:** A specification window separates the data by country and sorts it by date. Daily cases are calculated by subtracting the number of cases from the previous day from the current day's cases. The formula used is:

$$\text{DailyCases} = \begin{cases} 0, & \text{if PrevValue is null} \\ \text{Value} - \text{PrevValue}, & \text{otherwise} \end{cases} \tag{3.1}$$

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

4. **Grouping and Averaging Daily Cases:** The data is grouped by "Country", "Year" and "Month". For each group, the average number of cases per day is calculated. This gives the average number of confirmed cases per day for each month and country.

3.1.2 Query 2

The second task is to calculate the mean, standard deviation, minimum and maximum of the number of confirmed cases daily for each week and continent. This task is implemented by the class **Query2** in Appendix C. 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. **Assigning Continents (`__assign_continent method`):** Continents are assigned to each line of data on the basis of geographical coordinates. To do this, the method uses the GeoPandas "world_boundaries_gdf" DataFrame, loaded from a shapefile (1) containing continental boundaries, to identify the continent corresponding to a set of longitude and latitude coordinates. The method implements an internal determine_continent function, which searches for the matching continent by checking whether a geographic point (formed by its coordinates) lies within the boundaries of one of the continents defined in "world_boundaries_gdf". If no match is found in this DataFrame, the method applies predefined geographic conditions (2) to approximate the continent. Finally, this logic is applied to each row of the covidDataDf DataFrame to add the new "Continent" column. This method was created using a GeoPandas Tutorial (3).

3. **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", "Continent", "Date" and "Value"

4. **Computing Daily Confirmed Cases (`__compute_daily_confirmed_cases method`):** A window specification partitions the data by "Location" and orders it by "Date". Daily confirmed cases are calculated using the formula:

$$\text{DailyCases} = \begin{cases} 0, & \text{if PrevValue is null} \\ \text{Value} - \text{PrevValue}, & \text{otherwise} \end{cases} \tag{3.2}$$

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

5. **Computing Slopes (`__compute_slopes method`):** This method computes the slopes of daily confirmed cases for each location, preparing the data for linear regression. The slope is calculated using the formula:

$$\text{DailySlope} = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2} \tag{3.3}$$

   where:

   - x is the number of days since the start of the dataset.
   - y is the number of daily confirmed cases.
   - n is the number of observations.

6. **Filtering Top Affected Locations (`__filter_top_affected method`):** The top 100 affected locations are identified based on the slope of daily confirmed cases. To do this, the data is grouped by location and the maximum slope is calculated for each group. The top 100 locations are then selected.

7. **Aggregating Data (`__aggregate method`):** For each entry, a new "WeekRange" column is added with the start and end dates of the week on which the day in the "Date" column falls. The DataFrame is then grouped by continent and week. Statistical measures like mean, standard deviation, minimum, and maximum of daily cases are calculated for each group.

3.1.3 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:

$$\text{DailyCases} = \begin{cases} 0, & \text{if PrevValue is null} \\ \text{Value} - \text{PrevValue}, & \text{otherwise} \end{cases} \tag{3.4}$$

   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:

$$\text{MonthlySlope} = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2} \tag{3.5}$$

   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 (4).

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

3.2 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.

3.2.1 Common Performance Problems in Spark

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

3.2.2 Optimisation Solutions

Here is a list of techniques for optimising Apache Spark processes and overcoming performance problems (6):

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.

3.3 Pipeline of the Project

3.3.1 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.

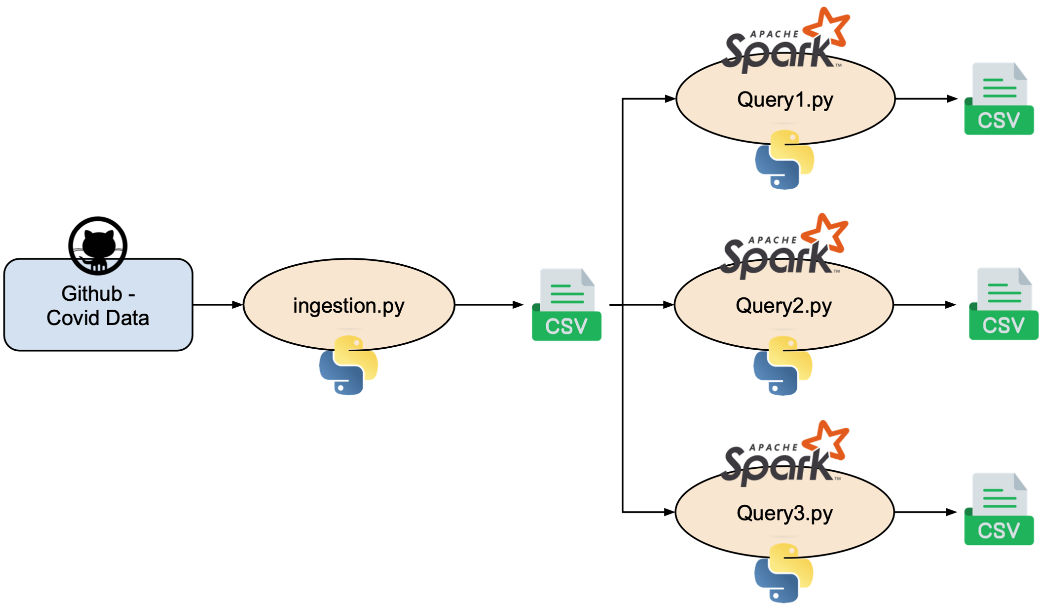


Figure 3.1: Pipeline Diagram

3.3.2 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 3.2 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.

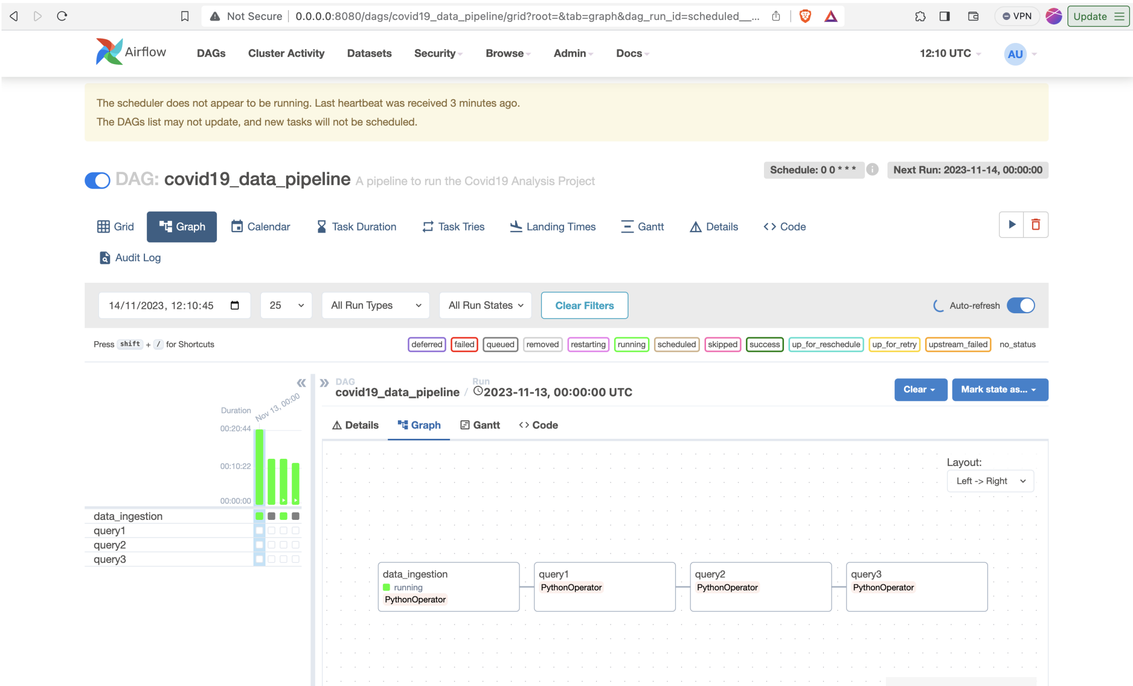


Figure 3.2: Apache Airflow DAG Graph

Chapter 4

Results & Discussion

4.1 Queries Results

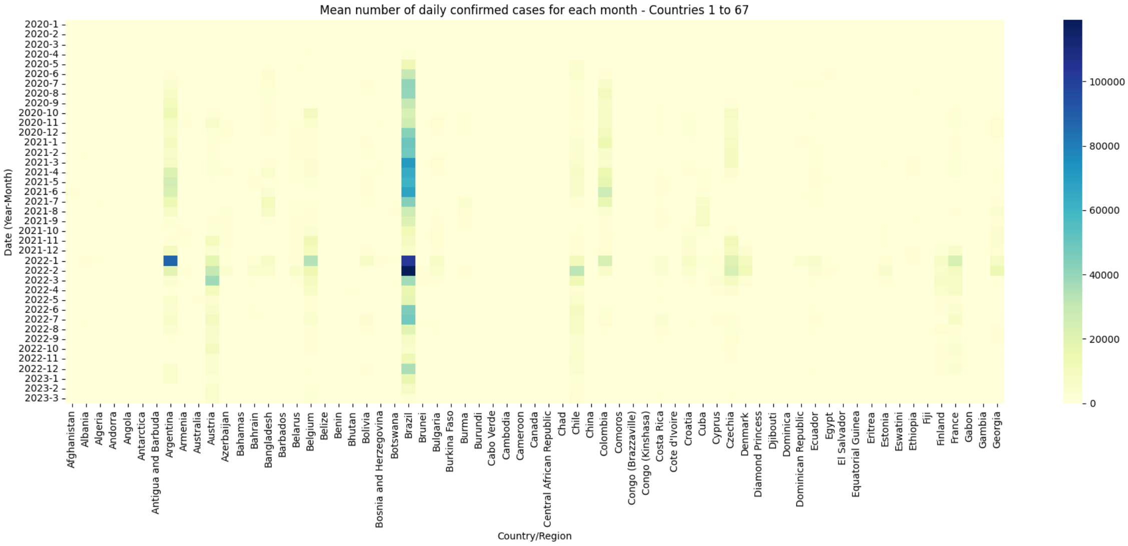
The programme was last run on 18 November 2023. The appendix E shows the output of the program on the terminal.

4.1.1 Query 1

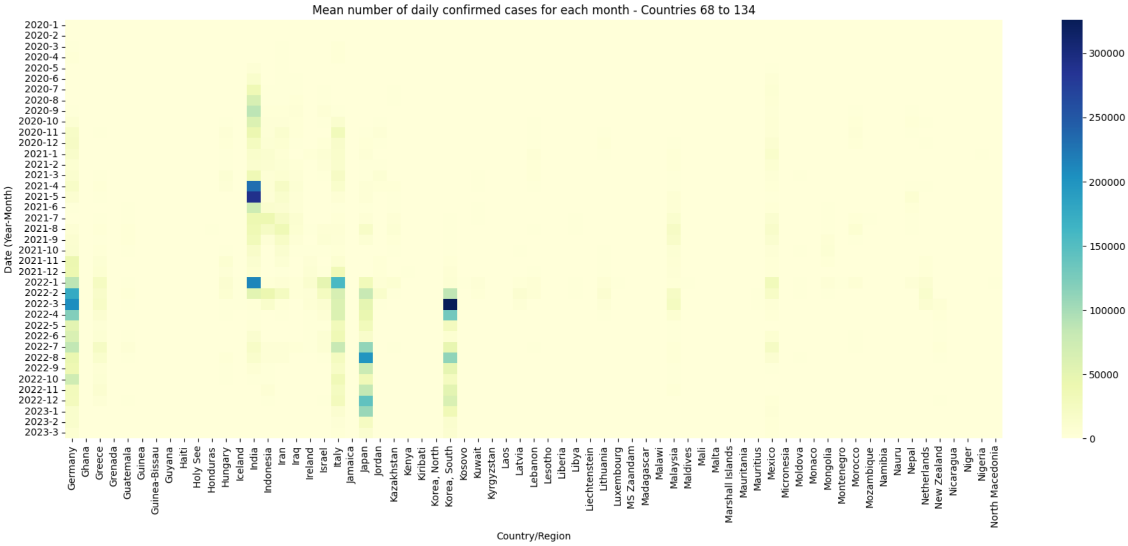
The first query takes around 3 seconds, and the table 4.1 shows a sample of the data calculated during the task. In order to evaluate performance, an equivalent script not using Spark was run. Execution time was 0.5 sec. The 4.2 section will cover this point. The results are consistent with what was expected (7). For example, the figure 4.1a shows that Brazil was heavily impacted by the pandemic, reaching an average peak in February 2022. The same is true for Korea in figure 4.1b and the United States in figure 4.1c, which were heavily impacted.

Table 4.1: Query 1 Results Sample

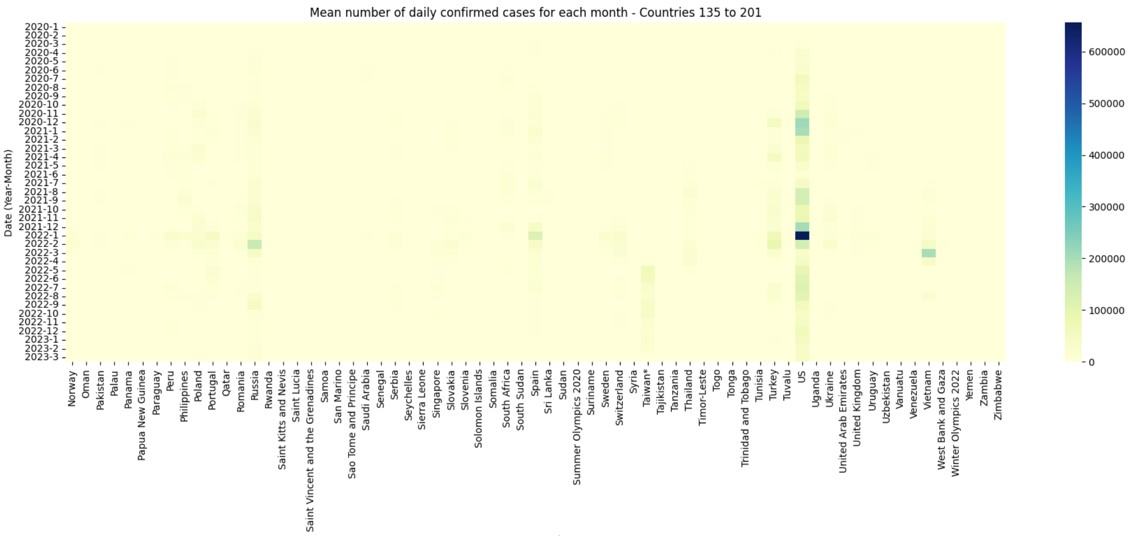
	Country/Region	Year	Month	Average
1	Afghanistan	2020	1	0.0
2	Afghanistan	2020	2	0.1724137931034483
3	Afghanistan	2020	3	5.193548387096774
4	Afghanistan	2020	4	55.36666666666667
5	Afghanistan	2020	5	430.741935483871



(a) Countries 1 to 67



(b) Countries 68 to 134



(c) Countries 135 to 201

Figure 4.1: Mean Daily Confirmed Cases Per Month

4.1.2 Query 2

The second query takes around 15 seconds, and the table 4.2 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 4.2 section will cover this point. Locations used to compute the statistics are shown on the map of figure 4.2. The area of the circles is proportional to how the location has been affected by the pandemic.

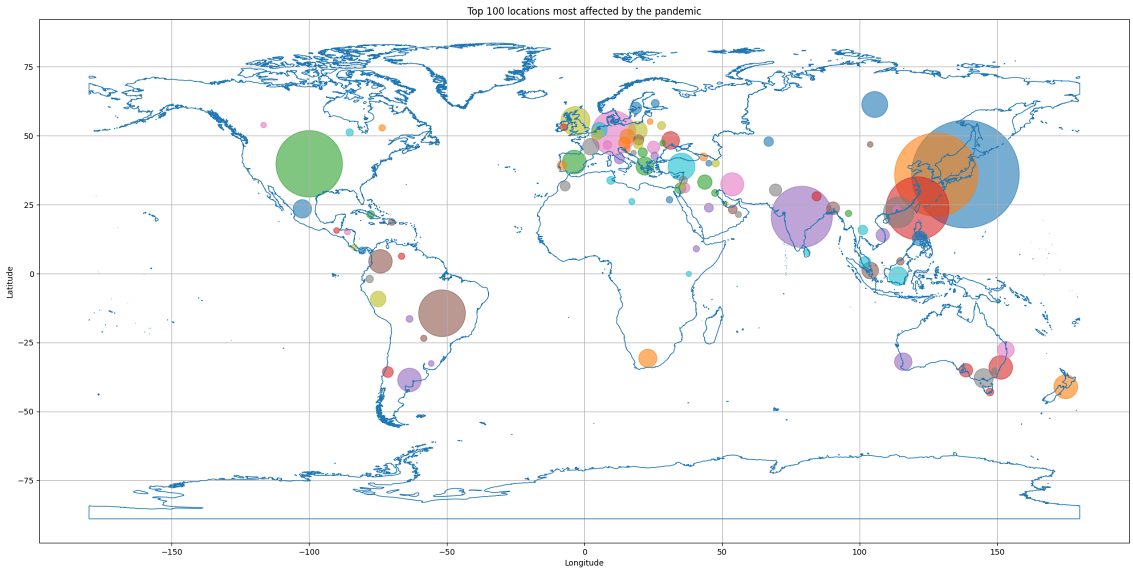


Figure 4.2: Top 100 Locations most affected by the pandemic

Table 4.2: 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 4.3 and 4.4 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 (7).

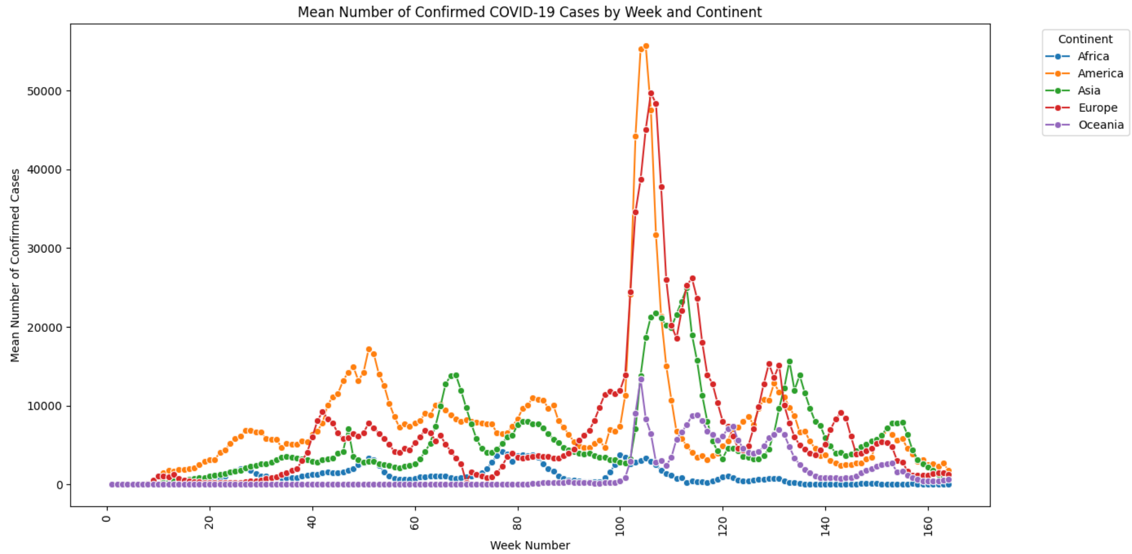


Figure 4.3: Mean Confimed Cases By Week and Continent

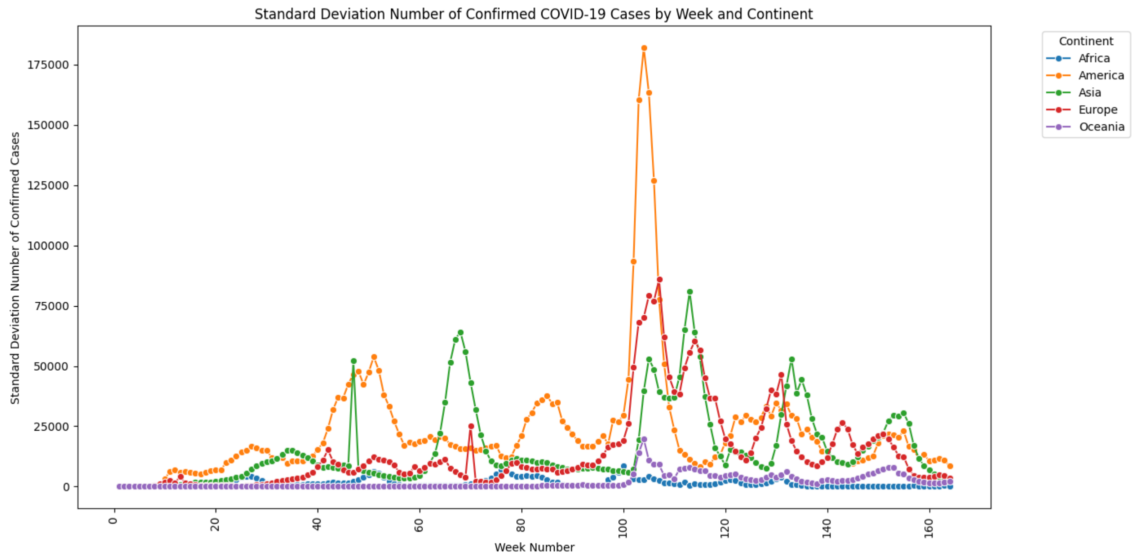


Figure 4.4: Standard Deviation Confimed Cases By Week and Continent

The figures 4.5 and 4.6 show the maximum and minimum of the number of confirmed cases by week and continent.

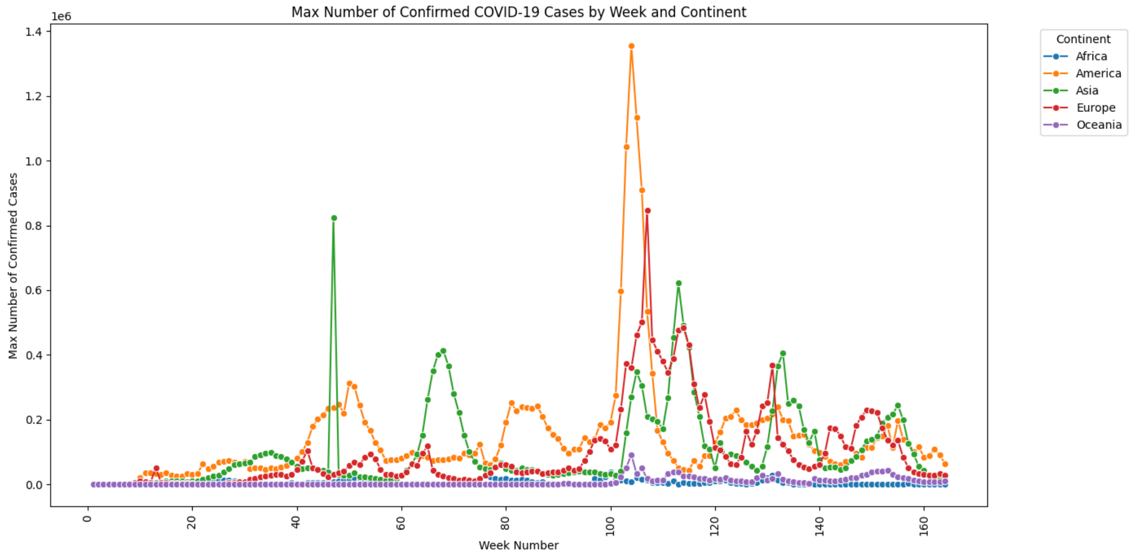


Figure 4.5: Maximum Confimed Cases By Week and Continent

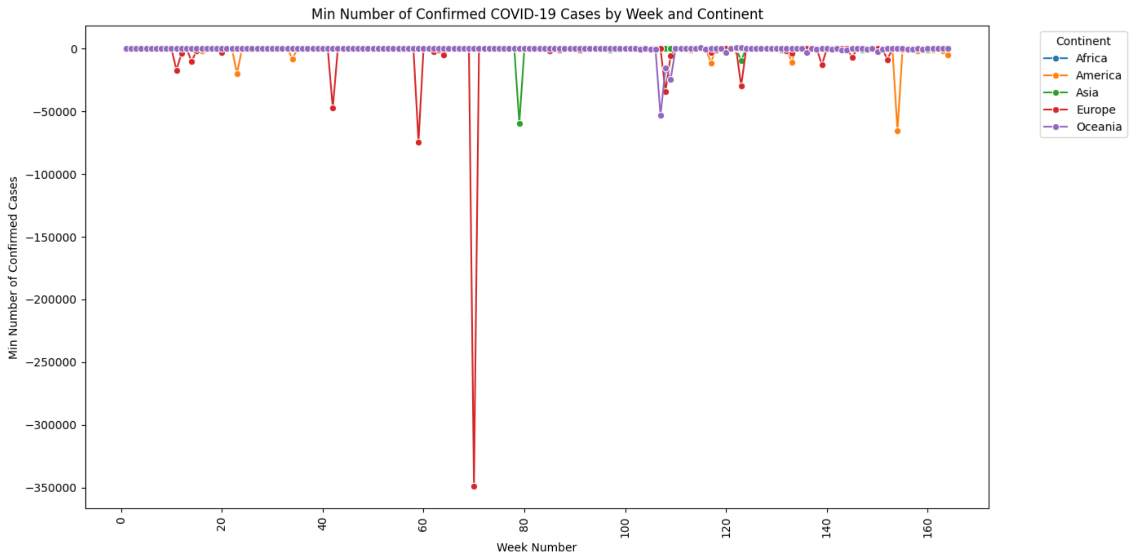


Figure 4.6: Minimum Confimed Cases By Week and Continent

4.1.3 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 4.3 shows a sample of the data calculated during the task with the custom implementation and the table 4.4 with the Spark MLlib implementation. Locations used to compute the statistics are shown on the map of figure 4.7. The area of the circles is proportional to how the location has been affected by the pandemic.

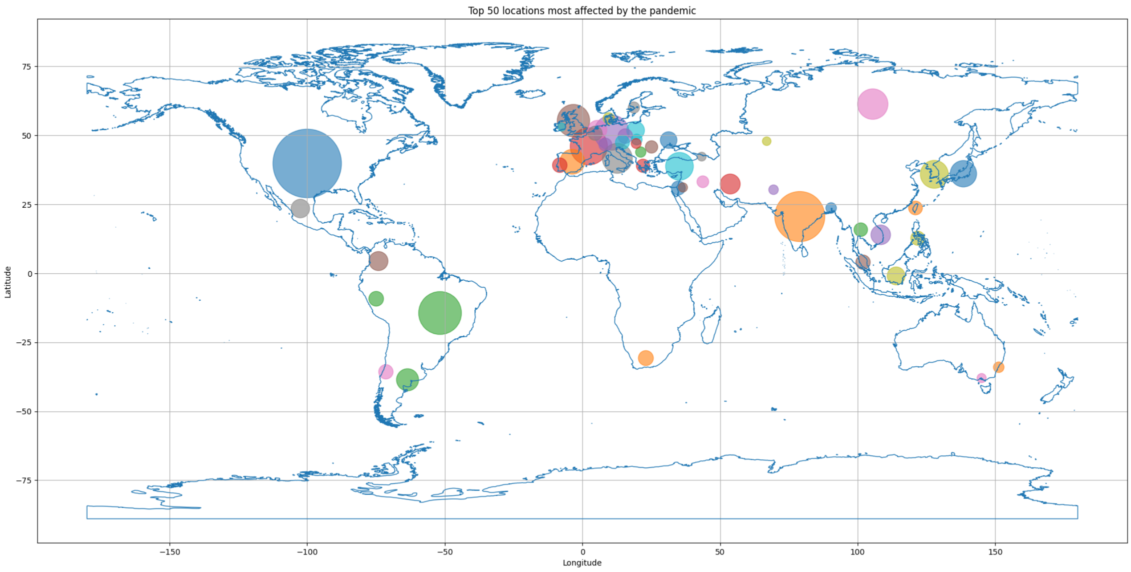


Figure 4.7: Top 50 Locations most affected by the pandemic

Table 4.3: 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 4.4: 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 4.8 and 4.9 show the clusters of the top 50 locations most affected by the pandemic in March 2020. The clusters are represented by different colours.

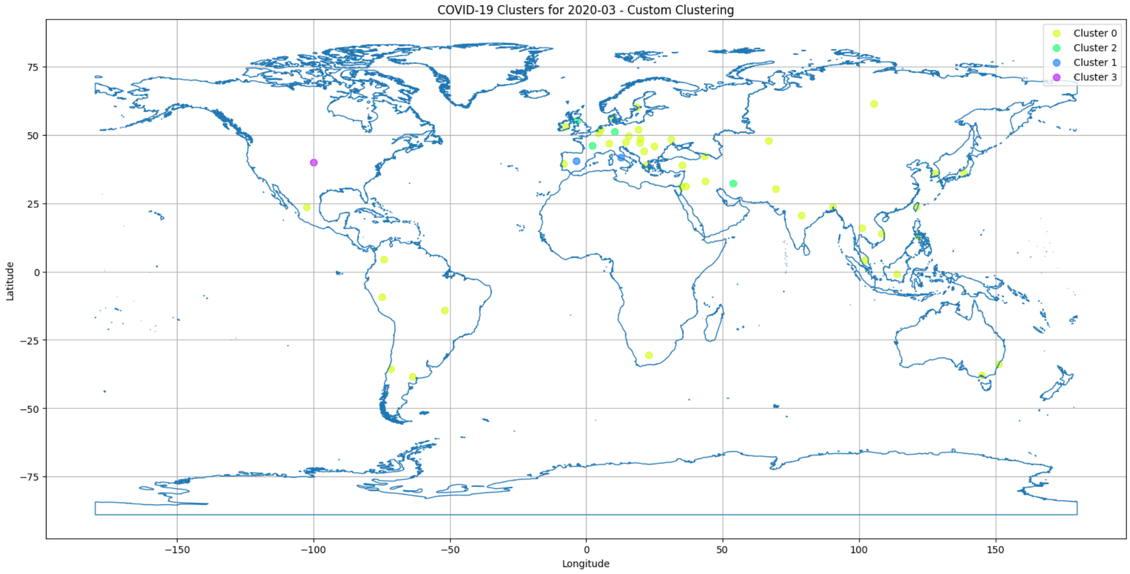


Figure 4.8: Custom KMeans Clustering on 03/2020

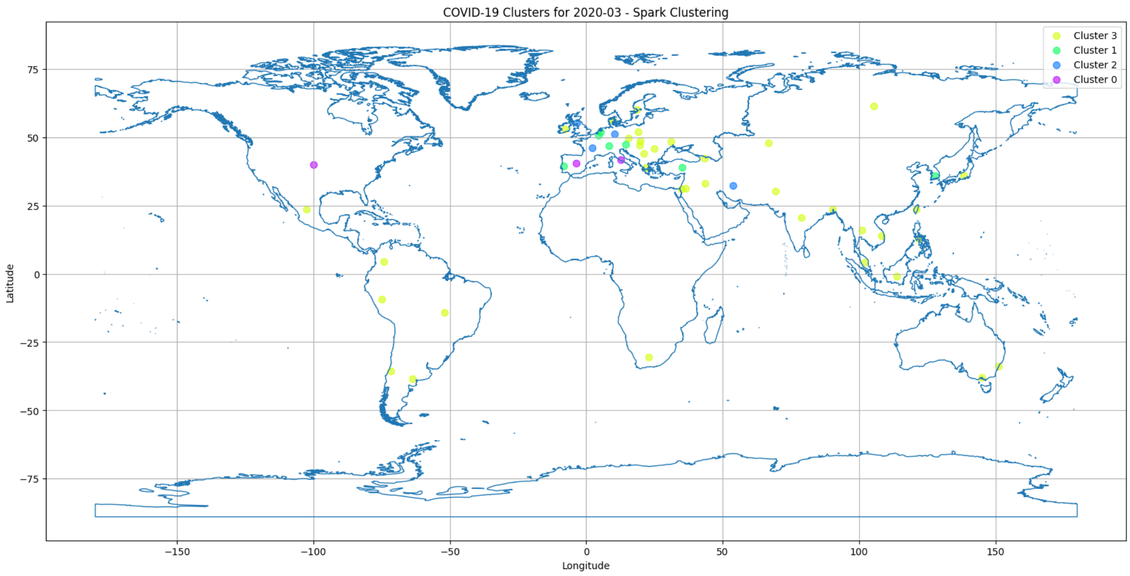


Figure 4.9: Spark MLlib KMeans Clustering on 03/2020

4.2 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.

4.3 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 5

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. ESRI. World Continents. ArcGIS Hub; 2013. Available at: https://hub.arcgis.com/datasets/esri::world-continents/about. Last Updated: February 9, 2023, (Accessed: October 29, 2023).

2. GISGeography. World Map with Latitudes and Longitudes; 2023. Available at: https://gisgeography.com/world-map-with-latitudes-and-longitudes/. Last Updated: October 29, 2023, (Accessed: October 29, 2023).

3. Luna JC. GeoPandas Tutorial: An Introduction to Geospatial Analysis. DataCamp; 2023. Available at: https://www.datacamp.com/tutorial/geopandas-tutorial-geospatial-analysis. (Accessed: October 29, 2023).

4. 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).

5. 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).

6. (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).

7. 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

```
Covid19AnalysisProjectLib/
    NonOptimised/
        CustomKMeans.py
        CustomQuery1.py
        CustomQuery2.py
        utils.py
    Queries/
        Query1.py
        Query2.py
        Query3.py
    ingestion.py
    Spark.py
dags/
    covid19AnalysisProject.py
data/
    covid_data/
        <last-ingestion-date>.csv
    World_Continents.zip
results/
    figures/
        mean-daily-confirmed-cases-per-month-1.png
        mean-daily-confirmed-cases-per-month-2.png
        mean-daily-confirmed-cases-per-month-3.png
        max-confirmed-cases-by-week-and-continent.png
        mean-confirmed-cases-by-week-and-continent.png
        min-confirmed-cases-by-week-and-continent.png
        std-confirmed-cases-by-week-and-continent.png
        covid-19-clusters-custom.png
        covid-19-clusters-spark.png
    query1/
        mean_confirmed_cases_per_month_pd.csv
        mean_daily_cases_per_month_spark/
            part-00000-<hash>.csv
```

```
    query2/
        weekly_stats_pd.csv
        weekly_stats_spark/
            part-00000-<hash>.csv
    query3/
        clusters_all_months/
            part-00000-<hash>.csv
        clusters_custom_all_months/
            part-00000-<hash>.csv
main.py
README.md
requirements.txt
setup.py
visualisation.ipynb
```

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

Query 1 Source Code

```python
from pyspark.sql import functions as F
from pyspark.sql.window import Window
import time


class Query1:
    """
    Class to run Query 1.

        Parameters:
            sparkSession (SparkSession): Spark Session
            covidDataDf (DataFrame): Covid DataFrame

        Attributes:
            sparkSession (SparkSession): Spark Session
            covidDataDf (DataFrame): Covid DataFrame
    """

    def __init__(self, sparkSession, covidDataDf):
        self.sparkSession = sparkSession  # Spark Session
        self.covidDataDf = covidDataDf  # Covid DataFrame

    def run(self):
        """
        Runs Query 1 and writes the result to a CSV file.
        """
        # Start the timer
        startTime = time.time()

        # Get the date columns (Remove State/Province, Country/Region, Lat, Long columns)
        date_cols = self.covidDataDf.columns[4:]

        # Create the stack expression to pivot the table to make the date columns into rows
        stack_expr = (
            "stack("
            + str(len(date_cols))
            + ", "
```

```
            + ", ".join(["'" + x + "', `" + x + "`" for x in date_cols])
            + ") as (Date, Value)"
        )

        # Pivot the table to make the date columns into rows
        covidDataDf_pivot = self.covidDataDf.select(
            "Country/Region", F.expr(stack_expr)
        ).withColumn(
            "Date", F.to_date("Date", "M/d/yy")
        )  # Convert the Date column to a date type

        # Window specification to get the previous day's cases for each country
        windowSpec = Window.partitionBy("`Country/Region`").orderBy("Date")

        # Calculate daily confirmed cases
        covidDataDf_daily = (
            covidDataDf_pivot.withColumn(
                "DailyCases", F.col("Value") - F.lag("Value").over(windowSpec)
            )
            .fillna({"DailyCases": 0})  # Fill the null values with 0
            .drop("Value")  # Drop the Value column
        )

        # Cache the dataframe
        covidDataDf_daily.cache()

        # Group by country and month to get average daily cases
        df_grouped = (
            covidDataDf_daily.groupBy(
                "Country/Region",
                F.year("Date").alias("Year"),
                F.month("Date").alias("Month"),
            )  # Group by country, year, and month
            .agg(
                F.avg("DailyCases").alias("Average")
            )  # Compute the average daily cases
            .orderBy(
                "Country/Region", "Year", "Month"
            )  # Order by country, year, and month
        )

        # Unpersist the dataframe
        covidDataDf_daily.unpersist()

        # Write the result to a CSV file
        try:
            df_grouped.write.csv(
                "results/query1/mean_daily_cases_per_month_spark",
                header=True,
                mode="overwrite",
            )
```

```
        except Exception as e:
            print(f"Failed to write the file: {e}")

        # Stop the timer
        endTime = time.time()
        print(f"Query 1 took {endTime - startTime} seconds.")
```

Appendix C

Query 2 Source Code

```python
import geopandas as gpd
from shapely.geometry import Point
from pyspark.sql import functions as F
from pyspark.sql.types import StringType
from pyspark.sql.window import Window
import time


class Query2:
    """
    Class to run Query 2.

        Parameters:
            sparkSession (SparkSession): Spark Session
            covidDataDf (DataFrame): Covid DataFrame

        Attributes:
            sparkSession (SparkSession): Spark Session
            covidDataDf (DataFrame): Covid DataFrame
            world_boundaries_gdf (GeoDataFrame): Shapefile of world boundaries (continents)
    """

    def __init__(self, sparkSession, covidDataDf):
        self.sparkSession = sparkSession  # Spark Session
        self.covidDataDf = covidDataDf  # Covid DataFrame
        self.world_boundaries_gdf = gpd.read_file(
            "data/World_Continents.zip"
        )  # Shapefile of world boundaries (continents)

    def __prepare_data(self):
        """
        Prepares the COVID data for processing.
        """
        # Location column : Province/State if not null, else Country/Region
        self.covidDataDf = (
            self.covidDataDf.withColumn(
                "Location", F.coalesce(F.col("Province/State"), F.col("Country/Region"))
```

```
            )  # Coalesce the Province/State and Country/Region columns
            .filter(
                F.col("Lat").isNotNull() & F.col("Long").isNotNull()
            )  # Filter out the rows with null values in Lat and Long columns
            .drop(
                "Province/State", "Country/Region"
            )  # Drop the Province/State and Country/Region columns
        )

    def __assign_continent(self):
        """
        Assign continents to each data row based on coordinates.
        """
        # Broadcast the world boundaries dataframe to all the nodes in the cluster for faster processing
        broadcasted_gdf = self.sparkSession.sparkContext.broadcast(
            self.world_boundaries_gdf
        )

        def determine_continent(longitude, latitude):
            """
            Determine the continent of a given set of coordinates.

                Parameters:
                    longitude (double): Longitude of the location (in degrees)
                    latitude (double): Latitude of the location (in degrees)

                Returns:
                    (str): Continent of the location
            """
            point = Point(
                longitude, latitude
            )  # Create a Point object from the given coordinates
            matching_continents = broadcasted_gdf.value[
                broadcasted_gdf.value.geometry.contains(point)
            ][
                "CONTINENT"
            ]  # Get the matching continents from the world boundaries geodataframe
            if len(matching_continents) > 0:
                # If the continent is Australia, return Oceania
                if matching_continents.values[0] == "Australia":
                    return "Oceania"
                # If the continent is North America or South America, return America
                if (
                    matching_continents.values[0] == "North America"
                    or matching_continents.values[0] == "South America"
                ):
```

```
                    return "America"
                return matching_continents.values[0]
            # If the continent is not found, determine the continent based on the coordinates (Approximate)
            elif -34 <= latitude <= 37 and -17 <= longitude <= 51:
                return "Africa"
            elif 34 <= latitude <= 82 and -25 <= longitude <= 60:
                return "Europe"
            elif -56 <= latitude <= 71 and -168 <= longitude <= -34:
                return "America"
            elif -1 <= latitude <= 81 and (25 <= longitude <= 171):
                return "Asia"
            elif -50 <= latitude <= 10 and 110 <= longitude <= 180:
                return "Oceania"
            elif -90 <= latitude <= -60:
                return "Antarctica"
            return None

        # Create a UDF to determine the continent of a given set of coordinates
        udf_determine_continent = F.udf(
            lambda long, lat: determine_continent(long, lat), StringType()
        )

        # Assign continents to each data row based on coordinates
        self.covidDataDf = self.covidDataDf.withColumn(
            "Continent", udf_determine_continent(F.col("Long"), F.col("Lat"))
        ).filter(
            F.col("Continent").isNotNull()
        )  # Filter out the rows with null values in the Continent column

    def __pivot_table(self):
        """
        Pivot the table to make the date columns into rows.
        """
        # Get the date columns (Remove Location, Continent, Lat, Long columns)
        date_cols = self.covidDataDf.columns[2:-2]

        # Create the stack expression to pivot the table to make the date columns into rows
        stack_expr = (
            "stack("
            + str(len(date_cols))
            + ", "
            + ", ".join(["'" + x + "', `" + x + "`" for x in date_cols])
            + ") as (Date, Value)"
        )

        # Pivot the table to make the date columns into rows
        self.covidDataDf = self.covidDataDf.select(
```

```
            "Location", "Continent", F.expr(stack_expr)
        ).withColumn(
            "Date", F.to_date("Date", "M/d/yy")
        )  # Convert the Date column to a date type

    def __compute_daily_confirmed_cases(self):
        # Window specification to get the previous day's cases for each country
        windowSpec = Window.partitionBy("Location").orderBy("Date")

        # Calculate daily confirmed cases
        self.covidDataDf = (
            self.covidDataDf.withColumn(
                "DailyCases", F.col("Value") - F.lag("Value").over(windowSpec)
            )  # Calculate the daily cases
            .fillna({"DailyCases": 0})  # Fill the null values with 0
            .drop("Value")  # Drop the Value column
        )

    def __compute_slopes(self):
        """
        Compute the slopes of the daily confirmed cases for each location.
        """
        # Determine the start date for computing DaysSinceStart
        start_date = self.covidDataDf.agg(F.min("Date")).first()[0]  # Start date
        days_since_start = F.datediff(
            F.col("Date"), F.lit(start_date)
        )  # Days since start

        # Prepare the data for linear regression
        windowSpec = Window.partitionBy("Location")
        regression_df = (
            self.covidDataDf.withColumn(
                "DaysSinceStart", days_since_start
            )  # Days since start (x)
            .withColumn(
                "DaysSquared", F.pow(F.col("DaysSinceStart"), 2)
            )  # Days squared
            .withColumn("n", F.col("DaysSinceStart") + 1)  # Number of observations (n)
            .withColumn(
                "CasesTimesDays", F.col("DaysSinceStart") * F.col("DailyCases")
            )  # Cases * Days
            .withColumn(
                "sumDays", F.sum("DaysSinceStart").over(windowSpec)
            )  # Sum of DaysSinceStart
            .withColumn(
                "sumCases", F.sum("DailyCases").over(windowSpec)
            )  # Sum of DailyCases
            .withColumn(
```

```
                "sumDaysSquared", F.sum("DaysSquared").over(
windowSpec)
            )  # Sum of DaysSquared
            .withColumn(
                "sumCasesTimesDays", F.sum("CasesTimesDays").over(
windowSpec)
            )  # Sum of CasesTimesDays
        )

        # Compute the slope of the linear regression
        numerator = F.col("n") * F.col("sumCasesTimesDays") - F.col
("sumDays") * F.col(
            "sumCases"
        )  # Numerator of the slope
        denominator = F.col("n") * F.col("sumDaysSquared") - F.pow(
            F.col("sumDays"), 2
        )  # Denominator of the slope
        self.covidDataDf = regression_df.withColumn(
            "DailySlope",
            F.when(denominator == 0, 0).otherwise(
                numerator / denominator
            ),  # Daily slope
        ).select(
            "Location", "Date", "DailyCases", "DailySlope", "
Continent"
        )  # Select the required columns

    def __filter_top_affected(self):
        """
        Select the top 100 affected locations.
        """
        # Aggregate the slopes by location, calculating the maximum
 slope
        top100LocationsDf = (
            self.covidDataDf.groupBy("Location")  # Group by
location
            .agg(F.max("DailySlope").alias("MaxSlope"))  # Maximum
slope
            .orderBy(F.desc("MaxSlope"))  # Order by the maximum
slope
            .limit(100)  # Select the top 100 locations
        )

        # Join the top 100 countries back to the original dataframe
        self.covidDataDf = self.covidDataDf.join(
            F.broadcast(
                top100LocationsDf
            ),  # Broadcast the top 100 locations dataframe to all
the nodes in the cluster for faster processing
            ["Location"],
        )

    def __aggregate(self):
        """
        Perform statistics calculations on the top 100 affected
locations.
```

```
        """
        # Group by continent and week
        self.covidDataDf = (
            self.covidDataDf.withColumn(
                "WeekStart",
                F.date_sub(
                    F.col("Date"), F.dayofweek(F.col("Date")) - 1
                ),  # Starting date of the week
            )
            .withColumn(
                "WeekEnd", F.date_add(F.col("WeekStart"), 6)  # Ending date of the week
            )
            .withColumn(
                "WeekRange",
                F.concat(
                    F.date_format(F.col("WeekStart"), "dd/MM/yyyy"),
                    F.lit(" - "),
                    F.date_format(F.col("WeekEnd"), "dd/MM/yyyy"),
                ),
            )
        )

        # Aggregate the data by continent and week
        weekly_stats = (
            self.covidDataDf.groupBy("Continent", "WeekRange")
            .agg(
                F.mean("DailyCases").alias("Mean"),  # Mean of DailyCases
                F.stddev("DailyCases").alias("Std"),  # Standard deviation of DailyCases
                F.min("DailyCases").alias("Min"),  # Minimum of DailyCases
                F.max("DailyCases").alias("Max"),  # Maximum of DailyCases
            )
            .orderBy(
                "Continent", F.min("WeekStart")
            )  # Order by continent and week start date
        )

        # Write the aggregated data to a CSV file
        try:
            weekly_stats.write.csv(
                "results/query2/weekly_stats_spark", header=True, mode="overwrite"
            )
        except Exception as e:
            print(f"Failed to write the file: {e}")

    def run(self):
        """
        Runs the query 2 and writes the results to a CSV file.
        """
        startTime = time.time()
```

```
        self.__prepare_data()
        self.__assign_continent()
        self.__pivot_table()
        self.covidDataDf.cache()  # Cache the dataframe in memory for faster processing
        self.__compute_daily_confirmed_cases()
        self.__compute_slopes()
        self.__filter_top_affected()
        self.__aggregate()
        self.covidDataDf.unpersist()  # Unpersist the dataframe from memory
        endTime = time.time()
        print(f"Query 2 took {endTime - startTime} seconds to complete.")
```

Appendix D

Query 3 Source Code

```
from pyspark.sql import functions as F
from pyspark.sql.window import Window
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.clustering import KMeans

import numpy as np
import time

from Covid19AnalysisProjectLib.NonOptimised.CustomKMeans import
    CustomKMeans


class Query3:
    """
    Class to run Query 3.

        Parameters:
            sparkSession (SparkSession): Spark Session
            covidDataDf (DataFrame): Covid DataFrame

        Attributes:
            sparkSession (SparkSession): Spark Session
            covidDataDf (DataFrame): Covid DataFrame
    """

    def __init__(self, sparkSession, covidDataDf):
        self.sparkSession = sparkSession  # Spark Session
        self.covidDataDf = covidDataDf  # Covid DataFrame

    def __prepare_data(self):
        """
        Prepares the COVID data for processing.
        """
        # Location column : Province/State if not null, else
    Country/Region
        self.covidDataDf = (
            self.covidDataDf.withColumn(
                "Location",
                F.coalesce(
```

```
                    F.col("Province/State"), F.col("Country/Region")
                ),  # If Province/State is null, use Country/Region
            )
            .filter(
                F.col("Lat").isNotNull() & F.col("Long").isNotNull()
            )  # Filter out rows with null Lat and Long values
            .drop(
                "Province/State", "Country/Region", "Lat", "Long"
            )  # Drop unnecessary columns
        )

    def __pivot_table(self):
        """
        Pivots the table to make the date columns into rows.
        """
        # Get the date columns (Remove Location column)
        date_cols = self.covidDataDf.columns[0:-1]

        # Create the stack expression to pivot the table to make the date columns into rows
        stack_expr = (
            "stack("
            + str(len(date_cols))
            + ", "
            + ", ".join(["'" + x + "', `" + x + "`" for x in date_cols])
            + ") as (Date, Value)"
        )

        # Pivot the table to make the date columns into rows
        self.covidDataDf = self.covidDataDf.select(
            "Location", F.expr(stack_expr)
        ).withColumn(
            "Date", F.to_date("Date", "M/d/yy")
        )  # Convert the Date column to a date type

    def __compute_daily_confirmed_cases(self):
        """
        Compute the daily confirmed cases for each location.
        """
        # Window specification to get the previous day's cases for each location
        windowSpec = Window.partitionBy("Location").orderBy("Date")
        # Calculate daily confirmed cases
        self.covidDataDf = (
            self.covidDataDf.withColumn(
                "DailyCases", F.col("Value") - F.lag("Value").over(windowSpec)
            )  # Calculate the difference between the current and previous day's cases
            .fillna({"DailyCases": 0})  # Fill the null values with 0
            .drop("Value")  # Drop the Value column
        )
```

```

    def __compute_monthly_slopes(self):
        """
        Compute the slopes of the monthly confirmed cases for each location.
        """
        # Extract the month from the date
        self.covidDataDf = self.covidDataDf.withColumn(
            "Month", F.date_format("Date", "yyyy-MM")
        )

        # Determine the start date for computing MonthsSinceStart
        start_date = self.covidDataDf.agg(F.min("Date")).first()[0]
        months_since_start_expr = (F.year("Date") - F.year(F.lit(start_date))) * 12 + (
            F.month("Date") - F.month(F.lit(start_date))
        )  # Calculate the number of months since the start date
        self.covidDataDf = self.covidDataDf.withColumn(
            "MonthsSinceStart", months_since_start_expr
        )  # Add the MonthsSinceStart column

        # Calculate monthly total cases for each location and include 'MonthsSinceStart'
        monthly_totals = self.covidDataDf.groupBy(
            "Location", "Month", "MonthsSinceStart"
        ).agg(F.sum("DailyCases").alias("MonthlyCases"))

        # Window specification to get the cumulative sum of MonthsSinceStart and MonthlyCases for each location
        window_spec = Window.partitionBy("Location").orderBy("Month")
        # Prepare the dataframe for linear regression
        regression_df = (
            monthly_totals.withColumn(
                "x", F.col("MonthsSinceStart")
            )  # MonthsSinceStart (x)
            .withColumn(
                "MonthsSquared", F.pow(F.col("MonthsSinceStart"), 2)
            )  # MonthsSinceStart squared
            .withColumn(
                "n", F.col("MonthsSinceStart") + 1
            )  # Number of observations (n)
            .withColumn(
                "CasesTimesMonths",
                F.col("MonthsSinceStart")
                * F.col("MonthlyCases"),  # MonthlyCases * MonthsSinceStart
            )
            .withColumn(
                "sumMonths", F.sum("MonthsSinceStart").over(window_spec)
            )  # Cumulative sum of MonthsSinceStart
            .withColumn(
                "sumCases", F.sum("MonthlyCases").over(window_spec)
            )  # Cumulative sum of MonthlyCases
```

```
            .withColumn(
                "sumMonthsSquared", F.sum("MonthsSquared").over(window_spec)
            )  # Cumulative sum of MonthsSquared
            .withColumn(
                "sumCasesTimesMonths",
                F.sum("CasesTimesMonths").over(
                    window_spec
                ),  # Cumulative sum of MonthlyCases * MonthsSinceStart
            )
        )

        # Calculate the slope for each location
        numerator = F.col("n") * F.col("sumCasesTimesMonths") - F.col(
            "sumMonths"
        ) * F.col("sumCases")
        denominator = F.col("n") * F.col("sumMonthsSquared") - F.pow(
            F.col("sumMonths"),
            2,
        )
        self.covidDataDf = regression_df.withColumn(
            "MonthlySlope",
            F.when(denominator == 0, 0).otherwise(numerator / denominator),
        ).select("Location", "Month", "MonthlyCases", "MonthlySlope")

    def __filter_top_affected(self):
        """
        Select the top 50 affected locations.
        """
        # Aggregate the slopes by location, calculating the maximum or average slope
        top50LocationsDf = (
            self.covidDataDf.groupBy("Location")
            .agg(
                F.mean("MonthlySlope").alias("MeanSlope")
            )  # Get the maximum slope for each location
            .orderBy(F.desc("MeanSlope"))  # Order by descending slope
            .limit(50)  # Select the top 50 locations
        )

        # Join the top 50 locations back to the original dataframe
        self.covidDataDf = (
            self.covidDataDf.join(
                F.broadcast(
                    top50LocationsDf
                ),  # Broadcast the top 50 locations dataframe to all nodes for efficient join
                ["Location"],
            )
            .drop(
```

```
                "Date", "MonthsSinceStart", "DailyCases"
            )  # Drop the unnecessary columns
            .dropDuplicates(["Location", "Month"])  # Drop
duplicate rows
            .orderBy("Month", "Location")  # Order by month and
location
        )

    def __apply_custom_clustering(self):
        """
        Apply custom clustering to the data for each month using
the CustomKMeans model.
        """
        # Start the timer
        start_time_custom = time.time()

        # Initialize CustomKMeans model
        custom_kmeans = CustomKMeans(n_clusters=4, max_iter=20,
seed=1, tol=1e-4)

        # Processing each month in parallel
        all_clusters = []
        unique_months = self.covidDataDf.select("Month").distinct()
.collect()
        for month_row in unique_months:
            # Filter the data for the current month
            month = month_row["Month"]
            monthly_data = self.covidDataDf.filter(F.col("Month")
== month)

            # Convert the MonthlySlope column to a numpy array
            slope_data = np.array(
                monthly_data.select("MonthlySlope").collect()
            ).reshape(-1, 1)

            # Train and predict using the custom KMeans model
            custom_kmeans.fit(slope_data)
            cluster_assignments = custom_kmeans.predict(slope_data)

            # Create a dataframe with the cluster assignments
            monthly_data = monthly_data.withColumn(
                "row_index", F.monotonically_increasing_id()
            )
            clusters_df = self.sparkSession.createDataFrame(
                [
                    (int(i), int(cluster_assignments[i]))
                    for i in range(len(cluster_assignments))
                ],
                ["row_index", "Cluster"],
            )

            # Join the cluster assignments to the original
dataframe
            clusters = monthly_data.join(clusters_df, "row_index").
select(
                "Location", "Month", "Cluster"
```

```
            )

            # Append to all_clusters list
            all_clusters.append(clusters)

        # Union all clusters and repartition for efficient writing
        final_clusters_df = self.sparkSession.createDataFrame(
            self.sparkSession.sparkContext.emptyRDD(), all_clusters
    [0].schema
        )
        for clusters in all_clusters:
            final_clusters_df = final_clusters_df.union(clusters)
    # Union all clusters

        # Repartition by month for efficient writing to a single
    CSV file
        final_clusters_df = final_clusters_df.repartition("Month")

        # Write the aggregated results to a single CSV file
        try:
            final_clusters_df.write.csv(
                "results/query3/clusters_custom_all_months",
                header=True,
                mode="overwrite",
            )
        except Exception as e:
            print(f"Failed to write the file: {e}")

        print(
            f"Clustering completed by custom implementation in {
    time.time() - start_time_custom} seconds"
        )

    def __apply_clustering(self):
        """
        Apply clustering to the data for each month using Spark
    MLlib.
        """
        # Start the timer
        start_time_spark = time.time()

        # Initialize VectorAssembler and KMeans model outside the
    loop
        vec_assembler = VectorAssembler(
            inputCols=["MonthlySlope"], outputCol="features"
        )
        kmeans = (
            KMeans()
            .setK(4)  # Number of clusters
            .setSeed(1)  # Random seed
            .setFeaturesCol("features")  # Input features
            .setPredictionCol("Cluster")  # Output cluster
        )

        # Vectorise the MonthlySlope column
```

```
        self.covidDataDf = vec_assembler.transform(self.covidDataDf)

        # Apply clustering to the data for each month using Spark MLlib
        all_clusters = []
        unique_months = self.covidDataDf.select("Month").distinct().collect()
        for month_row in unique_months:
            # Filter the data for the current month
            month = month_row["Month"]
            monthly_data = self.covidDataDf.filter(F.col("Month") == month)

            # Train and predict using the Spark MLlib KMeans model
            model = kmeans.fit(monthly_data)
            clusters = model.transform(monthly_data).select(
                "Location", "Month", "Cluster"
            )

            # Append to all_clusters list
            all_clusters.append(clusters)

        # Union all clusters and repartition for efficient writing
        final_clusters_df = self.sparkSession.createDataFrame(
            self.sparkSession.sparkContext.emptyRDD(), all_clusters[0].schema
        )
        for clusters in all_clusters:
            final_clusters_df = final_clusters_df.union(clusters)  # Union all clusters

        # Repartition by month for efficient writing to a single CSV file
        final_clusters_df = final_clusters_df.repartition("Month")

        # Write the aggregated results to a single CSV file
        try:
            final_clusters_df.write.csv(
                "results/query3/clusters_all_months", header=True, mode="overwrite"
            )
        except Exception as e:
            print(f"Failed to write the file: {e}")

        # Stop the timer
        stop_time_spark = time.time()
        print(f"Clustering completed by Spark MLlib in {stop_time_spark - start_time_spark} seconds")

    def run(self):
        """
        Runs the query 3 and writes the results to a CSV file.
        """
        # Execute methods in sequence
        self.__prepare_data()
```

```
        self.__pivot_table()
        self.__compute_daily_confirmed_cases()
        self.__compute_monthly_slopes()
        self.covidDataDf.cache()  # Cache the dataframe to speed up processing
        self.__filter_top_affected()
        self.__apply_custom_clustering()
        self.__apply_clustering()
        self.covidDataDf.unpersist()  # Unpersist the dataframe
```

Appendix E

Terminal Output

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR,
    use setLogLevel(newLevel).
23/11/18 18:04:06 WARN NativeCodeLoader: Unable to load native-
    hadoop library for your platform... using builtin-java classes
    where applicable

------------ STEP 1: FETCHING THE LATEST DATA ------------

1. Deleting the existing data...
   -> Deleting file: 2023-11-18.csv
Done deleting the existing data.

2. Fetching the latest data...
Done fetching the latest data.

3. Writing the data to a CSV file: 2023-11-18.csv
Done writing the data to a CSV file.


------------ STEP 2: LOADING THE DATA INTO A SPARK DATAFRAME
    ------------

Done loading the data into a Spark dataframe.


------------ STEP 3: Query 1 ------------

23/11/18 18:04:11 WARN SparkStringUtils: Truncated the string
    representation of a plan since it was too large. This behavior
    can be adjusted by setting 'spark.sql.debug.maxToStringFields'.
Query 1 took 3.1029131412506104 seconds.

------------ STEP 4: Query 2 ------------

23/11/18 18:04:20 WARN GarbageCollectionMetrics: To enable non-
    built-in garbage collector(s) List(G1 Concurrent GC), users
    should configure it(them) to spark.eventLog.gcMetrics.
    youngGenerationGarbageCollectors or spark.eventLog.gcMetrics.
    oldGenerationGarbageCollectors
```

```
Query 2 took 15.06498098373413 seconds to complete.

------------ STEP 5: Query 3 ------------

Clustering completed by custom implementation in 58.84758687019348
    seconds
23/11/18 18:05:32 WARN InstanceBuilder: Failed to load
    implementation from:dev.ludovic.netlib.blas.JNIBLAS
23/11/18 18:05:32 WARN InstanceBuilder: Failed to load
    implementation from:dev.ludovic.netlib.blas.VectorBLAS
23/11/18 18:07:17 WARN DAGScheduler: Broadcasting large task binary
     with size 2020.2 KiB
Clustering completed by Spark MLlib in 109.29923582077026 seconds
```