Members: Bao Phuc Duong (Patrick Duong), Shubham Gupta

ECE 4150 – Spring 2024

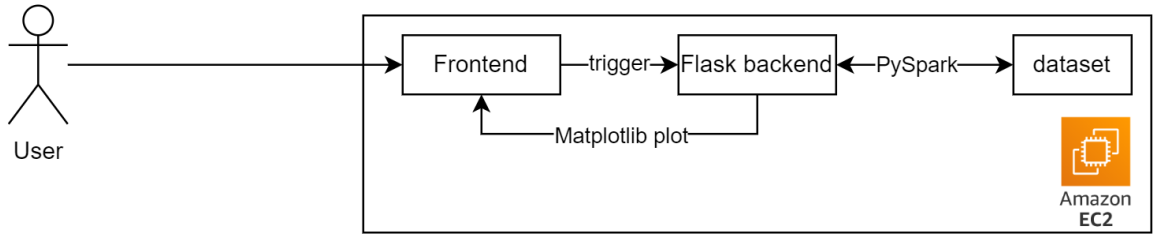
Apache Spark project on AWS EC2

A) Purpose

The objective of this project is to learn how to integrate Spark into AWS EC2 and leverage various Spark operations on extensive datasets. The power of Spark will be demonstrated on the simple website we built for this project.

In addition to Spark, we also use Matplotlib to visually represent the data manipulated by Spark. With Spark and Matplotlib, we can construct an intuitive analytics website, providing users with deeper insights into the dataset under examination.

B) Architecture and Process Flow



The users choose the details of the information they want to see on the frontend. When they click “submit”, the frontend triggers the corresponding Flask route in the backend to use Spark to manipulate, filter, and collect the appropriate data from the dataset. Then, the backend uses the collected data to sketch bar charts using Matplotlib, saves the plot as a ‘.png’ image, and send the image to the frontend.

C) Project Walkthrough

1) Choose a dataset

To demonstrate how efficiently Spark works with large datasets, we must first choose a suitable dataset for the project.

The dataset we chose is a csv file that contains data about the Mortality Number and Rate for 187 countries between 1970 and 2010. The dataset has 58906 rows and 6 columns, which offers a lot of information for me to work with using Spark. Below is the first few rows of the dataset:

Country Code	Country Name	Year	Age Group	Sex	Number of	Death Rate Per 100,000
AFG	Afghanistan	1970	0-6 days	Male	19,241	318,292.90
AFG	Afghanistan	1970	0-6 days	Female	12,600	219,544.20
AFG	Afghanistan	1970	0-6 days	Both	31,840	270,200.70
AFG	Afghanistan	1970	7-27 days	Male	15,939	92,701.00
AFG	Afghanistan	1970	7-27 days	Female	11,287	68,594.50
AFG	Afghanistan	1970	7-27 days	Both	27,226	80,912.50
AFG	Afghanistan	1970	28-364 days	Male	37,513	15,040.10
AFG	Afghanistan	1970	28-364 days	Female	32,113	13,411.80
AFG	Afghanistan	1970	28-364 days	Both	69,626	14,242.60
AFG	Afghanistan	1970	1-4 years	Male	36,694	4,288.20
AFG	Afghanistan	1970	1-4 years	Female	32,848	4,022.90
AFG	Afghanistan	1970	1-4 years	Both	69,542	4,158.60
AFG	Afghanistan	1970	5-9 years	Male	3,467	396.2
AFG	Afghanistan	1970	5-9 years	Female	2,492	306
AFG	Afghanistan	1970	5-9 years	Both	5,959	352.7
AFG	Afghanistan	1970	10-14 years	Male	1,723	230.8
AFG	Afghanistan	1970	10-14 years	Female	1,806	262.3
AFG	Afghanistan	1970	10-14 years	Both	3,529	245.9
AFG	Afghanistan	1970	15-19 years	Male	1,816	282.9
AFG	Afghanistan	1970	15-19 years	Female	1,902	324.7
AFG	Afghanistan	1970	15-19 years	Both	3,718	302.8

and more...

2) Choose an appropriate AWS EC2 instance:

We opted for Ubuntu 20.04 as the OS image for the EC2 instance because it is no cost and user-friendly. For streamlined development and testing of our application, we chose to run Spark in local mode. Additionally, setting up Flask locally for this project is straightforward.

Since Spark operates locally, we want it to use all CPU cores for parallel processing to ensure swift data filtering, collection, sketching, and presentation on the website for users. To achieve this, selecting an EC2 instance type that offers multiple CPU cores becomes a must. Hence, we decided on the t3.xlarge type, which boasts up to 4 cores and is very cost-effective (only $0.3328/h for Ubuntu OS).

We also allowed all inbound and outbound traffic for this EC2 instance.

3) Install Anaconda and PySpark:

After successfully launching the EC2 instance, the next step involves installing Anaconda and PySpark on it. While closely following the provided instructions, we made a few modifications to suit our preferences.

Initially, we installed Anaconda by executing two commands:

```
$ wget http://repo.continuum.io/archive/Anaconda3-4.1.1-Linux-x86_64.sh
$ bash Anaconda3-4.1.1-Linux-x86_64.sh
```

Following this, we agreed to the Anaconda license terms and opted for installation at the present location. To complete the Anaconda installation, we ran the command : `$ source .bashrc`

Subsequently, we proceeded to install JRE using:

```
$ sudo apt-get update
$ sudo apt-get install default-jre
```

Additionally, we installed Scala with the command:

```
$ sudo apt-get install scala
```

Py4J was installed using the following commands:

```
$ export PATH=$PATH:$HOME/anaconda/bin
$ conda install pip
$ pip install py4j
```

Moving forward, we installed Spark/Hadoop by downloading the package and extracting its contents:

```
$ wget http://archive.apache.org/dist/spark/spark-2.0.0/spark-2.0.0-bin-hadoop2.7.tgz
$ tar -zxvf spark-2.0.0-bin-hadoop2.7.tgz
```

Upon installing Spark/Hadoop, we configured Python to recognize Spark:

```
$ export SPARK_HOME='/home/ubuntu/spark-2.0.0-bin-hadoop2.7'
$ export PATH=$SPARK_HOME:$PATH
$ export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH
```

Finally, we installed PySpark with the command:

```
$ pip install pyspark -no-cache-dir
{use `pip install pyspark` if the above cmd gives errors}
```

These steps ensured the successful setup of Anaconda, PySpark, and related dependencies on the EC2 instance, paving the way for further development and analysis tasks. Now, let's move on to the coding portion of the project.

4) Import necessary modules and configure Spark session and data frame:

To implement Flask backend with Spark and Matplotlib, we must import the appropriate modules at the beginning.

```
from flask import Flask, render_template, redirect, session, request
from pyspark.sql import SparkSession, functions
import matplotlib
matplotlib.use('Agg') # Agg is non-interactive backend and compatible for flask frontend

import matplotlib.pyplot as plt
import io
import base64
```

We also must configure Spark Session as well. As mentioned earlier, we want Spark to run locally using as many available CPU cores as possible to suit its needs.

```
spark = SparkSession \
    .builder \
    .master("local[*]") \
    .appName("Mortality Analysis") \
    .getOrCreate()
```

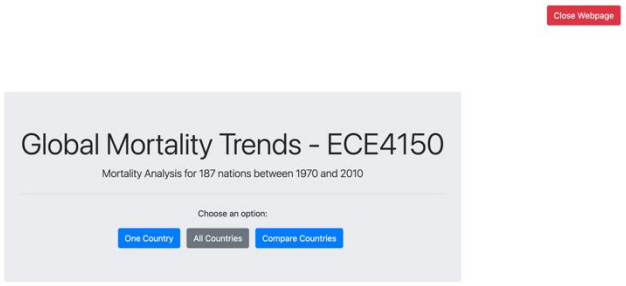
Then, we created a global DataFrame object **df** in PySpark, which is a distributed collection of data organized into named columns. This DataFrame **df** is created by reading the "mortality_age.csv" file into Spark. we also specified that the first row of the file contains column headers, and the columns are separated by commas.

```
df = spark.read.csv("file:///home/ubuntu/data/mortality_age.csv", header=True, sep=",")
```

we then perform some data cleaning and manipulation operations on **df**. Specifically, we remove commas from the "Number of Deaths" and "Death Rate per 100,000" columns and casts them to appropriate types. This is necessary because Spark cannot recognize numerical data if it contains comma in it. Also, **df** will be regularly used on multiple pages of the website, so we cache it to memory to avoid redundant computations and enhance performance.

```
# Remove commas from the "Number of Deaths" and "Death Rate per 100,000" columns and cast them to appropriate types
df = df.withColumn("Number of Deaths", functions.regexp_replace(df["Number of Deaths"], ",", "").cast("int")) \
    .withColumn("Death Rate Per 100,000", functions.regexp_replace(df["Death Rate Per 100,000"], ",", "").cast("float"))
df.cache() #caching to improve performance
```

5) Homepage



This is the default Homepage of the website. The user has two options to interact with the dataset: “One Country” and “All Countries”. If the user clicks on “Close Webpage”, the backend will safely close the Spark Session and redirect the user to “Google.com”.

```
# ROUTES
@app.route('/', methods=['GET'])
def homepage():
    prepopulate_data()
    return render_template('home.html')
```

When the user reaches this route, the backend runs the “prepopulate_data()” function to check if the Flask sessions “country_codes” and “age_group” are None. If the two sessions are None, then the backend will assign a suitable list to each session. We will use these two sessions to populate the dropdown in “oneAnalysis” and “allAnalysis” pages later.

```
# SUPPORTING FUNCTIONS
def prepopulate_data():
    if not session.get('country_codes') and not session.get('age_group'):
        # Collect all distinct country codes, sort them, then convert df to rdd, then flatten each row into its constituent elements.
        session['country_codes'] = df.select("Country Code").distinct().orderBy("Country Code").rdd.flatMap(lambda x: x).collect()

        # Collect all distinct age groups, sort them, then convert df to rdd, then flatten each row into its constituent elements.
        session['age_group'] = df.select("Age Group").distinct().orderBy("Age Group").rdd.flatMap(lambda x: x).collect()
```

Now, let’s examine how the list for session “country_codes” is created. From the DataFrame **df**, Spark selects the distinct values of the “Country Code” column, then sorts them in ascending order, converts the result to a resilient distributed dataset (RDD), flattens the RDD, and finally collects the elements into a list. This list is then assigned to the Flask session “country_codes” for later usage.

We can see the same Spark operations to create the list for session “age_group”. From **df**, Spark chooses the distinct values of the “Age Group” column, orders them in ascending order, converts the result to an RDD and flattens it, and collects the elements into a list. This list is assigned to Flask session “age_group” for later usage as well.

6) One Country (oneAnalysis) page:

After clicking on “One Country” button on homepage, the user is directed to this page. There will be 3 dropdowns for them to choose their desire values from. Their chosen values will be used as a filter for the DataFrame, and 2 corresponding plots will be created based on the filtered data.

Here’s the Flask route for this page:

```
@app.route('/oneAnalysis', methods=['GET', 'POST'])
def oneAnalysis():
    if request.method == 'POST':
        country_code = str(request.form['country_code'])
        age_group = str(request.form['age_group'])
        gender = str(request.form['gender'])

        df_filtered = df.filter((df["Country Code"] == country_code) & (df["Age Group"] == age_group) & (df["Sex"] == gender))

        # Gather data on Number of Deaths
        df_grouped_1 = df_filtered.groupBy("Year").agg(functions.sum("Number of Deaths").alias("Total Deaths"))

        # Gather data on Death rate
        df_grouped_2 = df_filtered.groupBy("Year").agg(functions.sum("Death Rate Per 100,000").alias("Death Rate"))

        # Collect data from data frame
        data_1 = df_grouped_1.collect()
        data_2 = df_grouped_2.collect()

        x_vals = [row["Year"] for row in data_1]
        y1_vals = [row["Total Deaths"] for row in data_1]
        y2_vals = [row["Death Rate"] for row in data_2]

        # Create plot 2
        fig, ax2 = plt.subplots()
        ax2.bar(x_vals, y2_vals)
        ax2.set_xlabel('Year')
        ax2.set_ylabel('Death rate')
        ax2.set_title("Death rate per 100,000 for {} by Year in {} ({})".format(gender, country_code, age_group))

        # Save plot2 in png format to send to frontend
        plot2_buf = io.BytesIO()
        plt.savefig(plot2_buf, format='png')
        plot2_buf.seek(0)
        plot2_base64 = base64.b64encode(plot2_buf.read()).decode('utf-8')
        plt.close(fig)

        return render_template('oneAnalysis.html', country_codes = session.get('country_codes'), age_group = session.get('age_group'),
                               plot1=plot1_base64, plot2=plot2_base64)
    return render_template('oneAnalysis.html', country_codes = session.get('country_codes'), age_group = session.get('age_group'),
                               plot1=None, plot2=None)
```

Looking at render_template() function calls of the code above, we find that the Flask sessions "country_codes" and "age_group" are sent to the frontend. They are used to populate the dropdowns "Country Code" and "Age Group" we see on the webpage.

We programmed the "Submit" button to be clickable only when 3 values from 3 dropdowns are selected by the user. When the user clicks the "Submit", a "POST" method will be triggered, and the backend jumps inside the 'if' block.

Now, let's see what happens inside the 'if' block. First, the selected values from the dropdowns "Country Code", "Age Group", and "Gender" are used to filter the DataFrame **df**, which results in the filtered DataFrame **df_filtered.**

```
df_filtered = df.filter((df["Country Code"] == country_code) & (df["Age Group"] == age_group) & (df["Sex"] == gender))
```

Then, Spark groups the rows of **df_filtered** by the "Year" column and aggregates the "Number of Deaths" column within each group, summing up the total number of deaths for each year. As a result, the DataFrame **df_grouped_1** is produced and it has two columns "Year" and "Total Deaths".

Similarly, to produce **df_grouped_2**, Spark groups the rows of **df_filtered** by the "Year" column and aggregates the "Death Rate Per 100,000" column within each group, summing up the total death rate per 100000 for each year. The DataFrame **df_grouped_2** has two columns "Year" and "Death Rate".

```
# Gather data on Number of Deaths
df_grouped_1 = df_filtered.groupBy("Year").agg(functions.sum("Number of Deaths").alias("Total Deaths"))

# Gather data on Death rate
df_grouped_2 = df_filtered.groupBy("Year").agg(functions.sum("Death Rate Per 100,000").alias("Death Rate"))
```

Then, Spark collects the data from both **df_grouped_1** and **df_grouped_2** before assigning them to "data_1" and "data_2" respectively.

```
# Collect data from data frame
data_1 = df_grouped_1.collect()
data_2 = df_grouped_2.collect()
```

"data_1" and "data_2" are lists of dictionaries, where each dictionary represents a row from the DataFrames **df_grouped_1** and **df_grouped_2** respectively. In "data_1" and "data_2", each row contains one dictionary. For example, we can visualize "data_1" and "data_2" as below:

data_1 = [
    {"Year": 1970, "Total Deaths": 23000},
    {"Year": 1980, "Total Deaths": 50000},
    ...
]

data_2 = [
    {"Year": 1970, "Death Rate": 10.0},
    {"Year": 1980, "Death Rate": 20.0},
    ...
]

(this is just an example, not real data)

After the “Submit” button is clicked, Spark filters and gathers data as shown above. Those data are used to sketch two plots in the backend – “Number of Deaths vs Year” and “Death Rate vs Year”.

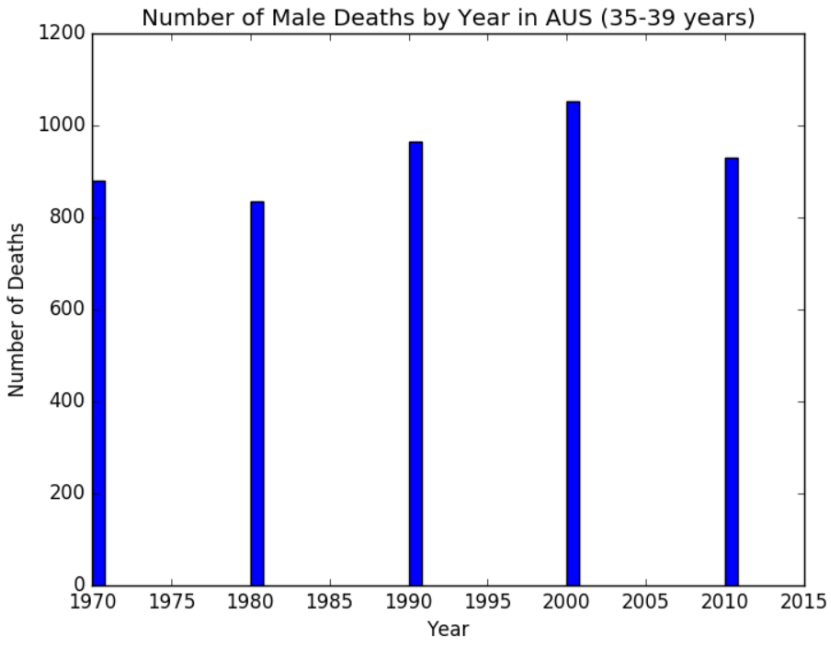
To sketch the plots, we need values for x-axes and y-axes. Since both plots have the same type of x-axis (Year), we only need 1 list of values for both plots’ x-axes. We need a list of “Total Deaths” values for the 1st plot while we need a list of “Death Rate” values for the 2nd plot. We can get those lists of values from the list of dictionaries “data_1” and “data_2” earlier.

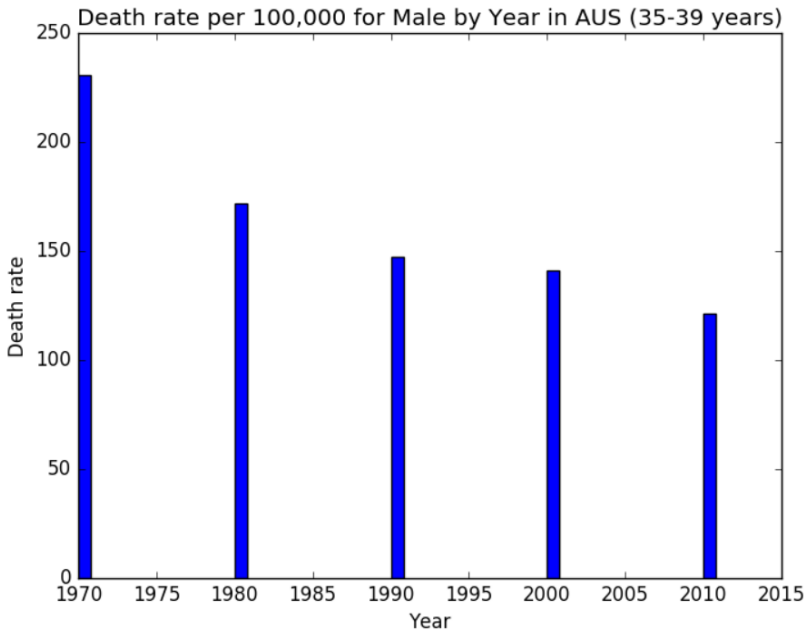
```
x_vals = [row["Year"] for row in data_1]
y1_vals = [row["Total Deaths"] for row in data_1]
y2_vals = [row["Death Rate"] for row in data_2]
```

After we get all the values for x-axes and y-axes, now we can plot the two plots, save them in ‘.png’ format, and send them to frontend in render_template() function call. To view the .png files/plots, we have added buttons to download the corresponding plots. Clicking on them downloads the plots locally on your machine.

```
# Create plot 1
fig, ax1 = plt.subplots()
ax1.bar(x_vals, y1_vals)
ax1.set_xlabel('Year')
ax1.set_ylabel('Number of Deaths')
ax1.set_title("Number of {} Deaths by Year in {} ({})".format(gender, country_code, age_group))

# Save plot1 in png format to send to frontend
plot1_buf = io.BytesIO()
plt.savefig(plot1_buf, format='png')
plot1_buf.seek(0)
plot1_base64 = base64.b64encode(plot1_buf.read()).decode('utf-8')
plt.close(fig)

# Create plot 2
fig, ax2 = plt.subplots()
ax2.bar(x_vals, y2_vals)
ax2.set_xlabel('Year')
ax2.set_ylabel('Death rate')
ax2.set_title("Death rate per 100,000 for {} by Year in {} ({})".format(gender, country_code, age_group))

# Save plot2 in png format to send to frontend
plot2_buf = io.BytesIO()
plt.savefig(plot2_buf, format='png')
plot2_buf.seek(0)
plot2_base64 = base64.b64encode(plot2_buf.read()).decode('utf-8')
plt.close(fig)
```

If the user chooses “AUS” for Country Code dropdown, “35-39 years” for Age Group dropdown, and “Male” for Gender dropdown, below is what he/she will see on the webpage:





7) All Countries (allAnalysis) page:

After clicking on “All Countries” button on homepage, the user reaches this page. There are 3 dropdowns for them to choose their desire values from. Their chosen values will be used as a filter for the DataFrame, and 2 corresponding plots will be created based on the filtered data.

The plots for this page shows the data for all 187 countries throughout the years, not just one specific country. The “Submit” button is disabled until all the user chooses 3 values from 3 dropdowns.

Here’s the Flask route that handles this page:

```
@app.route('/allAnalysis', methods=['GET', 'POST'])
def allAnalysis():
    if request.method == 'POST':
        age_group = str(request.form['age_group'])
        gender = str(request.form['gender'])
        year = str(request.form['year'])

        df_filtered = df.filter((df["Age Group"] == age_group) & (df["Sex"] == gender) & (df["Year"] == year))

        # Gather data on Number of Deaths
        df_grouped_1 = df_filtered.groupBy("Country Code").agg(functions.sum("Number of Deaths").alias("Total Deaths"))

        # Gather data on Death rate
        df_grouped_2 = df_filtered.groupBy("Country Code").agg(functions.sum("Death Rate Per 100,000").alias("Death Rate"))

        # Collect data from data frame
        data_1 = df_grouped_1.collect()
        data_2 = df_grouped_2.collect()

        x_ticks = list(range(len(data_1)))  # Create numerical ticks for x-axis
        bar_width = 0.5

        x_vals = [row["Country Code"] for row in data_1]
        y1_vals = [row["Total Deaths"] for row in data_1]
        y2_vals = [row["Death Rate"] for row in data_2]

        # Create plot1
        fig, ax1 = plt.subplots(figsize=(18, 7.5))
        ax1.set_title("Number of {} Deaths by Countries in {} ({})".format(gender, year, age_group))
        ax1.bar(x_ticks, y1_vals, width=bar_width)
        ax1.set_xlabel('Countries')
        ax1.set_ylabel('Number of Deaths')
        ax1.set_xticklabels([])    #no labels for x-axis
        plt.xlim([0, len(x_ticks)])
```

```
    # Add country codes on top of each bar
    for i, val in enumerate(y1_vals):
        if i % 2 == 0:
            #val + k (distance between x-val and its bar) must be sufficiently large
            #compared to plot values in order to see the shift of the bar labels
            ax1.text(x_ticks[i] + bar_width/2, val + 4000, x_vals[i], ha='center', va='bottom', fontsize=5.5, rotation=90)
        else:
            ax1.text(x_ticks[i] + bar_width/2, val + 1000, x_vals[i], ha='center', va='bottom', fontsize=5.5, rotation=90)

    # Save plot1 in png format to send to frontend
    plot1_buf = io.BytesIO()
    plt.savefig(plot1_buf, format='png')
    plot1_buf.seek(0)
    plot1_base64 = base64.b64encode(plot1_buf.read()).decode('utf-8')
    plt.close(fig)

    # Create plot2
    fig, ax2 = plt.subplots(figsize=(18, 7.5))
    ax2.set_title("Death rate per 100,000 for {} by Countries in {} ({})".format(gender, year, age_group))
    ax2.bar(x_ticks, y2_vals, width=bar_width)
    ax2.set_xlabel('Countries')
    ax2.set_ylabel('Death rate')
    ax2.set_xticklabels([])
    plt.xlim([0, len(x_ticks)])

    for i, val in enumerate(y2_vals):
        if i % 2 == 0:
            #val + k must be sufficiently large compared to plot values in order to see the shift of the bar labels
            ax2.text(x_ticks[i] + bar_width/2, val + 12, x_vals[i], ha='center', va='bottom', fontsize=5.5, rotation=90)
        else:
            ax2.text(x_ticks[i] + bar_width/2, val + 16, x_vals[i], ha='center', va='bottom', fontsize=5.5, rotation=90)

    # Save plot2 in png format to send to frontend
    plot2_buf = io.BytesIO()
    plt.savefig(plot2_buf, format='png')
    plot2_buf.seek(0)
    plot2_base64 = base64.b64encode(plot2_buf.read()).decode('utf-8')
    plt.close(fig)

    return render_template('allAnalysis.html', age_group = session.get('age_group'), plot1=plot1_base64, plot2=plot2_base64)
return render_template('allAnalysis.html', age_group = session.get('age_group'), plot1=None, plot2=None)
```

In the render_template() function calls, we see that the Flask sessions “age_group” is sent to the frontend as an argument. It’s used to populate the dropdowns “Age Group” we see on the webpage.

After the user choses items from all 3 dropdowns, they can click the “Submit” button, which triggers a “POST” response and the backend jumps inside the ‘if’ block. Now, let’s examine what happens inside the ‘if’ block.

First of all, Spark uses the selected values from the dropdowns “Age Group”, “Gender”, and “Year” to filter the DataFrame **df**, which results in the filtered DataFrame **df_filtered.**

```
df_filtered = df.filter((df["Age Group"] == age_group) & (df["Sex"] == gender) & (df["Year"] == year))
```

Then, Spark groups the rows of **df_filtered** by the “Country Code” column and aggregates the “Number of Deaths” column in each group, summing up the total number of deaths for each country code. Consequently, the DataFrame **df_grouped_1** is produced and it has two columns “Country Code” and “Total Deaths”.

Similarly, to create **df_grouped_2**, Spark groups the rows of **df_filtered** by the “Country Code” column and aggregates the “Death Rate Per 100,000” column in each group, summing up the total death rate per 100000 for each year. The DataFrame **df_grouped_2** has two columns “Country Code” and “Death Rate”.

```
# Gather data on Number of Deaths
df_grouped_1 = df_filtered.groupBy("Country Code").agg(functions.sum("Number of Deaths").alias("Total Deaths"))

# Gather data on Death rate
df_grouped_2 = df_filtered.groupBy("Country Code").agg(functions.sum("Death Rate Per 100,000").alias("Death Rate"))
```

Then, Spark collects the data from both **df_grouped_1** and **df_grouped_2** before assigning them to "data_1" and "data_2" respectively.

```
data_1 = df_grouped_1.collect()
data_2 = df_grouped_2.collect()
```

"data_1" and "data_2" are lists of dictionaries, where each dictionary represents a row from the DataFrames **df_grouped_1** and **df_grouped_2** respectively. In "data_1" and "data_2", each row contains one dictionary. For example, we can visualize "data_1" and "data_2" as below:

data_1 = [
{"Country Code": AFG, "Total Deaths": 45000},
{"Country Code": AUS, "Total Deaths": 30000},
...
]

data_2 = [
{"Country Code": AFG, "Death Rate": 20.0},
{"Country Code": AUS, "Death Rate": 15.0},
...
]

(this is just an example, not real data)

After the user clicks on "Submit" button, Spark manipulates data as shown above. Those data help sketch two plots in the backend – "Number of Deaths vs Countries" and "Death Rate vs Countries".

To sketch the plots, we require values for x-axes and y-axes. Since both plots have the same type of x-axis (Country Code), we only need 1 list of values for both plots' x-axes. We need a list of "Total Deaths" values for the 1st plot while we need a list of "Death Rate" values for the 2nd plot. We can get those list of values from the list of dictionaries "data_1" and "data_2" earlier.

```
x_vals = [row["Country Code"] for row in data_1]
y1_vals = [row["Total Deaths"] for row in data_1]
y2_vals = [row["Death Rate"] for row in data_2]
```

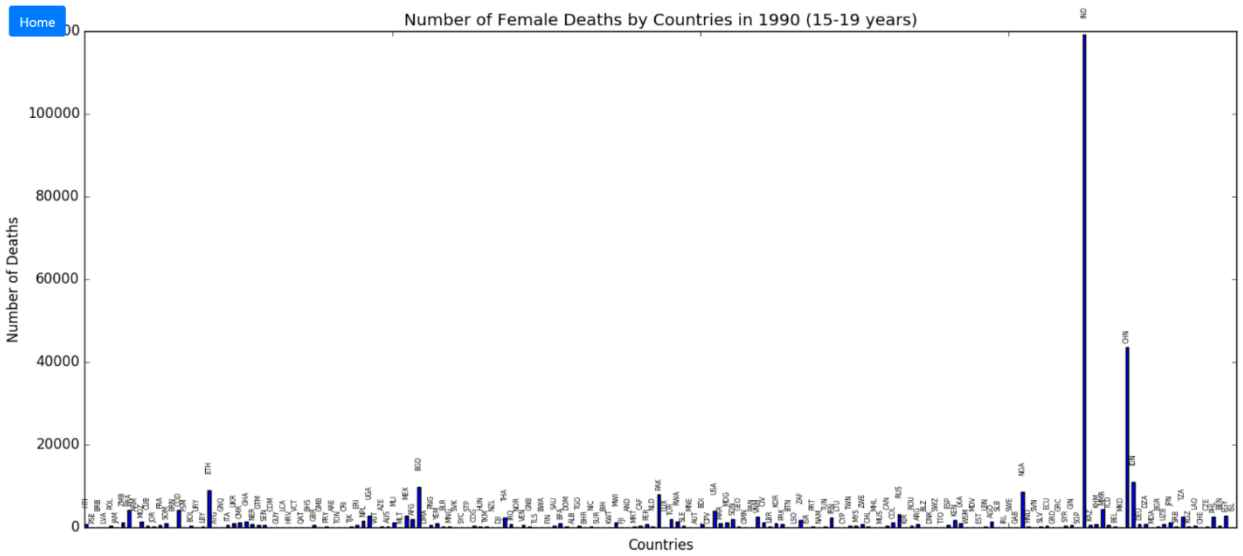
For these 2 plots, the x-values are vertically displayed on top of each bar, not under the x-axes.

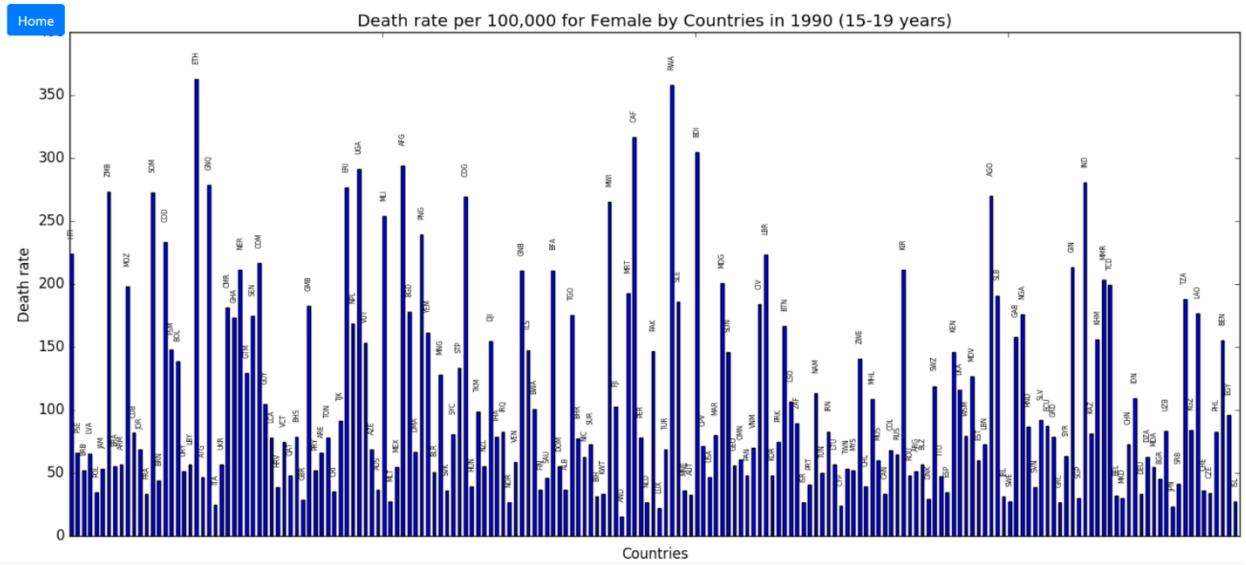
Also, the distances between x-values and their associated bars will be different to make sure the x-values do not overlap one another. Then, after the backend sketches the plots, it saves them in ".png" format and sends them to frontend in the render_template() call.

```
# Create plot1
fig, ax1 = plt.subplots(figsize=(18, 7.5))
ax1.set_title("Number of {} Deaths by Countries in {} ({})".format(gender, year, age_group))
ax1.bar(x_ticks, y1_vals, width=bar_width)
ax1.set_xlabel('Countries')
ax1.set_ylabel('Number of Deaths')
ax1.set_xticklabels([])    #no labels for x-axis
plt.xlim([0, len(x_ticks)])

# Add country codes on top of each bar
for i, val in enumerate(y1_vals):
    if i % 2 == 0:
        #val + k (distance between x-val and its bar) must be sufficiently large
        #compared to plot values in order to see the shift of the bar labels
        ax1.text(x_ticks[i] + bar_width/2, val + 4000, x_vals[i], ha='center', va='bottom', fontsize=5.5, rotation=90)
    else:
        ax1.text(x_ticks[i] + bar_width/2, val + 1000, x_vals[i], ha='center', va='bottom', fontsize=5.5, rotation=90)

# Save plot1 in png format to send to frontend
plot1_buf = io.BytesIO()
plt.savefig(plot1_buf, format='png')
plot1_buf.seek(0)
plot1_base64 = base64.b64encode(plot1_buf.read()).decode('utf-8')
plt.close(fig)
```

```
# Create plot2
fig, ax2 = plt.subplots(figsize=(18, 7.5))
ax2.set_title("Death rate per 100,000 for {} by Countries in {} ({})".format(gender, year, age_group))
ax2.bar(x_ticks, y2_vals, width=bar_width)
ax2.set_xlabel('Countries')
ax2.set_ylabel('Death rate')
ax2.set_xticklabels([])
plt.xlim([0, len(x_ticks)])

for i, val in enumerate(y2_vals):
    if i % 2 == 0:
        #val + k must be sufficiently large compared to plot values in order to see the shift of the bar labels
        ax2.text(x_ticks[i] + bar_width/2, val + 12, x_vals[i], ha='center', va='bottom', fontsize=5.5, rotation=90)
    else:
        ax2.text(x_ticks[i] + bar_width/2, val + 16, x_vals[i], ha='center', va='bottom', fontsize=5.5, rotation=90)

# Save plot2 in png format to send to frontend
plot2_buf = io.BytesIO()
plt.savefig(plot2_buf, format='png')
plot2_buf.seek(0)
plot2_base64 = base64.b64encode(plot2_buf.read()).decode('utf-8')
plt.close(fig)
```

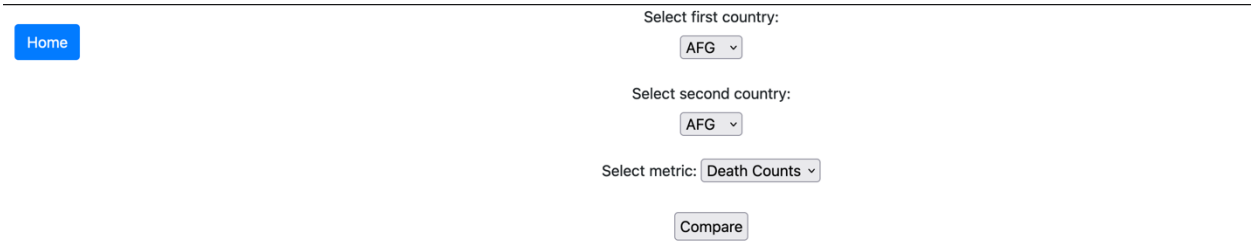
If the user chooses “15-19 years” for Age Group dropdown, “Female” for Gender dropdown, and “1990” for Year dropdown, below is what he/she will see on the webpage:





Note: The plots and country codes on top of the bars are much bigger and easier to view on the actual webpage. These are just small screenshots of the plots. Also, the user can conveniently scroll left, right, up, and down to see the plots from different angles on the real webpage**.**

7) Compare Countries page:



After clicking on “Compare Countries” button on homepage, the user reaches this page. There are 3 dropdowns for them to choose their desire values from. Their chosen values will be used as a filter for the DataFrame, and corresponding plots will be created based on the filtered data.

The plot for this page shows the data for comparison in death counts for the two countries selected as ex. USA and AUS, throughout the years.

The plot for other page shows the data for comparison in death ratio for the two countries selected as ex. USA and AUS, throughout the years.

Here’s the Flask route that handles this page:

```
@app.route('/compareCountries', methods=['GET', 'POST'])
def compareCountries():
    if request.method == 'POST':
        country_code1 = str(request.form['country1'])
        country_code2 = str(request.form['country2'])
        metric = str(request.form['metric'])  # New input for the selected metric

        # Filter the DataFrame for the selected countries
        df_filtered1 = df.filter(df["Country Code"] == country_code1)
        df_filtered2 = df.filter(df["Country Code"] == country_code2)

        # Gather data for the selected metric
        if metric == 'Total Deaths':
            # Gather data on Number of Deaths
            df_grouped_1 = df_filtered1.groupBy("Year").agg(functions.sum("Number of Deaths").alias("Total Deaths"))
            df_grouped_2 = df_filtered2.groupBy("Year").agg(functions.sum("Number of Deaths").alias("Total Deaths"))
        elif metric == 'Death Rate':
            # Gather data on Death rate
            df_grouped_1 = df_filtered1.groupBy("Year").agg(functions.sum("Death Rate Per 100,000").alias("Death Rate"))
            df_grouped_2 = df_filtered2.groupBy("Year").agg(functions.sum("Death Rate Per 100,000").alias("Death Rate"))

        # Collect data from data frames
        data_1 = df_grouped_1.collect()
        data_2 = df_grouped_2.collect()

        x_vals = [row["Year"] for row in data_1]  # Assuming the x-axis represents years
        y1_vals = [row[metric] for row in data_1]
        y2_vals = [row[metric] for row in data_2]

        # Create plot
        fig, ax1 = plt.subplots()
        ax1.plot(x_vals, y1_vals, label=country_code1)
        ax1.plot(x_vals, y2_vals, label=country_code2)
        ax1.set_xlabel('Year')
        ax1.set_ylabel(metric)
        ax1.set_title("{} Comparison between {} and {}".format(metric, country_code1, country_code2))
        ax1.legend()
```

```
        # Save plot in png format to send to frontend
        plot_buf = io.BytesIO()
        plt.savefig(plot_buf, format='png')
        plot_buf.seek(0)
        plot_base64 = base64.b64encode(plot_buf.read()).decode('utf-8')
        plt.close(fig)

        return render_template('compareCountries.html', countries=session.get('country_codes'),plot=plot_base64)
    return render_template('compareCountries.html', countries=session.get('country_codes'), plot = None)

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

In the render_template() function calls, we see that the Flask sessions “country_codes” is sent to the frontend as an argument. It’s used to populate the dropdowns “Country Code” we see on the webpage.

After the user choses items from all 3 dropdowns, they can click the “Compare” button, which triggers a “POST” response and the backend jumps inside the ‘if’ block. Now, let’s examine what happens inside the ‘if’ block.

First, the selected values from the dropdowns “Country Code” for both countries are used to filter the DataFrame **df**, which results in the filtered DataFrame **df_filtered1 and df_filtered2.**

```
        # Filter the DataFrame for the selected countries
        df_filtered1 = df.filter(df["Country Code"] == country_code1)
        df_filtered2 = df.filter(df["Country Code"] == country_code2)
```

Then, if the selected metric is “Total Deaths”, then Spark groups the rows of **df_filtered1 and df_filtered2** by the “Year” column and aggregates the “Number of Deaths” column within each group, summing up the total number of deaths for each year. As a result, the DataFrame **df_filtered1 and df_filtered2** are produced and it has two columns “Year” and “Total Deaths”.

Else, if the selected metric is “Death Rate”, then Spark groups the rows of **df_filtered1 and df_filtered2** by the “Year” column and aggregates the “Death Rate Per 100,000” column within each group, summing up the total number of deaths for each year. As a result, the DataFrame **df_filtered1 and df_filtered2** are produced and it has two columns “Year” and “Death Rate”.

```
        # Gather data for the selected metric
        if metric == 'Total Deaths':
            # Gather data on Number of Deaths
            df_grouped_1 = df_filtered1.groupBy("Year").agg(functions.sum("Number of Deaths").alias("Total Deaths"))
            df_grouped_2 = df_filtered2.groupBy("Year").agg(functions.sum("Number of Deaths").alias("Total Deaths"))
        elif metric == 'Death Rate':
            # Gather data on Death rate
            df_grouped_1 = df_filtered1.groupBy("Year").agg(functions.sum("Death Rate Per 100,000").alias("Death Rate"))
            df_grouped_2 = df_filtered2.groupBy("Year").agg(functions.sum("Death Rate Per 100,000").alias("Death Rate"))
```

Then, Spark collects the data from both **df_grouped_1** and **df_grouped_2** before assigning them to “data_1” and “data_2” respectively.

```
        # Collect data from data frames
        data_1 = df_grouped_1.collect()
        data_2 = df_grouped_2.collect()
```

“data_1” and “data_2” are lists of dictionaries, where each dictionary represents a row from the DataFrames **df_grouped_1** and **df_grouped_2** respectively. In “data_1” and “data_2”, each row contains one dictionary. For example, we can visualize “data_1” and “data_2” as below:

data_1 = [
    {“Year”: 1970, “Total Deaths”: 23000},
    {“Year”: 1980, “Total Deaths”: 50000},
    ...
]

data_2 = [
    {“Year”: 1970, “Death Rate”: 10.0},
    {“Year”: 1980, “Death Rate”: 20.0},
    ...
]

(this is just an example, not real data)

After the “Compare” button is clicked, Spark filters and gathers data as shown above. Those data are used to sketch plots corresponding to the selected metric in the backend – “Total Deaths vs Year” and “Death Rate vs Year”.

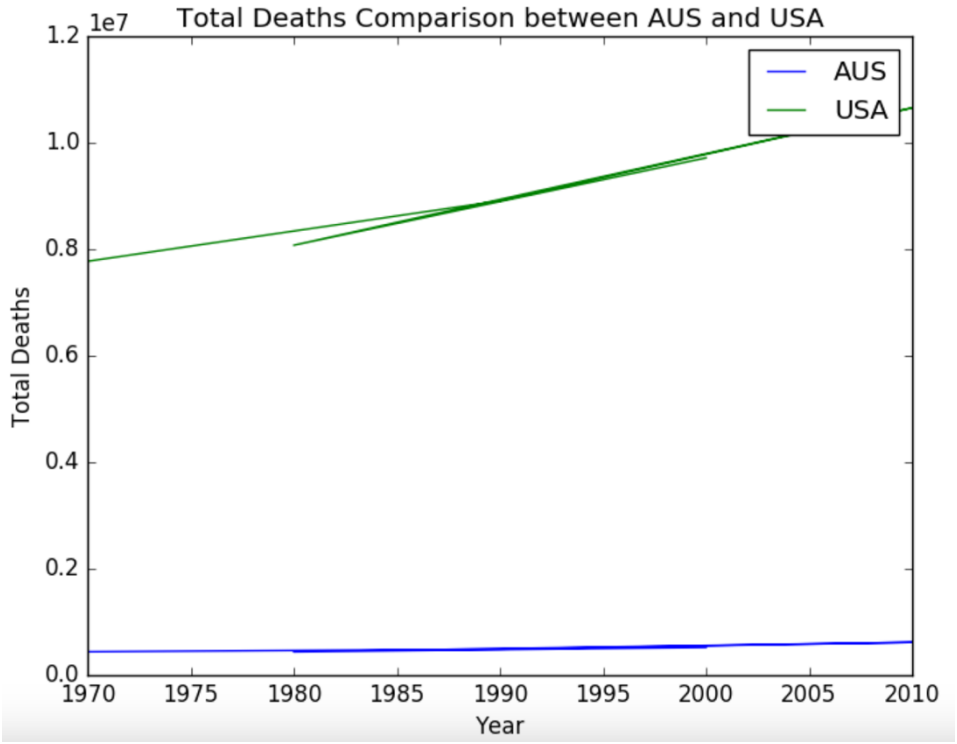
To sketch the plots, we need values for x-axes and y-axes. Since both plots have the same type of x-axis (Year), we only need 1 list of values for both plots’ x-axes. We need a list of “Total Deaths” values for the 1st plot while we need a list of “Death Rate” values for the 2nd plot. We can get those lists of values from the list of dictionaries “data_1” and “data_2” earlier.

```
x_vals = [row["Year"] for row in data_1]  # Assuming the x-axis represents years
y1_vals = [row[metric] for row in data_1]
y2_vals = [row[metric] for row in data_2]
```

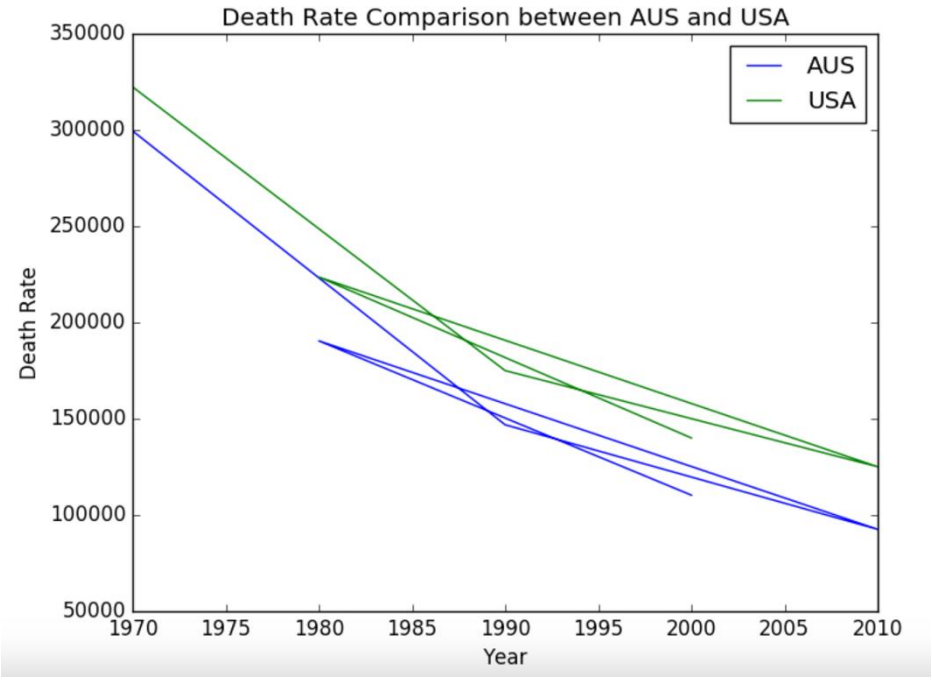
After we get all the values for x-axes and y-axes, now we can plot the two plots, save them in ‘.png’ format, and send them to frontend in render_template() function call.

```
# Create plot
fig, ax1 = plt.subplots()
ax1.plot(x_vals, y1_vals, label=country_code1)
ax1.plot(x_vals, y2_vals, label=country_code2)
ax1.set_xlabel('Year')
ax1.set_ylabel(metric)
ax1.set_title("{} Comparison between {} and {}".format(metric, country_code1, country_code2))
ax1.legend()

# Save plot in png format to send to frontend
plot_buf = io.BytesIO()
plt.savefig(plot_buf, format='png')
plot_buf.seek(0)
plot_base64 = base64.b64encode(plot_buf.read()).decode('utf-8')
plt.close(fig)
```

If the user chooses “AUS” for Country Code1 dropdown and “USA” for Country Code2 dropdown , “Death Counts ” for metric dropdown, below is what they will see on the webpage:



If the user chooses “AUS” for Country Code1 dropdown and “USA” for Country Code2 dropdown , “Death Ratio ” for metric dropdown, below is what they will see on the webpage:



D) Conclusion

Apache Spark is a powerful distributed computing system that is used for big data processing and analytics. Throughout this project, we gained valuable insights into Spark's ability at efficiently handling large datasets, alongside its seamless integration with AWS EC2 and Matplotlib to build a good visualization website.

At the beginning of this project, we learned how to choose an optimal instance type of AWS EC2 so that Spark can have enough resources to operate quickly on a large dataset. We also learned how to install and set up Spark on AWS EC2 virtual server so that it can run without errors.

During the development stages, we learned how to filter, manipulate, and use the data to craft insightful visualizations on the frontend. We also get familiar with the structures of Spark-produced Data Frames to use them effectively. Additionally, we tried our best to Optimize Spark's performance by caching the DataFrame and maximizing CPU cure utilization.

If you would like to see the complete code, documentation and the demo video for the Flask backend, the frontend, and the working of the app, please refer to the zipped file. Thank you!