## Problem I: MapReduce implementation using Python and mrjob

1. **total incomes** – The sum of all incomes in the dataset

The Google Colab environment was used for running and developing programs. The following code was used to calculate the total sum of values:

```
# total_incomes.py
total_incomes_code = """
from mrjob.job import MRJob

class TotalIncomes(MRJob):

    def mapper(self, _, line):
        income = float(line.strip())
        yield "total", income

    def reducer(self, key, values):
        total = sum(values)
        yield key, total

if __name__ == '__main__':
    TotalIncomes.run()
"""
```

In this code, all values are mapped with a common key, "total," and then the sum of these values is calculated. The Python code was executed to achieve this.

Subsequently, the following command was used to execute the file on the dataset:

```
!python total_incomes.py trial_incomes.csv

!python total_incomes.py test_incomes.csv
```

Result:

- Dataset: train

```
"total"   63168.0
```

- Dataset: test

```
"total"   210015551664.0
```

2. **Mean** – The mean of all incomes in the dataset.

```
mean_code = """
from mrjob.job import MRJob

class Mean(MRJob):

    def mapper(self, _, line):
        # Check if the line is a digit and yield key-value pairs
        if line.isdigit():
            yield "mean", int(line)

    def reducer(self, key, values):
        # Calculate the mean
        total = 0
        count = 0
        for value in values:
            total += value
            count += 1
        mean = total / count if count > 0 else 0
        yield key, mean

if __name__ == '__main__':
    Mean.run()

"""

with open("mean.py", "w") as f:

    f.write(mean_code)
```

In this code, similar to before, the total sum of values is calculated. However, in addition to that, for each mapped value, a count of 1 is incremented, representing the total number of data points.

Subsequently, the following command was used to execute the file on the dataset:

```
!python mean.py trial_incomes.csv

!python mean.py test_incomes.csv
```

Result:

- Dataset: train

"mean" 63.168

- Dataset: test

"mean" 21001.5551664

3. **Generalized mean** – The generalized mean of all incomes in the dataset .

```
# generalized_mean.py
generalized_mean_code = """
from mrjob.job import MRJob
from mrjob.step import MRStep

class GeneralizedMeanJob(MRJob):

    def mapper(self, _, line):
        income = float(line)
        yield None, income

    def reducer(self, _, incomes):
        # Replace 'p' with the desired order of the generalized mean
        p = 2  # You can change this value to calculate for different 
orders

        # Calculate the mean
        total = 0
        count = 0
        for value in incomes:
            total += value**p
            count += 1
        mean=(total/count)**(1/p)

        yield None, mean

if __name__ == '__main__':
    GeneralizedMeanJob.run()

"""
with open("generalized_mean.py", "w") as f:
    f.write(generalized_mean_code)
```

Here, the numeric value is mapped with the key "None" by the mapper. Then, it is raised to the power of p and divided by the total number of values according to the formula. Finally, it reaches the power of 1/p. It is worth mentioning that in this context, p is assumed to be 2.

Subsequently, the following command was used to execute the file on the dataset:

```
!python generalized_mean.py trial_incomes.csv

!python generalized_mean.py test_incomes.csv
```

Result:

- Dataset: train

  "mean" 665.8561901792308

- Dataset: test

  "mean" 52883028.36300636

4. **Maximum** – highest income in the dataset.

```
# maximum.py
maximum_code = """
from mrjob.job import MRJob

class Maximum(MRJob):

    def mapper(self, _, line):
        income = float(line.strip())
        yield "max", income

    def reducer(self, key, values):
        max=0
        for value in values:
          if value>max:
            max=value
        max_income=max
        #method 2
        #max_income = max(values)
        yield key, max_income

if __name__ == '__main__':
    Maximum.run()
"""
with open("maximum.py", "w") as f:
    f.write(maximum_code)
```

In this code, values are also mapped in the mapper, and in the reducer, the maximum value among them is calculated straightforwardly.

Subsequently, the following command was used to execute the file on the dataset:

```
!python maximum.py trial_incomes.csv


!python maximum.py test_incomes.csv
```

Result:

- Dataset: train

```
"max"   13473.0
```

- Dataset: test

```
"max" 164016448792.0
```

**Minimum** – lowest income in the dataset.

```
# minimum.py
minimum_code = """
from mrjob.job import MRJob

class Minimum(MRJob):

    def mapper(self, _, line):
        income = float(line.strip())
        yield "min", income

    def reducer(self, key, values):
        min=1e20
        for value in values:
          if value<min:
            min=value
        min_income=min
        # method 2
        #min_income = min(values)
        yield key, min_income

if __name__ == '__main__':
    Minimum.run()
"""
with open("minimum.py", "w") as f:
    f.write(minimum_code)
```

Similar to the minimum, this time the "income" values are mapped in the mapper. Subsequently, these values are received in the reducer, and the minimum value among them is calculated.

Subsequently, the following command was used to execute the file on the dataset:

```
!python minimum.py trial_incomes.csv


!python minimum.py test_incomes.csv
```

Result:

- Dataset: train

```
"min"    1.0
```

- Dataset: test

```
"min" 1.0
```

6. **Standard deviation** – the standard deviation of the incomes .

```
# std_deviation.py
std_deviation_code = """
# std_deviation.py
from mrjob.job import MRJob
from mrjob.step import MRStep

class StandardDeviationJob(MRJob):

    def mapper(self, _, line):
        income = float(line)
        yield None, income

    def reducer(self, _, incomes):
        # Calculate mean
        total = 0
        count = 0
        for value in incomes:
            total += value
            count += 1
        mean = total / count

        # Calculate the sum of squared differences from the mean
        sum_squared_diff = sum((x - mean) ** 2 for x in incomes)

        # Calculate the standard deviation
        std_deviation = (sum_squared_diff / count) ** 0.5

        yield None, std_deviation

if __name__ == '__main__':
    StandardDeviationJob.run()
"""
with open("standard_deviation.py", "w") as f:
    f.write(standard_deviation_code)
```

Similar to before, the values are mapped in the mapper with a common key (None) for use in the reducer. Then, their average is calculated, and using that, the values' differences from the mean are raised to the power of 2. After summing these squared differences, they are divided by the total count, and the square root is taken to calculate the standard deviation.

Subsequently, the following command was used to execute the file on the dataset:

```
!python standard_deviation.py test_incomes.csv

!python minimum.py test_incomes.csv
```

Result:

- Dataset: train

  662.853128359518

- Dataset: test

  52883024.19256389

**Problem II: Sparkifying world cities**
For solving the current set of questions, two types of data structures were utilized: RDD (Resilient Distributed Dataset) and DataFrame.
Using RDD:

```
from pyspark import SparkContext

# Create a Spark Context
sc = SparkContext("local[*]")

# Load the data from Google Drive as an RDD
file_path = "/content/worldcitiespop.txt"
rdd = sc.textFile(file_path)

# Extract header
header = rdd.first()

# Filter out the header from the RDD
data_rdd = rdd.filter(lambda row: row != header)

# Show the first few rows of the data RDD
for row in data_rdd.take(5):
    print(row)
```

1. **Simple cleaning worldcitiespo**p: Write a Spark python program that cleans the "worldcitiespop.txt" file to keep only valid lines. By valid we only consider rows with a given population.

```
from pyspark import SparkContext

# Define the minimum population threshold
min_population = 0

# Function to filter valid lines based on population
def is_valid_line(row):
    try:
        # Split the row into columns
        columns = row.split(',')

        # Extract the population column
        population = int(columns[4])

        # Check if the population is greater than or equal to the
threshold
        return population >= min_population
    except:
        return False

# Filter out invalid lines based on population
valid_data_rdd = data_rdd.filter(lambda row: row != header and
is_valid_line(row))

# Show the first few rows of the valid data RDD
for row in valid_data_rdd.take(5):
    print(row)
```

```
ad,andorra la vella,Andorra la Vella,07,20430,42.5,1.5166667
ad,canillo,Canillo,02,3292,42.5666667,1.6
ad,encamp,Encamp,03,11224,42.5333333,1.5833333
ad,la massana,La Massana,04,7211,42.55,1.5166667
ad,les escaldes,Les Escaldes,08,15854,42.5,1.5333333
```

```
# Count the number of records in the valid data RDD
record_count = valid_data_rdd.count()


# Print the count of valid records
print(f"Number of valid records: {record_count}")
```

Number of valid records: 47980

2. **Statistics:**Write a Spark Python program to display the following statistics for city populations: min, max, sum, and average.

```
# Parse the CSV data into a key-value pair RDD, where the key is the city
and the value is the population
city_population_rdd = valid_data_rdd.map(lambda line: line.split(',')) \
    .map(lambda fields: (fields[2], int(fields[4]) if fields[4].isdigit()
else 0))

# Filter out entries with population 0
filtered_city_population_rdd = city_population_rdd.filter(lambda x: x[1] >
0)

# Compute the statistics
population_stats = filtered_city_population_rdd.values().stats()

# Display the statistics
print("Minimum Population: {}".format(population_stats.min()))
print("Maximum Population: {}".format(population_stats.max()))
print("Total Population: {}".format(population_stats.sum()))
print("Average Population: {}".format(population_stats.mean()))
```

```
Minimum Population: 7.0
Maximum Population: 31480498.0
Total Population: 2289584998.999997
Average Population: 47719.57063359727
```

3. **Histograms:** Write a Spark Python program to calculate a frequency histogram of city populations. For the histogram, the equivalence classes will be chosen using a logarithmic scale. (class 0, cities of size [0..10[, class 1 cities of size [10..100[ ...).

```
# Step 1: Parse the data into tuples (population, 1)
parsed_data = valid_data_rdd.map(lambda line:
line.split(",")).filter(lambda parts: len(parts) == 7)
population_data = parsed_data.map(lambda parts: (int(parts[4]), 1))

# Step 2: Group the data by the logarithmic scale of city populations
# Use logarithmic scale for binning
log_bins = [0, 10, 100, 1000, 10000, 100000, 1000000, 10000000,
float('inf')]
population_bins = population_data.map(lambda x: (next(i-1 for i, v in
enumerate(log_bins) if x[0] < v), x[1]))

# Step 3: Count the number of cities in each group
histogram = population_bins.reduceByKey(lambda x, y: x + y)

# Step 4: Sort the results by bin
sorted_histogram = histogram.sortByKey()

# Step 5: Collect the result
result = sorted_histogram.collect()

# Print the sorted result
print(result)
```

```
[(0, 5), (1, 174), (2, 2187), (3, 20537), (4, 21550), (5, 3248), (6, 269), (7, 10)]
```

**4. TopK:** Write a Spark Python program to calculate and display the 10 cities with the largest population.

```
# Step 1: Parse the CSV data into a structured format
def parse_csv(line):
    fields = line.split(",")
    country, city, accent_city, region, population, latitude, longitude = 
fields
    return (country, city, accent_city, region, int(population), latitude, 
longitude)

parsed_data_rdd = valid_data_rdd.map(parse_csv)

# Step 2: Transform the data to create key-value pairs
# Key: Population, Value: City Information
population_city_rdd = parsed_data_rdd.map(lambda x: (x[4], x))
k=10
# Step 3: Use shufflable transformation to find the top 10 cities by 
population
top_10_cities = population_city_rdd.sortByKey(ascending=False).take(k)

# Display the result
for i, (population, (country, city, accent_city, region, int_population, 
latitude, longitude)) in enumerate(top_10_cities, 1):
    print(f"{i}.country: {country}, City: {city},region: {region}, 
Population: {int_population}, Location: {latitude}, {longitude}")
```

```
1.country: jp, City: tokyo,region: 40, Population: 31480498, Location: 
35.685, 139.751389
2.country: cn, City: shanghai,region: 23, Population: 14608512, Location: 
31.045556, 121.399722
3.country: in, City: bombay,region: 16, Population: 12692717, Location: 
18.975, 72.825833
4.country: pk, City: karachi,region: 05, Population: 11627378, Location: 
24.9056, 67.0822
5.country: in, City: delhi,region: 07, Population: 10928270, Location: 
28.666667, 77.216667
6.country: in, City: new delhi,region: 07, Population: 10928270, Location: 
28.6, 77.2
7.country: ph, City: manila,region: D9, Population: 10443877, Location: 
14.6042, 120.9822
8.country: ru, City: moscow,region: 48, Population: 10381288, Location: 
55.752222, 37.615556
9.country: kr, City: seoul,region: 11, Population: 10323448, Location: 
37.5985, 126.9783
10.country: br, City: sao paulo,region: 27, Population: 10021437, 
Location: -23.473293, -46.665803
```

5. Calculate the percentage of the total population represented by the top K populous cities globally.

```
# with the city and its population
city_population_rdd = valid_data_rdd.map(lambda line:
line.split(",")).map(lambda cols: (cols[2], int(cols[4])))

# Calculate the total population
total_population = city_population_rdd.values().sum()

# Define the value of K (e.g., top 10 cities)
K = 10

# Get the top K populous cities
top_K_cities = city_population_rdd.takeOrdered(K, key=lambda x: -x[1])

# Calculate the total population represented by the top K cities
total_population_top_K = sum(city[1] for city in top_K_cities)

# Calculate the percentage
percentage_top_K = (total_population_top_K / total_population) * 100

# Print the result
print(f"The top {K} cities represent {percentage_top_K:.2f}% of the total
population globally.")
```

```
The top 10 cities represent 5.83% of the total population globally.
```

---

6. Calculate the total population of each region and find the region with the highest population.

```
# Parse the data and extract relevant columns
def parse_line(line):
    fields = line.split(',')
    return (fields[3], int(fields[4]))

# Map each line to a tuple of (Region, Population)
region_population_rdd = valid_data_rdd.map(parse_line)

# Use reduceByKey to calculate the total population for each region
total_population_rdd = region_population_rdd.reduceByKey(lambda x, y: x +
y)

# Find the region with the highest population using sortBy
total_population_region = total_population_rdd.sortBy(lambda x: x[1],
ascending=False)
max_population_region = total_population_region.first()

# Display the result
print("Region with the highest population:")
print("Region: {}, Total Population: {}".format(max_population_region[0],
max_population_region[1]))
print("Total population by region:")
# Collect the results and iterate over them
result_list = total_population_region.collect()
for i, (region, population) in enumerate(result_list, 1):
    print(f"{i}. Region: {region}, Population: {population}")
```

```
Region with the highest population:
Region: 04, Total Population: 112249869
Total population by region:
1. Region: 04, Population: 112249869
2. Region: 07, Population: 99634521
3. Region: 02, Population: 90271668
4. Region: 05, Population: 88272385
5. Region: 16, Population: 71422264
6. Region: 08, Population: 69386610
7. Region: 09, Population: 68742631
8. Region: 11, Population: 61477197
9. Region: 01, Population: 60187116
10. Region: 06, Population: 58191980
…
```

7. Find the countries where the sum of the populations of their cities is greater than a specified value.

```
# Define the specified value for population sum
specified_population_sum = 1000000  # Change this value as needed

# Parse the CSV data and filter out invalid entries
def parse_and_filter(line):
    try:
        fields = line.split(",")
        country = fields[0]
        population = int(fields[4])
        return (country, population)
    except:
        return ("Invalid", 0)

# Apply the transformation to parse and filter the data
filtered_data_rdd = valid_data_rdd.map(parse_and_filter).filter(lambda x: 
x[0] != "Invalid")

# Compute the sum of populations for each country
sum_population_by_country_rdd = filtered_data_rdd.reduceByKey(lambda x, y: 
x + y)

# Filter countries based on the specified population sum
result_countries_rdd = sum_population_by_country_rdd.filter(lambda x: x[1] 
> specified_population_sum)

# Collect the result and print the output
result_countries = result_countries_rdd.collect()
for country, population_sum in result_countries:
    print(f"Country: {country}, Population Sum: {population_sum}")
```

```
Country: at, Population Sum: 4475257
Country: bg, Population Sum: 4858204
Country: br, Population Sum: 133449921
Country: cf, Population Sum: 1721313
Country: cg, Population Sum: 1456811
Country: cl, Population Sum: 11618910
Country: cm, Population Sum: 8336059
Country: cn, Population Sum: 218884084
Country: es, Population Sum: 27772466
Country: gr, Population Sum: 7273653
…
```

8. Determine the average population for cities in each country, considering only cities with a population above a certain threshold.

```
# Define the minimum population threshold
population_threshold = 100000  # Set your desired threshold

# Parse the CSV data and filter cities with population above the threshold
parsed_data_rdd = valid_data_rdd.map(lambda line: line.split(","))
filtered_data_rdd = parsed_data_rdd.filter(lambda fields: int(fields[4]) >
population_threshold)

# Create a key-value pair with (Country, (Population, 1))
country_population_rdd = filtered_data_rdd.map(lambda fields: (fields[0],
(int(fields[4]), 1)))

# Use combineByKey to calculate the sum and count of populations for each
country
sum_count_rdd = country_population_rdd.combineByKey(
    lambda value: (value[0], 1),
    lambda x, value: (x[0] + value[0], x[1] + 1),
    lambda x, y: (x[0] + y[0], x[1] + y[1])
)

# Calculate the average population for each country
average_population_rdd = sum_count_rdd.mapValues(lambda x: x[0] / x[1])

# Collect the results
results = average_population_rdd.collect()

# Print the results
for country, avg_population in results:
    print(f"Country: {country}, Average Population: {avg_population}")
```

```
Country: at, Average Population: 446116.2
Country: bg, Average Population: 331211.4285714286
Country: br, Average Population: 410966.37272727274
Country: cf, Average Population: 684190.0
Country: cg, Average Population: 1115773.0
Country: cl, Average Population: 381148.0
Country: cm, Average Population: 399620.4
Country: cn, Average Population: 502467.7545691906
Country: es, Average Population: 319685.76
Country: gq, Average Population: 164537.5
Country: gr, Average Population: 251161.85714285713
Country: id, Average Population: 339203.0680272109
…
```

9. Identify cities with a population above the average population of their respective countries.

```
# Step 1: Parse the CSV data into key-value pairs
def parse_line(line):
    fields = line.split(',')
    country = fields[0]
    city = fields[1]
    population = int(fields[4])
    return (country, (city, population))

# Apply the parse_line function to each line in the RDD
parsed_data_rdd = valid_data_rdd.map(parse_line)

# Step 2: Calculate the average population for each country
country_population_avg_rdd = parsed_data_rdd \
    .mapValues(lambda x: (x[1], 1)) \
    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
    .mapValues(lambda x: x[0] / x[1])

# Step 3: Filter cities with populations above their respective country
averages
result_rdd = parsed_data_rdd.join(country_population_avg_rdd) \
    .filter(lambda x: x[1][0][1] > x[1][1])

# Display the result
result_rdd.collect()
```

```
[('an', (('willemstad', 97590), 8399.142857142857)),
 ('at', (('amstetten', 22832), 20250.031674208145)),
 ('at', (('baden', 24893), 20250.031674208145)),
 ('at', (('bregenz', 26928), 20250.031674208145)),
 ('at', (('dornbirn', 43013), 20250.031674208145)),
 ('at', (('feldkirch', 29446), 20250.031674208145)),
 ('at', (('graz', 222326), 20250.031674208145)),
 ('at', (('innsbruck', 112467), 20250.031674208145)),
 ('at', (('kapfenberg', 21819), 20250.031674208145)),
 ('at', (('klagenfurt', 90610), 20250.031674208145)),
 ('at', (('klosterneuburg', 24843), 20250.031674208145)),
 ('at', (('krems', 24092), 20250.031674208145)),
 ('at', (('leoben', 24809), 20250.031674208145)),
 ('at', (('leonding', 22736), 20250.031674208145)),
 ('at', (('linz', 181163), 20250.031674208145)),
 ('at', (('modling', 20710), 20250.031674208145)),
 ('at', (('salzburg', 145310), 20250.031674208145)),
 ('at', (('sankt polten', 49000), 20250.031674208145)),
 ('at', (('steyr', 39567), 20250.031674208145)),
 ('at', (('traun', 23959), 20250.031674208145)),
…
```

**10. Re-cleaning:** By analyzing the results of 4), we see that "Delhi" and "New Delhi" are the same cities. Propose a solution to remove duplicates (different cities in the same place) by not keeping the cities with the highest population in case of overlap. Recalculate your histograms etc...
Note:
In this section, we were tasked with finding overlaps in the data. Since these duplicate data points are not exact duplicates but are semantically repetitive, finding them is challenging. A suitable approach was to use geographical coordinate features, namely latitude and longitude. Unfortunately, the format used for these data points is not consistent. Consider the following example:

Country: in, City: delhi, Region: 07, Population: 10928270, Location: 28.666667, 77.216667
Country: in, City: new delhi, Region: 07, Population: 10928270, Location: 28.6, 77.2

In this case, both cities refer to Delhi, but the first one has several decimal places, while the second has only one decimal place. Therefore, direct use of geographical coordinates did not yield satisfactory results.

It was decided to use geographical coordinates but with only one decimal place. However, rounding and removing information might cause some entries to be incorrectly considered equal. For example, after this process, the record for the sixth most populous city (excluding duplicates) was mistakenly removed:

Country: ph, City: manila, Region: D9, Population: 10443877, Location: 14.6042, 120.9822

Therefore, it was decided to use additional information to identify duplicate records. Considering that, semantically, the country and region values for duplicate records should be the same, this information was utilized. The code for this verification is as follows:

```
# Parse the RDD into a structured format
def parse_line(line):
    fields = line.split(",")
    return ((math.floor(float(fields[5])*10)/10,
math.floor(float(fields[6])*10)/10,fields[0],fields[3]), fields[0],
fields[1], fields[2], fields[3], int(fields[4]))

parsed_data_rdd = valid_data_rdd.map(parse_line)

# Define a function to keep the record with the highest population within
each key
def reduce_duplicates(a, b):
    if a[5] > b[5]:
        return b
    else:
        return a
# Use reduceByKey to eliminate duplicates based on the highest population
within each key
result_rdd = parsed_data_rdd.map(lambda x: (x[0],
x)).reduceByKey(reduce_duplicates).values()
```

```python
# Print the result
for row in result_rdd.collect():
    print("City: {}, Population: {}, Location: {}, {}".format(row[1],
row[5], row[0][0], row[0][1]))
```

```
City: us, Population: 12115, Location: 41.6, -93.5
City: us, Population: 31421, Location: 41.5, -90.6
City: us, Population: 122945, Location: 42.0, -91.7
City: us, Population: 15212, Location: 43.4, -94.2
…
```

2.

```
# Count the number of records in the valid data RDD
record_count = result_rdd.count()


# Print the count of valid records
print(f"Number of valid records: {record_count}")
```

Number of valid records: 38510

---

```
# Extract population values for statistics
population_stats = result_rdd.map(lambda row: row[5])

# Display the statistics
print("Minimum Population: {}".format(population_stats.min()))
print("Maximum Population: {}".format(population_stats.max()))
print("Total Population: {}".format(population_stats.sum()))
print("Average Population: {}".format(population_stats.mean()))
```

```
Minimum Population: 7
Maximum Population: 31480498
Total Population: 1933830945
Average Population: 50216.331991690415
```

3.

```python
# Step 1: Parse the data into tuples (population, 1)
population_data = result_rdd.map(lambda row: (row[0][5], 1))

# Step 2: Group the data by the logarithmic scale of city populations
# Use logarithmic scale for binning
log_bins = [0, 10, 100, 1000, 10000, 100000, 1000000, 10000000,
float('inf')]
population_bins = population_data.map(lambda x: (next(i-1 for i, v in
enumerate(log_bins) if x[0] < v), x[1]))

# Step 3: Count the number of cities in each group
histogram = population_bins.reduceByKey(lambda x, y: x + y)

# Step 4: Sort the results by bin
sorted_histogram = histogram.sortByKey()

# Step 5: Collect the result
result = sorted_histogram.collect()

# Print the sorted result
print(result)
```

```
[(0, 4), (1, 77), (2, 1500), (3, 15184), (4, 17882), (5, 2781), (6, 242), (7, 8)]
```

---

```python
# Define a function to format the result_rdd into the desired format
def format_result(row):
    return (row[1], row[2], row[3], row[4], row[5], row[0][0], row[0][1])

# Apply the format_result function to each element in result_rdd
final_rdd = result_rdd.map(format_result)

# Print the first 5 rows of final_rdd
for row in final_rdd.take(5):
    print("Country: {}, City: {}, AccentCity: {}, Region: {}, Population:
{}, Latitude: {}, Longitude: {}".format(row[0], row[1], row[2], row[3],
row[4], row[5], row[6]))
```

```
Country: ad, City: les escaldes, AccentCity: Les Escaldes, Region: 08,
Population: 15854, Latitude: 42.5, Longitude: 1.5
Country: ad, City: sant julia de loria, AccentCity: Sant Juli� de L�ria,
Region: 06, Population: 8020, Latitude: 42.4, Longitude: 1.5
Country: ae, City: dubai, AccentCity: Dubai, Region: 03, Population:
1137376, Latitude: 25.2, Longitude: 55.3
Country: af, City: carikar, AccentCity: Carikar, Region: 40, Population:
53693, Latitude: 35.0, Longitude: 69.1
```

4.

```
# Step 2: Transform the data to create key-value pairs
# Key: Population, Value: City Information
population_city_rdd = final_rdd.map(lambda x: (x[4], x))
k=10
# Step 3: Use shufflable transformation to find the top 10 cities by
population
top_10_cities = population_city_rdd.sortByKey(ascending=False).take(k)

# Display the result
for i, (population, (country, city, accent_city, region, int_population,
latitude, longitude)) in enumerate(top_10_cities, 1):
    print(f"{i}.country: {country}, City: {city},region: {region},
Population: {int_population}, Location: {latitude}, {longitude}")
```

```
1.country: jp, City: tokyo,region: 40, Population: 31480498, Location:
35.6, 139.7
2.country: cn, City: shanghai,region: 23, Population: 14608512, Location:
31.0, 121.3
3.country: in, City: bombay,region: 16, Population: 12692717, Location:
18.9, 72.8
4.country: pk, City: karachi,region: 05, Population: 11627378, Location:
24.9, 67.0
5.country: in, City: delhi,region: 07, Population: 10928270, Location:
28.6, 77.2
6.country: ph, City: manila,region: D9, Population: 10443877, Location:
14.6, 120.9
7.country: ru, City: moscow,region: 48, Population: 10381288, Location:
55.7, 37.6
8.country: kr, City: seoul,region: 11, Population: 10323448, Location:
37.5, 126.9
9.country: br, City: sao paulo,region: 27, Population: 10021437, Location:
-23.5, -46.7
10.country: tr, City: istanbul,region: 34, Population: 9797536, Location:
41.0, 28.9
```

5.

```
# Assuming final_rdd is the RDD with the desired format (Country, City,
AccentCity, Region, Population, Latitude, Longitude)

# Define the top K value
top_k = 5

# Sort final_rdd by Population in descending order
sorted_rdd = final_rdd.sortBy(lambda x: x[4], ascending=False)

# Take the top K populous cities
top_k_cities = sorted_rdd.take(top_k)

# Calculate the total population
total_population = final_rdd.map(lambda x: x[4]).sum()

# Calculate the population of the top K cities
top_k_population = sum(city[4] for city in top_k_cities)

# Calculate the percentage
percentage_top_k_population = (top_k_population / total_population) * 100

# Print the result
print("Percentage of total population represented by the top {} populous
cities: {:.2f}%".format(top_k, percentage_top_k_population))
```

```
Percentage of total population represented by the top 5 populous cities: 4.21%
```

6.

```
# Map the final_rdd to a new RDD with key-value pairs where the key is the
region and the value is the population
region_population_rdd = final_rdd.map(lambda x: (x[3], x[4]))

# Use reduceByKey to calculate the total population for each region
total_population_rdd = region_population_rdd.reduceByKey(lambda a, b: a +
b)

# Find the region with the highest population
max_population_region = total_population_rdd.max(lambda x: x[1])

# Print the region with the highest population
print("\nRegion with the highest population:")
print("Region: {}, Total Population: {}".format(max_population_region[0],
max_population_region[1]))

# Order the total population of each region by population in descending
order
ordered_total_population_rdd = total_population_rdd.sortBy(lambda x: x[1],
ascending=False)

# Print the total population of each region in descending order
print("\nTotal population of each region (ordered by population in
descending order):")
for row in ordered_total_population_rdd.collect():
    print("Region: {}, Total Population: {}".format(row[0], row[1]))
```

```
Region with the highest population:
Region: 04, Total Population: 95893419

Total population of each region (ordered by population in descending
order):
Region: 04, Total Population: 95893419
Region: 02, Total Population: 83399132
Region: 05, Total Population: 81248168
Region: 07, Total Population: 75975207
Region: 08, Total Population: 62700707

...
```

7.

```python
# Define the specified population threshold
specified_population = 10000000  # Adjust this value as needed

# Create a new RDD with the key as the country and the value as the
population of the city
country_population_rdd = final_rdd.map(lambda x: (x[0], x[4]))

# Use reduceByKey to calculate the sum of populations for each country
country_sum_population_rdd = country_population_rdd.reduceByKey(lambda a,
b: a + b)

# Filter the countries based on the specified population threshold
selected_countries_rdd = country_sum_population_rdd.filter(lambda x: x[1]
> specified_population)

# Print the result
for country in selected_countries_rdd.collect():
    print("Country: {}, Sum of Populations: {}".format(country[0],
country[1]))
```

```
Country: br, Sum of Populations: 122529195
Country: cl, Sum of Populations: 11446732
Country: cn, Sum of Populations: 215141618
Country: es, Sum of Populations: 22901511
Country: id, Sum of Populations: 51283581
Country: mm, Sum of Populations: 11283081
Country: ru, Sum of Populations: 95510179
```

...

8.

```
# Set the population threshold
population_threshold = 100000

# Filter final_rdd to include only cities with population above the
threshold
filtered_rdd = final_rdd.filter(lambda x: x[4] > population_threshold)

# Map the data to (Country, (Population, 1)) to calculate the sum and
count for each country
country_population_rdd = filtered_rdd.map(lambda x: (x[0], (x[4], 1)))

# Reduce by key to calculate the sum and count for each country
sum_count_rdd = country_population_rdd.reduceByKey(lambda a, b: (a[0] +
b[0], a[1] + b[1]))

# Calculate the average population for each country
average_population_rdd = sum_count_rdd.map(lambda x: (x[0], x[1][0] /
x[1][1]))

# Print the result
for row in average_population_rdd.collect():
    print("Country: {}, Average Population: {:.2f}".format(row[0],
row[1]))
```

```
Country: at, Average Population: 657601.33
Country: bg, Average Population: 331211.43
Country: br, Average Population: 416413.68
Country: cl, Average Population: 381148.00
Country: cm, Average Population: 399620.40
```

...

9.

```
# Calculate average population for each country
country_population_avg = final_rdd.map(lambda x: (x[0], (x[4], 1))) \
    .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1])) \
    .mapValues(lambda x: x[0] / x[1])

# Broadcast the average population to all nodes
broadcast_avg = sc.broadcast(dict(country_population_avg.collect()))

# Filter cities with population above their respective country's average
above_avg_cities = final_rdd.filter(lambda x: x[4] >
broadcast_avg.value[x[0]])

# Print the identified cities
for row in above_avg_cities.collect():
    print("Country: {}, City: {}, Population: {}, Latitude: {}, Longitude:
{}".format(row[0], row[1], row[4], row[5], row[6]))
```

```
Country: ht, City: les cayes, Population: 59323, Latitude: 18.2,
Longitude: -73.8
Country: ht, City: petionville, Population: 108235, Latitude: 18.5,
Longitude: -72.3
Country: hu, City: batonyterenye, Population: 14214, Latitude: 47.9,
Longitude: 19.8
Country: hu, City: bicske, Population: 11072, Latitude: 47.4, Longitude:
18.6
…
```

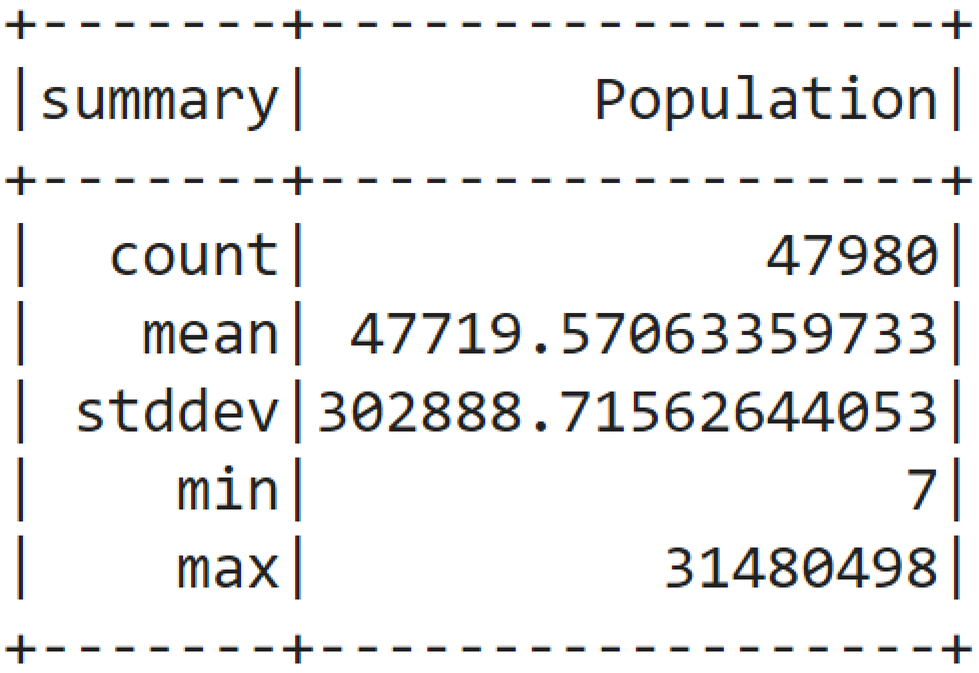
Using dataframe:

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
# Import SparkSession
# Create a Spark Session
spark = SparkSession.builder.master("local[*]").getOrCreate()
# Check Spark Session Information
print(spark)
# Import a Spark function from library
from pyspark.sql.functions import col
# Load the data from Google Drive
file_path = "/content/worldcitiespop.txt"
df = spark.read.csv(file_path, header=True, inferSchema=True)

# Show the first few rows of the DataFrame
df.show()
```

```
<pyspark.sql.session.SparkSession object at 0x7b75f6010730>
+-------+------------------+------------------+------+----------+----------+---------+
|Country|              City|        AccentCity|Region|Population|  Latitude|Longitude|
+-------+------------------+------------------+------+----------+----------+---------+
|     ad|             aixas|             Aix�s|    06|      NULL|42.4833333|1.4666667|
|     ad|        aixirivali|        Aixirivali|    06|      NULL|42.4666667|      1.5|
|     ad|        aixirivall|        Aixirivall|    06|      NULL|42.4666667|      1.5|
|     ad|         aixirvall|         Aixirvall|    06|      NULL|42.4666667|      1.5|
|     ad|          aixovall|          Aixovall|    06|      NULL|42.4666667|1.4833333|
|     ad|           andorra|           Andorra|    07|      NULL|      42.5|1.5166667|
|     ad|  andorra la vella|  Andorra la Vella|    07|     20430|      42.5|1.5166667|
|     ad|   andorra-vieille|   Andorra-Vieille|    07|      NULL|      42.5|1.5166667|
|     ad|           andorre|           Andorre|    07|      NULL|      42.5|1.5166667|
|     ad|andorre-la-vieille|Andorre-la-Vieille|    07|      NULL|      42.5|1.5166667|
|     ad|   andorre-vieille|   Andorre-Vieille|    07|      NULL|      42.5|1.5166667|
|     ad|         ansalonga|         Ansalonga|    04|      NULL|42.5666667|1.5166667|
|     ad|             anyos|             Any�s|    05|      NULL|42.5333333|1.5333333|
|     ad|             arans|             Arans|    04|      NULL|42.5833333|1.5166667|
|     ad|           arinsal|           Arinsal|    04|      NULL|42.5666667|1.4833333|
|     ad|           aubinya|           Aubiny�|    06|      NULL|     42.45|      1.5|
|     ad|           auvinya|           Auvinya|    06|      NULL|     42.45|      1.5|
|     ad|         bicisarri|         Bi�isarri|    06|      NULL|42.4833333|1.4666667|
|     ad|        bixessarri|        Bixessarri|    06|      NULL|42.4833333|1.4666667|
|     ad|         bixisarri|         Bixisarri|    06|      NULL|42.4833333|1.4666667|
+-------+------------------+------------------+------+----------+----------+---------+
only showing top 20 rows
```

```
# Clean the data by removing rows with non-integer values for populations
cleaned_df = df.filter(col("Population").isNotNull())

# Show the cleaned DataFrame
cleaned_df.show()
```
2.

```
# Calculate statistics for city populations
statistics_df = cleaned_df.select(
    col("Population").cast("int").alias("Population")
).describe(["Population"])

# Show the statistics DataFrame
statistics_df.show()
```



```
+-------+------------------+
|summary|        Population|
+-------+------------------+
|  count|             47980|
|   mean| 47719.57063359733|
| stddev|302888.7156264053|
|    min|                 7|
|    max|          31480498|
+-------+------------------+
```

3.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, log10

# Extract the "Population" column and apply a logarithmic transformation
populations =
cleaned_df.select(log10(col("Population").cast("double")).alias("log_popul
ation"))

# Extract the log-transformed population values as an RDD
log_populations_rdd = populations.rdd.flatMap(lambda x: x)
```

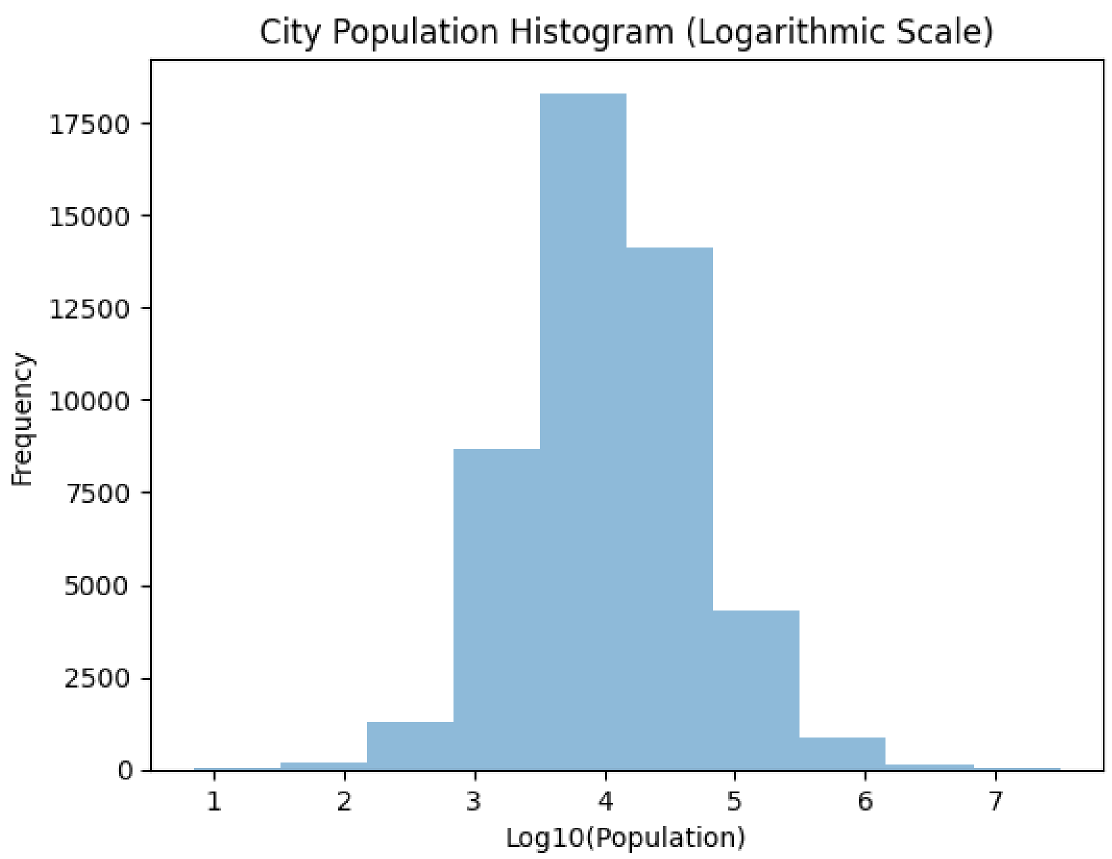
```
# Define the bins for the histogram using a logarithmic scale
bins = [i for i in range(0, 9)]

# Compute the histogram using Spark RDD's histogram method
hist_values = log_populations_rdd.histogram(bins)

# Create a dictionary from the histogram values
hist_dict = dict(zip(hist_values[0][:-1], hist_values[1]))

# Print the histogram as a dictionary
print(hist_dict)
```

```
{0: 5, 1: 174, 2: 2187, 3: 20537, 4: 21550, 5: 3248, 6: 269, 7: 10}
```

---

plotting the histograms

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, log10
import matplotlib.pyplot as plt


# Extract the "Population" column and apply a logarithmic transformation
populations =
cleaned_df.select(log10(col("Population").cast("double")).alias("log_popul
ation"))

# Convert to Pandas DataFrame for plotting
populations_pd = populations.toPandas()

# Plot the histogram
plt.hist(populations_pd["log_population"], bins=10, alpha=0.5)
plt.title("City Population Histogram (Logarithmic Scale)")
plt.xlabel("Log10(Population)")
plt.ylabel("Frequency")
plt.show()
```



4.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Select the columns needed for the calculation
top_cities_df = cleaned_df.select(
        col("Country"),
    col("City"),
    col("AccentCity"),
    col("Population").cast("int").alias("Population"),
    col('Region'),
    col('Latitude'),
    col('Longitude')
)

# Get the top 10 cities by population
top_10_cities = top_cities_df.orderBy(col("Population").desc()).limit(10)

# Show the result
```

```
top_10_cities.show()
```

```
+-------+---------+----------+----------+------+----------+----------+
|Country|     City|AccentCity|Population|Region|  Latitude| Longitude|
+-------+---------+----------+----------+------+----------+----------+
|     jp|    tokyo|     Tokyo|  31480498|    40|    35.685|139.751389|
|     cn| shanghai|  Shanghai|  14608512|    23| 31.045556|121.399722|
|     in|   bombay|    Bombay|  12692717|    16|    18.975| 72.825833|
|     pk|  karachi|   Karachi|  11627378|    05|   24.9056|   67.0822|
|     in|    delhi|     Delhi|  10928270|    07| 28.666667| 77.216667|
|     in|new delhi| New Delhi|  10928270|    07|      28.6|      77.2|
|     ph|   manila|    Manila|  10443877|    D9|   14.6042|  120.9822|
|     ru|   moscow|    Moscow|  10381288|    48| 55.752222| 37.615556|
|     kr|    seoul|     Seoul|  10323448|    11|   37.5985|  126.9783|
|     br|sao paulo| S�o Paulo|  10021437|    27|-23.473293|-46.665803|
+-------+---------+----------+----------+------+----------+----------+
```

5.

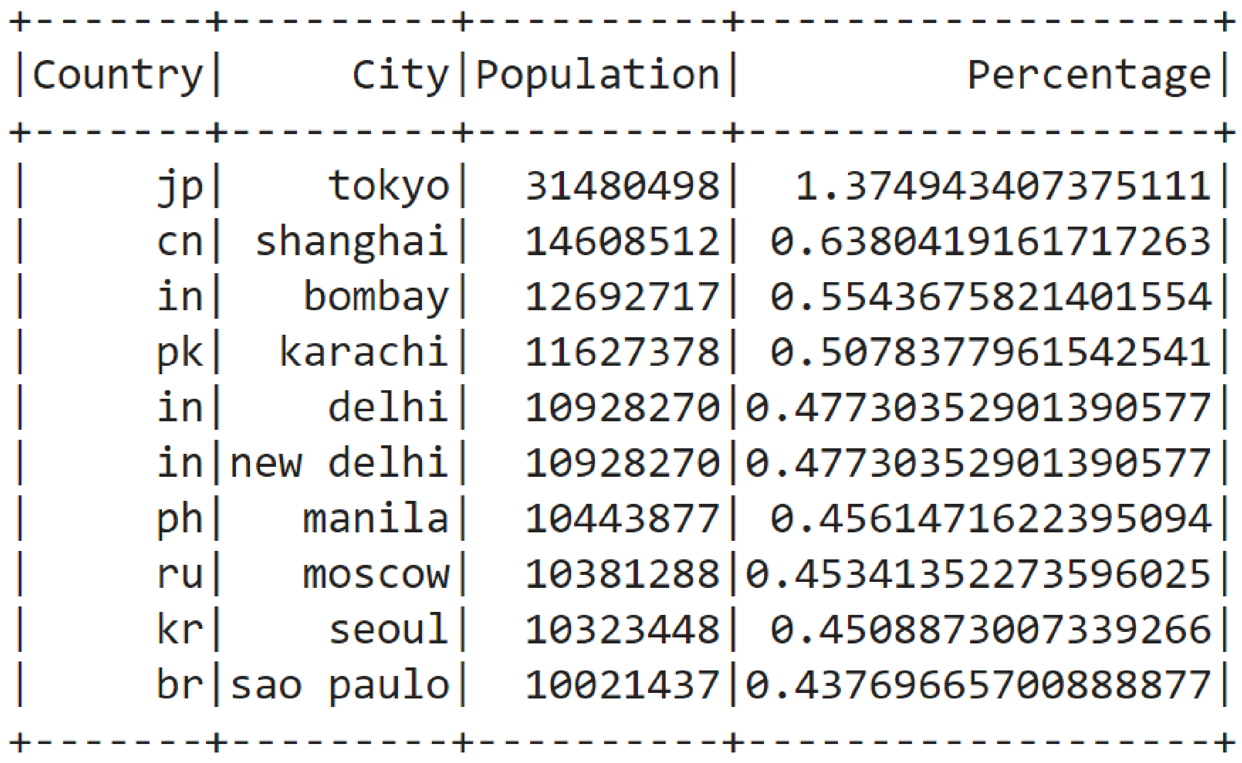
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col


# Select the columns needed for the calculation
top_cities_df = cleaned_df.select(
    col("Country"),
    col("City"),
    col("Population").cast("int").alias("Population")
)

# Define the value of K
K = 10

# Calculate the total population
total_population = top_cities_df.agg({"Population":
"sum"}).collect()[0][0]

# Calculate the total population of the top K cities
top_k_population =
top_cities_df.orderBy(col("Population").desc()).limit(K).agg({"Population"
: "sum"}).collect()[0][0]

# Calculate the percentage
percentage = (top_k_population / total_population) * 100
```

```
print(f"The top {K} cities represent {percentage:.2f}% of the total
population.")
```

The top 10 cities represent 5.83% of the total population.

---

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Select the columns needed for the calculation
top_cities_df = cleaned_df.select(
    col("Country"),
    col("City"),
    col("Population").cast("int").alias("Population")
)

# Define the value of K
K = 10

# Calculate the total population
total_population = top_cities_df.agg({"Population":
"sum"}).collect()[0][0]

# Calculate the total population of the top K cities
top_k_cities_df = top_cities_df.orderBy(col("Population").desc()).limit(K)
top_k_population = top_k_cities_df.agg({"Population":
"sum"}).collect()[0][0]

# Calculate the percentage for each city
percentage_columns = [(col("Population") / total_population *
100).alias(f"Percentage") ]
result_df = top_k_cities_df.select("*", *percentage_columns)

# Show the result
result_df.show()
```



```
+-------+---------+----------+--------------------+
|Country|     City|Population|          Percentage|
+-------+---------+----------+--------------------+
|     jp|    tokyo|  31480498|   1.374943407375111|
|     cn| shanghai|  14608512|  0.6380419161717263|
|     in|   bombay|  12692717|  0.5543675821401554|
|     pk|  karachi|  11627378|  0.5078377961542541|
|     in|    delhi|  10928270| 0.47730352901390577|
|     in|new delhi|  10928270| 0.47730352901390577|
|     ph|   manila|  10443877|  0.4561471622395094|
|     ru|   moscow|  10381288| 0.45341352273596025|
|     kr|    seoul|  10323448|  0.4508873007339266|
|     br|sao paulo|  10021437| 0.43769665700888877|
+-------+---------+----------+--------------------+
```

6.

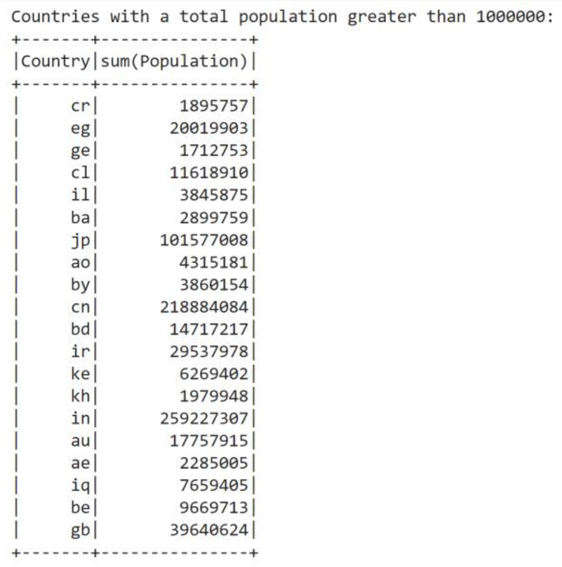
```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col


# Select the columns needed for the calculation
region_population_df = cleaned_df.select(
    col("Region"),
    col("Population").cast("int").alias("Population")
)

# Calculate the total population for each region
region_total_population =
region_population_df.groupBy("Region").agg({"Population":
"sum"}).orderBy(col("sum(Population)").desc())

# Find the region with the highest population
max_population_region = region_total_population.first()["Region"]

# Show the result
print("Total Population of Each Region:")
region_total_population.show()
```

```python
print(f"\nThe region with the highest population is:
{max_population_region}")
```

```
Total Population of Each Region:
+------+---------------+
|Region|sum(Population)|
+------+---------------+
|    04|      112249869|
|    07|       99634521|
|    02|       90271668|
|    05|       88272385|
|    16|       71422264|
|    08|       69386610|
|    09|       68742631|
|    11|       61477197|
|    01|       60187116|
|    06|       58191980|
|    15|       57366367|
|    10|       52912641|
|    13|       50762945|
|    19|       49067537|
|    23|       48056875|
|    30|       47565528|
|    25|       44541463|
|    12|       44192013|
|    40|       43122879|
|    27|       42540035|
+------+---------------+
only showing top 20 rows


The region with the highest population is: 04
```

7.

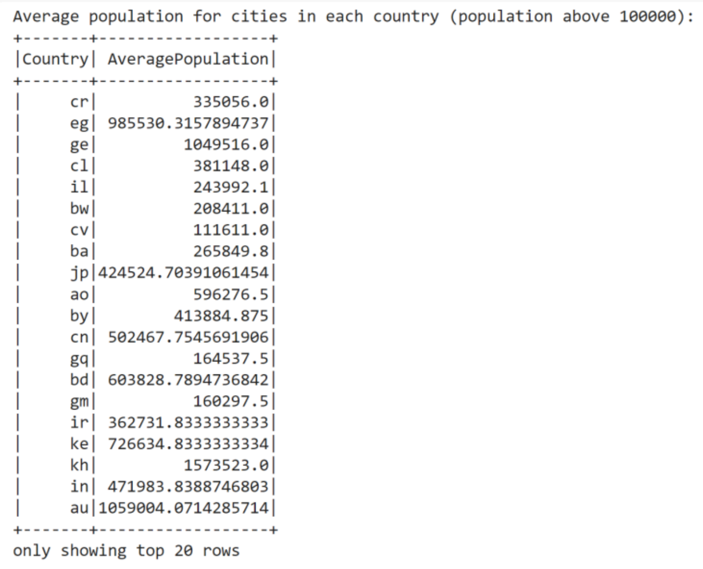
```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col


# Select the columns needed for the calculation
country_population_df = cleaned_df.select(
    col("Country"),
    col("Population").cast("int").alias("Population")
)

# Define the specified value
specified_value = 1000000  # Adjust as needed

# Calculate the total population for each country
country_total_population =
country_population_df.groupBy("Country").agg({"Population": "sum"})

# Filter countries where the sum of populations is greater than the
specified value
selected_countries =
country_total_population.filter(col("sum(Population)") > specified_value)

# Show the result
print(f"Countries with a total population greater than
{specified_value}:")
selected_countries.show()
```



```
Countries with a total population greater than 1000000:
+-------+---------------+
|Country|sum(Population)|
+-------+---------------+
|     cr|        1895757|
|     eg|       20019903|
|     ge|        1712753|
|     cl|       11618910|
|     il|        3845875|
|     ba|        2899759|
|     jp|      101577008|
|     ao|        4315181|
|     by|        3860154|
|     cn|      218884084|
|     bd|       14717217|
|     ir|       29537978|
|     ke|        6269402|
|     kh|        1979948|
|     in|      259227307|
|     au|       17757915|
|     ae|        2285005|
|     iq|        7659405|
|     be|        9669713|
|     gb|       39640624|
+-------+---------------+
```

8.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg

# Select the columns needed for the calculation
country_population_df = cleaned_df.select(
    col("Country"),
    col("Population").cast("int").alias("Population")
)

# Define the population threshold
population_threshold = 100000  # Adjust as needed

# Filter cities with a population above the threshold
filtered_cities_df = country_population_df.filter(col("Population") >
population_threshold)

# Calculate the average population for cities in each country
average_population_by_country =
filtered_cities_df.groupBy("Country").agg(avg("Population").alias("Average
Population"))

# Show the result
print(f"Average population for cities in each country (population above
{population_threshold}):")
average_population_by_country.show()
```



```
Average population for cities in each country (population above 100000):
+-------+------------------+
|Country| AveragePopulation|
+-------+------------------+
|     cr|          335056.0|
|     eg| 985530.3157894737|
|     ge|         1049516.0|
|     cl|          381148.0|
|     il|          243992.1|
|     bw|          208411.0|
|     cv|          111611.0|
|     ba|          265849.8|
|     jp|424524.70391061454|
|     ao|          596276.5|
|     by|        413884.875|
|     cn| 502467.7545691906|
|     gq|          164537.5|
|     bd| 603828.7894736842|
|     gm|          160297.5|
|     ir| 362731.8333333333|
|     ke| 726634.8333333334|
|     kh|         1573523.0|
|     in| 471983.8388746803|
|     au|1059004.0714285714|
+-------+------------------+
only showing top 20 rows
```

9.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg


# Select the columns needed for the calculation
city_population_df = cleaned_df.select(
        col("Country"),
    col("City"),
    col("AccentCity"),
    col("Population").cast("int").alias("Population"),
    col('Region'),
    col('Latitude'),
    col('Longitude')
)

# Calculate the average population for cities in each country
average_population_by_country =
city_population_df.groupBy("Country").agg(avg("Population").alias("Average
Population"))

# Join the original DataFrame with the average population by country
joined_df = city_population_df.join(average_population_by_country,
on="Country")

# Filter cities with a population above the average population of their
respective countries
selected_cities = joined_df.filter(col("Population") >
col("AveragePopulation"))

# Show the result
print("Cities with a population above the average population of their
respective countries:")
selected_cities.show()
```

```
Cities with a population above the average population of their respective countries:
+-------+---------------+---------------+----------+------+---------+----------+------------------+
|Country|           City|     AccentCity|Population|Region| Latitude| Longitude| AveragePopulation|
+-------+---------------+---------------+----------+------+---------+----------+------------------+
|     cr|      turrialba|      Turrialba|     28960|    02| 9.904666|-83.683519|19543.886597938144|
|     cr|    san vicente|    San Vicente|     34453|    08| 9.960158| -84.04762|19543.886597938144|
|     cr|    san vicente|    San Vicente|     34453|    04|  9.98433| -84.08223|19543.886597938144|
|     cr|     san rafael|     San Rafael|     25415|    08| 9.875563|-84.076613|19543.886597938144|
|     cr|san rafael abajo|San Rafael Abajo|     27424|    08| 9.831002|-84.290079|19543.886597938144|
|     cr|      san pedro|      San Pedro|     27482|    08| 9.928293|-84.050735|19543.886597938144|
|     cr|      san pablo|      San Pablo|     21666|    04|10.030131|-84.122144|19543.886597938144|
|     cr|     san miguel|     San Miguel|     28827|    08| 9.871206| -84.06084|19543.886597938144|
|     cr|       san juan|       San Juan|     26051|    08| 9.959738|-84.081653|19543.886597938144|
|     cr|       san jose|       San Jos�|    335056|    08| 9.933333|-84.083333|19543.886597938144|
|     cr|       san jose|       San Jos�|     31430|    01|10.951731|-85.136096|19543.886597938144|
|     cr|  san francisco|  San Francisco|     55933|    08|     9.95|-84.066667|19543.886597938144|
|     cr|  san francisco|  San Francisco|     55933|    04|  10.0035|-84.072127|19543.886597938144|
|     cr|     san felipe|     San Felipe|     24985|    08| 9.904882|-84.105512|19543.886597938144|
|     cr|        quesada|        Quesada|     27316|    01|10.323807|-84.427141|19543.886597938144|
|     cr|         purral|         Purral|     30034|    08| 9.958084|-84.030498|19543.886597938144|
|     cr|     puntarenas|     Puntarenas|     35655|    07| 9.976149|-84.838754|19543.886597938144|
|     cr|        paraiso|        Para�so|     39709|    02|  9.83832|-83.865565|19543.886597938144|
|     cr|       mercedes|       Mercedes|     26013|    04|10.006955|-84.133963|19543.886597938144|
|     cr|          limon|          Lim�n|     63094|    06| 9.992861|-83.030533|19543.886597938144|
+-------+---------------+---------------+----------+------+---------+----------+------------------+
only showing top 20 rows
```

## 10. Recleaning

```
from pyspark.sql.functions import col, format_number

# Multiply Latitude and Longitude values by 10, truncate, and then divide
by 10
rounded_df = cleaned_df.withColumn("Latitude", (col("Latitude") *
10).cast("int").cast("double") / 10)
rounded_df = rounded_df.withColumn("Longitude", (col("Longitude") *
10).cast("int").cast("double") / 10)


# Show the modified DataFrame
rounded_df.show()

from pyspark.sql import SparkSession
from pyspark.sql import Row
import math

# Assuming cleaned_df is your DataFrame with the original data
schema = ["Country", "AccentCity", "City", "Region", "Population",
"Latitude", "Longitude"]

# Parse the DataFrame into the desired format
def parse_dataframe(row):
    latitude = math.floor(row["Latitude"] * 10) / 10
    longitude = math.floor(row["Longitude"] * 10) / 10
    country = row["Country"]
    region = row["Region"]
    return ((country, region, latitude, longitude), row)

parsed_data_rdd = cleaned_df.rdd.map(parse_dataframe).reduceByKey(lambda
a, b: b if a["Population"] > b["Population"] else a).values()

# Convert RDD to DataFrame
result_df = parsed_data_rdd.map(lambda x: Row(**x.asDict())).toDF(schema)

# Show the result
result_df.show()
```

10.2

```
# Calculate statistics for city populations
statistics_df = result_df.select(
    col("Population").cast("int").alias("Population")
).describe(["Population"])

# Show the statistics DataFrame
statistics_df.show()
```

10.3

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, log10

# Extract the "Population" column and apply a logarithmic transformation
populations =
result_df.select(log10(col("Population").cast("double")).alias("log_popula
tion"))

# Extract the log-transformed population values as an RDD
log_populations_rdd = populations.rdd.flatMap(lambda x: x)

# Define the bins for the histogram using a logarithmic scale
bins = [i for i in range(0, 9)]

# Compute the histogram using Spark RDD's histogram method
hist_values = log_populations_rdd.histogram(bins)

# Create a dictionary from the histogram values
hist_dict = dict(zip(hist_values[0][:-1], hist_values[1]))

# Print the histogram as a dictionary
```

```
print(hist_dict)

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, log10
import matplotlib.pyplot as plt


# Extract the "Population" column and apply a logarithmic transformation
populations = 
result_df.select(log10(col("Population").cast("double")).alias("log_popula
tion"))

# Convert to Pandas DataFrame for plotting
populations_pd = populations.toPandas()

# Plot the histogram
plt.hist(populations_pd["log_population"], bins=10, alpha=0.5)
plt.title("City Population Histogram (Logarithmic Scale)")
plt.xlabel("Log10(Population)")
plt.ylabel("Frequency")
plt.show()
```

10.4

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Select the columns needed for the calculation
top_cities_df = result_df.select(
    col("Country"),
    col("City"),
    col("AccentCity"),
    col("Population").cast("int").alias("Population"),
    col('Region'),
    col('Latitude'),
    col('Longitude')
)

# Get the top 10 cities by population
top_10_cities = top_cities_df.orderBy(col("Population").desc()).limit(10)

# Show the result
top_10_cities.show()
```

10.5

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col


# Select the columns needed for the calculation
top_cities_df = result_df.select(
    col("Country"),
    col("City"),
    col("Population").cast("int").alias("Population")
)

# Define the value of K
K = 5

# Calculate the total population
total_population = top_cities_df.agg({"Population":
"sum"}).collect()[0][0]

# Calculate the total population of the top K cities
top_k_population =
top_cities_df.orderBy(col("Population").desc()).limit(K).agg({"Population"
: "sum"}).collect()[0][0]

# Calculate the percentage
percentage = (top_k_population / total_population) * 100

print(f"The top {K} cities represent {percentage:.2f}% of the total
population.")
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
final_df=result_df
# Select the columns needed for the calculation
top_cities_df = final_df.select(
    col("Country"),
    col("City"),
    col("Population").cast("int").alias("Population")
)

# Define the value of K
K = 10
```

```
# Calculate the total population
total_population = top_cities_df.agg({"Population":
"sum"}).collect()[0][0]

# Calculate the total population of the top K cities
top_k_cities_df = top_cities_df.orderBy(col("Population").desc()).limit(K)
top_k_population = top_k_cities_df.agg({"Population":
"sum"}).collect()[0][0]

# Calculate the percentage for each city
percentage_columns = [(col("Population") / total_population *
100).alias(f"Percentage") ]
result_df = top_k_cities_df.select("*", *percentage_columns)

# Show the result
result_df.show()
```

10.6

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col


# Select the columns needed for the calculation
region_population_df = final_df.select(
    col("Region"),
    col("Population").cast("int").alias("Population")
)

# Calculate the total population for each region
region_total_population =
region_population_df.groupBy("Region").agg({"Population":
"sum"}).orderBy(col("sum(Population)").desc())

# Find the region with the highest population
max_population_region = region_total_population.first()["Region"]

# Show the result
print("Total Population of Each Region:")
region_total_population.show()

print(f"\nThe region with the highest population is:
{max_population_region}")
```

10.7

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col


# Select the columns needed for the calculation
country_population_df = final_df.select(
    col("Country"),
    col("Population").cast("int").alias("Population")
)

# Define the specified value
specified_value = 1000000  # Adjust as needed

# Calculate the total population for each country
country_total_population =
country_population_df.groupBy("Country").agg({"Population": "sum"})

# Filter countries where the sum of populations is greater than the
specified value
selected_countries =
country_total_population.filter(col("sum(Population)") > specified_value)

# Show the result
print(f"Countries with a total population greater than
{specified_value}:")
selected_countries.show()
```

10.8

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg

# Select the columns needed for the calculation
country_population_df = final_df.select(
    col("Country"),
    col("Population").cast("int").alias("Population")
)

# Define the population threshold
population_threshold = 100000  # Adjust as needed

# Filter cities with a population above the threshold
filtered_cities_df = country_population_df.filter(col("Population") >
population_threshold)
```

```
# Calculate the average population for cities in each country
average_population_by_country =
filtered_cities_df.groupBy("Country").agg(avg("Population").alias("Average
Population"))

# Show the result
print(f"Average population for cities in each country (population above
{population_threshold}):")
average_population_by_country.show()
```

10.9

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg


# Select the columns needed for the calculation
city_population_df = final_df.select(
        col("Country"),
    col("City"),
    col("AccentCity"),
    col("Population").cast("int").alias("Population"),
    col('Region'),
    col('Latitude'),
    col('Longitude')
)

# Calculate the average population for cities in each country
average_population_by_country =
city_population_df.groupBy("Country").agg(avg("Population").alias("Average
Population"))

# Join the original DataFrame with the average population by country
joined_df = city_population_df.join(average_population_by_country,
on="Country")

# Filter cities with a population above the average population of their
respective countries
selected_cities = joined_df.filter(col("Population") >
col("AveragePopulation"))

# Show the result
print("Cities with a population above the average population of their
respective countries:")
selected_cities.show()
```