

Getting Started with PySpark and Pandas

Estimated time needed: **60** minutes

PySpark is the Python API for Apache Spark, a distributed computing system designed for handling large-scale data processing. Its ability to perform computations across multiple nodes makes it an ideal choice for Big Data scenarios, where data sets can reach terabytes or even petabytes in size. PySpark allows data scientists and analysts to harness the power of distributed computing, scaling up their workflows and processing capabilities significantly.

In contrast, Pandas is a powerful library tailored for data manipulation and analysis in Python, primarily used for handling structured data. It provides rich functionality for data cleaning, transformation, aggregation, and visualization, making it particularly suited for smaller data sets typically fitting into memory. Pandas excels at operations requiring quick turnaround times for data analysis and exploration, facilitating tasks such as data wrangling and exploratory data analysis.

To illustrate the strengths of both PySpark and Pandas, we will utilize a sample COVID-19 data set containing daily cases, deaths, and vaccinations across various continents.

Objectives

- Understand PySpark and Pandas: Explain the core functionalities and use cases of PySpark for big data processing and Pandas for data manipulation.

- Set up the environment: Install and configure PySpark and Pandas to work together in a Python environment.

- Load and explore data: Import data into Pandas and PySpark DataFrames and perform basic data exploration.

- Convert between DataFrames: Convert a Pandas DataFrame to a Spark DataFrame for distributed processing.

- Perform data manipulation: Create new columns, filter data, and perform aggregations using PySpark.

- Utilize SQL queries: Use Spark SQL for querying data and leveraging user-defined functions (UDFs).

1. PySpark overview

Overview

PySpark is the Python API for Apache Spark, designed for large-scale data processing and analysis. It offers tools for working with RDDs and DataFrames, enabling efficient, fault-tolerant distributed computing.

Key features

- **Distributed computing:** Handles data across multiple nodes in a cluster.
- **High performance:** Outperforms traditional frameworks in speed.
- **Big data handling:** Manages data sets larger than a single machine's memory.
- **Python integration:** Compatible with Python libraries like Pandas and NumPy.

Use cases

- **Large-scale data processing:** Ideal for processing large volumes of data that exceed the capacity of a single machine.
- **Data analysis:** Useful for complex data manipulations and analysis using distributed computing.

Strengths

- High-speed data processing.
- Fault-tolerant and scalable.

Limitations

- Complex setup and configuration.
- Steeper learning curve compared to some data processing tools.

2. Understanding Pandas

Overview

Pandas is a Python library designed for data manipulation and analysis. It provides two primary data structures: Series and DataFrame, which facilitate handling and organizing structured data.

Key features

- **Data structures:** Series for one-dimensional data and DataFrame for two-dimensional data.
- **Data I/O:** Reads and writes data in various formats such as CSV, Excel, and SQL.
- **Data cleaning:** Functions for handling missing or duplicate data.
- **Data analysis:** Includes statistical functions for detailed data analysis.

Use cases

- **Data manipulation:** Efficient handling of structured data.
- **Data analysis:** Comprehensive analysis and transformation of data sets.

Strengths

- User-friendly API for data manipulation.
- Extensive support for various data formats.

Limitations

- Limited scalability for extremely large datasets compared to distributed frameworks.

3. Setting up the environment

Installation

- First, let's install the necessary libraries if they are not already installed.

In [1]:
```
!pip install pyspark
```

```
Collecting pyspark
  Downloading pyspark-3.4.4.tar.gz (311.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 311.4/311.4 MB 1.2 MB/s eta
0:00:0000:0100:01
  Preparing metadata (setup.py) ... done
Collecting py4j==0.10.9.7 (from pyspark)
  Downloading py4j-0.10.9.7-py2.py3-none-any.whl (200 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 200.5/200.5 kB 18.2 MB/s eta
0:00:00
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.4.4-py2.py3-none-any.whl s
ize=311905466 sha256=91e6c3163338098b446d66a12aa2d8959a5fbfc0ce1ff110e8ecb
747e1e01e1c
  Stored in directory: /home/jupyterlab/.cache/pip/wheels/4e/66/db/939eb1c
49afb8a7fd2c4e393ad34e12b77db67bb4cc974c00e
Successfully built pyspark
Installing collected packages: py4j, pyspark
Successfully installed py4j-0.10.9.7 pyspark-3.4.4
```

In [2]:
```
!pip install findspark
```

```
Collecting findspark
  Downloading findspark-2.0.1-py2.py3-none-any.whl (4.4 kB)
Installing collected packages: findspark
Successfully installed findspark-2.0.1
```

In [3]:
```
!pip install pandas
```

```
Requirement already satisfied: pandas in /home/jupyterlab/conda/envs/pytho
n/lib/python3.7/site-packages (1.3.5)
Requirement already satisfied: python-dateutil>=2.7.3 in /home/jupyterlab/
conda/envs/python/lib/python3.7/site-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in /home/jupyterlab/conda/env
s/python/lib/python3.7/site-packages (from pandas) (2023.3)
Requirement already satisfied: numpy>=1.17.3 in /home/jupyterlab/conda/env
s/python/lib/python3.7/site-packages (from pandas) (1.21.6)
Requirement already satisfied: six>=1.5 in /home/jupyterlab/conda/envs/pyt
hon/lib/python3.7/site-packages (from python-dateutil>=2.7.3->pandas) (1.1
6.0)
```

4. Initializing a Spark session

A Spark session is crucial for working with PySpark. It enables DataFrame creation, data loading, and various operations.

Importing libraries

- `findspark` is used to locate the Spark installation.
- `pandas` is imported for data manipulation.

Creating a Spark session

- `SparkSession.builder.appName("COVID-19 Data Analysis").getOrCreate()` initializes a Spark session with the specified application name.

Checking Spark session

- The code checks if the Spark session is active and prints an appropriate message.

In [4]:
```
import findspark  # This helps us find and use Apache Spark
findspark.init()  # Initialize findspark to locate Spark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, Intege
import pandas as pd
# Initialize a Spark Session
spark = SparkSession \
    .builder \
    .appName("COVID-19 Data Analysis") \
    .config("spark.sql.execution.arrow.pyspark.enabled", "true") \
    .getOrCreate()

# Check if the Spark Session is active
if 'spark' in locals() and isinstance(spark, SparkSession):
    print("SparkSession is active and ready to use.")
else:
    print("SparkSession is not active. Please create a SparkSession.")
```

```
25/10/12 08:47:56 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setL
ogLevel(newLevel).
SparkSession is active and ready to use.
```

5. Importing data into Pandas from various sources

Let's read the COVID-19 data from the provided URL.

In [5]:
```
# Read the COVID-19 data from the provided URL
vaccination_data = pd.read_csv('https://cf-courses-data.s3.us.cloud-objec
```

6. Displaying the first five records

To retrieve and print the first five records

- `vaccination_data.head()` retrieves the first five rows of the DataFrame vaccination_data.This gives us a quick look at the data contained within the data set.

- The `print` function is used to display a message indicating what is being shown, followed by the actual data.

Selecting specific columns:

- Let's define a list called `columns_to_display`, which contains the names of the columns as : `['continent', 'total_cases', 'total_deaths', 'total_vaccinations', 'population']`.
- By using `vaccination_data[columns_to_display].head()`, let's filter the DataFrame to only show the specified columns and again display the first five records of this subset.
- The continent column is explicitly converted to string, while the numeric columns (total cases, total deaths, total vaccinations, population) are filled with zeros for NaN values and then converted to int64 (which is compatible with LongType in Spark).
- The use of fillna(0) ensures that NaN values do not cause type issues during the Spark DataFrame creation.

In [6]:
```
print("Displaying the first 5 records of the vaccination data:")
columns_to_display = ['continent', 'total_cases', 'total_deaths', 'total_
# Show the first 5 records
print(vaccination_data[columns_to_display].head())
```

```
Displaying the first 5 records of the vaccination data:
  continent  total_cases  total_deaths  total_vaccinations    population
0      Asia     235214.0        7998.0                 NaN  4.112877e+07
1       NaN   13145380.0      259117.0                 NaN  1.426737e+09
2    Europe     335047.0        3605.0                 NaN  2.842318e+06
3    Africa     272139.0        6881.0                 NaN  4.490323e+07
4   Oceania       8359.0          34.0                 NaN  4.429500e+04
```

7. Converting the Pandas DataFrame to a Spark DataFrame

Let's convert the Pandas DataFrame, which contains our COVID-19 vaccination data, into a Spark DataFrame. This conversion is crucial as it allows us to utilize Spark's distributed computing capabilities, enabling us to handle larger datasets and perform operations in a more efficient manner.

Defining the schema:

- **StructType**:
  - A class that defines a structure for a DataFrame.
- **StructField**:
  - Represents a single field in the schema.
  - **Parameters**:

1. **Field name**: The name of the field.
2. **Data type**: The type of data for the field.
3. **Nullable**: A boolean indicating whether null values are allowed.

- **Data types**:
  - **StringType()**: Used for text fields.
  - **LongType()**: Used for numerical fields.

Data type conversion:

- **astype(str)**:
  - Used to convert the `'continent'` column to string type.
- **fillna(0)**:
  - Replaces any NaN values with 0, ensuring that the numerical fields do not contain any missing data.
- **astype('int64')**:
  - Converts the columns from potentially mixed types to 64-bit integers for consistent numerical representation.

Creating a Spark DataFrame:

- **createDataFrame**:
  - The `createDataFrame` method of the Spark session (`spark`) is called with `vaccination_data` (the Pandas DataFrame) as its argument.
  - **Parameters**:
    - It takes as input a subset of the pandas DataFrame that corresponds to the fields defined in the schema, accessed using `schema.fieldNames()` .
- This function automatically converts the Pandas DataFrame into a Spark DataFrame, which is designed to handle larger data sets across a distributed environment.

- The resulting spark_df will have the defined schema, which ensures consistency and compatibility with Spark's data processing capabilities.

Storing the result:

In [7]:
```
# Convert to Spark DataFrame directly
# Define the schema
schema = StructType([
    StructField("continent", StringType(), True),
    StructField("total_cases", LongType(), True),
    StructField("total_deaths", LongType(), True),
    StructField("total_vaccinations", LongType(), True),
```

```
    StructField("population", LongType(), True)
])

# Convert the columns to the appropriate data types
vaccination_data['continent'] = vaccination_data['continent'].astype(str)
vaccination_data['total_cases'] = vaccination_data['total_cases'].fillna(
vaccination_data['total_deaths'] = vaccination_data['total_deaths'].filln
vaccination_data['total_vaccinations'] = vaccination_data['total_vaccinat
vaccination_data['population'] = vaccination_data['population'].fillna(0)

spark_df = spark.createDataFrame(vaccination_data[schema.fieldNames()])
# Show the Spark DataFrame
spark_df.show()
```

```
[Stage 0:>                                                          (0 +
1) / 1]
+-------------+-----------+------------+------------------+----------+
|    continent|total_cases|total_deaths|total_vaccinations|population|
+-------------+-----------+------------+------------------+----------+
|         Asia|     235214|        7998|                 0|  41128772|
|          nan|   13145380|      259117|                 0|1426736614|
|       Europe|     335047|        3605|                 0|   2842318|
|       Africa|     272139|        6881|                 0|  44903228|
|      Oceania|       8359|          34|                 0|     44295|
|       Europe|      48015|         159|                 0|     79843|
|       Africa|     107481|        1937|                 0|  35588996|
|North America|       3904|          12|                 0|     15877|
|North America|       9106|         146|                 0|     93772|
|South America|   10101218|      130663|                 0|  45510324|
|         Asia|     452273|        8777|                 0|   2780472|
|North America|      44224|         292|                 0|    106459|
|          nan|  301499099|     1637249|        9104304615|4721383370|
|      Oceania|   11861161|       25236|                 0|  26177410|
|       Europe|    6082444|       22534|                 0|   8939617|
|         Asia|     835757|       10353|                 0|  10358078|
|North America|      39127|         849|                 0|    409989|
|         Asia|     696614|        1536|                 0|   1472237|
|         Asia|    2051348|       29499|                 0| 171186368|
|North America|     108582|         593|                 0|    281646|
+-------------+-----------+------------+------------------+----------+
only showing top 20 rows
```

8. Checking the structure of the Spark DataFrame

In this section, Let's examine the structure of the Spark DataFrame that we created from the Pandas DataFrame. Understanding the schema of a DataFrame is crucial as it provides insight into the data types of each column and helps ensure that the data is organized correctly for analysis.

Displaying the schema:

- The method `spark_df.printSchema()` is called to output the structure of the Spark DataFrame.

- This method prints the names of the columns along with their data types (e.g., `StringType`, `IntegerType`, `DoubleType`, etc.), providing a clear view of how the data is organized.

In [8]:
```
print("Schema of the Spark DataFrame:")
spark_df.printSchema()
# Print the structure of the DataFrame (columns and types)
```

```
Schema of the Spark DataFrame:
root
 |-- continent: string (nullable = true)
 |-- total_cases: long (nullable = true)
 |-- total_deaths: long (nullable = true)
 |-- total_vaccinations: long (nullable = true)
 |-- population: long (nullable = true)
```

9. Basic data exploration

In this section, let's perform basic data exploration on the Spark DataFrame. This step is essential for understanding the data set better, allowing us to gain insights and identify any patterns or anomalies. Let's demonstrate how to view specific contents of the DataFrame, select certain columns, and filter records based on conditions.

9.1 Viewing DataFrame contents

- To view the contents in the DataFrame, use the following code:

In [9]:
```
# List the names of the columns you want to display
columns_to_display = ['continent', 'total_cases', 'total_deaths', 'total_
# Display the first 5 records of the specified columns
spark_df.select(columns_to_display).show(5)
```

```
+---------+-----------+------------+------------------+----------+
|continent|total_cases|total_deaths|total_vaccinations|population|
+---------+-----------+------------+------------------+----------+
|     Asia|     235214|        7998|                 0|  41128772|
|      nan|   13145380|      259117|                 0|1426736614|
|   Europe|     335047|        3605|                 0|   2842318|
|   Africa|     272139|        6881|                 0|  44903228|
|  Oceania|       8359|          34|                 0|     44295|
+---------+-----------+------------+------------------+----------+
only showing top 5 rows
```

9.2 Picking specific columns

- To display only certain columns, use the following code:

In [10]:
```
print("Displaying the 'continent' and 'total_cases' columns:")
# Show only the 'continent' and 'total_cases' columns
```

```
spark_df.select('continent', 'total_cases').show(5)
```

```
Displaying the 'continent' and 'total_cases' columns:
+---------+-----------+
|continent|total_cases|
+---------+-----------+
|     Asia|     235214|
|      nan|   13145380|
|   Europe|     335047|
|   Africa|     272139|
|  Oceania|       8359|
+---------+-----------+
only showing top 5 rows
```

9.3 Sifting Through Data

- To filter records based on a specific condition, use the following code:

In [11]:
```
print("Filtering records where 'total_cases' is greater than 1,000,000:")
 # Show records with more than 1 million total cases
spark_df.filter(spark_df['total_cases'] > 1000000).show(5)
```

```
Filtering records where 'total_cases' is greater than 1,000,000:
+-------------+-----------+------------+------------------+----------+
|    continent|total_cases|total_deaths|total_vaccinations|population|
+-------------+-----------+------------+------------------+----------+
|          nan|   13145380|      259117|                 0|1426736614|
|South America|   10101218|      130663|                 0|  45510324|
|          nan|  301499099|     1637249|        9104304615|4721383370|
|      Oceania|   11861161|       25236|                 0|  26177410|
|       Europe|    6082444|       22534|                 0|   8939617|
+-------------+-----------+------------+------------------+----------+
only showing top 5 rows
```

10. Working with columns

In this section, let's create a new column called `death_percentage`, which calculates the death rate during the COVID-19 pandemic. This calculation is based on the total_deaths (the count of deaths) and the population (the total population) columns in our Spark DataFrame. This new metric will provide valuable insight into the impact of COVID-19 in different regions.

- Let's import the functions module from `pyspark.sql` as `F`, which contains built-in functions for DataFrame operations.

Calculating the death percentage:

- Let's create a new DataFrame `spark_df_with_percentage` by using the `withColumn()` method to add a new column called `death_percentage`.

- The formula `(spark_df['total_deaths'] / spark_df['population']) * 100` computes the death percentage by dividing the total deaths by the total population and multiplying by 100.

Formatting the percentage:

- Let's update the `death_percentage` column to format its values to two decimal places using `F.format_number()`, and concatenate a percentage symbol using `F.concat()` and `F.lit('%')`.
- This makes the death percentage easier to read and interpret.

Selecting relevant columns:

- Let's define a list `columns_to_display` that includes `'total_deaths', 'population', 'death_percentage', 'continent', 'total_vaccinations', and 'total_cases'`.
- Finally, let's display the first five records of the modified DataFrame with the new column by calling `spark_df_with_percentage.select(columns_to_display).show(5)`.

In [12]:
```
from pyspark.sql import functions as F

spark_df_with_percentage = spark_df.withColumn(
    'death_percentage',
    (spark_df['total_deaths'] / spark_df['population']) * 100
)
spark_df_with_percentage = spark_df_with_percentage.withColumn(
    'death_percentage',
    F.concat(
        # Format to 2 decimal places
        F.format_number(spark_df_with_percentage['death_percentage'], 2),
        # Append the percentage symbol
        F.lit('%')
    )
)
columns_to_display = ['total_deaths', 'population', 'death_percentage', '
spark_df_with_percentage.select(columns_to_display).show(5)
```

```
+------------+----------+----------------+---------+------------------+---
--------+
|total_deaths|population|death_percentage|continent|total_vaccinations|tot
al_cases|
+------------+----------+----------------+---------+------------------+---
--------+
|        7998|  41128772|           0.02%|     Asia|                 0|
235214|
|      259117|1426736614|           0.02%|      nan|                 0|
13145380|
|        3605|   2842318|           0.13%|   Europe|                 0|
335047|
|        6881|  44903228|           0.02%|   Africa|                 0|
272139|
|          34|     44295|           0.08%|  Oceania|                 0|
8359|
+------------+----------+----------------+---------+------------------+---
--------+
only showing top 5 rows
```

11. Grouping and summarizing

Let's calculate the total number of deaths per continent using the data in our Spark DataFrame. Grouping and summarizing data is a crucial aspect of data analysis, as it allows us to aggregate information and identify trends across different categories.

Grouping the data

The `spark_df.groupby(['continent'])` method groups the data by the `continent` column. This means that all records associated with each continent will be aggregated together.

Aggregating the deaths

The `agg({"total_deaths": "SUM"})` function is used to specify the aggregation operation. In this case, we want to calculate the sum of the `total_deaths` for each continent. This operation will create a new DataFrame where each continent is listed alongside the total number of deaths attributed to it.

Displaying the results

The `show()` method is called to display the results of the aggregation. This will output the total number of deaths for each continent in a tabular format.

In [13]:
```
print("Calculating the total deaths per continent:")
# Group by continent and sum total death rates
spark_df.groupby(['continent']).agg({"total_deaths": "SUM"}).show()
```

```
Calculating the total deaths per continent:
[Stage 14:==============================>                          (43 + 14)
/ 75]
```

```
+-------------+-----------------+
|    continent|sum(total_deaths)|
+-------------+-----------------+
|       Europe|          2102483|
|       Africa|           259117|
|          nan|         22430618|
|North America|          1671178|
|South America|          1354187|
|      Oceania|            32918|
|         Asia|          1637249|
+-------------+-----------------+
```

12. Exploring user-defined functions (UDFs)

UDFs in PySpark allow us to create custom functions that can be applied to individual columns within a DataFrame. This feature provides increased flexibility and customization in data processing, enabling us to define specific transformations or calculations that are not available through built-in functions. In this section, let's define a UDF to convert total deaths in the dataset.

Importing pandas_udf

The `pandas_udf` function is imported from `pyspark.sql.functions`. This decorator allows us to define a UDF that operates on Pandas Series

Defining the UDF

This function `convert_total_deaths()` takes in a parameter `total_deaths` and returns double its value. You can replace the logic with any transformation you want to apply to the column data.

Registering the UDF

The line `spark.udf.register("convert_total_deaths", convert_total_deaths, IntegerType())` registers the UDF with Spark indicating that the function returns an integer, allowing us to use it in Spark SQL queries and DataFrame operations.

In [14]:
```
from pyspark.sql import SparkSession
from pyspark.sql.types import IntegerType
# Function definition
def convert_total_deaths(total_deaths):
    return total_deaths * 2
# Here you can define any transformation you want
# Register the UDF with Spark
spark.udf.register("convert_total_deaths", convert_total_deaths, IntegerT
```

Out[14]:
```
<function __main__.convert_total_deaths(total_deaths)>
```

13. Using Spark SQL

Spark SQL enables us to execute SQL queries directly on DataFrames.

In [15]:
```
# Drop the existing temporary view if it exists
spark.sql("DROP VIEW IF EXISTS data_v")

# Create a new temporary view
spark_df.createTempView('data_v')

# Execute the SQL query using the UDF
spark.sql('SELECT continent, total_deaths, convert_total_deaths(total_dea
```

```
+-------------+------------+----------------------+
|    continent|total_deaths|converted_total_deaths|
+-------------+------------+----------------------+
|         Asia|        7998|                 15996|
|          nan|      259117|                518234|
|       Europe|        3605|                  7210|
|       Africa|        6881|                 13762|
|      Oceania|          34|                    68|
|       Europe|         159|                   318|
|       Africa|        1937|                  3874|
|North America|          12|                    24|
|North America|         146|                   292|
|South America|      130663|                261326|
|         Asia|        8777|                 17554|
|North America|         292|                   584|
|          nan|     1637249|               3274498|
|      Oceania|       25236|                 50472|
|       Europe|       22534|                 45068|
|         Asia|       10353|                 20706|
|North America|         849|                  1698|
|         Asia|        1536|                  3072|
|         Asia|       29499|                 58998|
|North America|         593|                  1186|
+-------------+------------+----------------------+
only showing top 20 rows
```

14. Running SQL queries

In this step, let's execute SQL queries to retrieve specific records from the temporary view which was created earlier. Let's demonstrate how to display all records from the data table and filter those records based on vaccination totals. This capability allows for efficient data exploration and analysis using SQL syntax.

Displaying All Records

The first query retrieves all records from the temporary view data using the SQL command SELECT * FROM data_v. The show() method is called to display the results in a tabular format. This is useful for getting an overview of the entire dataset.

In [16]:
```
spark.sql('SELECT * FROM data_v').show()
```

```
+-------------+-----------+------------+------------------+----------+
|    continent|total_cases|total_deaths|total_vaccinations|population|
+-------------+-----------+------------+------------------+----------+
|         Asia|     235214|        7998|                 0|  41128772|
|          nan|   13145380|      259117|                 0|1426736614|
|       Europe|     335047|        3605|                 0|   2842318|
|       Africa|     272139|        6881|                 0|  44903228|
|      Oceania|       8359|          34|                 0|     44295|
|       Europe|      48015|         159|                 0|     79843|
|       Africa|     107481|        1937|                 0|  35588996|
|North America|       3904|          12|                 0|     15877|
|North America|       9106|         146|                 0|     93772|
|South America|   10101218|      130663|                 0|  45510324|
|         Asia|     452273|        8777|                 0|   2780472|
|North America|      44224|         292|                 0|    106459|
|          nan|  301499099|     1637249|        9104304615|4721383370|
|      Oceania|   11861161|       25236|                 0|  26177410|
|       Europe|    6082444|       22534|                 0|   8939617|
|         Asia|     835757|       10353|                 0|  10358078|
|North America|      39127|         849|                 0|    409989|
|         Asia|     696614|        1536|                 0|   1472237|
|         Asia|    2051348|       29499|                 0| 171186368|
|North America|     108582|         593|                 0|    281646|
+-------------+-----------+------------+------------------+----------+
only showing top 20 rows
```

Filtering records

The second query is designed to filter the data set to show only those continents where the total vaccinations exceed 1 million. The SQL command used here is `SELECT continent FROM data_v WHERE total_vaccinations > 1000000`. The `show()` method is again used to display the results, specifically listing the continents that meet the filter criteria.

In [17]:
```
print("Displaying continent with total vaccinated more than 1 million:")
# SQL filtering
spark.sql("SELECT continent FROM data_v WHERE total_vaccinations > 100000
```

```
Displaying continent with total vaccinated more than 1 million:
+-------------+
|    continent|
+-------------+
|          nan|
|North America|
|       Europe|
|       Europe|
|          nan|
|          nan|
|          nan|
|         Asia|
|         Asia|
|       Europe|
|          nan|
|         Asia|
|      Oceania|
|          nan|
|          nan|
|          nan|
|          nan|
+-------------+
```

Conclusion

This hands-on-lab is designed to help you understand the core functionalities of robust tools for data manipulation and analysis. By mastering these tools, you gain the ability to choose the right one for your data processing needs, whether working with large data sets requiring distributed computing or smaller data sets that lend themselves to quick manipulation in memory.

Author

Ritika Joshi

<!--## Change Log |Date (YYYY-MM-DD)|Version|Changed By|Change Description| |-|-|-|-| |2024-09-23|0.1|Ritika|First Draft| --!>

In [ ]: