Assignment week 5

By Mohammad Movahedi

#introduction This assignment delves into the realm of Big Data Processing using Apache Spark, focusing on a Word Count application applied to the "Meditation" text from the Gothenburg Project, all within the Google Colab environment. Following the comprehensive instructions outlined in the "Big Data Processing with Apache Spark – Part 1: Introduction," I installed the Java Development Kit (JDK) and Apache Spark. Subsequently, I adapted the Word Count application to analyze the contents of the "Meditation" text. This assignment is a practical exploration of Apache Spark's capabilities for processing extensive datasets and extracting meaningful insights from textual content.

Analysis

In this part I will perform the following tasks:

- Download the text file from Project Gutenberg
- Read the text file into an RDD
- Perform Word Count
- Find the Most Common Word
- Find the Word with the Fewest Occurrences

This code segment installs Apache Spark in Google Colab. It begins by installing the OpenJDK 8 JDK (Java Development Kit) and then downloads and extracts the Spark distribution (version 3.1.2) with Hadoop 2.7. Finally, the findspark library is installed using pip. findspark is a Python library that allows the integration of Spark with Jupyter Notebooks or other Python environments. This setup enables the use of Apache Spark for distributed data processing in a Google Colab environment.

```
#Install Spark in Google Colab
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q https://archive.apache.org/dist/spark/spark-3.1.2/spark-
3.1.2-bin-hadoop2.7.tgz
!tar xf spark-3.1.2-bin-hadoop2.7.tgz
!pip install -q findspark
```

This sets environment variables for Java and Spark in the Google Colab environment. Specifically, it designates the Java Home path as "/usr/lib/jvm/java-8-openjdk-amd64" and the Spark Home path as "/content/spark-3.1.2-bin-hadoop2.7". These environment variables are crucial for ensuring that the Python environment can locate and communicate with the Java and Spark installations.

```
#Set Environment Variables
import os
```

```
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.1.2-bin-hadoop2.7"
```

the findspark library. findspark is a Python library that helps Python programs locate Spark's installation on a system, facilitating the integration of Spark with Python environments like Jupyter Notebooks. By running findspark.init(), it configures the necessary environment variables to enable the seamless use of Spark within the Python environment in Google Colab.

```
#Install FindSpark
import findspark
findspark.init()
```

This creates a SparkSession named spark by invoking SparkSession.builder.master("local[*]").getOrCreate(). The specified master URL, "local[*]", indicates that Spark should run in local mode using all available cores on the machine. This Spark session serves as the entry point for interacting with Spark functionalities, enabling distributed data processing.

```
#Create a Spark Session
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[*]").getOrCreate()
```

This downloads the text content of a meditation text from Project Gutenberg using the requests library. The specified URL points to the text file, and the content is retrieved using the get method. Subsequently, the obtained text content is saved to a local file named "sample.txt" in UTF-8 encoding.

```
import requests

# URL of the meditation text from Project Gutenberg
url = "https://www.gutenberg.org/cache/epub/2680/pg2680.txt"

# Download the text file
response = requests.get(url)
text_content = response.text

# Save the content to a local file named sample.txt
with open("sample.txt", "w", encoding="utf-8") as file:
    file.write(text_content)

# Print a message indicating the successful download and save
print("Meditation text downloaded from Project Gutenberg and saved as sample.txt.")
```

```
Meditation text downloaded from Project Gutenberg and saved as
sample.txt.
```

This code performs a basic Word Count analysis using Apache Spark in Google Colab. It reads the text file "sample.txt" into a Resilient Distributed Dataset (RDD) using Spark's textFile method. The Word Count operation is then carried out by first using flatMap to split each line into words, followed by map to associate each word with the count of 1. The reduceByKey operation is then employed to aggregate the counts for each word. The final results are collected and printed, displaying the count for each unique word in the provided text.

```
# Read the text file into an RDD
text_data = spark.sparkContext.textFile("sample.txt")

# Perform Word Count
word_count_data = text_data.flatMap(lambda line: line.split("
")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)

# Collect the Word Count Results
result = word_count_data.collect()

# Display the Word Count Results
for (word, count) in result:
    print(f"{word}: {count}")
```

This code identifies the most common word and its occurrence count from the Word Count results obtained earlier. The max function is used with a lambda function to find the element with the maximum count in the word_count_data RDD. The result is then printed, revealing the most common word in the provided text ("and") and its frequency (3149 occurrences)

```
# Find the Most Common Word
most_common_word = word_count_data.max(lambda x: x[1])

# Display the Most Common Word and its Occurrence Count
print(f"The most common word is '{most_common_word[0]}' with
{most_common_word[1]} occurrences.")
```

```
The most common word is 'and' with 3149 occurrences.
```

This code identifies the word with the fewest occurrences and its count from the Word Count results obtained earlier. The min function is used with a lambda function to find the element with the minimum count in the word_count_data RDD. The result is then printed, revealing the word with the fewest occurrences in the provided text ("Author:") and its frequency (1 occurrence).

```
# Find the Word with the Fewest Occurrences
fewest_occurrences_word = word_count_data.min(lambda x: x[1])

# Display the Word with the Fewest Occurrences and its Occurrence
Count
print(f"The word with the fewest occurrences is
'{fewest_occurrences_word[0]}' with {fewest_occurrences_word[1]}
occurrences.")
```

```
The word with the fewest occurrences is 'Author:' with 1 occurrences.
```

This code introduces a timing mechanism to measure the execution duration of the Word Count job using Apache Spark. The time module is employed to record the start time before executing the Word Count operation and the end time afterward. The elapsed time is then calculated by subtracting the start time from the end time. The Word Count results are displayed, followed by the elapsed time, providing insights into the computational efficiency of the Spark job. In this specific example, the Word Count job took approximately 0.04 seconds to complete.

```
import time
# Start the Timer
start_time = time.time()

# Perform Word Count
word_count_data = text_data.flatMap(lambda line: line.split("
")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)

# Stop the Timer
end_time = time.time()

# Calculate the Elapsed Time
elapsed_time = end_time - start_time

# Display the Word Count Results
#result = word_count_data.collect()
#for (word, count) in result:
#    print(f"{word}: {count}")

# Display the Elapsed Time
print(f"\nThe word count job took {elapsed_time:.2f} seconds to
complete.")
```

```

The word count job took 0.04 seconds to complete.
```

conclusion

In conclusion, our journey through this assignment provided a hands-on experience with Apache Spark, tailored to the unique context of processing the "Meditation" text. By adapting the Word Count application, we gained valuable insights into the frequency of words, identified the most and least common terms, and measured the time taken for the word count job, all within the Google Colab environment. This practical exercise not only solidified our understanding of Apache Spark's functionality but also demonstrated its applicability to diverse datasets. As we continue our exploration of data analytics, this assignment stands as a foundational step in leveraging powerful tools for efficient processing and analysis of substantial textual datasets.

reference

Srini Penchikala (2015). Big Data Processing with Apache Spark – Part 1: Introduction. [online] InfoQ. Available at: https://www.infoq.com/articles/apache-spark-introduction/ [Accessed 4 Dec. 2023].