

Assignment - 2

Title: Write a Hadoop/PySpark program that counts the number of occurrences of each word in a text file.

Theory

1. Introduction:

Big Data processing requires efficient distributed computing frameworks, and Apache Hadoop along with PySpark is widely used for such tasks. One common example of big data processing is word count, which determines the frequency of each word in a given text file. PySpark, the Python API for Apache Spark, enables distributed processing by leveraging Resilient Distributed Datasets (RDDs). This write-up explains the implementation of a word count program using PySpark.

2. Key Components:

The word count program in PySpark utilizes the following key components:

- **SparkContext:** The entry point for interacting with Spark's functionality, used to create RDDs.
- **RDD (Resilient Distributed Dataset):** A fault-tolerant and parallel collection of data that supports distributed processing.
- **Transformations:** Operations like `flatMap()`, `map()`, and `reduceByKey()` that transform data without immediate execution.
- **Actions:** Operations like `collect()` that trigger execution and return results.
- **Local Mode Execution:** Running Spark locally instead of a cluster, useful for development and testing.

3. Input and Output:

Input

- A text file (`input.txt`) containing sentences. Example content:
Hello world, Hello Hadoop, Hello PySpark, PySpark is great.

Output

- A collection of words along with their respective occurrence counts.
[('Hello', 3), ('world', 1), ('Hadoop', 1), ('PySpark', 2), ('is', 1), ('great', 1)]

4. Program Flow:

1. Initialize SparkContext:

- `SparkContext("local", "WordCount")` initializes a Spark session in local mode.
- This is required to enable Spark to create and manipulate RDDs.

2. Load the text file into an RDD:

- `textFile("input.txt")` reads the contents of the file and stores each line as an element in an RDD.
- This operation allows Spark to process data in a distributed fashion.

3. Split each line into words & assign an initial count of 1 to each word:

- `flatMap(lambda line: line.split(" "))` splits each line into a list of words.
- `map(lambda word: (word, 1))` maps each word to a tuple (word, 1), signifying its initial occurrence count.

4. Aggregate counts of the same words using reduceByKey():

- `reduceByKey(lambda x, y: x + y)` combines tuples with the same word by summing their counts.
- This step results in a final RDD where each word is paired with its total occurrence count.

5. Collect and print the final word counts:

- `collect()` retrieves the computed word counts from the RDD and brings them into local memory.
- `print(word_count_rdd.collect())` displays the results in a structured format.

6. Stop the SparkContext:

- `sc.stop()` stops the Spark session and releases resources.
- This step is necessary to properly shut down Spark after execution.

5. Explanation of Code:

1. Step 1: Import Required Libraries from pyspark
`import SparkContext`
 - The `SparkContext` class is imported from the `pyspark` library.
 - It acts as the main entry point for utilizing Spark's capabilities.
2. Step 2: Initialize SparkContext
`sc = SparkContext("local", "WordCount")`
 - specifies that Spark will run on a single machine, suitable for development and testing.
 - "WordCount" serves as the application name.
3. Step 3: Load the Text File into an RDD
`text_rdd = sc.textFile("input.txt")`
 - Reads the input text file and creates an RDD, where each element represents a line from the file.
4. Step 4: Split Lines into Words and Assign an Initial Count
`words_rdd = text_rdd.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1))`
 - Splits each line into individual words.
 - Unlike `map()`, `flatMap()` flattens the output into a single sequence.
 - `map(lambda word: (word, 1))`
 - Converts each word into a key-value pair (word, 1), where 1 represents the initial count.
5. Step 5: Aggregate the Word Counts
`word_count_rdd = words_rdd.reduceByKey(lambda x, y: x + y)`
 - Groups words based on their key (word) and sums up their associated counts.
 - For instance, if the word "Hello" appears three times, the counts (1,1,1) are summed to 3.

6. Step 6: Collect and Display the Result

`Print(word_count_rdd.collect())collect()`

- gathers the computed word counts from the distributed cluster and returns them to the driver program.
- The results are printed as a list of tuples representing word-frequency pairs.

7. Step 7: Stop `SparkContext`.`stop()`

- Terminates the Spark session to release system resources and free up memory.

6. Conclusion:

The word count program demonstrates how PySpark processes data in a distributed environment using RDDs. By leveraging transformations and actions, we efficiently count word occurrences in a given text file. This simple example highlights the power of Spark for handling large-scale data processing tasks.

```
from pyspark import SparkContext

sc = SparkContext("local", "WordCount")


# Load text file into an RDD
text_rdd = sc.textFile("input.txt")

# Map Phase: Split lines into words and assign count 1 to each word
words_rdd = text_rdd.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1))

# Reduce Phase: Sum counts of the same words
word_count_rdd = words_rdd.reduceByKey(lambda x, y: x + y)

# Collect and print results
print(word_count_rdd.collect())

sc.stop()
```



```
[('hello', 1), ('my', 1), ('name', 1), ('is', 1), ('harshal', 2), ('mali', 2), ('i', 1), ('am', 1), ('', 1), ('pimpri', 2),
```

