Name of Student: Pushkar Sane		
Roll Number: 45	**Lab Assignment Number: 6**	
Title of Lab Assignment: To run spark commands and functions		
DOP: 01-10-2024	**DOS: 03-10-2024**	
CO Mapped:	**PO Mapped:**	**Signature:**

Practical No. 6

Aim:

1. To run spark commands and functions.
2. Downloading Data Set and Processing it Spark
3. Word Count in Apache Spark.

Theory:

Downloading Dataset and Processing it in Apache Spark

In Apache Spark, the process of handling large datasets starts with downloading or loading data from various sources, including local files, Hadoop HDFS, cloud storage, or databases. After the environment is set up, Spark can be utilized to read and transform the data through its robust APIs.

Key Concepts:

- Dataset Sources: Data may be in formats such as CSV, JSON, Parquet, or plain text. Spark offers specific functions for reading these formats.
- Data Loading: Once the dataset is determined, it is loaded into Spark as a Resilient Distributed Dataset (RDD) or a DataFrame, which are the core data structures in Spark.
- Transformations and Actions: Spark employs a "lazy evaluation" strategy. Transformations like filtering, mapping, or reducing the data remain unexecuted until an action, like collecting results or writing output, is initiated.
- Parallel Processing: Spark distributes the dataset across a cluster, allocating tasks for quicker execution. This enables efficient processing of large-scale data.

When downloading a dataset, it can involve retrieving it from the internet or loading it from an existing distributed file system like HDFS. After the dataset is loaded, Spark can perform a variety of operations to process the data, such as filtering, aggregating, and analyzing.

Data Processing Steps:

1. Loading: Data from files is loaded into RDDs or DataFrames.
2. Transformation: Operations such as filtering, splitting, mapping, and aggregating are applied to manipulate the data.
3. Action: Initiating an action prompts Spark to execute the transformations and generate results, such as writing to a file or displaying the output.

Word Count in Apache Spark

The Word Count problem serves as a fundamental yet illustrative example in Apache Spark, frequently utilized to showcase its capabilities in distributed data processing. It involves counting the frequency of each word within a given dataset, typically a text file. This case effectively demonstrates Spark's parallel processing strengths.

Word Count Workflow:

Reading Input: The input dataset is usually a text file containing multiple lines of text. Spark reads this file and loads the data into an RDD or DataFrame, where each element corresponds to a line of text.

Splitting Lines into Words: The text is divided into individual words. Spark utilizes its transformation functions, such as flatMap(), to decompose each line into words, transforming each word into an element in the dataset.

Mapping Words to Key-Value Pairs: Following the splitting of the text into words, each word is mapped to a key-value pair, where the word serves as the key and 1 signifies the occurrence of that word. This step sets the stage for aggregation.

Reducing by Key (Word Count): After mapping the words to key-value pairs, Spark performs aggregation. It totals the occurrences of each word using the reduceByKey() transformation, where the values for each word are combined (e.g., summing the counts of 1 for each word).

Collecting Results: Once the word count operation is finalized, Spark initiates an action (like collect() or saveAsTextFile()) to compile the results. The output consists of a list of words along with their respective counts.

Key Concepts in Word Count:

- Parallelization: Each phase of the Word Count process (reading data, splitting text, mapping, and reducing) is distributed across nodes in a Spark cluster. This allows for the rapid processing of large datasets.
- Transformations: Operations such as flatMap(), map(), and reduceByKey() are classified as transformations in Spark. These are executed in a lazy manner and only run when an action is triggered.
- Action: Actions in Spark, like collecting results, initiate the computation of transformations. Without an action, no processing takes place.

Applications of Word Count:

- Text Analytics: Determining word frequencies in documents, which is beneficial for tasks like keyword extraction or search engine indexing.
- Log File Analysis: Counting occurrences of specific log messages in system logs for debugging or monitoring purposes.
- Data Mining: Word count is a simple yet critical operation in broader data mining tasks, especially in the processing of large-scale text data.

Set Up Spark and HDFS in Cloudera VM

1. Open the Terminal in Cloudera VM:
    - Log into the Cloudera VM and open the terminal.

2. Verify HDFS is Running:
    - Ensure that HDFS is running:

      hdfs dfsadmin -report

```
[cloudera@quickstart ~]$ hdfs dfsadmin -report
Configured Capacity: 58531520512 (54.51 GB)
Present Capacity: 46694797257 (43.49 GB)
DFS Remaining: 45820696076 (42.67 GB)
DFS Used: 874101181 (833.61 MB)
DFS Used%: 1.87%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0

-------------------------------------------------
Live datanodes (1):

Name: 10.0.2.15:50010 (quickstart.cloudera)
Hostname: quickstart.cloudera
Decommission Status : Normal
Configured Capacity: 58531520512 (54.51 GB)
DFS Used: 874101181 (833.61 MB)
Non DFS Used: 8588236355 (8.00 GB)
DFS Remaining: 45820696076 (42.67 GB)
DFS Used%: 1.49%
DFS Remaining%: 78.28%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 6
Last contact: Tue Oct 01 16:19:21 PDT 2024


[cloudera@quickstart ~]$
```

      This should show a report of the HDFS status.

Download and Upload Dataset to HDFS

1. Download the Dataset:
    - Open a terminal and use wget to download the dataset, or download it from any public source.

      Example:

      wget

      https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data

```
[cloudera@quickstart ~]$ wget https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data
--2024-10-01 16:32:09--  https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data
Resolving archive.ics.uci.edu... 128.195.10.252
Connecting to archive.ics.uci.edu|128.195.10.252|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified
Saving to: “processed.cleveland.data”

    [  <=>

2024-10-01 16:32:19 (70.6 KB/s) - “processed.cleveland.data” saved [18461]

[cloudera@quickstart ~]$
```

2. Upload Dataset to HDFS:
    - Create a directory in HDFS to store your dataset:

      hdfs dfs -mkdir /user/cloudera/datasets

```
[cloudera@quickstart ~]$ hdfs dfs -mkdir /user/cloudera/datasets
[cloudera@quickstart ~]$
```

    - Upload the dataset to HDFS:

      hdfs dfs -put processed.cleveland.data /user/cloudera/datasets/

```
[cloudera@quickstart ~]$ hdfs dfs -put processed.cleveland.data /user/cloudera/datasets/
[cloudera@quickstart ~]$
```

    - Verify that the dataset is uploaded successfully:

      hdfs dfs -ls /user/cloudera/datasets/

```
[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera/datasets/
Found 1 items
-rw-r--r--   1 cloudera cloudera      18461 2024-10-01 16:42 /user/cloudera/datasets/processed.cleveland.data
[cloudera@quickstart ~]$
```

Process the Dataset Using Spark

1. Open Spark Shell in Cloudera VM:
   - For Python (PySpark):

     Pyspark

```
[cloudera@quickstart ~]$ pyspark
Python 2.6.6 (r266:84292, Jul 23 2015, 15:22:56)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar
!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!
/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/parquet/lib/slf4j-log4j12-1.7.5.jar!/
org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/avro/avro-tools-1.7.6-cdh5.13.0.jar!/
org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
24/10/01 16:51:37 WARN util.NativeCodeLoader: Unable to load native-hadoop libra
ry for your platform... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Python version 2.6.6 (r266:84292, Jul 23 2015 15:22:56)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

2. Read the Dataset from HDFS:

   For PySpark:

   # In the PySpark shell, run the following:

   from pyspark import SparkContext

   from pyspark.sql import SQLContext

   # Create a SparkContext

   sc = SparkContext.getOrCreate()

   # Create a SQLContext

   sqlContext = SQLContext(sc)

   # Read the dataset as a text file

   lines = sc.textFile("/user/cloudera/datasets/processed.cleveland.data")

```
# Split each line into a list of values
rows = lines.map(lambda line: line.split(","))
# Convert to DataFrame using the SQLContext
df = sqlContext.createDataFrame(rows)
# Show the first 5 rows
df.show(5)
```

```
>>> from pyspark.sql import SQLContext
>>> from pyspark import SparkContext
>>> sc = SparkContext.getOrCreate()  # Create SparkContext
>>> sqlContext = SQLContext(sc)  # Create SQLContext
>>> # Read the dataset as a text file
... lines = sc.textFile("/user/cloudera/datasets/processed.cleveland.data")
>>>
>>> # Split each line into a list of values
... rows = lines.map(lambda line: line.split(","))
>>>
>>> # Convert to DataFrame using the SQLContext
... df = sqlContext.createDataFrame(rows)
24/10/01 16:55:19 WARN shortcircuit.DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.
>>> df.show(5)
+----+---+---+-----+-----+---+---+-----+---+---+---+---+---+---+
|  _1| _2| _3|   _4|   _5| _6| _7|   _8| _9|_10|_11|_12|_13|_14|
+----+---+---+-----+-----+---+---+-----+---+---+---+---+---+---+
|63.0|1.0|1.0|145.0|233.0|1.0|2.0|150.0|0.0|2.3|3.0|0.0|6.0|  0|
|67.0|1.0|4.0|160.0|286.0|0.0|2.0|108.0|1.0|1.5|2.0|3.0|3.0|  2|
|67.0|1.0|4.0|120.0|229.0|0.0|2.0|129.0|1.0|2.6|2.0|2.0|7.0|  1|
|37.0|1.0|3.0|130.0|250.0|0.0|0.0|187.0|0.0|3.5|3.0|0.0|3.0|  0|
|41.0|0.0|2.0|130.0|204.0|0.0|2.0|172.0|0.0|1.4|1.0|0.0|3.0|  0|
+----+---+---+-----+-----+---+---+-----+---+---+---+---+---+---+
only showing top 5 rows

>>>
```

3. Perform Processing on the Dataset:

   Example: Filtering Data where the first column (_c0) is greater than 2:

   PySpark:

```
# Check the schema
df.printSchema()
# Filter based on the actual column name
filtered_df = df.filter(df['_1'] > 2)  # Adjust the column index based on your schema
filtered_df.show()
```

```
>>> # Check the schema
... df.printSchema()
root
 |-- _1: string (nullable = true)
 |-- _2: string (nullable = true)
 |-- _3: string (nullable = true)
 |-- _4: string (nullable = true)
 |-- _5: string (nullable = true)
 |-- _6: string (nullable = true)
 |-- _7: string (nullable = true)
 |-- _8: string (nullable = true)
 |-- _9: string (nullable = true)
 |-- _10: string (nullable = true)
 |-- _11: string (nullable = true)
 |-- _12: string (nullable = true)
 |-- _13: string (nullable = true)
 |-- _14: string (nullable = true)

>>>
>>> # Filter based on the actual column name
... filtered_df = df.filter(df['_1'] > 2)  # Adjust the column index based on your schema
>>> filtered_df.show()
+----+---+---+-----+-----+---+---+-----+---+---+---+---+---+---+
|  _1| _2| _3|   _4|   _5| _6| _7|   _8| _9|_10|_11|_12|_13|_14|
+----+---+---+-----+-----+---+---+-----+---+---+---+---+---+---+
|63.0|1.0|1.0|145.0|233.0|1.0|2.0|150.0|0.0|2.3|3.0|0.0|6.0|  0|
|67.0|1.0|4.0|160.0|286.0|0.0|2.0|108.0|1.0|1.5|2.0|3.0|3.0|  2|
|67.0|1.0|4.0|120.0|229.0|0.0|2.0|129.0|1.0|2.6|2.0|2.0|7.0|  1|
|37.0|1.0|3.0|130.0|250.0|0.0|0.0|187.0|0.0|3.5|3.0|0.0|3.0|  0|
|41.0|0.0|2.0|130.0|204.0|0.0|2.0|172.0|0.0|1.4|1.0|0.0|3.0|  0|
|56.0|1.0|2.0|120.0|236.0|0.0|0.0|178.0|0.0|0.8|1.0|0.0|3.0|  0|
|62.0|0.0|4.0|140.0|268.0|0.0|2.0|160.0|0.0|3.6|3.0|2.0|3.0|  3|
|57.0|0.0|4.0|120.0|354.0|0.0|0.0|163.0|1.0|0.6|1.0|0.0|3.0|  0|
|63.0|1.0|4.0|130.0|254.0|0.0|2.0|147.0|0.0|1.4|2.0|1.0|7.0|  2|
|53.0|1.0|4.0|140.0|203.0|1.0|2.0|155.0|1.0|3.1|3.0|0.0|7.0|  1|
|57.0|1.0|4.0|140.0|192.0|0.0|0.0|148.0|0.0|0.4|2.0|0.0|6.0|  0|
|56.0|0.0|2.0|140.0|294.0|0.0|2.0|153.0|0.0|1.3|2.0|0.0|3.0|  0|
|56.0|1.0|3.0|130.0|256.0|1.0|2.0|142.0|1.0|0.6|2.0|1.0|6.0|  2|
|44.0|1.0|2.0|120.0|263.0|0.0|0.0|173.0|0.0|0.0|1.0|0.0|7.0|  0|
|52.0|1.0|3.0|172.0|199.0|1.0|0.0|162.0|0.0|0.5|1.0|0.0|7.0|  0|
|57.0|1.0|3.0|150.0|168.0|0.0|0.0|174.0|0.0|1.6|1.0|0.0|3.0|  0|
|48.0|1.0|2.0|110.0|229.0|0.0|0.0|168.0|0.0|1.0|3.0|0.0|7.0|  1|
|54.0|1.0|4.0|140.0|239.0|0.0|0.0|160.0|0.0|1.2|1.0|0.0|3.0|  0|
|48.0|0.0|3.0|130.0|275.0|0.0|0.0|139.0|0.0|0.2|1.0|0.0|3.0|  0|
|49.0|1.0|2.0|130.0|266.0|0.0|0.0|171.0|0.0|0.6|1.0|0.0|3.0|  0|
+----+---+---+-----+-----+---+---+-----+---+---+---+---+---+---+
only showing top 20 rows

>>>
```

4. Save Processed Data Back to HDFS:

   After performing some operations, you can write the output back to HDFS

   PySpark:

   # Combine all columns into a single string separated by commas

   filtered_df_rdd = filtered_df.rdd.map(lambda row: ",".join(row))

   # Save the RDD as a text file, which will be in CSV format

   filtered_df_rdd.saveAsTextFile("/user/cloudera/output/filtered_wine_data.csv")

```
>>> # Combine all columns into a single string separated by commas
... filtered_df_rdd = filtered_df.rdd.map(lambda row: ",".join(row))
>>>
>>> # Save the RDD as a text file, which will be in CSV format
... filtered_df_rdd.saveAsTextFile("/user/cloudera/output/filtered_wine_data.csv
")
>>>
```

Perform Word Count Example in Apache Spark

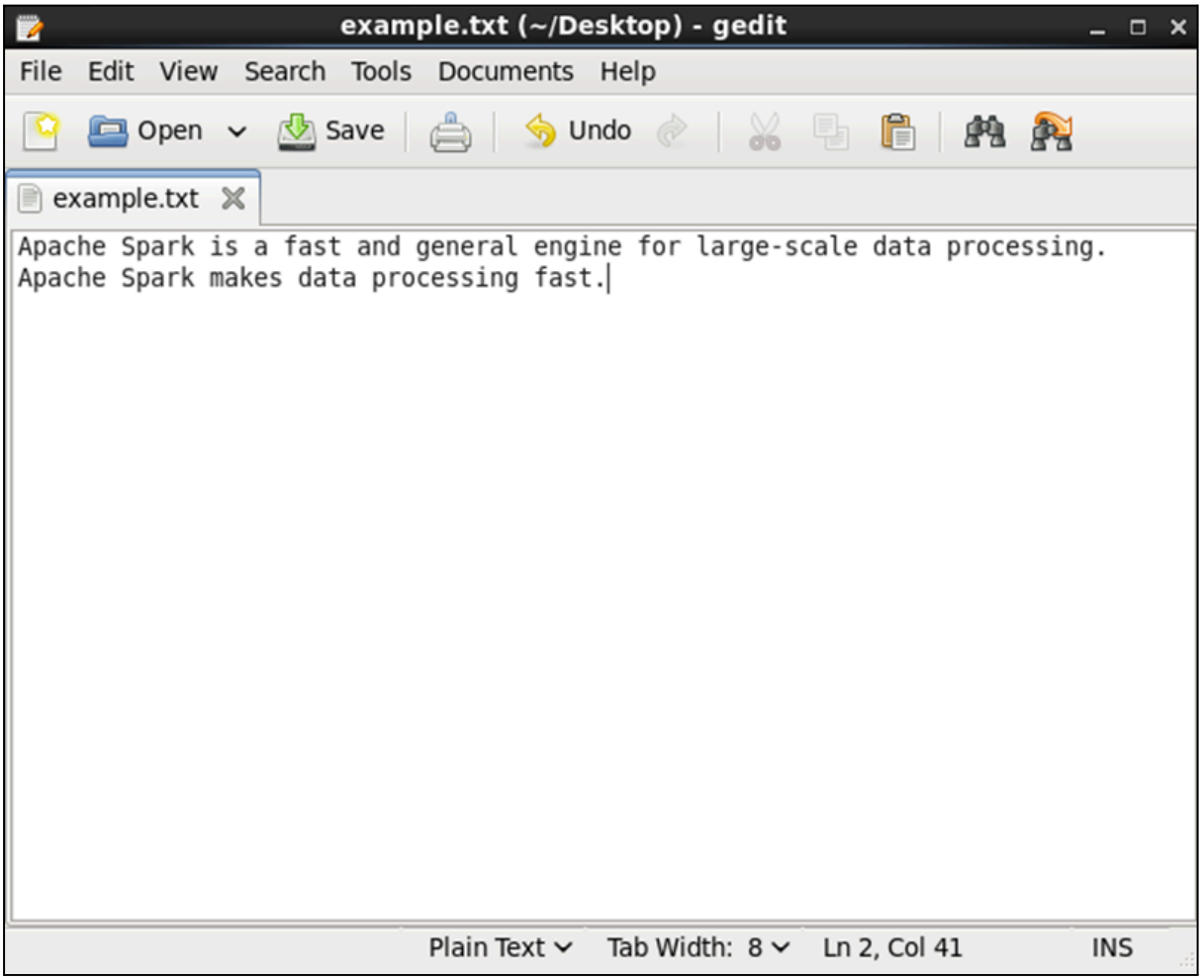
1. Create or Download a Text File for Word Count:

   Example content for a simple text file:

   Apache Spark is a fast and general engine for large-scale data processing.

   Apache Spark makes data processing fast.

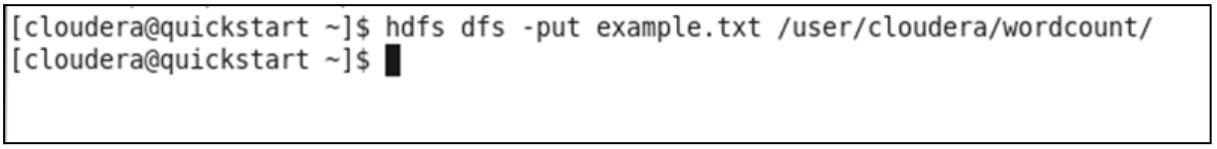
   Save this as example.txt.



2. Upload the Text File to HDFS:

   hdfs dfs -mkdir /user/cloudera/wordcount

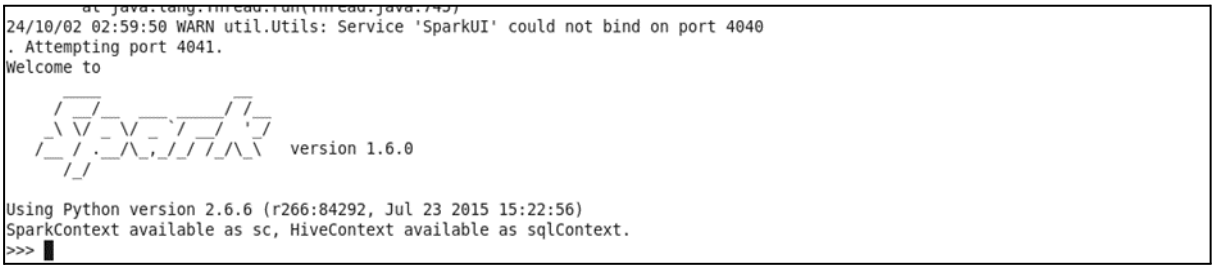
   hdfs dfs -put example.txt /user/cloudera/wordcount/



3. Perform Word Count Using PySpark:

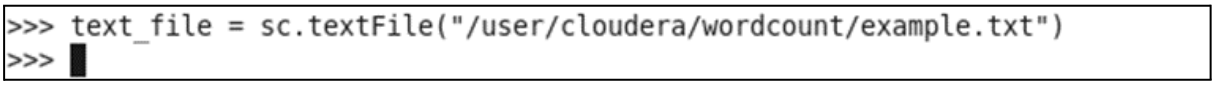
   a. Launch PySpark:

      pyspark



   b. Read the Text File:

      text_file = sc.textFile("/user/cloudera/wordcount/example.txt")



   c. Perform Word Count:

```
# Perform the word count operation
word_counts = text_file.flatMap(lambda line: line.split(" ")) \
                       .map(lambda word: (word, 1)) \
                       .reduceByKey(lambda a, b: a + b)

# Collect and print the results
for word, count in word_counts.collect():
    print("{0}: {1}".format(word, count))
```

```
>>> # Perform the word count operation
... word_counts = text_file.flatMap(lambda line: line.split(" ")) \
...                        .map(lambda word: (word, 1)) \
...                        .reduceByKey(lambda a, b: a + b)
>>>
>>> # Collect and print the results
... for word, count in word_counts.collect():
...     print("{0}: {1}".format(word, count))
...
a: 1
and: 1
processing.: 1
for: 1
large-scale: 1
data: 2
is: 1
processing: 1
fast: 1
general: 1
engine: 1
fast.: 1
Apache: 2
Spark: 2
makes: 1
>>>
```

Conclusion:

Apache Spark enables efficient distributed data processing by applying parallelized transformations and actions, as demonstrated by the foundational Word Count example.