Scala

Web Log Processing

```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.sql.{Column, SparkSession}
import org.apache.spark.sql.functions._
```

This imports Level and Logger from org.apache.log4j, Column and SparkSession from org.apache.spark.sql, and all functions from org.apache.spark.sql.functions. This way, you can directly use functions like regexp_extract, sum, col, to_date, udf, to_timestamp, desc, dayofyear, and year without specifying the package each time.

```
// Create a SparkSession with the given application name "WebLog" and set the master to run locally using all available cores.
val spark = SparkSession.builder()
  .appName("WebLog")       // Setting the application name
  .master("local[*]")      // Running Spark in local mode using all available cores
  .getOrCreate()           // Creating the SparkSession if it doesn't exist, or getting the existing one

// Read the content of a CSV file located at "weblog.csv" into a DataFrame
val base_df = spark.read.text("weblog.csv")

// Print the schema of the DataFrame
base_df.printSchema()

import spark.implicits._
base_df.show(3, false)
```

The provided code performs the following tasks:

1. **Import Spark implicits**: `import spark.implicits._` imports the implicit methods available in Spark. These implicits allow you to implicitly convert standard Scala objects into DataFrame operations.
2. **Read a CSV file into a DataFrame**: `val base_df = spark.read.text("/home/deptii/Web_Log/weblog.csv")` reads the content of a CSV file located at "weblog.csv" into a DataFrame named `base_df`. Since `.read.text()` is used, each line of the text file will be treated as a separate record in the DataFrame, and there will be a single column named "value" containing the content of each line.
3. **Print DataFrame schema**: `base_df.printSchema()` prints the schema of the DataFrame `base_df`. This shows the data types of each column in the DataFrame.

4. **Display DataFrame content**: `base_df.show(3, false)` displays the first 3 rows of the DataFrame `base_df`. The `false` parameter indicates that the truncation of the displayed data should be disabled. So, it displays the entire content of each row.

```
val parsed_df = base_df.select(
   regexp_extract($"value", """^(\S+)""", 1).alias("host"),
   regexp_extract($"value", """\[(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2})""", 1).as("timestamp"),
   regexp_extract($"value", """\s(\S+)\s+HTTP""", 1).as("path"),
   regexp_extract($"value", """,(\d+)$""", 1).cast("int").alias("status")
)
```

Certainly, let's break down each regular expression used in the code:

1. **Host Regex**:
   - `^(\S+)`
     - `^` asserts the start of the line.
     - `(\S+)` captures one or more non-whitespace characters.
   - This regex captures the host part of the log entry, which is typically the IP address or hostname.
2. **Timestamp Regex**:
   - `\[(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2})`
     - `\[` matches the opening square bracket `[`.
     - `(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2})` captures the timestamp in the format `dd/MMM/yyyy:HH:mm:ss`.
       - `\d{2}` matches two digits for the day.
       - `\w{3}` matches three word characters for the month abbreviation (e.g., Jan, Feb).
       - `\d{4}` matches four digits for the year.
       - `:\d{2}` matches two digits for the hour.
       - `:\d{2}` matches two digits for the minutes.
       - `:\d{2}` matches two digits for the seconds.
   - This regex captures the timestamp part of the log entry enclosed within square brackets.

3. **Path Regex**:
    - `\s(\S+)\s+HTTP`
        - `\s` matches a whitespace character.
        - `(\S+)` captures one or more non-whitespace characters, which represent the path.
        - `\s+HTTP` matches one or more whitespace characters followed by "HTTP".
    - This regex captures the path part of the log entry, which represents the URL or endpoint accessed.
4. **Status Regex**:
    - `,\d+$`
        - `,` matches the comma character.
        - `\d+` matches one or more digits.
        - `$` asserts the end of the line.
    - This regex captures the status code at the end of the log entry, typically separated by a comma. It then converts it to an integer.

```
// Count the number of rows with null values in the column "value" in the initial dataset
println("Number of bad row in the initial dataset: " + base_df.filter($"value".isNull).count())

// Filter out rows with null values in columns "host", "timestamp", "path", or "status"
val bad_rows_df = parsed_df.filter($"host".isNull || $"timestamp".isNull || $"path".isNull || $"status".isNull)

// Print the count of rows with null values in specified columns
println("Number of bad rows: " + bad_rows_df.count())

// Remove rows with any null values in any column
val cleaned_df = parsed_df.na.drop()

// Print the count of null values after cleaning
println("The count of null value: " + cleaned_df.filter($"host".isNull || $"timestamp".isNull || $"path".isNull || $"status".isNull).count())

// Print the count of rows before and after cleaning
println("Before: " + parsed_df.count() + " | After: " + cleaned_df.count())

// Convert the "timestamp" column to a timestamp format and drop the original column, caching the result
val logs_df = cleaned_df.withColumn("time", to_timestamp($"timestamp", "dd/MMM/yyyy:HH:mm:ss")).drop("timestamp").cache()

// Print the schema of the DataFrame
logs_df.printSchema()
```

```
// Show the first two rows of the DataFrame
logs_df.show(2)

// Show descriptive statistics for the "status" column
logs_df.describe("status").show()
```

The overall purpose of this code is to preprocess and analyze a DataFrame containing log data. Here's a breakdown of each step:

1. **Filter out rows with null values**: The code filters out rows from the DataFrame `parsed_df` where any of the specified columns ("host", "timestamp", "path", or "status") have null values. This identifies rows with missing or incomplete data.
2. **Print the count of bad rows**: After filtering, it prints the count of rows that were identified as having null values in any of the specified columns. This provides insight into the extent of missing data in the dataset.
3. **Remove rows with any null values**: It removes rows with any null values in any column from the DataFrame `parsed_df`. This step is aimed at cleaning the data by eliminating incomplete records.
4. **Print the count of null values after cleaning**: It prints the count of null values in the specified columns after cleaning. This helps verify the effectiveness of the cleaning process.
5. **Print the count of rows before and after cleaning**: It prints the counts of rows in the DataFrame `parsed_df` before and after cleaning. This provides information on the impact of the cleaning process on the dataset size.
6. **Convert timestamp column and cache the result**: It converts the "timestamp" column of the cleaned DataFrame to a timestamp format using the specified pattern ("dd/MMM/yyyy:HH:mm:ss"). The original "timestamp" column is dropped, and the resulting DataFrame `logs_df` is cached for faster access in subsequent operations.
7. **Print the schema of the DataFrame**: It prints the schema of the DataFrame `logs_df`, showing the data types of each column.
8. **Show the first two rows of the DataFrame**: It displays the first two rows of the DataFrame `logs_df`, allowing a quick inspection of the data.
9. **Show descriptive statistics for the "status" column**: It displays descriptive statistics (e.g., count, mean, standard deviation, min, max) for the "status" column in the DataFrame `logs_df`. This provides insights into the distribution and characteristics of the status codes in the log data.

1. Problem Statement No. 05
Write a Scala Program to process a log file of a system and perform following analytics on the given dataset.
(I) Display the list of top 10 frequent hosts.
(II) Display the list of top 5 URLs or paths
(III) Display the number of unique Hosts

```
import org.apache.spark.sql.functions._

// (I) Display the list of top 10 frequent hosts
val top10Hosts = logs_df.groupBy($"host").count().orderBy(desc("count")).limit(10)
println("Top 10 frequent hosts:")
top10Hosts.show()

// (II) Display the list of top 5 URLs or paths
val top5Paths = logs_df.groupBy($"path").count().orderBy(desc("count")).limit(5)
println("Top 5 URLs or paths:")
top5Paths.show()

// (III) Display the number of unique Hosts
val uniqueHostsCount = logs_df.select(countDistinct($"host")).collect()(0)(0)
println(s"Number of unique hosts: $uniqueHostsCount")
```

2. Problem Statement No. 06
Write a Scala Program to process a log file of a system and perform following analytics on the given dataset.
(I) Display the count of 404 Response Codes
(II) Display the list of Top Twenty-five 404 Response Code Hosts
(III) Display the number of Unique Daily Hosts

```
import org.apache.spark.sql.functions._

// (I) Display the count of 404 Response Codes
val count404 = logs_df.filter($"status" === 404).count()
println(s"Count of 404 Response Codes: $count404")

// (II) Display the list of Top Twenty-five 404 Response Code Hosts
val top25Hosts404 = logs_df.filter($"status" ===
404).groupBy($"host").count().orderBy(desc("count")).limit(25)
println("Top Twenty-five 404 Response Code Hosts:")
top25Hosts404.show()

// (III) Display the number of Unique Daily Hosts
val uniqueDailyHosts = logs_df.groupBy(to_date($"time").alias("date"),
$"host").agg(countDistinct($"host")).groupBy("date").count()
println("Number of Unique Daily Hosts:")
uniqueDailyHosts.show()
```

3. Problem Statement No. 17

1. Write a Scala program that counts the number of occurrences of each word in the given input file using Spark framework.

2. Write a Scala Program to find out if the number is Positive, Negative or Zero.

```
// Open the file "wcount.txt" in the Vim text editor
vim wcount.txt

// Launch the Spark shell to interactively run Spark applications
spark-shell

// Read the contents of the file "wcount.txt" into an RDD named inputfile
val inputfile = sc.textFile("wcount.txt")

// Perform word count on the inputfile RDD:
// 1. Split each line into words
// 2. Map each word to a tuple (word, 1) for counting
// 3. Reduce the tuples by key (word) to get the count of each word
val counts = inputfile.flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey(_+_)

// Print debug information about the RDD counts, showing its lineage and dependencies
counts.toDebugString

// Cache the RDD counts into memory for faster subsequent operations
counts.cache

// Save the RDD counts as text files in a directory named "Output"
counts.saveAsTextFile("Output")
```

```
object NumberType {
  def main(args: Array[String]): Unit = {
    // Input number
    println("Enter a number:")
    val number = scala.io.StdIn.readDouble()

    // Determine number type
    val result = if (number > 0) "Positive" else if (number < 0) "Negative" else "Zero"

    // Print the result
    println(s"The number $number is $result.")
  }
}
```

4. Problem Statement No. 18

1. Write a Scala program that counts the number of occurrences of each word in the given input file using Spark framework.

2. Write a Scala Program to find out the largest of two numbers.

```
object LargestNumber {
  def main(args: Array[String]): Unit = {
    // Input two numbers
    println("Enter the first number:")
    val num1 = scala.io.StdIn.readDouble()

    println("Enter the second number:")
    val num2 = scala.io.StdIn.readDouble()

    // Determine the largest number
    val largest = if (num1 > num2) num1 else num2

    // Print the result
    println(s"The largest number is: $largest")
  }
}
```

Hadoop

7. Problem Statement No. 13,14,19

Write a code in JAVA for a simple Word Count application that counts the number of occurrences of each word in a given input set using the Hadoop Map-Reduce framework on local-standalone set-up.

```
# Switch to the user 'hadoop' with the environment variables set to that user
su – hadoop

# Change directory to a directory named 'hadoop'
cd hadoop

# Open the Vim text editor to create/edit a file named 'input.txt'
vim input.txt

# Start all Hadoop daemons by running the 'start-all.sh' script
start-all.sh

# Create a directory named 'wordcount' in the Hadoop Distributed File System (HDFS)
hdfs dfs -mkdir /wordcount

# Copy a file named 'input.txt' from the local file system to the 'wordcount' directory in HDFS
hdfs dfs -put /home/hadoop/hadoop/input.txt /wordcount
```

```
nano Mapper1.java
nano Reducer1.java
nano WC_Runner.java

# Compile the Java files (Mapper1.java, Reducer1.java, WC_Runner.java)
# and specify the classpath using the output of the `hadoop classpath` command
# The `-d .` flag specifies the output directory as the current directory
javac -classpath "$(hadoop classpath)" -d . Mapper1.java Reducer1.java WC_Runner.java

# Create a JAR file named "Wordcount.jar" containing the compiled Java classes
# The `jar` command is used to create, modify, or extract files from JAR archives
# The `-cvf` options specify the action (create), verbose output, and file name
# The `com` directory contains the compiled Java classes
jar -cvf Wordcount.jar com
```

```
# Run a Hadoop job using the JAR file "/home/hadoop/hadoop/Wordcount.jar"
# The main class of the job is "com.wc.WC_Runner"
# The input file is "/test.wc/input.txt" and the output directory is "/output"
hadoop jar /home/hadoop/hadoop/Wordcount.jar com.wc.WC_Runner /test.wc/input.txt /output

# After the Hadoop job completes, display the contents of the output file
# The `hdfs dfs -cat` command is used to display the contents of files in HDFS
# In this case, we are displaying the contents of the file "part-00000" in the "/output" directory
hadoop$ hdfs dfs -cat /output/part-00000
```