PROGRAM-1

Convert Column

from pyspark.sql import SparkSession: This line imports the SparkSession class from the pyspark.sql module, which is responsible for creating and managing the Spark application's entry point.

spark = SparkSession.builder.master("local[1]") \: This line starts building a SparkSession by using the SparkSession.builder method. The master argument specifies the Spark cluster's URL, and in this case, it is set to "local[1]". This means that Spark will run in local mode with one worker thread.

.appName('SparkByExamples.com') \: This line sets the name of the Spark application to 'SparkByExamples.com' using the appName method. The name is used for identification purposes in the Spark cluster's UI and logs.

.getOrCreate(): This line completes the building of the SparkSession and retrieves an existing SparkSession or creates a new one if it doesn't already exist. The getOrCreate method ensures that only one SparkSession is active in the application.

data = [("James","Smith","USA","CA"),("Michael","Rose","USA","NY"), \: This line defines a list of tuples called data, where each tuple represents a row of data. Each tuple contains values for the columns "firstname", "lastname", "country", and "state".

columns=["firstname","lastname","country","state"]: This line defines a list called columns that specifies the names of the columns in the DataFrame.

df=spark.createDataFrame(data=data,schema=columns): This line uses the spark.createDataFrame function to create a DataFrame called df. The data argument specifies the input data (list of tuples), and the schema argument specifies the column names. The function creates the DataFrame with the specified columns and infers the data types of each column based on the input data.

df.show(): This line displays the contents of the DataFrame df using the show method. It prints the first 20 rows of the DataFrame in a tabular format.

print(df.collect()): This line retrieves all the rows from the DataFrame df using the collect method and prints them. The collect method returns the rows as a list of Row objects.

states1=df.rdd.map(lambda x: x[3]).collect(): This line converts the DataFrame df into an RDD using the rdd property. The map transformation is applied to extract the values from the fourth column (index 3) of each row using a lambda function. The collect action is then used to retrieve the values of the RDD and store them in the states1 list.

print(states1): This line prints the contents of the states1 list, which contains the values extracted from the RDD. The list represents the values of the "state" column from the DataFrame df.

from collections import OrderedDict: This line imports the OrderedDict class from the collections module. OrderedDict is a dictionary subclass that remembers the order of key-value pairs.

res = list(OrderedDict.fromkeys(states1)): This line creates a new list res by removing duplicate values from the states1 list while preserving the original order. It uses the fromkeys method of OrderedDict to create a dictionary with the values of states1 as keys. Then, the list function is used to convert the keys of the dictionary back into a list.

print(res): This line prints the contents of the res list, which contains the unique values of the "state" column from the DataFrame df, maintaining their original order.

Overall, this code demonstrates how to extract values from a specific column of a data frame using RDD operations in Spark. It then removes duplicate values from the extracted list while preserving the original order using the OrderedDict class.

states3=df.select(df.state).collect(): This line selects the "state" column from the DataFrame df using the select method. The select method allows you to choose specific columns from a DataFrame. The result is a new DataFrame that only contains the "state" column. The collect action is then applied to retrieve the rows of the DataFrame and store them in the states3 list.

print(states3): This line prints the contents of the states3 list, which contains the rows extracted from the "state" column. Each row is represented as a Row object, and in this case, each Row object has a single field named "state" with the corresponding value.

states4=df.select(df.state).rdd.flatMap(lambda x: x).collect(): This line performs the following operations:

df.select(df.state) selects the "state" column from the DataFrame df using the select method, resulting in a new DataFrame with a single column.
.rdd converts the DataFrame to an RDD (Resilient Distributed Dataset), which is a fundamental data structure in Spark that allows distributed processing.
.flatMap(lambda x: x) applies a transformation to each element of the RDD, in this case, flattening each row to a single value. The lambda function lambda x: x is used to return each element as is.
.collect() triggers the execution of the RDD operations and collects the resulting values into a list.

states5=df.select(df.state).toPandas()['state']: This line performs the following operations:

df.select(df.state) selects the "state" column from the DataFrame df using the select method, resulting in a new DataFrame with a single column.
.toPandas() converts the DataFrame to a Pandas DataFrame, which is a popular data manipulation library in Python.
['state'] selects the "state" column from the Pandas DataFrame, resulting in a Pandas Series.
states5 is assigned the extracted "state" column as a Pandas Series.

states6=list(states5): This line converts the Pandas Series states5 to a Python list.

print(states6): This line prints the contents of the states6 list.

PROGRAM-2

Current Date

StructType is a class that represents a schema for a DataFrame or a structured data record. It is a collection of StructField objects.

StructField is a class that represents a field in a StructType schema. It defines the name, data type, and nullable properties of a column or field.

StringType is a data type representing a string column.

In this specific code snippet, a schema is defined with a single column named "seq" of type StringType and nullable set to True. The schema is used to define the structure of the data in the DataFrame.

from pyspark.sql.types import StringType imports the StringType class from the pyspark.sql.types module. This class represents the data type for a string column.

dates = ['1'] defines a list called dates containing a single string value '1'.

schema = StringType() creates an instance of the StringType class and assigns it to the variable schema. This specifies that the column will have a string data type.

df = spark.createDataFrame(dates, schema=schema) creates a DataFrame called df by passing the dates list and the schema to the createDataFrame() method of the spark object. This method creates a DataFrame with a single column named "_1" and assigns the specified data type to it.

df.show() displays the contents of the DataFrame.

In summary, the code creates a DataFrame with a single column containing the string value '1'. The DataFrame is then displayed using the show() method.

PROGRAM-3

Pandas Pyspark examples

pd.DataFrame is a function from the Pandas library that creates a new DataFrame object.

data is the input data for the DataFrame. It can be in various formats such as a NumPy array, a dictionary, or another DataFrame. The specific format depends on the data you want to populate in the DataFrame.

columns=['Name', 'Age'] is an optional parameter that specifies the column names for the DataFrame. In this case, the column names are set as 'Name' and 'Age'.

pandasDF is the variable that holds the created DataFrame.

from pyspark.sql import SparkSession imports the SparkSession module from the pyspark.sql package. SparkSession is the entry point to interact with Spark SQL functionalities.

spark = SparkSession.builder creates a builder object for configuring the SparkSession.

.master("local[1]") sets the master URL to run Spark locally with one worker thread.

The "**local[1]**" means one worker thread will be used.

.appName("SparkByExamples.com") sets the name of the Spark application to "SparkByExamples.com".

.getOrCreate() gets an existing SparkSession or creates a new one if it doesn't exist.

spark.createDataFrame(pandasDF) creates a Spark DataFrame (sparkDF) from the Pandas DataFrame (pandasDF). This allows you to leverage Spark's distributed processing capabilities on the data.

sparkDF.printSchema() prints the schema of the Spark DataFrame. The schema provides information about the names and data types of the columns in the DataFrame.

sparkDF.show() displays the content of the Spark DataFrame. It prints the first 20 rows in a tabular format, allowing you to inspect the data.

StructType is a class that represents the schema of a DataFrame. It is used to define the structure of the DataFrame's columns.

StructField is a class that represents a single field (column) in the DataFrame schema. It defines the name, data type, and whether the field can be nullable.

StringType() and IntegerType() are data types from the pyspark.sql.types module. In this case, "First Name" is defined as a string column (StringType()), and "Age" is defined as an integer column (IntegerType()).

The **True** argument passed to the nullable parameter of StructField indicates that the columns can have null values.

spark.createDataFrame(pandasDF, schema=mySchema) creates a Spark DataFrame (sparkDF2) from a Pandas DataFrame (pandasDF) and applies the custom schema (mySchema) to the DataFrame. The schema parameter is set to mySchema, which specifies the structure and data types of the columns in the DataFrame.

sparkDF2.printSchema() prints the schema of the Spark DataFrame (sparkDF2), displaying the column names and their corresponding data types.

sparkDF2.show() displays the content of the Spark DataFrame (sparkDF2) in a tabular format, similar to how pandasDF.head() works in Pandas.

spark.conf.set("spark.sql.execution.arrow.enabled", "true") enables the use of Apache Arrow for accelerating data transfer between Spark and Pandas. Apache Arrow is an in-memory columnar data format that provides efficient data interchange between different data processing systems.

spark.conf.set("spark.sql.execution.arrow.pyspark.fallback.enabled", "true") enables the fallback mechanism in PySpark for executing certain operations using Arrow. If a particular operation is not supported by Arrow, PySpark will fallback to the traditional execution mechanism without Arrow.

sparkDF2.select("*") selects all columns from the Spark DataFrame sparkDF2.

toPandas() converts the selected Spark DataFrame sparkDF2 to a Pandas DataFrame.

This operation brings the data from the distributed Spark DataFrame into a local Pandas DataFrame.

pandasDF2 is assigned the resulting Pandas DataFrame.

print(pandasDF2) prints the Pandas DataFrame pandasDF2 to the console.

spark.conf.get("spark.sql.execution.arrow.enabled") retrieves the value of the Spark configuration property "spark.sql.execution.arrow.enabled". This property determines whether Spark will use Apache Arrow for efficient data transfer between Spark and Pandas. It returns the current value of the property.The value of the property is assigned to the variable test.

print(test) prints the value of test to the console.

spark.conf.get("spark.sql.execution.arrow.pyspark.fallback.enabled") retrieves the value of the Spark configuration property.

"spark.sql.execution.arrow.pyspark.fallback.enabled". This property determines whether to enable the fallback mechanism for Arrow-based data transfer when Arrow is not available. It returns the current value of the property.The value of the property is assigned to the variable test123.

print(test123) prints the value of test123 to the console.

PROGRAM-4

Pyspark add month

Import the necessary functions col and expr from the **pyspark.sql.functions module**.

Define a list of tuples called **data**, where each tuple represents a row of data with two columns: "**date**" and "**increment**".

Create a DataFrame from the data using **spark.createDataFrame(data)**. Since no column names are provided, the columns will be named "0" and "1" by default.

Rename the columns of the DataFrame **using .toDF("date", "increment")**.

Select the "**date**" column, "**increment**" column, and a new column named **"inc_date"**.

The expression **expr("add_months(to_date(date, 'yyyy-MM-dd'), cast(increment as int))")** is used to calculate the incremental date based on the "**date**" and "**increment**" columns. It converts the "**date**" column to a date type using **to_date**, casts the "**increment**" column to an integer using cast, and adds the specified number of months using **add_months.**

Alias the resulting column as **"inc_date" using .alias("inc_date").**

Finally, **use .show()** to display the resulting DataFrame with the selected columns.

PROGRAM-5

Add new column

from pyspark.sql import SparkSession: This line imports the SparkSession class from the pyspark.sql module. SparkSession is the main entry point for working with structured data in Spark.

spark = SparkSession.builder: This creates a new instance of the SparkSessionBuilder, which is used to configure and create a SparkSession.

.appName('SparkByExamples.com'): This sets the name of the Spark application to 'SparkByExamples.com'. The application name is a user-defined string that helps identify the application in the Spark cluster.

.getOrCreate(): This method gets an existing SparkSession or creates a new one if none exists. If a SparkSession has already been created in the current context, it returns that instance; otherwise, it creates a new one.

data = [...]: This line defines a list called data that holds multiple tuples. Each tuple represents a data record.

Each tuple contains four elements representing different attributes of a person:

The first element represents the person's first name (a string).
The second element represents the person's last name (a string).
The third element represents the person's gender (a string, typically 'M' or 'F').
The fourth element represents the person's salary (an integer).

columns = ["firstname","lastname","gender","salary"]: This line defines a list called columns which contains the column names for the DataFrame. The order of the column names corresponds to the order of the fields in each tuple of the data list.

df = spark.createDataFrame(data=data, schema=columns): This line creates a PySpark DataFrame called df using the spark.createDataFrame() method. It takes two parameters:

data=data: This specifies the input data for the DataFrame, which is the data list that contains tuples representing the data records.

schema=columns: This specifies the schema for the DataFrame, which is the columns list that contains the column names.

df.show(): This line displays the contents of the DataFrame by invoking the show() method on the DataFrame df. The show() method prints the first 20 rows of the DataFrame in a tabular format.

if 'salary1' not in df.columns: print("aa"): This line checks if the column 'salary1' exists in the DataFrame by using the in operator to check if the string 'salary1' is present in the list of column names df.columns. If the column does not exist, it prints the string "aa".

from pyspark.sql.functions import lit: This line imports the lit function from the pyspark.sql.functions module. The lit function is used to create a column with a constant value.

df.withColumn("bonus_percent", lit(0.3)): This line adds a new column called "bonus_percent" to the DataFrame df. The withColumn() method is used to create a new column or replace an existing column. In this case, it creates a new column named "bonus_percent" with a constant value of 0.3.

.show(): This line displays the contents of the DataFrame after adding the new column using the show() method. The show() method prints the first 20 rows of the DataFrame with the new column included.

df.withColumn("bonus_amount", df.salary*0.3): This line adds a new column called "bonus_amount" to the DataFrame df. The withColumn() method is used to create a new column or replace an existing column. In this case, it creates a new column named "bonus_amount" by multiplying the values in the "salary" column by 0.3.

.show(): This line displays the contents of the DataFrame after adding the new column using the show() method. The show() method prints the first 20 rows of the DataFrame with the new column included.

df.withColumn("name", concat_ws(",", "firstname", "lastname")): This line adds a new column called "name" to the DataFrame df. The withColumn() method is used to create a new column or replace an existing column. In this case, it creates a new column named "name" by concatenating the values from the "firstname" and "lastname" columns, separated by a comma (","). The concat_ws() function is used for concatenation, with the first argument specifying the delimiter and the subsequent arguments specifying the columns to be concatenated.

.show(): This line displays the contents of the DataFrame after adding the new column using the show() method. The show() method prints the first 20 rows of the DataFrame with the new column included.

df.withColumn("current_date", current_date()): This line adds a new column called "current_date" to the DataFrame df. The withColumn() method is used to create a new column or replace an existing column. In this case, it creates a new column named "current_date" and assigns it the current date value using the current_date() function. The current_date() function returns the current date as a DateType.

.show(): This line displays the contents of the DataFrame after adding the new column using the show() method. The show() method prints the first 20 rows of the DataFrame with the new column included.

df.withColumn("grade", ...) creates a new column named "grade" in the DataFrame df using the withColumn() method.

The **when()** function from **pyspark.sql.functions** is used to define the conditions for assigning values to the "grade" column. It takes a boolean expression as the first argument and a value to be assigned if the condition is true as the second argument.

The first **when()** condition checks if the "**salary**" column is less than 4000. If true, it assigns the value "A" to the "grade" column using the lit() function.

The second **when()** condition checks if the "**salary**" column is between 4000 and 5000 (inclusive). If true, it assigns the value "B" to the "grade" column using the lit() function.

The **otherwise()** function is used to assign a default value of "C" to the "grade" column when none of the previous conditions are true.

The **.show()** method is used to display the resulting DataFrame with the new "grade" column.

df.withColumn("grade", ...) creates a new column named "grade" in the DataFrame df using the withColumn() method.

The **when()** function from **pyspark.sql.functions** is used to define the conditions for assigning values to the "grade" column. It takes a boolean expression as the first argument and a value to be assigned if the condition is true as the second argument.

The first **when()** condition checks if the "salary" column is less than 4000. If true, it assigns the value "A" to the "grade" column using the lit() function.

The second **when()** condition checks if the "salary" column is between 4000 and 5000 (inclusive). If true, it assigns the value "B" to the "grade" column using the lit() function.

The **otherwise()** function is used to assign a default value of "C" to the "grade" column when none of the previous conditions are true.

The **.show()** method is used to display the resulting DataFrame with the new "grade" column.

df.createOrReplaceTempView("PER") creates a temporary view named "PER" from the DataFrame df. This allows you to run SQL queries on the DataFrame.

spark.sql("select firstname,salary, '0.3' as bonus from PER").show() executes an SQL query that selects the "firstname" and "salary" columns from the "PER" view. It also adds a new column called "bonus" with a constant value of '0.3'. The .show() method is used to display the result.

spark.sql("select firstname,salary, salary * 0.3 as bonus_amount from PER").show() executes an SQL query that calculates a new column called "bonus_amount" by multiplying the "salary" column by 0.3. The .show() method is used to display the result.

spark.sql("select firstname,salary, current_date() as today_date from PER").show() executes an SQL query that adds a new column called "today_date" with the current date using the current_date() function. The .show() method is used to display the result.

spark.sql("select firstname,salary, case salary when salary < 4000 then 'A' else 'B' END as grade from PER").show() executes an SQL query that adds a new column called "grade" using a CASE statement. It checks the value of the "salary" column and assigns 'A' if the salary is less than 4000, and 'B' otherwise. The .show() method is used to display the result.

PROGRAM-6

Aggregate

df = spark.createDataFrame(data=simpleData, schema=schema) creates a DataFrame df using the createDataFrame method of the SparkSession. It takes the simpleData as the data source and schema as the schema.

df.printSchema() prints the schema of the DataFrame df, which displays the structure and data types of each column.

df.show(truncate=False) displays the content of the DataFrame df without truncating the column values.

df.select(approx_count_distinct("salary")).collect()[0][0] computes the approximate count of distinct values in the "salary" column using the approx_count_distinct function. The result is collected as a list, and [0][0] is used to extract the value from the first row and first column.

df.select(avg("salary")).collect()[0][0] calculates the average value of the "salary" column using the avg function. Similarly, the result is collected as a list, and [0][0] is used to extract the average value from the first row and first column.

df.select(collect_list("salary")).show(truncate=False) collects all the values of the "salary" column into a list using the collect_list function. The result is displayed without truncating the column values.

df.select(collect_set("salary")).show(truncate=False) collects all the unique values of the "salary" column into a set using the collect_set function. The result is displayed without truncating the column values.

df2 = df.select(countDistinct("department", "salary")) calculates the distinct count of values in the "department" and "salary" columns using the countDistinct function. The result is stored in the DataFrame df2.

df2.show(truncate=False) displays the distinct count of values in the "department" and "salary" columns.

df2.collect()[0][0] retrieves the distinct count value from the first row and first column of the DataFrame df2.

df.select(count("salary")).collect()[0] calculates the count of non-null values in the "salary" column using the count function. The result is collected as a list, and [0] is used to extract the count value.

df.select(first("salary")).show(truncate=False) retrieves the first value in the "salary" column using the first function.

df.select(last("salary")).show(truncate=False) retrieves the last value in the "salary" column using the last function.

df.select(kurtosis("salary")).show(truncate=False) calculates the kurtosis (fourth moment) of the "salary" column using the kurtosis function.

df.select(max("salary")).show(truncate=False) retrieves the maximum value in the "salary" column using the max function.

df.select(min("salary")).show(truncate=False) retrieves the minimum value in the "salary" column using the min function.

df.select(mean("salary")).show(truncate=False) calculates the mean (average) value of the "salary" column using the mean function.

df.select(skewness("salary")).show(truncate=False) calculates the skewness (third moment) of the "salary" column using the skewness function.

df.select(stddev("salary"), stddev_samp("salary"), stddev_pop("salary")).show(truncate=False) calculates the standard deviation using different methods (stddev, stddev_samp, stddev_pop) for the "salary" column. The results are displayed without truncating the column values.

df.select(sum("salary")).show(truncate=False) calculates the sum of values in the "salary" column using the sum function.

df.select(sumDistinct("salary")).show(truncate=False) calculates the sum of distinct values in the "salary" column using the sumDistinct function.

df.select(variance("salary"), var_samp("salary"), var_pop("salary")).show(truncate=False) calculates different measures of variance (variance, var_samp, var_pop) for the "salary" column. The results are displayed without truncating the column values.

PROGRAM-7

Array string

from pyspark.sql.functions import col, concat_ws: This line imports the necessary functions col and concat_ws from the pyspark.sql.functions module. col is used to reference a column by name, and concat_ws is used to concatenate values with a specified delimiter.

df2 = df.withColumn("languagesAtSchool", concat_ws(",", col("languagesAtSchool"))): This line creates a new DataFrame df2 by adding a new column called "languagesAtSchool". The values in the "languagesAtSchool" column are concatenated with a comma delimiter using the concat_ws function. The col("languagesAtSchool") expression specifies the column to be modified.

df2.printSchema(): This line prints the schema of the DataFrame df2, which shows the structure and data types of its columns.

df2.show(truncate=False): This line displays the contents of the DataFrame df2 without truncating the column values. It provides a tabular representation of the DataFrame, showing all rows and columns.

df.createOrReplaceTempView("ARRAY_STRING"): This line creates a temporary view named "ARRAY_STRING" from the DataFrame df. A temporary view allows you to query the DataFrame using SQL syntax.

spark.sql("select name, concat_ws(',',languagesAtSchool) as languagesAtSchool,currentState from ARRAY_STRING").show(truncate=False): This line executes a SQL query on the temporary view "ARRAY_STRING". The query selects the "name", "languagesAtSchool", and "currentState" columns from the view. The concat_ws(',', languagesAtSchool) function is used to concatenate the values in the "languagesAtSchool" column with a comma delimiter. The result of the query is then displayed using the show() function.

PROGRAM-8

Array type

ArrayType(StringType(), False): This line defines an ArrayType column type with elements of type StringType. The False argument indicates that the array can contain null values.

data: This variable is a list of tuples, where each tuple represents a row of data. Each tuple contains five elements:

The first element is a string representing a person's name in the format "FirstName,MiddleName,LastName".
The second element is a list of strings representing programming languages known by the person.
The third element is a list of strings representing technologies known by the person.
The fourth element is a string representing the person's current state.
The fifth element is a string representing the person's home state.

StructType([...]): This line creates a StructType object that represents the schema of a DataFrame. The schema is defined as a list of StructField objects.

StructField("name", StringType(), True): This line defines a StructField named "name" with a StringType. It indicates that the "name" field can contain null values (True).

StructField("languagesAtSchool", ArrayType(StringType()), True): This line defines a StructField named "languagesAtSchool" with an ArrayType of StringType. It indicates that the "languagesAtSchool" field can contain null values (True).

StructField("languagesAtWork", ArrayType(StringType()), True): This line defines a StructField named "languagesAtWork" with an ArrayType of StringType. It indicates that the "languagesAtWork" field can contain null values (True).

StructField("currentState", StringType(), True): This line defines a StructField named "currentState" with a StringType. It indicates that the "currentState" field can contain null values (True).

StructField("previousState", StringType(), True): This line defines a StructField named "previousState" with a StringType. It indicates that the "previousState" field can contain null values (True).

spark.createDataFrame(data=data, schema=schema): This line creates a DataFrame called df using the createDataFrame method of the spark session. It takes two arguments:

data: The data used to populate the DataFrame. In this case, it's the data list containing rows of data.
schema: The schema of the DataFrame. It specifies the structure and data types of the columns in the DataFrame. In this case, it's the schema object defined earlier.

df.printSchema(): This line prints the schema of the DataFrame df using the printSchema method. It displays the data types and nullable properties of each column in the DataFrame.

df.show(): This line shows the content of the DataFrame df using the show method. It displays the first 20 rows of the DataFrame in a tabular format.

df.select(df.name, explode(df.languagesAtSchool)).show(): This line selects the name column from the DataFrame df and applies the explode function to the languagesAtSchool column. The explode function transforms an array column into multiple rows, with each row containing one element from the array. The resulting DataFrame will have additional rows for each element in the languagesAtSchool array.

df.select(split(df.name, ",").alias("nameAsArray")).show(): This line selects the name column from the DataFrame df and applies the split function to it. The split function splits a string column into an array of substrings based on a delimiter. In this case, the delimiter is a comma (","), and the resulting array is assigned an alias "nameAsArray" in the output DataFrame.

df.select(df.name, array(df.currentState, df.previousState).alias("States")).show(): This line selects the name column from the DataFrame df and applies the array function to create a new array column named "States". The array function constructs an array column from the specified input columns. In this case, the "States" array column contains values from the currentState and previousState columns.

df.select(df.name, array_contains(df.languagesAtSchool, "Java").alias("array_contains")).show(): This line selects the name column from the DataFrame df and applies the array_contains function to the languagesAtSchool column. The array_contains function checks if the given value ("Java" in this case) exists in the array column. The result is a boolean column named "array_contains" indicating whether each row's languagesAtSchool array contains the value "Java".

PROGRAM-9

Broadcast Dataframe

This code is creating a broadcast variable named broadcastStates using Spark's **SparkContext.broadcast() method**. The broadcast variable contains a dictionary that maps state abbreviations to their corresponding full names.

states = {"NY":"New York", "CA":"California", "FL":"Florida"}: This line defines a dictionary states where state abbreviations (keys) are mapped to their full names (values). For example, "NY" maps to "New York".

broadcastStates = spark.sparkContext.broadcast(states): This line creates a broadcast variable named broadcastStates by calling the broadcast() method on the SparkContext (spark.sparkContext) with the states dictionary as the argument. Broadcasting the variable allows it to be efficiently shared among all the worker nodes in the Spark cluster.

data = [("James","Smith","USA","CA"), ("Michael","Rose","USA","NY"), ("Robert","Williams","USA","CA"), ("Maria","Jones","USA","FL")]: This line defines the data list, where each element represents a row of data to be included in the DataFrame. Each row is a tuple containing the first name, last name, country ("USA"), and state abbreviation.

columns = ["firstname","lastname","country","state"]: This line defines the column names for the DataFrame. The column names are specified as strings in the columns list. The order of the column names should match the order of values in each tuple of the data list.

df = spark.createDataFrame(data=data, schema=columns): This line creates the DataFrame df by calling the createDataFrame() method on the spark object. The data argument is set to the data list, which contains the rows of data. The schema argument is set to the columns list, which specifies the column names.

df.printSchema(): This line prints the schema of the DataFrame, which shows the names and data types of each column.

df.show(truncate=False): This line displays the contents of the DataFrame. The show() method is called on the DataFrame with truncate=False to display the full contents of each column without truncation.

def state_convert(code):: This line defines a function named state_convert that takes a code parameter representing a state code.

return broadcastStates.value[code]: This line retrieves the value associated with the code key from the broadcastStates variable. Since broadcastStates is a broadcast variable, it allows for efficient and distributed access to the shared states map.

result = df.rdd.map(lambda x: (x[0],x[1],x[2],state_convert(x[3]))).toDF(columns): This line applies the state_convert function to each row of the DataFrame df using the RDD (Resilient Distributed Dataset) representation of df. It maps each row to a new tuple, where the fourth element is replaced with the corresponding state name obtained by calling state_convert function. Finally, toDF(columns) converts the resulting RDD back to a DataFrame with the specified columns.

result.show(truncate=False): This line displays the contents of the result DataFrame, showing all columns without truncation.

df['state']: This selects the 'state' column from the DataFrame df.

broadcastStates.value: This retrieves the value of the broadcast variable broadcastStates, which is a dictionary of state codes and their corresponding names.

df['state'].isin(broadcastStates.value): This checks if each value in the 'state' column of df is present in the values of the broadcastStates.value dictionary. It returns a Boolean column that indicates whether each value is in the dictionary.

df.where(...): This applies the filter condition to the DataFrame df. Rows that satisfy the condition (i.e., their 'state' value is present in the broadcastStates.value dictionary) are included in the filtered DataFrame filteDf.

PROGRAM-10

CAST COLUMN

The columns list defines the schema for the DataFrame. Each element in the list represents a column in the DataFrame, and the order of the elements corresponds to the order of the columns in the data.

The spark.createDataFrame function is called with the data parameter set to simpleData and the schema parameter set to columns. This creates a DataFrame where each row corresponds to a tuple in simpleData, and the column names and data types are defined by the columns list.

The **df.printSchema()** statement prints the schema of the DataFrame, which shows the names and data types of each column.

The **df.show(truncate=False)** statement displays the contents of the DataFrame, showing all columns and their respective values. The truncate=False argument ensures that the full contents of each column are displayed without truncation.

The **from pyspark.sql.functions** import statement imports the col function, which is used to reference columns in DataFrame transformations.

The from **pyspark.sql.**types import statement imports the data types needed for casting columns.

The **df2 = df.withColumn(...)** statement creates a new DataFrame named df2 by applying a series of transformations to the original DataFrame df.

The **withColumn function** is used to create a new column or replace an existing column with the same name. In this case, it is used to update the columns "age", "isGraduated", and "jobStartDate".

The **df3 = df2.selectExpr(...)** statement creates a new DataFrame named df3 by selecting and transforming columns from the DataFrame df2.

The **selectExpr function** allows specifying complex expressions and transformations on columns in a concise way. It takes string expressions as arguments.

The arguments to **selectExpr** specify the columns to select and the transformations to apply. Each argument is a string expression in the format **"cast(column_name as data_type) alias",** where column_name is the name of the column, data_type is the target data type, and alias is an optional alias for the column.

The **cast** function is used to cast the column to the specified data type, similar to the previous code snippet.

The transformed columns are specified as new column aliases in the format **"cast(column_name as data_type) alias".** For example, **"cast(age as int) age" casts the "age"** column to an integer data type and assigns the alias "age" to the transformed column in the new DataFrame.

The **df3.printSchema()** statement prints the schema of the transformed DataFrame df3, showing the updated data types of the columns.

The **df3.show(truncate=False)** statement displays the content of the DataFrame df3, showing the transformed columns and their values.

The statement **df3.createOrReplaceTempView("CastExample")** creates a temporary view named "CastExample" for the DataFrame df3. This allows you to query the DataFrame using Spark SQL.

The **spark.sql function** is used to execute SQL queries on the DataFrame. In this case, the SQL query "SELECT STRING(age), BOOLEAN(isGraduated), DATE(jobStartDate) from CastExample" is executed.

The SQL query selects the transformed columns from the "**CastExample**" view and applies additional type conversions using the SQL functions STRING, BOOLEAN, and DATE.

The result of the SQL query is stored in the DataFrame df4.

The **df4.printSchema()** statement prints the schema of the DataFrame df4, showing the data types of the columns.

The **df4.show(truncate=False)** statement displays the content of the DataFrame df4, showing the transformed columns and their values.

PROGRAM-11

Change string double

This code showcases different approaches to casting column data types in Spark DataFrames using both DataFrame functions and SQL expressions.

PROGRAM-12

Collecting

This code creates a data frame, collects data from it, and demonstrates how to access and manipulate the collected data.

PROGRAM-13

Column Functions

alias: Renames the columns using aliases.
asc and desc: Sorts the DataFrame in ascending and descending order.
cast: Converts the data type of a column.
between: Filters rows based on a range of values.
contains: Filters rows based on a substring match.
startswith and endswith: Filters rows based on the start or end of a string.
isNull and isNotNull: Filters rows based on null or non-null values.
like and rlike: Filters rows based on pattern matching.
substr: Extracts a substring from a column.
when and otherwise: Implements conditional expressions.
isin: Filters rows based on a list of values.
getItem: Retrieves an element from an array or map column.
getField: Retrieves a field from a struct or map column.

The code snippet showcases the usage of these operations to manipulate and filter data in a Spark DataFrame.

PROGRAM-14

Column Operations

Creating a DataFrame: Data is created using a list of tuples and converted into a DataFrame with specific column names.

Printing schema and data: The schema and content of the DataFrame are printed using printSchema() and show() functions.
Selecting columns: Columns are selected using dot notation (df.columnName) or indexing (df["columnName"]).

Applying functions: Various functions such as substr, startswith, and col are used to manipulate and filter column values.
Renaming columns: Columns with dot notation are renamed by replacing dots with underscores.
Accessing columns with backticks: Columns with special characters or dots in their names are accessed using backticks.
Accessing struct columns: Columns within a struct column are accessed using dot notation or indexing.
Column operators: Arithmetic and comparison operators (+, -, *, /, %, >, <, ==) are applied to columns.

The code snippet demonstrates how to perform column selection, function application, column renaming, and struct column access using different syntax variations.

PROGRAM-15

Convert Map to Column

Creating a DataFrame: Data is created using a list of tuples, where each tuple represents a row with a name and properties (a map).
Printing schema and data: The schema and content of the DataFrame are printed using printSchema() and show() functions.
Accessing nested fields: The nested fields in the properties map are accessed using the getItem function or indexing (properties["key"]).

Transforming the DataFrame: The DataFrame is transformed by extracting the hair and eye properties into separate columns using withColumn and dropping the original properties column.
Working with map keys: The unique keys from the properties map are extracted using explode and map_keys. The keys are collected as a list and then used to create dynamic columns using col and getItem.

The code showcases different ways to access and manipulate nested structures and maps within a DataFrame. It demonstrates accessing map values, transforming the DataFrame structure, and dynamically generating columns based on map keys using functions like explode, map_keys, getItem, and col.

PROGRAM-16

Convert column to Map

First, we define the schema for the DataFrame, specifying the data types of each column. Then, we create the DataFrame using the provided data and schema.

The DataFrame is printed to display its schema and contents.

Next, we use the withColumn() function to create a new column called "propertiesMap". Inside the create_map() function, we specify key-value pairs using lit() to provide the keys and col() to reference the columns to be mapped. We pass these pairs to create_map() to create a map column.

Finally, we drop the original "salary" and "location" columns using the drop() function to remove them from the DataFrame.

The modified DataFrame is printed again to show the updated schema and contents.

The methods used in this code include createDataFrame(), printSchema(), show(), withColumn(), lit(), col(), create_map(), and drop().

PROGRAM-17

Count distinct

The code demonstrates different ways to obtain distinct values from a DataFrame in PySpark. It shows how to use the `distinct()` method, the `countDistinct()` function, and Spark SQL to retrieve distinct values and perform distinct count operations on specific columns or the entire DataFrame.

PROGRAM-18

create dataframe dictionary

This code demonstrates the creation of DataFrames with map columns, accessing map values using `getItem()` and square brackets ([]), extracting map keys, and manipulating complex data structures in PySpark. It highlights the flexibility of PySpark's DataFrame API in handling structured and semi-structured data.

PROGRAM-19

Create dataframe

This code showcases different approaches to create DataFrames in PySpark. It covers creating DataFrames directly from RDD, creating DataFrames from RDD with explicit column names, creating DataFrames from RDD and data list with column names, and creating DataFrames from `Row` objects. It provides flexibility in creating DataFrames from various data sources and highlights the versatility of PySpark's DataFrame API.

PROGRAM-20

Create list

In summary, this code illustrates how to create DataFrames in PySpark using lists of tuples, custom schemas, lists of Row objects, and RDDs. It showcases different approaches to define the schema and convert data into a DataFrame.

PROGRAM-21

Current date timestamp

In summary, this code showcases the usage of date and timestamp functions in PySpark. It demonstrates how to add current date and timestamp columns to a DataFrame, format date and timestamp values, and perform these operations using both DataFrame functions and Spark SQL.

PROGRAM-22

Dataframe flatmap

In summary, this code creates a DataFrame `df` using PySpark by specifying column names and providing data in the form of a list of tuples. It then prints the schema and displays the contents of the DataFrame.

PROGRAM-23

Dataframe repartition

In summary, this code demonstrates how to check and manipulate the number of partitions in a PySpark DataFrame using methods like `repartition()`, `coalesce()`, and `getNumPartitions()`. It also shows an example of writing the DataFrame to a CSV file.

PROGRAM-24

Dataframe

This code essentially creates a SparkSession object and prints its details for informational purposes.

PROGRAM-25

Date string

Overall, this code demonstrates how to use date and timestamp functions in Spark SQL to perform various transformations on date and timestamp data.

PROGRAM-26

Date timestamp functions

The code essentially showcases how to perform various date and time operations and extract specific components from date and timestamp values using PySpark's built-in functions.

PROGRAM-27

Date diff

Overall, the code illustrates how to manipulate and calculate date differences, months between dates, and rounding values using PySpark's date and time functions both in DataFrame operations and SQL queries.

PROGRAM-28

Distinct

Overall, the code demonstrates different techniques to handle duplicates in a DataFrame, including removing all duplicates, removing duplicates from specific columns, and printing the count and content of the resulting DataFrames.

PROGRAM-29

Drop-column

In summary, the code showcases different ways to drop columns from a PySpark DataFrame using the `drop` method, whether by directly referencing column names, using column objects, or passing multiple column names at once.

PROGRAM-30

Drop-null

In summary, the code showcases different ways to handle missing or null values in a PySpark DataFrame using the `na.drop()` and `dropna()` methods. The `na.drop()` method can be used with various options, such as removing rows with any null values or specifying specific columns to consider. The `dropna()` method provides a similar functionality for dropping rows with null values.