Apache Spark

1. Setting up apache spark.

* Python IDE enivornmaent should be there or should be installed,(i used anaconda)
* Jdk ,apache spark runs on scala which news java 11 or jdk.

(i have installed java 8 in installation there is some path spaces be careful with that or else ir will create errors) C:\jdk

Also apache spark does not work with newer version of Java 16 and 17 it most compatible with java 11 or 8.

* Install spark ,to open spark files you have to have rarlabs installed in your windows

About the directories and path.

cd c:\SparkCourse

dir

spark-submit file\_name with python code

What is RDD ?

How to create an RDD would be the first thing to know

Key/value concept and how it is done through RDD APIs

Variable =rdd.map(lambda x: (x,1)) > this how you create a key/ value in rdd.

Now that was just a key /value pair Now if you want to create a list then,

reduceByKey(): combines values with the same key using some function.

goupbyKey():Group values with the same keys

sortByKey(): sort by key values

keys (),value() -creates an RDD of just the keys ,or just the values

Only use mapValue() or flatMapValues() if it does not affect the keys.and if you call mapValues( ) then it will create everz values that will be turned into tuples.

Flitering RDD’s

filter() - if we want to filter out entries that have specific values in the first item for eg.minTemps = parsedlines. filter(lambda x: ‘’TMIN’’ in x[1])

Map VS FlatMAp

map() transforms each element of an RDD into one new element

flatMap() can create many new elements from each one

Re expression (import re)

re.compile(r'\W+', re.UNICODE).split(text.lower())

/W means splitting regular expression on word and we rae reducing it into lower case so that we don't get different results.

sortBykey() to sort the values (so if your code haves count in values then make sure you flip it to bring count to key and then use sortBYKey()).

SPARKSQL (where RDD is extended to dataFrames) most them use this

Here is the boilerplate how to use spark sql in python

From pyspark.sql, import SparkSession,Row

Spark =aprkSession.builder.appName(“SparkSQL”).getorCreate() [get if want to get some spark code which is already connected]

inputData =spark.read.json(dataFile)

inputData.cretaeOrREplaceTEmpView(“myStrcuturedStuff”)

myResultDataFrame =spark.sql(“SELECT foo FRom bar ORDER BY foobar”)

What is the difference between dataFrame and Datasets?

So, DataFrames are like flexible tables in a spreadsheet that let you work with different types of data easily. Datasets are more strict collections of data that are efficient for big tasks. Depending on what you need to do, you might choose one over the other.

#infer the schema and register the DtaFrame as a table

variable= spark.createDataFrame(people).cache() (creating a dataframe from rdd)

To query this as database table, we have to a view

variable.createOrReplaceTempView(‘’people”)

dataframe.select(“name”).show() > will print all column names

dataframe.filter() >filter out things

dataframe.groupby().cout().show()

After I would or done with spark we enter this

spark.stop()

Next, we have to use Sparksql for extracting or counting words.

Pyspark.sql import function as func

func.explode () >similar to flatmap()

(explodes columnsinto rows)

func.split()

func.lower()

func,col(‘’calnmae”) to call a column name

Using custom schema using DataFrame and nor RDD

Min temperature(dataset) now if we dont have headers in our columns we import

from pyspark.sql.types import StructType, StructField, StringType, IntegerType, FloatType

To apply the schema

schema = StructType([ \

StructField("stationID", StringType(), True), \

StructField("date", IntegerType(), True), \

StructField("measure\_type", StringType(), True), \

StructField("temperature", FloatType(), True)])

This like giving the colum name and aslo we are telling what type data type would be there in each column

Df,filter (filters the value given in the function)

df.select() would whatever you want to select from the columns

df.groupby() functions groups all the according to the different column names given in the the function.

minTempsByStationF = minTempsByStation.withColumn("temperature",

func.round(func.col("min(temperature)") \* 0.1 \* (9.0 / 5.0) + 32.0, 2))\

.select("stationID", "temperature").sort("temperature")

Here its creating a new column ‘’tempetature’’ fucn.round () round to the nearest decimal value,sort() function sorts the values for min to max.

In Total\_spend-customer-sorted (py file)

totalByCustomer = customersDF.groupBy("cust\_id").agg(func.round(func.sum("amount\_spent"), 2) \

.alias("total\_spent"))

customersDF: This is a DataFrame that presumably contains data about customers and their spending habits. It likely has columns like "cust\_id" (customer identifier) and "amount\_spent" (the amount of money spent by each customer).

.groupBy("cust\_id"): This operation groups the rows of the DataFrame by the "cust\_id" column. In other words, it groups the data so that all rows with the same customer ID are in the same group.

.agg(func.round(func.sum("amount\_spent"), 2).alias("total\_spent"))`: This part of the code performs aggregation on each group. It does the following: func.sum("amount\_spent"): This function calculates the sum of the "amount\_spent" values within each group. It adds up the total amount spent by each customer.

func.round(..., 2): This function rounds the result of the sum to 2 decimal places, ensuring that the total spent is displayed as a monetary value with two decimal places.

.alias("total\_spent"): This renames the aggregated column to "total\_spent". The resulting DataFrame will have two columns: "cust\_id" (customer identifier) and "total\_spent" (the total amount spent by each customer).

For data analysis, dataframes are easier than RDD.

**ADVANCE Examples of SPARK PROGRAMS**

def lookupName(movieID):

return nameDict.value[movieID]

lookupNameUDF = func.udf(lookupName)

# Add a movieTitle column using our new udf

moviesWithNames = movieCounts.withColumn("movieTitle", lookupNameUDF(func.col("movieID")))

lookupNameUDF = func.udf(lookupName): This line defines a User-Defined Function (UDF) named lookupNameUDF using PySpark's func.udf function. It essentially wraps the lookupName Python function, making it available for use within a Spark DataFrame transformation.

**Broadcast**

**:popular-movies-nice-dataframe.py**

**def loadMovieNames():**

**movieNames = {}**

**# CHANGE THIS TO THE PATH TO YOUR u.ITEM FILE:**

**with codecs.open("C:/SparkCourse/ml-100k/ml-100k/u.item", "r", encoding='ISO-8859-1', errors='ignore') as f:**

**for line in f:**

**fields = line.split('|')**

**movieNames[int(fields[0])] = fields[1]**

**return movieNames**

**spark = SparkSession.builder.appName("PopularMovies").getOrCreate()**

**nameDict = spark.sparkContext.broadcast(loadMovieNames())**

Imagine you have a large dataset that you want to process using Spark. You also have a small piece of data, like a dictionary, that you want to use while processing the large dataset. Instead of sending a copy of that dictionary to every worker node (which could be wasteful in terms of network and memory usage), you can use broadcasting. Here's a simpler explanation: Without Broadcasting: If you didn't broadcast the dictionary, Spark would send a separate copy of that dictionary to each worker node. This could consume a lot of network bandwidth and memory, especially if the dictionary is large. With Broadcasting: When you broadcast the dictionary, Spark sends it only once to all worker nodes. Each worker node gets a read-only copy of the dictionary, and it doesn't need to transfer the entire dictionary again and again. This saves network bandwidth and memory. Broadcasting is useful when you have a relatively small piece of data that you need on all worker nodes to perform some operation efficiently. It's a way to optimize distributed data processing in Spark by reducing data transfer overhead.

lines = spark.read.text("file:///SparkCourse/Marvel-graph.txt")

Takes each row as value from the csv file

connections = lines.withColumn("id", func.split(func.trim(func.col("value")), " ")[0]) \

.withColumn("connections", func.size(func.split(func.trim(func.col("value")), " ")) - 1) \

.groupBy("id").agg(func.sum("connections").alias("connections"))

func.trim Function:

* func.trim is used to remove leading and trailing white spaces (including spaces, tabs, and other whitespace characters) from a string column in a DataFrame.
* It is primarily used for cleaning and preprocessing text data.
* func.split is used to split a string column into an array of substrings based on a specified delimiter. It's particularly useful when you want to split a string into multiple parts.
* It is often used for tasks like parsing log files or breaking down a single column into multiple columns.

connections.sort(func.col("connections").desc()).first()

first() just taking the first row of that dataframe

**Breadth First Search**

**Find The Degree** of separation (6 degrees of separation) (iterative algorithm)

FIRST we have to convert each row or value into a node representation

:degree-of-separation.py

Difference between flatMap() and Map()

map Function:

* The map function applies a given function to each element of a collection and returns a new collection of the same length. Each element in the original collection is mapped to exactly one element in the resulting collection.

flatMap Function:

* The flatMap function applies a given function to each element of a collection and returns a new collection. However, unlike map, the result of applying the function can be zero, one, or multiple elements for each input element.
* It is used for one-to-zero, one-to-many, or many-to-many element transformations.

Explanation of the code given in degree-of.spearation py file (how to BFS)

This code is an implementation of a Breadth-First Search (BFS) algorithm using Apache Spark. It's used to find the shortest path between two characters in a graph. Here's a brief overview of the code:

1. \*\*Setup Spark\*\*: It configures and sets up a SparkContext for local execution.

2. \*\*Define Characters\*\*: It specifies two character IDs, `startCharacterID` and `targetCharacterID`, for which the shortest path will be found.

3. \*\*Accumulator\*\*: `hitCounter` is an accumulator used to signal when the target character is found during BFS traversal.

4. \*\*Data Transformation Functions\*\*:

- `convertToBFS`: Converts input lines into a format suitable for BFS traversal by splitting the input line into character ID, connections, color, and distance.

- `createStartingRdd`: Reads the input data and maps it to a format suitable for BFS. It creates the initial RDD.

5. \*\*BFS Map and Reduce Functions\*\*:

- `bfsMap`: Performs the BFS mapping step. It explores neighbors of gray nodes, updates distances, and colors nodes.

- `bfsReduce`: Combines data from different paths to determine the shortest path and preserve node properties.

6. \*\*Main BFS Loop\*\*:

- The code enters a loop where BFS iterations are performed.

- In each iteration, it prints the current iteration number.

- It uses `flatMap` to create new vertices and update node properties in the mapping stage.

- It checks if the target character has been found by examining the accumulator.

- If the target character is found, it breaks out of the loop.

- In the reducing stage, it combines data to determine the shortest path and preserve node properties.

- The loop continues for a maximum of 10 iterations.

7. \*\*Result Output\*\*: If the target character is found, the code prints the number of directions from which the target character was reached.

In summary, this code uses Spark to perform BFS on a graph represented as an RDD. It iteratively explores the graph, looking for the shortest path from a starting character to a target character. It uses accumulators to signal when the target character is found, and it combines data to find the shortest path.

**ITEM BASED Collabrating Filtering** in Spark recommendation of similar movies

This means you need database of similar movies,

Explanation : if I like a movie and also like another movie, which becomes a pair, and any other person also likes that pair of movies, and a third person watches one of the movies in the pair and likes it, the other movie in the pair can be recommended to him.

Cosine similarity score is used to check how similar the rating of the two movies are

:movies-similarties-dataframe.py

**RUNNIG SPARK ON CLUSTER**

**Using ElasticMApreduce**

**Hadoop YARN component as cluster manager**

**.partitionBy()**

Any operation your are using like join(),groupby(),lookyp()...etc as the data is in millions,it is good to use partitionBy()

**INTRODUCING MACHINE LEARNING LIBRARY IN SPARK**

**Linear regression Decision Tree**

**Advanced Analytics with Spark**

**spark Streaming structure streaming, and Graph x**