Kmeans Clustering in Pyspark:---

Software required:-

1) If we are working on a cluster we do not need anything just type :- pyspark to start the pythonspark shell..

2) if we are on stand-alone system we need following:

a)java 7 and above

b)python 2.7 or higher

c) and spark 1.6.2 tgz extracted to C drive.

Q1:- Why we need to use spark(pyspark)?

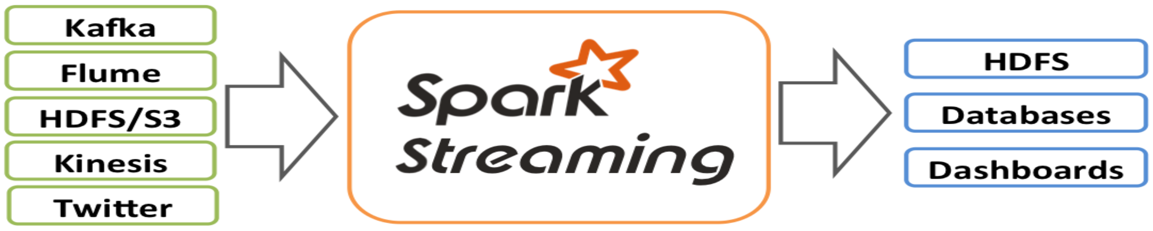
Ans:- Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.Where as in spark due to RDDs the speed of execution is enhanced a lot.

RDDS :

Resilient Distributed Datasets

- RDD) is a fundamental data structure of Spark
- an immutable distributed collection of objects(cannot delete)
- Dataset in RDD is divided into logical partitions, which may be computed on different nodes of the clusters.
- RDD is a read-only, partitioned collection of records

- **Importing Data from Databases To work with Spark:-**



Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to file systems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.

- **For Importing From Some databses:-**

  We can use Sqoop (commands):
  Example: we will import from oracle Databse:-

  - Import the data of the table emp present in Oracle database to HDFS.
  - oracle connector should be present in the sqoop directory and the command should be executed from the sqoop library
    Command:-
    [dms@BAN-SDU-OVS4-VM3 ~ sqoop import --connect jdbc:oracle:thin:@localhost:1521/orcl --username pawandb-- password dms123 --table ACTIVITY

GOAL: To apply k-means cluster model from MLIB of spark to check number of cluster for our crime data:-

Note:- We are using our local/hdfs for pulling data.

Tools:-pyspark,hive,notepad:

Libraries:-MLib,numpy,math,pandas,sqlContext

Data:- Insurance_data()

Explaination:--

- **Step 1:- We open our pyspark shell (**type pyspark**)**
- **Step2:-**Load our required library
- **Step3:-**Load the data to a rdd
- **Step4:-**Select the required column and create a new rdd
- **Step5:-**make a float array with data
- **Step6:-**Create our Kmean-model with the data
- **Step7:** Create a function to check the eculidiean distance(To check WSSE for our clusters)
- **Step8:**Repeat step5 with different number of clsuters as input
- **Step9:-**When there is significant drop in WSSE value stop that's our required number of cluster
- **Step10:-**Once we get the WSSE to minimal,now we can predict the cluster for the data.
- **Step11**:- Load the predicted cluster to a rdd
- **Step12**:-join the original data with the new cluster data using zip.
- **Step13**:-Save the final output in a CSV format using a user-defined function.

1**.The spark.mllib implementation includes a parallelized variant of the k-means++ method called kmeans||.**

---k is the number of desired clusters.

---maxIterations is the maximum number of iterations to run.

---initializationMode specifies either random initialization or initialization via k-means||.

---runs is the number of times to run the k-means algorithm (k-means is not guaranteed to find a globally optimal solution,

and when run multiple times on given dataset,algorithm returns best clustering result).

---initializationSteps determines the number of steps in the k-means|| algorithm.

---epsilon determines the distance threshold within which we consider k-means to have converged.

---initialModel is an optional set of cluster centers used for initialization. If this parameter is supplied, only one run is performed.

NOTE:- whenever u get not a rdd list type error just convert ur data to a rdd using sc.parallelize(data)

STEP1:-

```
from pyspark.mllib.clustering import KMeans, KMeansModel

from numpy import array

from math import sqrt

from pyspark.sql import SQLContext, Row

from pyspark.sql import SQLContext

sqlContext = SQLContext(sc)
```

STEP2:-

```
# Load and parse the data

mydata = sc.textFile("hdfs:/// user/ dms/ Analytics/ Pawanwork/ bala1/ sampleData.csv")

# it will from a rdd with default number of partition We can also mention if we want specific number of partitions

mydata = sc.textFile("("hdfs:/// user/ dms/ Analytics/ Pawanwork/ bala1/ sampleData.csv
",numPartitions=5)
```

NOTE:- Any Time during the process If we want to see the data into a tabular form Just Type:-

My_data = sqlContext.createDataFrame(rdd name)

My_data.show()

STEP3,4&5:-

# to remove headers as it is not a numeric data

Header = mydata.first() # to get the first row as header out to rdd/list

data 1=my data.filter(lambda x: x!=Header) # filters out the first row from the data

#To peep inside our Rdd anytime we can use : rdd_name.first():- to view frist row

: rdd_name.take(n):-to view nrows

: rdd_collect() :- To see all data

#To select Required columns for making the clusters(only numeric column we need)

data2 =data1.map(lambda x: x.split(","))

data3 = data2.map(lambda x: (x[2],x[4],x[9],x[11]))

# here since indexing starts from 0 we chose

# we are selecting only the 3rd,5th,10th,12th column of our data

# to split the contents and convert to float

parsedData = data3.map(lambda line: array([float(x) for x in line]))

# we have converted the data to float format

STEP6:-

# Build the model (cluster the data)

```
clusters = KMeans.train(parsedData, 2, maxIterations=10,runs=10, initializationMode="random")
```

# here '2' indicates the number of clusters,At first we take two then increase or decrease this number on basis on WSSE value

STEP 7:-

# Evaluate clustering by computing Within Set Sum of Squared Errors(Eculidiean distance)

```
def error(point):
    center = clusters.centers[clusters.predict(point)] #   clusters should be our model name
    return sqrt(sum([x**2 for x in (point - center)]))
```

NOTE:- To avoid indentation error code above def code should when typed in pyspark should look exactly as wriiten.Nothing should be directly below the def,we need to take of spacing .

STEP 9:-

#calculate the WSSE for the number of clusters

```
WSSSE = parsedData.map(lambda point: error(point)).reduce(lambda x, y: x + y)

print("Within Set Sum of Squared Error = " + str(WSSSE))
```

# we can check this error value to see for significant drops and decide our cluster number. For this we need to change number of cluster parameter and check for WSSSE value and Using elbow criteria finalise or number of clusters

# after this step we can change number of cluster to see where the drop is significant and that will be our final cluster number..

Go Back to change Number of cluster

STEP 10:- # Save and load model

clusters.save(sc, "myModelPath")

sameModel = KMeansModel.load(sc, "myModelPath")......I t is saved in a format called parquet in hdfs

# We save model for future data of same format from same source.This model will tell the cluster Id for the future data.

STEP11:-

Once the Number of cluster is finalized we can predict clusters for our data sets:-

clusters.predict(parsedData).collect() # here clusters is the model name.

[0, 0, 0, 3, 0, 3, 1, 0, 0, 3, 2, 1, 0, 1, 2, 1, 1, 0, 2, 0, 3, 0, 2, 0, 3, 1, 1, 0, 2, 3, 0, 0, 0, 2, 1, 3, 3, 1, 3, 0, 2, 3, 3, 1, 2, 3, 3, 2, 2, 3]

STEP12:-

We need to create a master table using our cluster Id and our original data to do that we will use ZIP **function.**

clusterRdd = clusters.predict(parsedData) # **ClusterRdd is the rdd in which we are storing the cluster** Id of each row.

final_table = data3.zip(clusterRdd) # we are concating the cluster Id with their respective data

# final table is our master table.

#To peep inside our Rdd anytime we can use : rdd_name.first():- to view frist row

: rdd_name.take(n):-to view nrows

: rdd_collect() :- To see all data

Step13:-

We can save this result to a text file/csv file in our HDFS:-- first Create a function to join the data on basis of (,): Since we need csv output.

```
def  toCSVLine(data):

      return ','.join(str(d) for d in data)
```

# Take care of spacing while creating this function: Other- wise it will throw indentation error.

#Pass the output to the function ,We pass the master table to the above defined fuction to change it to a csv file.

```
lines =final_table.map(toCSVLine)
```

#Save the file to desired loaction

```
lines.saveAsTextFile('hdfs://adress)
```

To load the cluster attributes we can ..use :--(reading parquet format) # This will give centroids for each clsuters

Centriods:

```
from pyspark.sql import SQLContext

sqlContext = SQLContext(sc)

data = sqlContext.read.parquet("hdfs:///user/dms/myModel/data/") # this is the address of the mymodel path which i saves..

data.first()
```