**Spark Components**

* **Spark Streaming:** Real Time data processing. Log data can be analyzed in window of time.
* **Spark SQL:** SQL interface to Spark.
* **MLLib:** Library of ML and data mining tools.
* **GraphX:** Network and graph library on Spark.
* **Spark Core:** Core engine for Spark.

Scala is language of choice on Spark, next preferred language is Python and then Java.

**RDD’s**

Spark is built around RDD Resilient Distributed Dataset. **SparkConte**xt(sc) is Environment created by drive program or spark shell. sc creates RDD’s.

**Creating RDD’s:**

**nums** = parallelize ([1, 2, 3, 4])

**sc.textFile**(‘file:///c:/us/gos.txt’) or hdfs:/// or S3N:///

by loading from text file

**hiveCtx** = HiveContext(sc) rows = hiveCtx.sql(“SELECT name, age from usrs”)

Can also be created from JDBC, Cassandra, Hbase, Elasticsearch, Json, CSV, Sequence file etc.

**Transforming RDD’s**

Transformation creates new RDD from input RDD.

* **map:** Used in case of 1 to 1 relationship
* **flatmap:** Any relationship b/w in and out.
* **Filter:** To filter records from input RDD
* **Distinct:** Gives distinct unique values.
* **Sample:** Random Sample
* **union, intersection, subtract, Cartesian**

**Map example:**

rdd = sc.parallelize([1, 2, 3, 4])

squarRDD = rdd.map(lambda x: x\*x)

o/p => 1, 4 9, 16

Many RDD methods accepts function as a parameter. #Functional Programming.

**RDD Actions**

* **collect:** Reduces result and returns object.
* **count:** count of no. of rows in RDD.
* **countByValue:** no. of times a value occurred.
* **take:** to take a sample of a RDD.
* **top:** to take top few rows of a RDD.
* **Reduce:** combine all values etc.

In Spark driver program nothing happens till action is called. This is called **lazy evaluation**

from pyspark import SparkConf, SparkContext

def loadMovieNames():

movieNames = {}

with open("ml-100k/u.item") as f:

for line in f:

fields = line.split('|')

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

return movieNames

def parseInput(line):

fields = line.split()

return (int(fields[1]), (float(fields[2]), 1.0))

if \_\_name\_\_ == "\_\_main\_\_":

**conf** = SparkConf().setAppName("WorstMovies")

sc = SparkContext(conf = conf)

movieNames = loadMovieNames()

**lines** = sc.textFile("hdfs:///user/maria\_dev/ml-100k/u.data")

**movieRatings** = lines.map(parseInput)

**ratingTotalsAndCount** = movieRatings.reduceByKey(lambda movie1, movie2: ( movie1[0] + movie2[0], movie1[1] + movie2[1] ) )

**averageRatings** = ratingTotalsAndCount.mapValues(lambda totalAndCount : totalAndCount[0] / totalAndCount[1])

**sortedMovies** = averageRatings.sortBy(lambda x: x[1])

results = sortedMovies.take(10)

for result in results:

print(movieNames[result[0]], result[1])

**spark-submit** command is used to submit spark code. *spark-submit lowestRatedMoviesSpark.py*

**Spark SQL**

Data Frames in Spark are like tables, we can run SQL query over data frames. Data frames has schema so we can read and write it in structured data format. We can also communicate using Tableau, JDBC etc.

**Using SparkSQL in python**:

1. From pyspark.sql **import** *SQLContext, row.*
2. HiveContext = HiveContext(sc)
3. inputData = spark.read.json(datafile)
4. inputData.createOrReplaceTempView(“myStructuredStuff”) create table.
5. MyResultDataFrame = hiveContext.sql(“”” SELECT foo FROM bar ORDER BY foobar ”””)

**Other Data Frame Operations**

* *myDataframe.show()*
* *myDataframe.select(“someFieldName”)*
* *myDF.filter(myDF(“someFieldName”>200))*
* *myDF.groupBy(myDF(“someField”)).mean()*
* *myDataframe.rdd().map(mapperFunction)*

rdd() used to get underlying RDD from data frame.

**Datasets**: DataFrame is a dataset of row objects. Dataset includes typed information too.

**Thrift Server**

Spark SQL exposes a JDBC/ODBC connection using thrift server, if we build Spark with Hive support. We can start it by following command

*sbin/start-thriftserver.sh (port: 10000)*

and to connect to thrift server:

*bin/beeline –u jdbc:hive2://localhost:10000*

We can create new table or query existing cached tables using *hiveCtx.cacheTable(“tableName”)*

**UDF’s**

*from pyspark.sql.types import IntegerType*

*hiveCtx.registerFunction(“square”, lambda x: x\*x, IntegerType())*

*df = hiveCtx.sql(“SELECT square(col1) FORM tabl”)*

Sample Spark Code:

Refer attached python file for codes.

* LowestRatedMovieDataFrame.py
* LowestRatedMovieSpark.py

*export SPARK\_MAJOR\_VERSON=2*

*spark-submit LowestRatedMovieDataFrame.py*

Spark is moving away from the RDD syntax in favor of a simpler to understand DataFrame syntax.

**from** **pyspark.sql** **import** SparkSession

spark = SparkSession.builder.appName ("Operations").getOrCreate()

df = spark.read.csv('appl\_stock.csv', inferSchema=**True**,header=**True**)

df.printSchema()

*# Using SQL*

df.filter("Close<500").show()

*# Using SQL with .select()*

df.filter("Close<500").select('Open').show()

*# Using SQL with .select()*

df.filter("Close<500").select(['Open','Close']).show()

df.filter(df["Close"] < 200).show()

*# Make sure to add in the parenthesis separating the statements!*

df.filter( (df["Close"] < 200) & (df['Open'] > 200) ).show()

*# Collecting results as Python objects*

df.filter(df["Low"] == 197.16).collect()

result = df.filter(df["Low"] == 197.16).collect()

type(result[0])

pyspark.sql.types.Row

#rows can be turned into dictionary

row.asDict()

**for** item **in** result[0]:

print(item)