**Spark Components** [**http://locahost:4040**](http://locahost:4040)

* **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)

**Spark Scheduling Mode**

spark.scheduler.modeSpark Property

FAIR

FIFO

val conf = new SparkConf().setMaster(...).setAppName(...)  
conf.set("spark.scheduler.mode", "FAIR")

val sc = new SparkContext(conf)

**Speculative execution in Spark**

spark.speculation: TRUE for Speculative Execution.

spark.speculation.interval(ms): speculate interval.

spark.speculation.multiplier: This entry defines how many times slower a task must be to be considered for speculation. Default value is 1.5. It means that tasks running 1.5 times slower than the median will be taken into account for speculation.

spark.speculation.quantile: it specifies how many tasks must finish before enabling the speculation **for given stage. It's expressed by the fraction and by default the value is 0.75 (75%).**

**SparkContext**

**SparkModes:** Batchmode, Interactive mode

**Persistence:**

By default, Spark loads an RDD whenever it required. It drops it once the action is over

• It will load and re-compute the ROD chain, each time a different operation is performed

• Persistence allows the intermediate RDD to be persisted so it need not have to be recomputed.

• persist() can persist the RDD in memory, disk, shared or in other third party sinks

• cache() provides the default persist() — in memory

**Partitioning:** By default, all RDDs are partitioned

• spark.default.parallelism parameter. Default is the total no. of cores available across the entire cluster. Can be specified during RDD creation explicitly. Derived RDD take same number as source.

scala> val textFile = sc.textFile("file:///home/kiran/partitioning\_wc")

textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[29] at textFile at <console>:16

scala> val counts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).partitionBy(new HashPartitioner(10))

counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[32] at partitionBy at <console>:18

scala> counts.reduceByKey(\_+\_).saveAsTextFile("/home/kiran/partition\_spark/hash")

scala>

**Broadcast Variables:** A read-only variable that is shared by all nodes. Used for lookup tables or similar functions. Spark optimizes distribution and storage for better performance.

**Accumulators:** A shared variable across nodes that can be updated by each node. Helps compute items not done through reduce operations. Spark optimizes distribution and takes care of race conditions

#initiaIize accumulator

sedancount = sc. accumulator( )

hatchbackcount =sc.accumulat r(Ø)

#Set Broadcast variable

sedanrext=sc. broadcast( " sedan")

hatchbackText=sc. broadcast( " hatchback" )

**KeyValueRDD’s:**

val pairs = lines.map(x => (x.split(" ")(0), x))

pairs.filter{case (key, value) => value.length < 20}

rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x.\_1 + y.\_1, x.\_2 + y.\_2))

**Workcount example:**

val input = sc.textFile("s3://...")

val words = input.flatMap(x => x.split(" "))

val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)

**SparkSubmit and its Parameters**

--master yarn-client --driver-memory 5g --num-executors 40 --executor-memory 1g --executor-cores 2

**Scala and pySpark:**

Launching Spark Shell:

1. $ SPARK\_HOME/bin/spark-shell \\Launches Spark Scala shell
2. $ SPARK\_HOME/bin/pyspark \\Launches Spark Python shell

Spark-submit:

1. $ mvn **package**
2. $ spark-submit --**class** com.cloudera.sparkwordcount.SparkWordCount \ --master **local** --deploy-mode client --executor-memory 1g \ --name wordcount --conf "http://spark.app.id=wordcount" \ sparkwordcount-1.0-SNAPSHOT-jar-**with**-dependencies.jar <http://hdfs://namenode_host:8020/path/to/inputfile.txt>
3. $ spark-submit --master yarn --deploy-mode client --executor-memory 1g \

--name wordcount --conf "http://spark.app.id=wordcount" http://wordcount.py http://hdfs://namenode\_host:8020/path/to/inputfile.txt 2

**SparkStraming**

Used for Real Time Analytics like, Fraud detection, Spam Filtering, Network Intrusion Detection, Click stream analytics, Stock Market Analytics etc.

**Sources**: Flat Files (Logs etc.), TCP/IP, Apache Flume, Apache Kafka, Amazon Kinesis, Twitter, Facebook and other social Media.

**DStream:**

• A Streaming context is created from the Spark context to enable streaming.

• Streaming creates a DStream (Discretized Stream) on which processing occurs.

• A micro-batch window is setup for the DStream

• Data is received, accumulated as a micro-batch and processed as a micro-batch.

• Each micro-batch is an RDD so regular RDD operations can be applied on the DStream RDD.

**DStream processing**

• Spark collects incoming data for each interval (micro-batch)

• Data is collected as an RDD for that interval.

• It then calls all transformations and operations that applies for that DStream or derived DStreams

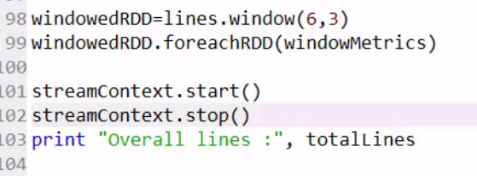
• Global variables can be used to track data across DStreams

• Windowing functions are available for computing across multiple DStreams.

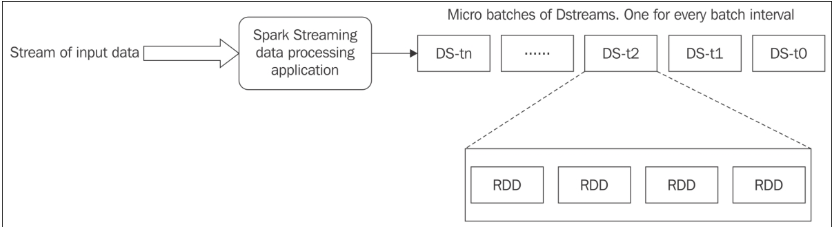
• Window size multiple of interval

• Sliding interval multiple of interval

**Windowing:**



**Microbatch:**



Every Spark Streaming data processing application will be running continuously till it is terminated. This application will be constantly listening to the data source to receive the incoming stream of data. The Spark Streaming data processing application would have a configured batch interval. At the end of every batch interval, it will produce a data abstraction named Discretized Stream (DStream) which works very similar to Spark's RDD. Just like RDD, a DStream supports an equivalents method for the commonly used Spark transformations and Spark actions.

**from** **pyspark** **import** SparkContext

**from** **pyspark.streaming** **import** StreamingContext

import org.apache.spark.streaming.StreamingContext

import org.apache.spark.streaming.dstream.InputDStream

val ssc = new StreamingContext(spark.sparkContext, Minutes(2))

val **inputStream** = ssc.queueStream(dvc)

val **stream** = TwitterUtils.createStream(ssc, None, Seq("challenge"))

**import** **org.apache.spark.\_**  
**import** **org.apache.spark.streaming.\_**  
**import** **org.apache.spark.streaming.StreamingContext.\_** *// not necessary since Spark 1.3*

*// Create a local StreamingContext with two working thread and batch interval of 1 second.*

*// The master requires 2 cores to prevent from a starvation scenario.*  
**val** conf **=** **new** **SparkConf**().setMaster("local[2]").setAppName("NetworkWordCount")

**val** ssc **=** **new** **StreamingContext**(conf, **Seconds**(1))

*// Create a DStream that will connect to hostname:port, like localhost:9999*

**val** lines **=** ssc.socketTextStream("localhost", 9999)

*// Split each line into words*

**val** words **=** lines.flatMap(**\_**.split(" "))

**SparkML and SparkSQL:**

**from** **pyspark.sql** **import** SQLContext

**from** **pyspark.sql.functions** **import** desc

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

spark = SparkSession.builder.appName('hack\_find').getOrCreate()

**from** **pyspark.ml.clustering** **import** KMeans

**from** **pyspark.ml.linalg** **import** Vectors

**from** **pyspark.ml.feature** **import** VectorAssembler

**from** **pyspark.ml.feature** **import** StandardScaler

scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures", withStd=**True**, withMean=**False**)

*# Compute summary statistics by fitting the StandardScaler*

scalerModel = scaler.fit(final\_data)

cluster\_final\_data = scalerModel.transform(final\_data)

**Writing SQL query in Spark:**

**import** *org.apache.spark.sql.catalyst.encoders.ExpressionEncoder*

*import org.apache.spark.sql.Encoder*

*import spark.implicits.\_*

*val employeeDF = spark.sparkContext.textFile("examples/src/main/resources/employee.txt").map(\_.split(",")).map(attributes =&amp;amp;amp;amp;gt; Employee(attributes(0), attributes(1).trim.toInt)).toDF()*

*employeeDF.createOrReplaceTempView("employee")*

*val youngstersDF = spark.sql("SELECT name, age FROM employee WHERE age BETWEEN 18 AND 30")*

*youngstersDF.map(youngster =&amp;amp;amp;amp;gt; "Name: " + youngster(0)).show()*