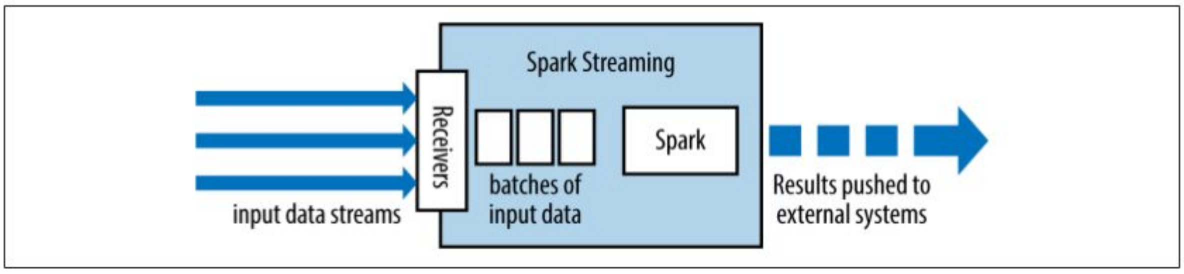
MODULE 14

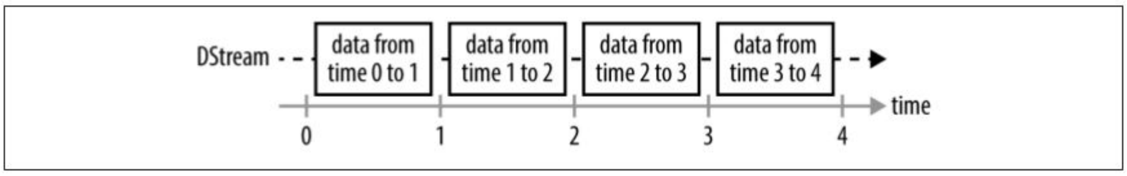
CREATE AN SPARK STREAMING APP

1. Architecture and Abstraction

Spark Streaming uses a “micro-batch” architecture, where the streaming computa-
tion is treated as a continuous series of batch computations on small batches of data.
Spark Streaming receives data from various input sources and groups it into small
batches. New batches are created at regular time intervals. At the beginning of each
time interval a new batch is created, and any data that arrives during that interval
gets
added to that batch. At the end of the time interval the batch is done growing. The
size of the time intervals is determined by a parameter called the batch interval. The
batch interval is typically between 500 milliseconds and several seconds, as config-
ured by the application developer. Each input batch forms an RDD, and is processed
using Spark jobs to create other RDDs. The processed results can then be pushed
out
to external systems in batches



The programming abstraction in Spark Streaming is a discretized stream or a
DStream, which is a sequence of RDDs, where each RDD has one time slice of the
data in the stream



You can create DStreams either from external input sources, or by applying transformations to other DStreams. In our simple example, we created a DStream from data received through a socket, and then applied a filter() transformation to it

Transformations

Transformations on DStreams can be grouped into either stateless or stateful:

- In stateless transformations the processing of each batch does not depend on the data of its previous batches. They include the common RDD transformations
- Stateful transformations, in contrast, use data or intermediate results from previous batches to compute the results of the current batch. They include transformations based on sliding windows and on tracking state across time

Output Operations

Output operations specify what needs to be done with the final transformed data in a stream (e.g., pushing it to an external database or printing it to the screen).

2. Creating a WordCount Application

We will receive a stream of newline-delimited lines of text from a server running at port 7777, filter only the lines that contain the word error, and print them

- Move to the SPARK_HOME folder:

```
$ cd $SPARK_HOME
```

- Create a directory to save the source code

```
$ mkdir -p examples/socket-stream/src/main/scala
```

- Create a file name SocketStream.scala in $SPARK_HOME/examples/socket-stream/src/main/scala folder:

```
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.Duration
import org.apache.spark.streaming.Seconds

object SocketStream {
    def main(args: Array[String]) {
      val conf = new SparkConf().setAppName("Socket-Stream")
      // Create a StreamingContext with a 1-second batch size from
a SparkConf
      val ssc = new StreamingContext(conf, Seconds(1))
      // Create a DStream using data received after connecting to
port 7777 on the
      // local machine
      val lines = ssc.socketTextStream("localhost", 7777)
      // Filter our DStream for lines with "error"
      val errorLines = lines.filter(_.contains("error"))
      // Print out the lines with errors
      errorLines.print()
      // Start our streaming context and wait for it to "finish"
      ssc.start()
      // Wait for the job to finish
      ssc.awaitTermination()
    }
}
```

- Create file build.sbt in $SPARK_HOME/examples/wordcount-app

```
name := "socket-stream"

version := "0.0.1"

scalaVersion := "2.11.12"
// additional libraries
```

```
libraryDependencies ++= Seq(
"org.apache.spark" %% "spark-core" % "2.4.1" % "provided",
"org.apache.spark" %% "spark-streaming" % "2.4.1"
)
```

- Build application :

```
$ sbt clean package
```

- Submit and run in Spark :

```
$ $SPARK_HOME/bin/spark-submit --class SocketStream
target/scala-2.11/socket-stream_2.11-0.0.1.jar
```

- Open another terminal and send text from port 7777:
  nc -l localhost -p 7777

3. Create a Log Analyzer

- Move to the SPARK_HOME folder:

```
$ cd $SPARK_HOME
```

- Create a directory to save the source code

```
$ mkdir -p examples/logs-analyzer/src/main/scala
```

- Download logs sample:
  https://drive.google.com/file/d/184RPO2pxbyDXUXIb3__nWxnI5iz-WNOC/view?usp=sharing

- Create a file name ApacheAccessLog.scala in
  $SPARK_HOME/examples/logs-analyzer/src/main/scala folder:

```
/** An entry of Apache access log. */
case class ApacheAccessLog(ipAddress: String,
  clientIdentd: String,
  userId: String,
  dateTime: String,
  method: String,
  endpoint: String,
  protocol: String,
  responseCode: Int,
  contentSize: Long) {
}

object ApacheAccessLog {
  val PATTERN = """^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\]
"(\S+) (\S+) (\S+)" (\d{3}) (\d+)""".r

  /**
    * Parse log entry from a string.
    *
    * @param log A string, typically a line from a log file
    * @return An entry of Apache access log
    * @throws RuntimeException Unable to parse the string
    */
  def parseLogLine(log: String): ApacheAccessLog = {
    log match {
      case PATTERN(ipAddress, clientIdentd, userId, dateTime,
method, endpoint, protocol, responseCode, contentSize)
      => ApacheAccessLog(ipAddress, clientIdentd, userId,
dateTime, method, endpoint, protocol, responseCode.toInt,
        contentSize.toLong)
      case _ => throw new RuntimeException(s"""Cannot parse log
line: $log""")
    }
  }
}
```

- Create a file name LogAnalyzerStreaming.scala in $SPARK_HOME/examples/logs-analyzer/src/main/scala folder:

```
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.{Seconds, StreamingContext}


object LogAnalyzerStreaming {
 def main(args: Array[String]) {
  val WINDOW_LENGTH = Seconds(30)
  val SLIDE_INTERVAL = Seconds(10)

  val sparkConf = new SparkConf().setAppName("Log Analyzer
Streaming in Scala")
```

```
  val streamingContext = new StreamingContext(sparkConf,
SLIDE_INTERVAL)

  val logLinesDStream: DStream[String] =
streamingContext.socketTextStream("localhost", 9999)

  val accessLogsDStream: DStream[ApacheAccessLog] =
logLinesDStream.map(ApacheAccessLog.parseLogLine).cache()
  val windowDStream: DStream[ApacheAccessLog] =
accessLogsDStream.window(WINDOW_LENGTH, SLIDE_INTERVAL)

  windowDStream.foreachRDD(accessLogs => {
    if (accessLogs.count() == 0) {
      println("No access logs received in this time interval")
    } else {
      // Calculate statistics based on the content size.
      val contentSizes: RDD[Long] =
accessLogs.map(_.contentSize).cache()
      println("Content Size Avg: %s, Min: %s, Max: %s".format(
        contentSizes.reduce(_ + _) / contentSizes.count,
        contentSizes.min,
        contentSizes.max
      ))

      // Compute Response Code to Count.
      val responseCodeToCount: Array[(Int, Long)] = accessLogs
        .map(_.responseCode -> 1L)
        .reduceByKey(_ + _)
        .take(100)
      println( s"""Response code counts:
${responseCodeToCount.mkString("[", ",", "]")}""")

      // Any IPAddress that has accessed the server more than 10
times.
      val ipAddresses: Array[String] = accessLogs
        .map(_.ipAddress -> 1L)
        .reduceByKey(_ + _)
        .filter(_._2 > 10)
        .map(_._1)
        .take(100)
      println( s"""IPAddresses > 10 times:
${ipAddresses.mkString("[", ",", "]")}""")

      // Top Endpoints.
      val topEndpoints: Array[(String, Long)] = accessLogs
```

```
         .map(_.endpoint -> 1L)
         .reduceByKey(_ + _)
         .top(10)(Ordering.by[(String, Long), Long](_._2))
       println( s"""Top Endpoints: ${topEndpoints.mkString("[",
",", "]")}""")
     }
   })

   // Start the streaming server.
   streamingContext.start() // Start the computation
   streamingContext.awaitTermination() // Wait for the computation
to terminate
}
}
```

- Create file build.sbt in $SPARK_HOME/examples/logs-analyzer

```
name := "log-analyzer"

version := "0.0.1"

scalaVersion := "2.11.12"
// additional libraries

libraryDependencies ++= Seq(
"org.apache.spark" %% "spark-core" % "2.4.1" % "provided",
"org.apache.spark" %% "spark-streaming" % "2.4.1"
)
```

- Create the shell script named "*stream.sh*" that emulates network stream by periodically sending portions of the sample log file to a network socket:

```
#!/bin/sh

set -o nounset
set -o errexit

test $# -eq 1 || ( echo "Incorrect number of arguments" ; exit 1 )

file="$1"
```

```
network_port=9999
lines_in_batch=100
interval_sec=10

n_lines=$(cat $file | wc -l)
cursor=1
while test $cursor -le $n_lines
do
      tail -n +$cursor $file | head -$lines_in_batch | nc -l
$network_port
      cursor=$(($cursor + $lines_in_batch))
      sleep $interval_sec
done
```

- Build application :

```
$ sbt clean package
```

- Submit and run in Spark :

```
$ $SPARK_HOME/bin/spark-submit --class "LogAnalyzerStreaming"
target/scala-2.11/log-analyzer_2.11-0.0.1.jar
```

- Open another terminal and send text from port 9999:

```
./stream.sh log.txt
```