TwitterUtils.createStream(...)
  .filter(_.getText.contains("Spark"))
  .countByWindow(Seconds(5))
Complex algorithms can be expressed using:
- Spark transformations: map(), reduce(), join(), etc
- MLlib + GraphX
- SQL

- Scalable
- High-throughput
- Fault-tolerant
Batch

Realtime

hadoop Map Reduce

Spark STREAMING

One unified API

APACHE STORM™
Distributed • Resilient • Real-time
Tathagata Das (TD)
- Lead developer of Spark Streaming + Committer on Apache Spark core
- Helped re-write Spark Core internals in 2012 to make it 10x faster to support Streaming use cases
- On leave from UC Berkeley PhD program
- Ex: Intern @ Amazon, Intern @ Conviva, Research Assistant @ Microsoft Research India

- Scales to 100s of nodes
- Batch sizes as small at half a second
- Processing latency as low as 1 second
- Exactly-once semantics no matter what fails
USE CASES (live statistics)

Page views → Kafka for buffering → Spark for processing
USE CASES

Smart meter readings

Join 2 live data sources

Live weather data

Spark

STREAMING

(Anomaly Detection)
Input data stream

Spark STREAMING

Batches every X seconds

Spark CORE

Batches of processed data
Input data streams

Batches of processed data

Batches every X seconds
DSTREAM
(Discretized Stream)

Batch interval = 5 seconds

Input DStream

Block #1  Block #2  Block #3

T = 5

One RDD is created every 5 seconds

Block #1  Block #2  Block #3

T = +5

RDD @ T=5

RDD @ T=+5

1

2
TRANSFORMING DSTREAMS

Block #1 ➔ Block #2 ➔ Block #3

Materialize!

Part. #1 ➔ Part. #2 ➔ Part. #3

flatMap()

Part. #1 ➔ Part. #2 ➔ Part. #3

linesDStream ➔ linesDStream ➔ wordsDStream

5 sec ➔ 10 sec ➔ 15 sec
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

// 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 linesStream = ssc.socketTextStream("localhost", 7777)

// Filter our DStream for lines with "error"
val errorLinesStream = linesStream.filter(_.contains("error"))

// Print out the lines with errors
errorLinesStream.print()

// Start our streaming context and wait for it to "finish"
ssc.start()

// Wait for the job to finish
ssc.awaitTermination()
$ nc localhost 7777

all is good
there was an error
good good

error 4 happened
all good now

$ spark-submit --class com.examples.Scala.StreamingLogInput \ $ASSEMBLY_JAR local[4]

... 

--------------------------
--------------------------
there was an error
--------------------------
--------------------------
error 4 happened
## Spark Stages

Total Duration: 6 s  
Scheduling Mode: FIFO  
Active Stages: 1  
Completed Stages: 4  
Failed Stages: 0

### Active Stages (1)

<table>
<thead>
<tr>
<th>Stage Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Tasks: Succeeded/Total</th>
<th>Input</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>start at StreamingLogInput.scala:21</td>
<td>2014/10/21 12:44:25</td>
<td>5 s</td>
<td>0/1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Completed Stages (4)

<table>
<thead>
<tr>
<th>Stage Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Tasks: Succeeded/Total</th>
<th>Input</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>getCallSite at DStream.scala:294</td>
<td>2014/10/21 12:44:31</td>
<td>25 ms</td>
<td>1/1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>getCallSite at DStream.scala:294</td>
<td>2014/10/21 12:44:31</td>
<td>13 ms</td>
<td>1/1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>getCallSite at DStream.scala:294</td>
<td>2014/10/21 12:44:29</td>
<td>27 ms</td>
<td>1/1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>getCallSite at DStream.scala:294</td>
<td>2014/10/21 12:44:27</td>
<td>30 ms</td>
<td>1/1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Failed Stages (0)
Streaming

Time since start: 27 minutes 20 seconds
Network receivers: 1
Batch interval: 1 second
Processed batches: 1641
Waiting batches: 0

Statistics over last 100 processed batches

Receiver Statistics

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TwitterReceiver-0</td>
<td>ACTIVE</td>
<td>localhost</td>
<td>39</td>
<td>0</td>
<td>61</td>
<td>151</td>
<td>-</td>
</tr>
</tbody>
</table>

Batch Processing Statistics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Last batch</th>
<th>Minimum</th>
<th>25th percentile</th>
<th>Median</th>
<th>75th percentile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Time</td>
<td>31 ms</td>
<td>5 ms</td>
<td>39 ms</td>
<td>56 ms</td>
<td>457 ms</td>
<td>2 seconds 289 ms</td>
</tr>
<tr>
<td>Scheduling Delay</td>
<td>0 ms</td>
<td>0 ms</td>
<td>0 ms</td>
<td>0 ms</td>
<td>1 ms</td>
<td>803 ms</td>
</tr>
<tr>
<td>Total Delay</td>
<td>31 ms</td>
<td>31 ms</td>
<td>40 ms</td>
<td>57 ms</td>
<td>499 ms</td>
<td>2 seconds 289 ms</td>
</tr>
</tbody>
</table>
Spark 1.4.0

New UI for Streaming

Streaming Statistics

Running batches of 1 second for 1 minute 36 seconds since 2015/06/08 19:41:44 (96 completed batches, 4176 records)

- Input Rate
  Avg: 43.50 events/sec

- Scheduling Delay
  Avg: 0 ms

- Processing Time
  Avg: 113 ms

- Total Delay
  Avg: 114 ms

Active Batches (0)

Completed Batches (last 96 out of 96)

<table>
<thead>
<tr>
<th>Batch Time</th>
<th>Input Size</th>
<th>Scheduling Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015/06/08 19:42:00</td>
<td>19 events</td>
<td>0 ms</td>
</tr>
<tr>
<td>2015/06/08 19:43:18</td>
<td>44 events</td>
<td>0 ms</td>
</tr>
<tr>
<td>2015/06/08 19:43:17</td>
<td>59 events</td>
<td>0 ms</td>
</tr>
<tr>
<td>2015/06/08 19:43:16</td>
<td>21 events</td>
<td>0 ms</td>
</tr>
</tbody>
</table>
Batch interval = 600 ms

spark.streaming.blockInterval = 200 ms
Batch interval = 600 ms

200 ms later
Batch interval = 600 ms

200 ms later
Batch interval = 600 ms
Batch interval = 600 ms
Batch interval = 600 ms

Driver

Internal Threads

W

Ex

block, P3
block, P3
block, P1
block, P2

Internal Threads

W

Ex

block, P2
block, P2
block, P1
block, P1

Internal Threads

W

Ex

block, P1
block, P2
block, P3
block, P3

Internal Threads

OS Disk

SSD

SSD

OS Disk

SSD

SSD

OS Disk

SSD

SSD
Batch interval = 600 ms

Materialize!
Batch interval = 600 ms

Union!
Sources directly available in StreamingContext API

BASIC
- File systems
- Socket Connections
- Akka Actors

ADVANCED
- Kafka
- Flume
- Twitter

CUSTOM
- Anywhere

Requires linking against extra dependencies

Requires implementing user-defined receiver

val logData = ssc.textFileStream(logDirectory)
Spark Streaming + Flume Integration Guide

Apache Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. Here we explain how to configure Flume and Spark Streaming to receive data from Flume. There are two approaches to this.

Approach 1: Flume-style Push-based Approach

Flume is designed to push data between Flume agents. In this approach, Spark Streaming essentially sets up a receiver that acts as an Avro agent for Flume, to which Flume can push the data. Here are the configuration steps.

General Requirements

Choose a machine in your cluster such that

- When your Flume + Spark Streaming application is launched, one of the Spark workers must run on that machine.
- Flume can be configured to push data to a port on that machine.

Due to the push model, the streaming application needs to be up, with the receiver scheduled and listening on the chosen port, for Flume to be able to push data.

Configuring Flume

Configure Flume agent to send data to an Avro sink by having the following in the configuration file.
Spark Streaming + Kafka Integration Guide

Apache Kafka is publish-subscribe messaging rethought as a distributed, partitioned, replicated commit log service. Here we explain how to configure Spark Streaming to receive data from Kafka.

1. **Linking:** In your SBT/Maven project definition, link your streaming application against the following artifact (see Linking section in the main programming guide for further information).

   ```
   groupId = org.apache.spark
   artifactId = spark-streaming-kafka_2.10
   version = 1.2.0
   ```

2. **Programming:** In the streaming application code, import KafkaUtils and create input DStream as follows.

   ```scala
   import org.apache.spark.streaming.kafka._

   val kafkaStream = KafkaUtils.createStream(...)
Spark Streaming Custom Receivers

Spark Streaming can receive streaming data from any arbitrary data source beyond the ones for which it has in-built support (that is, beyond Flume, Kafka, Kinesis, files, sockets, etc.). This requires the developer to implement a receiver that is customized for receiving data from the concerned data source. This guide walks through the process of implementing a custom receiver and using it in a Spark Streaming application. Note that custom receivers can be implemented in Scala or Java.

Implementing a Custom Receiver

This starts with implementing a Receiver (Scala doc, Java doc). A custom receiver must extend this abstract class by implementing two methods:

- `onstart()`: Things to do to start receiving data.
- `onstop()`: Things to do to stop receiving data.

Both `onstart()` and `onstop()` must not block indefinitely. Typically, `onstart()` would start the threads that responsible for receiving the data and `onstop()` would ensure that the receiving by those threads are stopped. The receiving threads can also use `isStopped()`, a Receiver method, to check whether they should stop receiving data.

Once the data is received, that data can be stored inside Spark by calling `store(data)`, which is a method provided by the Receiver class. There are number of flavours of `store()` which allow you store the received data record at...
OUTPUT OPERATIONS ON DSTREAMS

print()

saveAsTextFile(prefix, [suffix])

saveAsObjectFiles(prefix, [suffix])

saveAsHadoopFiles(prefix, [suffix])