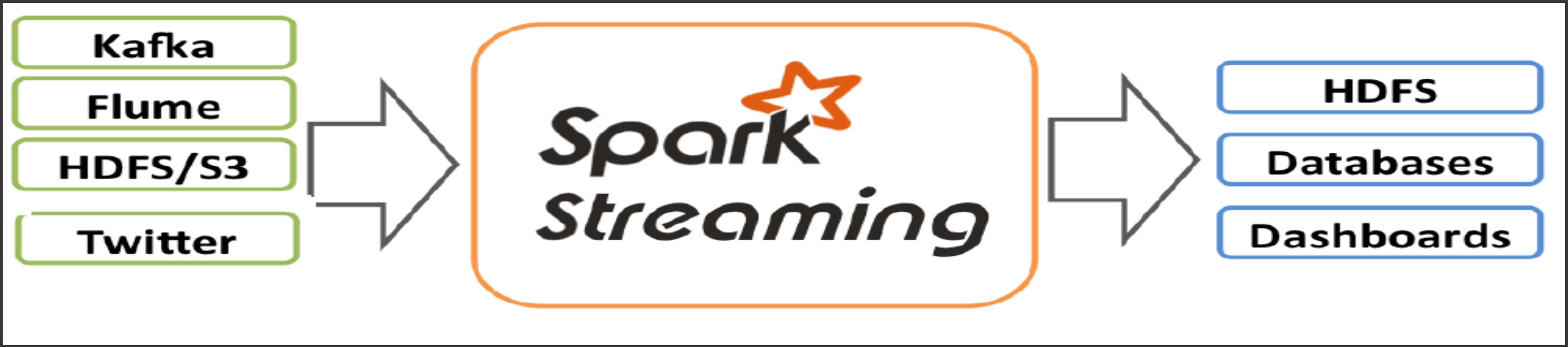
SPARK Streaming

If you don't drive your business,
you will be driven out of business

What is Spark Streaming?

- ❑ Large-scale near-real-time stream processing engine
- ❑ Extends Spark for doing big data stream processing
- ❑ Spark Streaming is scalable, high-throughput, fault-tolerant stream processing of live data streams.

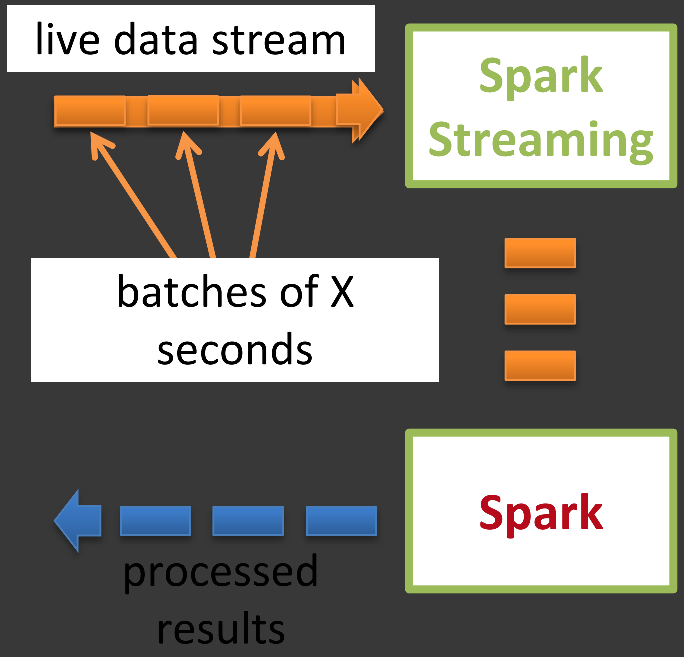
- ❑ Scales to 100s of nodes and achieves second scale latencies
- ❑ Integrates with Spark's batch and interactive processing
- ❑ Provides a simple batch-like API for implementing complex algorithms
- ❑ Can absorb live data streams from Kafka, Flume, HDFS



Discretized Stream Processing

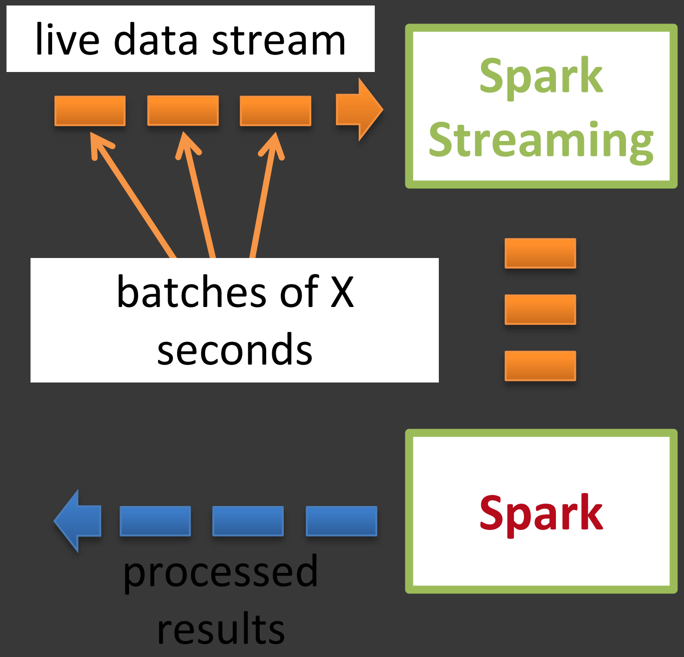
Run a streaming computation as a **series of very small, deterministic batch jobs**

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Discretized Stream Processing

- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



StreamingContext

- Main entry point for Spark Streaming functionality.
- Provides methods used to create DStreams from various input sources.
- It can be created by providing
  - A Spark master URL and an appName,
  - and from a org.apache.spark.SparkConf configuration,
  - and from an existing org.apache.spark.SparkContext. The associated SparkContext can be accessed using context.sparkContext.
- After creating and transforming DStreams, the streaming computation can be started and stopped using **context.start()** and **context.stop()**, respectively. **context.awaitTermination()** allows the current thread to wait for the termination of the context by stop() or by an exception.

Streaming Context

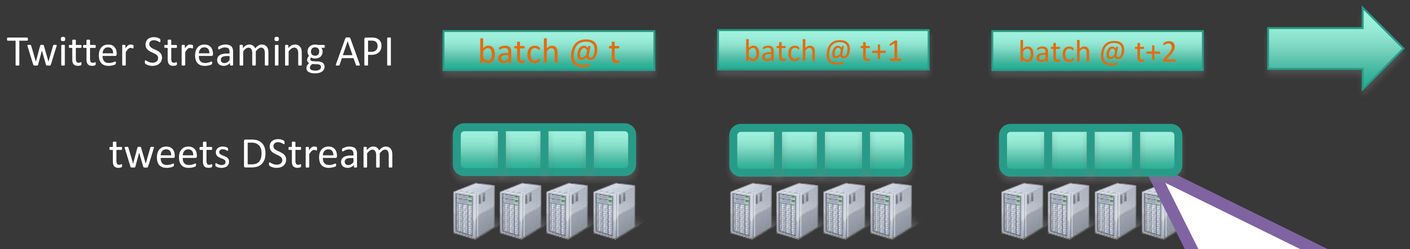
After a context is defined, we have to do the following.

- ✓ Define the input sources by creating input DStreams.
- ✓ Define the streaming computations by applying transformation and output operations to DStreams.
- ✓ Start receiving data and processing it using streamingContext.start().
- ✓ Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().
- ✓ The processing can be manually stopped using streamingContext.stop().
- ✓ Once a context has been started, no new streaming computations can be set up or added to it.
- ✓ Once a context has been stopped, it cannot be restarted.
- ✓ Only one StreamingContext can be active in a JVM at the same time.
- ✓ stop() on StreamingContext also stops the SparkContext. To stop only the StreamingContext, set the optional parameter of stop() called stopSparkContext to false.
- ✓ A SparkContext can be re-used to create multiple StreamingContexts, as long as the previous StreamingContext is stopped (without stopping the SparkContext) before the next StreamingContext is created.

Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data



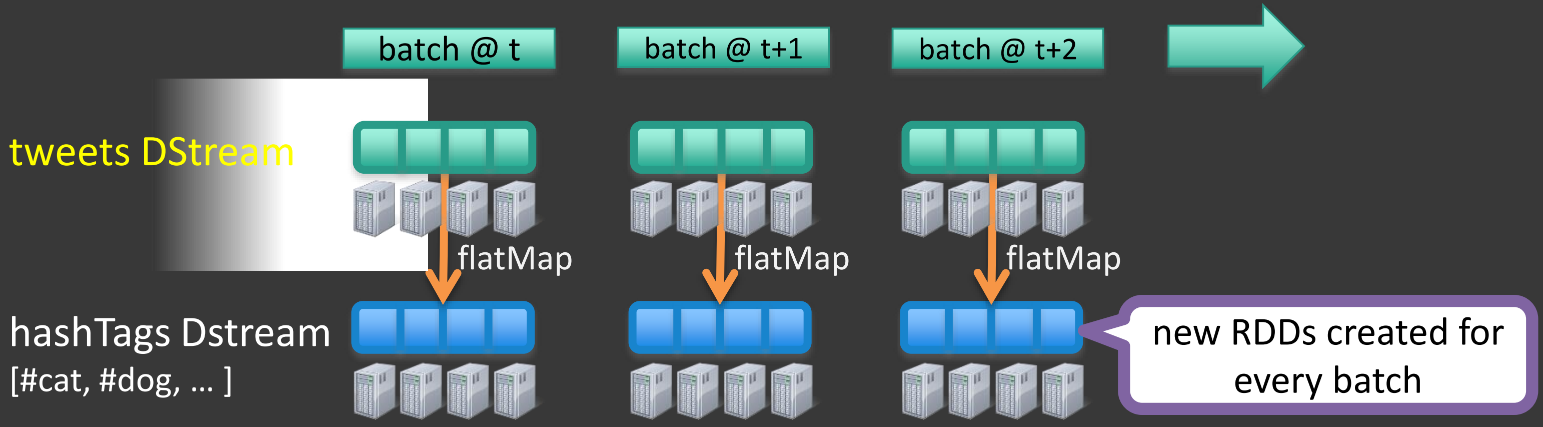
stored in memory as an RDD (immutable, distributed)

Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```

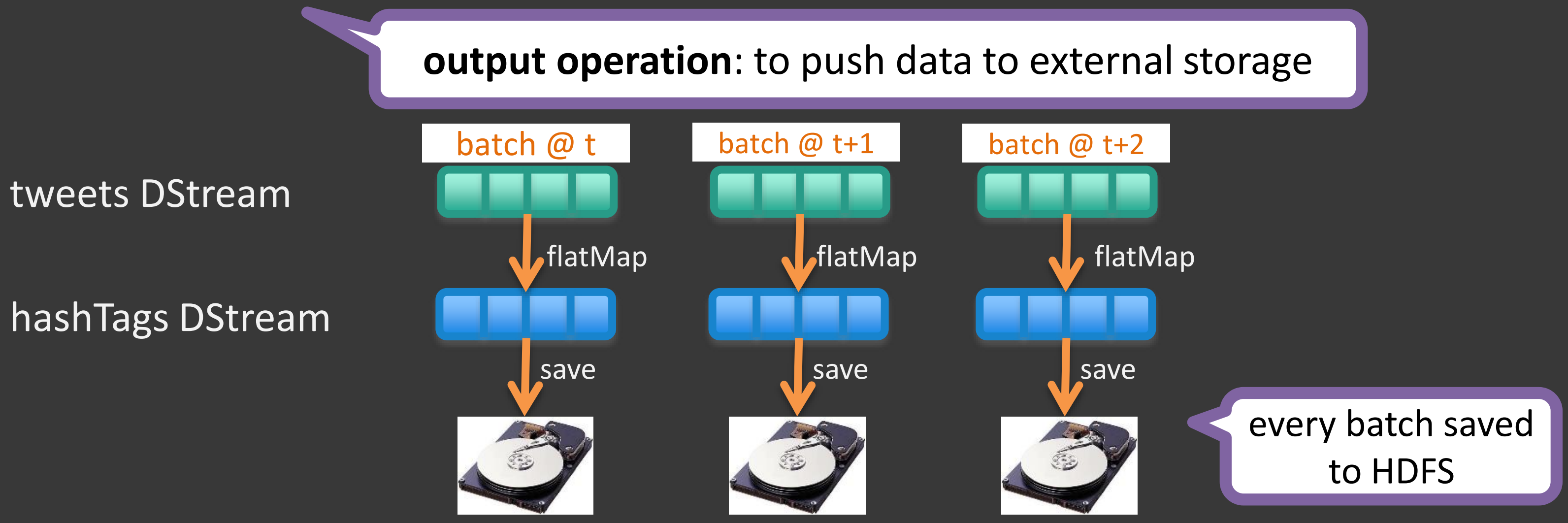
new DStream

transformation: modify data in one Dstream to create another DStream



Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```



Java Example

Scala

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

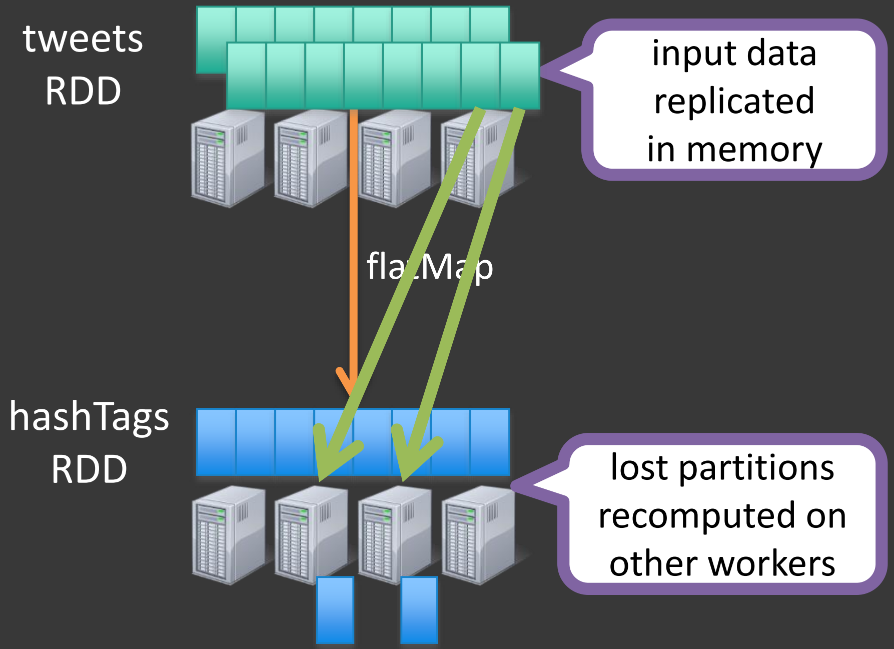
Java

```
JavaDStream<Status> tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
JavaDstream<String> hashTags = tweets.flatMap(new Function<...> {  })
hashTags.saveAsHadoopFiles("hdfs://...")
```

Function object to define the transformation

Fault-tolerance

- RDDs remember the sequence of operations that created it from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data


tweets RDD
input data replicated in memory
flatMap
hashTags RDD
lost partitions recomputed on other workers

Key concepts

- **DStream** – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, Akka Actor, TCP sockets
- **Transformations** – modify data from one DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, …
  - Stateful operations – window, countByValueAndWindow, …
- **Output Operations – send data to external entity**
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results

Basic Streaming Example

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.{Seconds, StreamingContext}

//create StreamingContext using sparkcontext object as sc.
val ssc = new StreamingContext(sc, Seconds(10))
val lines = ssc.textFileStream("hdfs file name")
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)

//Print the first ten elements of each RDD generated in this
DStream to the console
wordCounts.print()
ssc.start()
ssc.awaitTermination()
```

Linking

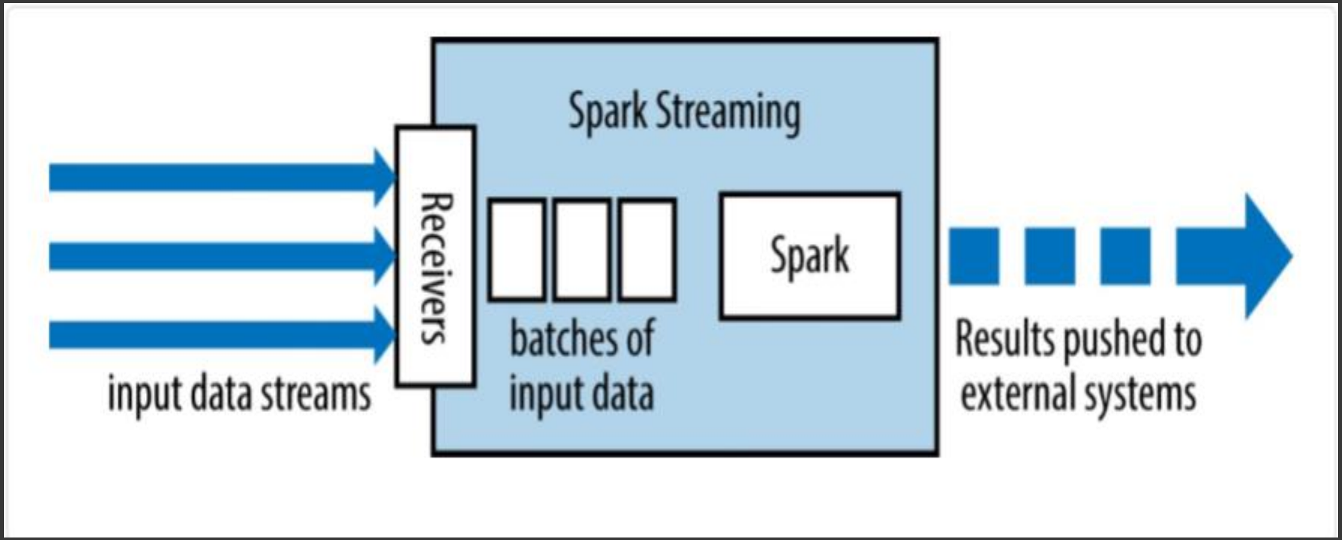
- ✓ Similar to Spark, Spark Streaming is available through Maven Central.
- ✓ To write our own Spark Streaming program, we will have to add the following dependency to our SBT or Maven project.

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.11</artifactId>
    <version>1.6.0</version>
</dependency>
```

Source	**Artifact**
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Twitter	spark-streaming-twitter_2.10

Architecture and Abstraction

Spark Streaming uses a “micro-batch” architecture, where the streaming computation is treated as a continuous series of batch computations on small batches of data.



Error Lines Filter

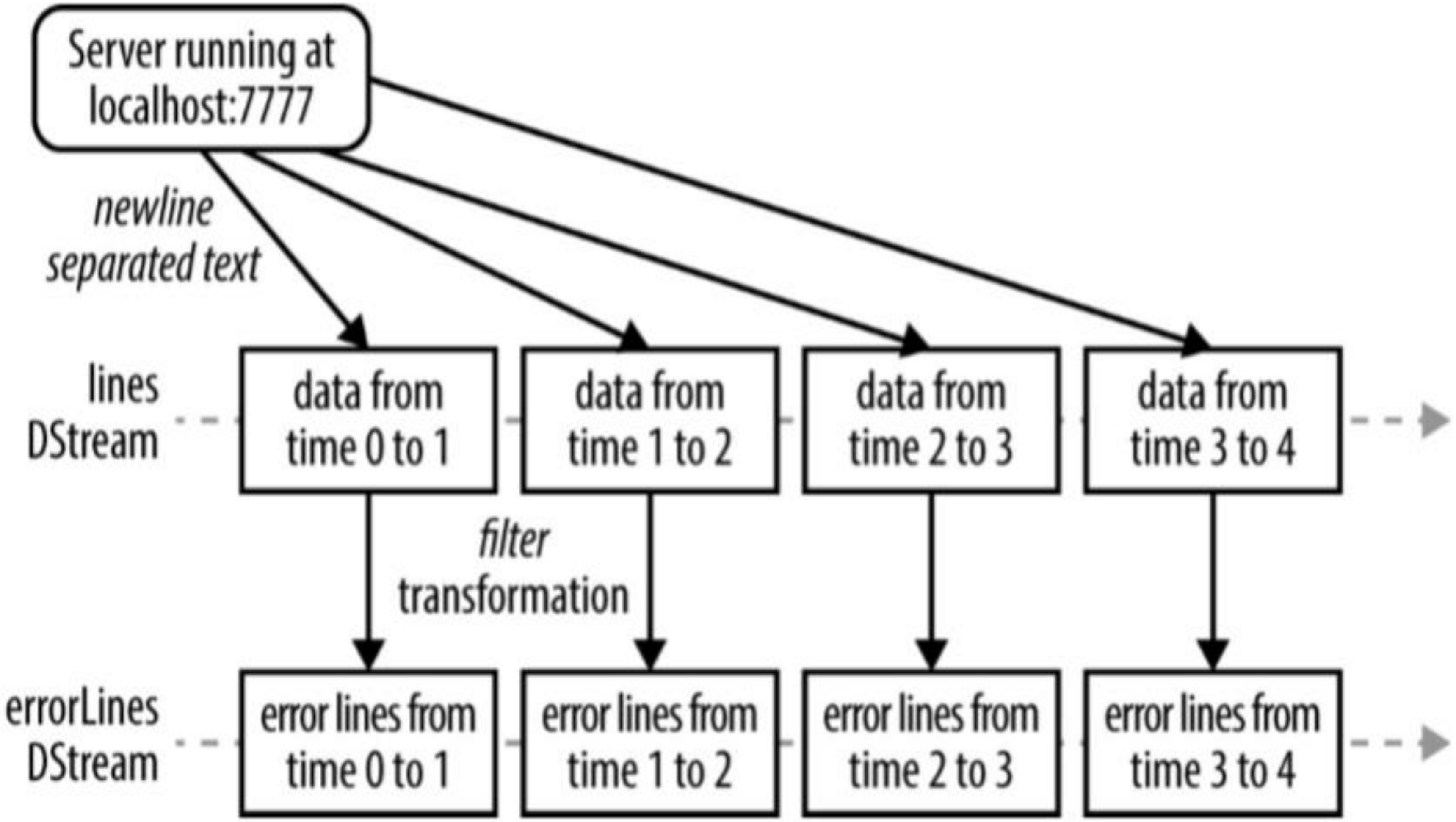
```
val conf = new SparkConf()
     .setAppName("The swankiest Spark app ever")
        .setMaster("local[2]")

val sc = new SparkContext(conf)

val ssc = new StreamingContext(sc, Seconds(1))
// Create a DStream using data received at connecting to port 7777
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()
    ssc.start()
    ssc.awaitTermination()
```

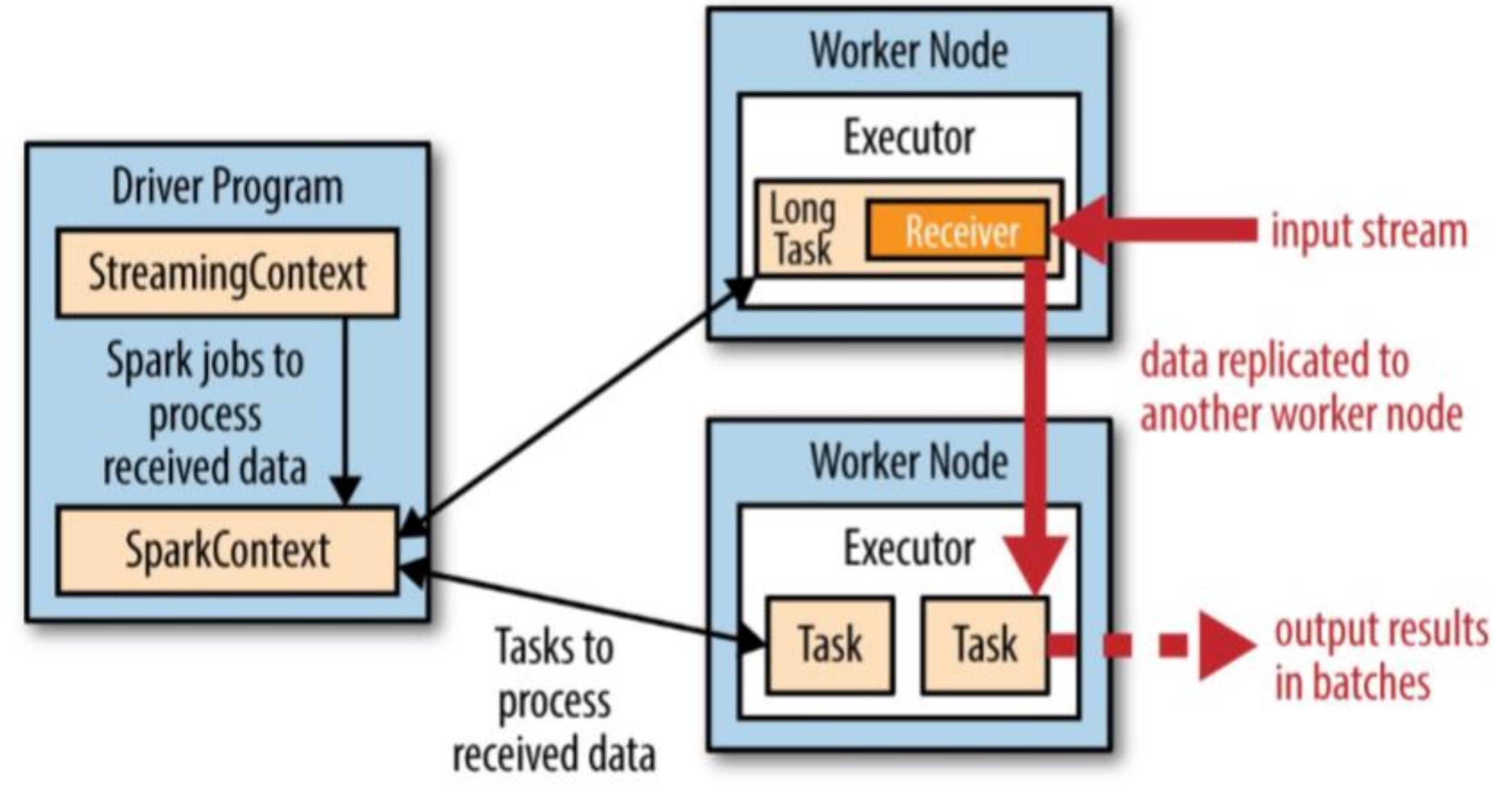
Command to Open a connection to network port

```
$ nc –l localhost 7777
   or
$ nc -lk 7777
```



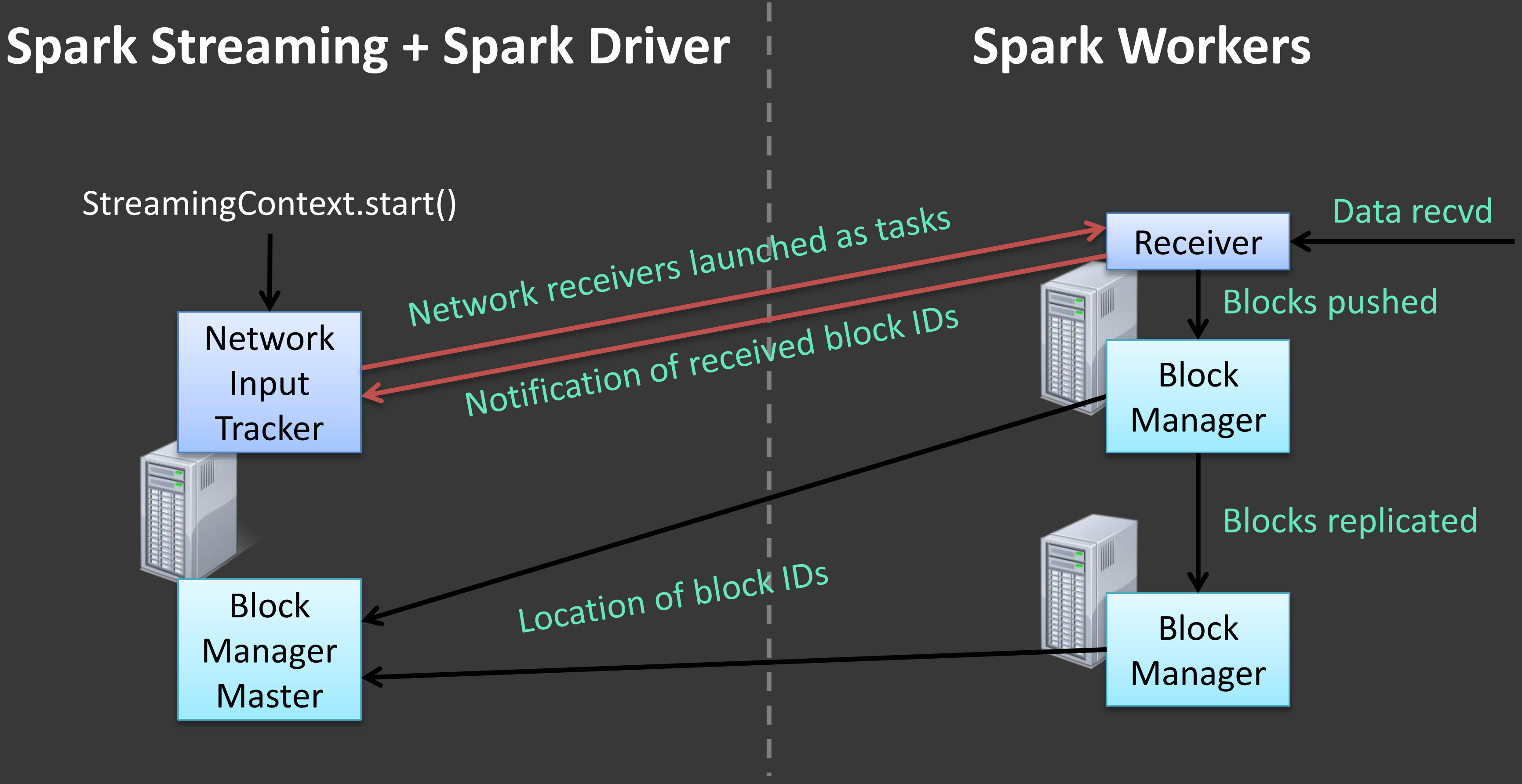
DStreams and transformation

Execution of Spark Streaming


Worker Node
Executor
Long Task
Receiver
input stream
Driver Program
StreamingContext
Spark jobs to process received data
SparkContext
data replicated to another worker node
Worker Node
Executor
Task
Task
output results in batches
Tasks to process received data

Execution of Spark Streaming within Spark's components

Execution Model – Receiving Data



Input Dstreams & Receivers

- Input DStreams are DStreams representing the stream of input data received from streaming sources
- In the quick example, lines was an input DStream as it represented the stream of data received from the netcat server
- Every input Dstream is associated with a Receiver object which receives the data from a source and stores it in Spark's memory for processing..
- Spark Streaming provides two categories of built-in streaming sources.
- Basic sources:
  - Sources directly available in the StreamingContext API. Examples: file systems, socket connections, and Akka actors.
- Advanced sources:
  - Sources like Kafka, Flume, Twitter, etc. are available through extra utility classes. These require linking against extra dependencies as discussed in the linking section
- **When running a Spark Streaming program locally, do not use "local" or "local[1]"** as the master URL. Either of these means that only one thread will be used for running tasks locally. If you are using a input DStream based on a receiver (e.g. sockets, Kafka, Flume, etc.), then the single thread will be used to run the receiver, leaving no thread for processing the received data. Hence, when running locally, always use "local[n]" as the master URL, where n > number of receivers to run.
- Extending the logic to running on a cluster, the number of cores allocated to the Spark Streaming application must be more than the number of receivers. Otherwise the system will receive data, but not be able to process it.

File Streams

- For reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.), a DStream can be created as

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)
```

- Spark Streaming will monitor the directory dataDirectory and process any files created in that directory (files written in nested directories not supported). Note that
  - The files must have the same data format.
  - The files must be created in the dataDirectory by atomically moving or renaming them into the data directory.
  - Once moved, the files must not be changed. So if the files are being continuously appended, the new data will not be read.
- For simple text files, there is an easier method `streamingContext.textFileStream(dataDirectory)`. And file streams do not require running a receiver, hence does not require allocating cores.

Advanced Sources (Twitter & Flume)

- To minimize issues related to version conflicts of dependencies, the functionality to create DStreams from these sources has been moved to separate libraries that can be linked to explicitly when necessary. For example, if you want to create a DStream using data from Twitter's stream of tweets, we have to do the following:

- Linking: Add the artifact **spark-streaming-twitter_2.10** to the SBT/Maven project dependencies.
- Programming: Import the **TwitterUtils** class and create a DStream with TwitterUtils.createStream as shown below.
- Deploying: Generate an uber JAR with all the dependencies (including the dependency spark-streaming-twitter_2.10 and its transitive dependencies) and then deploy the application.

```
import org.apache.spark.streaming.twitter._
TwitterUtils.createStream(ssc, None)


import org.apache.spark.streaming.flume._
val flumeStream = FlumeUtils.createStream(streamingContext, receiver's
hostname, receiver port)
```

Receiver Reliability

- There can be two kinds of data sources based on their reliability. Sources (like Kafka and Flume) allow the transferred data to be acknowledged. If the system receiving data from these reliable sources acknowledges the received data correctly, it can be ensured that no data will be lost due to any kind of failure. This leads to two kinds of receivers:

- **Reliable Receiver** - A reliable receiver correctly sends acknowledgment to a reliable source when the data has been received and stored in Spark with replication.
- **Unreliable Receiver** - An unreliable receiver does not send acknowledgment to a source. This can be used for sources that do not support acknowledgment, or even for reliable sources when one does not want or need to go into the complexity of acknowledgment.

Transformations on DStreams

- 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 like map(), filter(), and reduceByKey().
- In 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.

Stateless DStream Transformations

Function Name	Purpose	Example
map()	Apply a function to each element in the DStream and return a DStream of the result.	ds.map(x => x + 1)
flatMap()	Apply a function to each element in the DStream and return a DStream of the contents of the iterators returned.	ds.flatMap(x => x.split(" "))
filter()	Return a DStream consisting of only elements that pass the condition passed to filter.	ds.filter(x => x != 1)
repartition()	Change the number of partitions of the DStream.	ds.repartition(10)
reduceByKey()	Combine values with the same key in each batch.	ds.reduceByKey((x, y) => x + y)
groupByKey()	Group values with the same key in each batch.	ds.groupByKey()

Stateless DStream Transformations

Function Name	Purpose
join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
transform(func)	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
cogroup(otherStream, [numTasks])	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.

Stateful transformations are operations on DStreams that track data across time; that is, some data from previous batches is used to generate the results for a new batch.
The two main types are
- ✓ windowed operations, which act over a sliding window of time periods
- ✓ updateStateByKey(), which is used to track state across events for each key

Stateful transformations require checkpointing to be enabled in your StreamingContext for fault tolerance

Stateful DStream Transformations

Function Name	Purpose	Example
updateStateByKey(func)	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.	
window(windowLength, slideInterval)	Return a new DStream which is computed based on windowed batches of the source DStream.	ds.window(Seconds(30), Seconds(10))
countByWindow(windowLength, slideInterval)	Return a sliding window count of elements in the stream.	ds.countByWindow(Seconds(30), Seconds(10))
reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function func over batches in a sliding window. Note: By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property spark.default.parallelism) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks.	val ipDStream = accessLogsDStream.map(logEntry => (logEntry.getIpAddress(), 1)) val ipCountDStream = ipDStream.reduceByKeyAndWindow({(x, y) => x + y}, // Adding elements in the new batches entering the window {(x, y) => x - y}, // Removing elements from the oldest batches exiting the window Seconds(30), // Window duration Seconds(10))
countByValueAndWindow(windowLength, slideInterval, [numTasks])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument.	val ipDStream = accessLogsDStream.map{entry => entry.getIpAddress()} val ipAddressRequestCount = ipDStream.countByValueAndWindow(Seconds(30), Seconds(10))

UpdateStateByKey Operation

The **updateStateByKey** operation allows you to maintain arbitrary state while continuously updating it with new information. To use this, we will have to do two steps.

- Define the state - The state can be an arbitrary data type.
- Define the state update function - Specify with a function how to update the state using the previous state and the new values from an input stream.

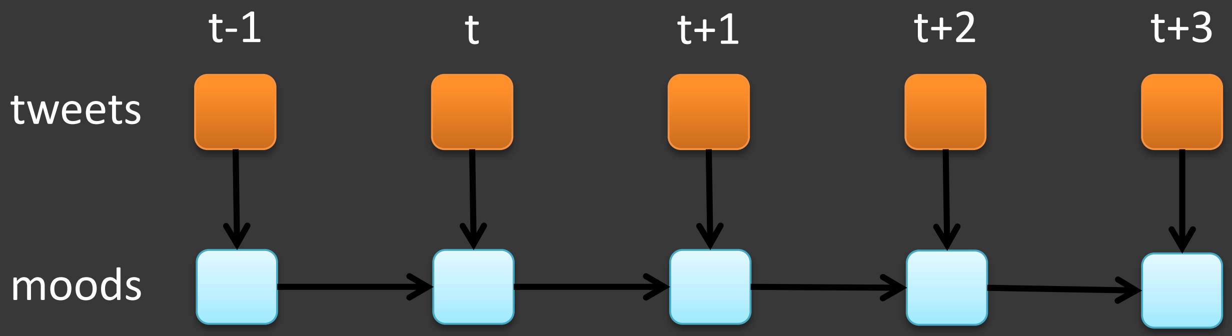
In every batch, Spark will apply the state update function for all existing keys, regardless of whether they have new data in a batch or not. If the update function returns None then the key-value pair will be eliminated.

```
def updateRunningSum(values: Seq[Long], state: Option[Long]) = {
Some(state.getOrElse(0L) + values.size)
}
val responseCodeDStream = accessLogsDStream.map(log =>
(log.getResponseCode(), 1L))
val responseCodeCountDStream =
responseCodeDStream.updateStateByKey(updateRunningSum _)
```

Arbitrary Stateful Computations

- Maintain arbitrary state, track sessions
  – Maintain per-user mood as state, and update it with his/her tweets

```
moods = tweets.updateStateByKey(tweet => updateMood(tweet))
updateMood(newTweets, lastMood) => newMood
```



WINDOW OPERATIONS

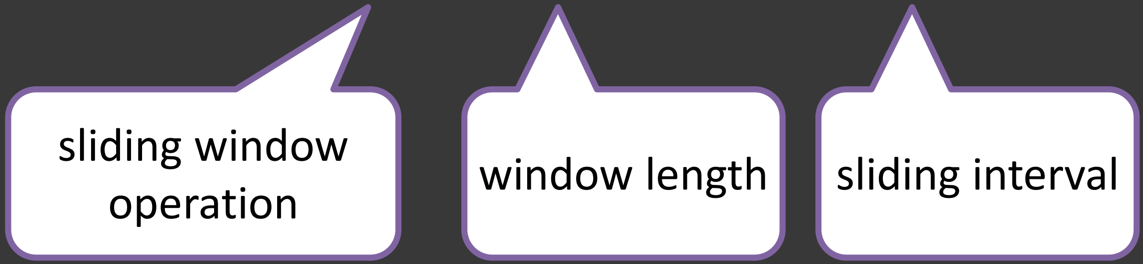
Spark streaming provides window based operations

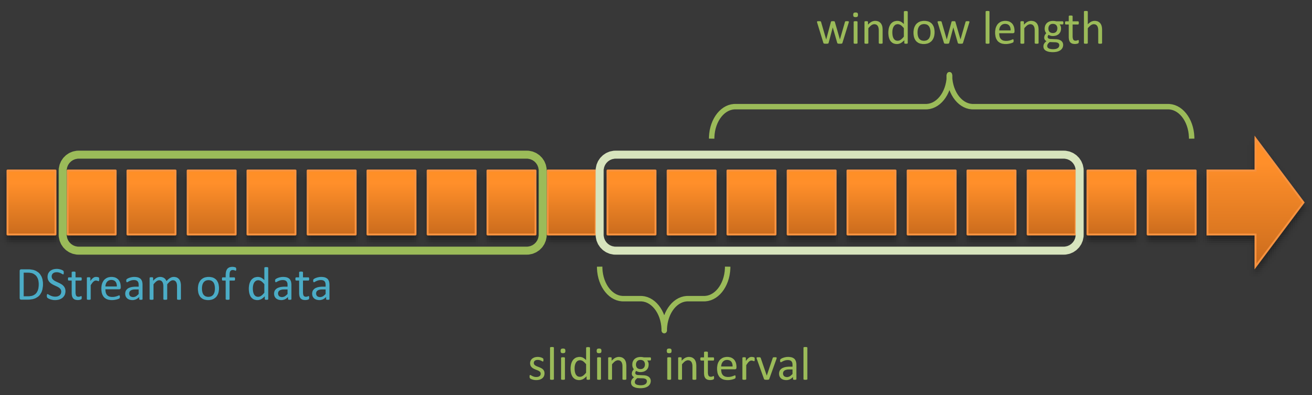
- ✓ get streaming data at specified window duration of data for given period of time.
- ✓ apply transformations over a sliding window of data.
- ✓ Windowed operations compute results across a longer time period than the StreamingContext's batch interval, by combining results from multiple batches
- ✓ All windowed operations need two parameters, window duration and sliding duration, both of which must be a multiple of the StreamingContext's batch interval
- ✓ The window duration controls how many previous batches of data are considered, namely the last windowDuration/batchInterval.
- ✓ If we had a source DStream with a batch interval of 10 seconds and wanted to create a sliding window of the last 30 seconds (or last 3 batches) we would set the windowDuration to 30 seconds

```
val accessLogsWindow = accessLogsDStream.window(Seconds(30), Seconds(10))
val windowCounts = accessLogsWindow.count()
```

Example – Count the hashtags over last 1 min

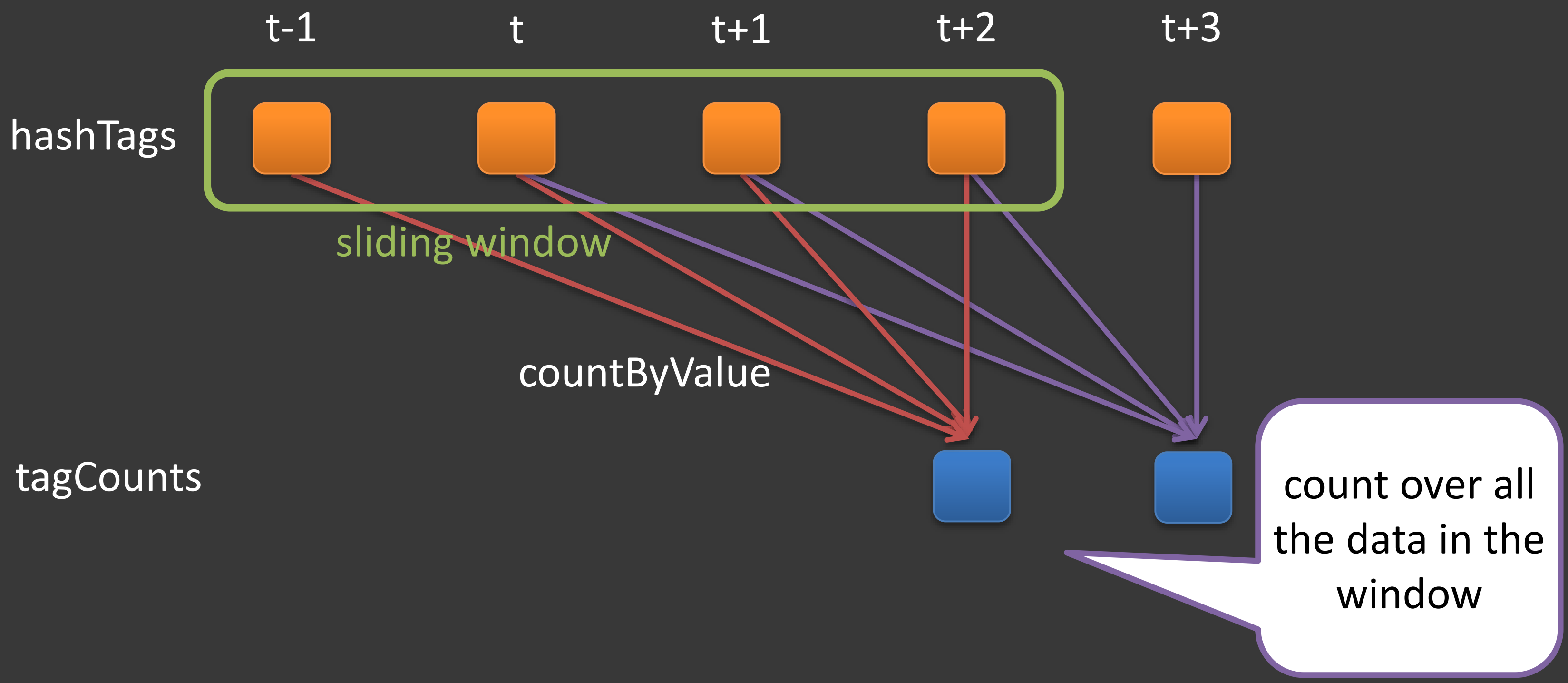
```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(1)).countByValue()
```





Example – Count the hashtags over last 1 min

```
val tagCounts = hashTags.window(Minutes(1),
Seconds(1)).countByValue()
```



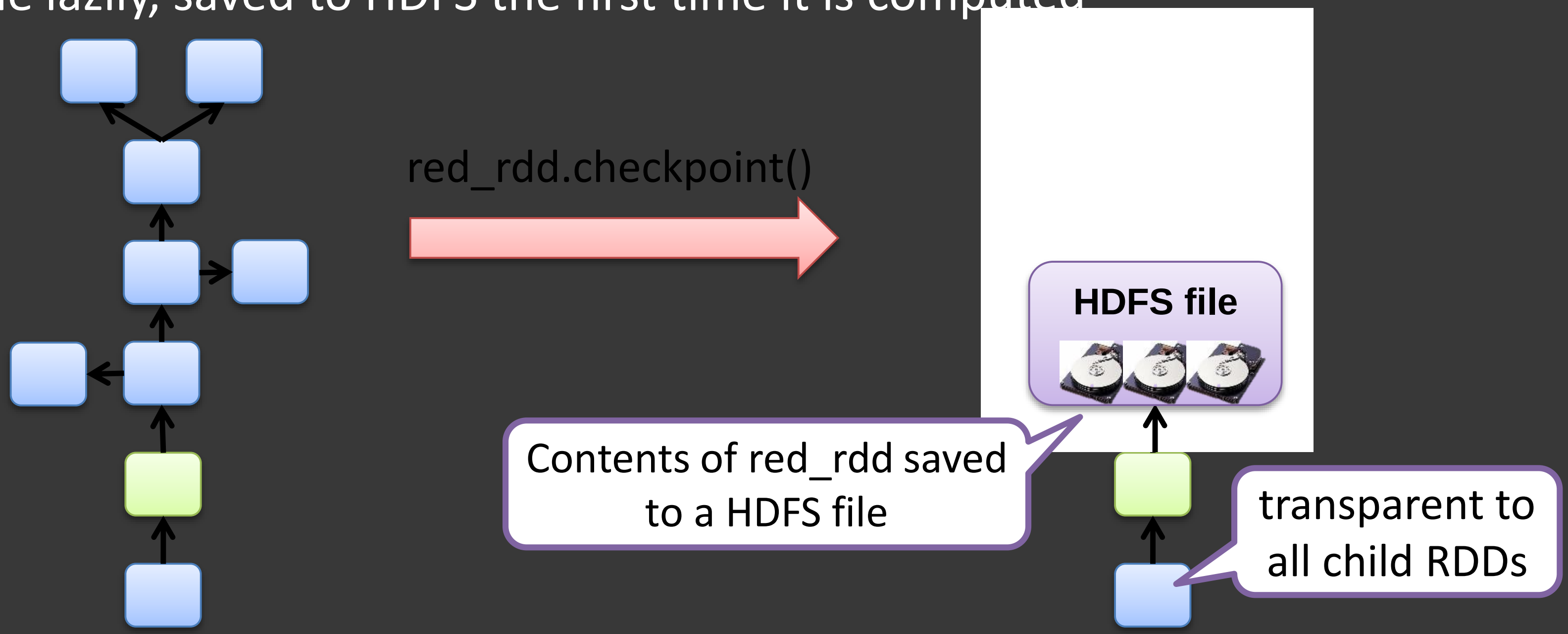
Output Operations on DStreams

Function	Purpose	Example
print()	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging.	ds.print()
saveAsTextFiles(prefix, [suffix])	Save this DStream's contents as text files. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".	ds.saveAsTextFiles("outputDir", "txt"
saveAsObjectFiles(prefix, [suffix])	Save this DStream's contents as SequenceFiles of serialized Java objects. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".	ds. saveAsObjectFiles ("outputDir", "txt"
saveAsHadoopFiles(prefix, [suffix])	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".	writableIpAddressRequestCount.saveAsHadoopFiles[SequenceFileOutputFormat[Text, LongWritable]]("outputDir", "txt"
foreachRDD(func)	The most generic output operator that applies a function, func, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database.	ds.foreachRDD { rdd => rdd.foreachPartition { partition => // Open connection to storage //system (e.g. a database //connection) partition.foreach { item => // Use connection to push item to system } // Close connection } }

What is RDD checkpointing?

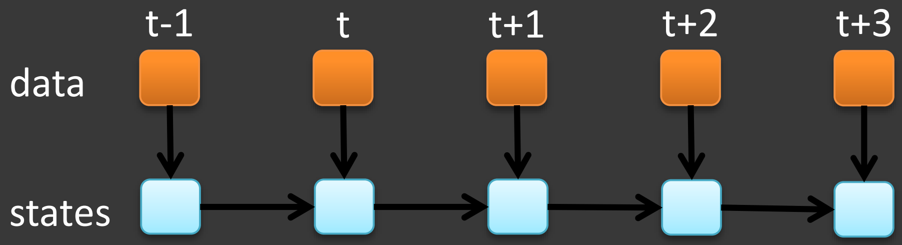
Saving RDD to HDFS to prevent RDD graph from growing too large

- Done internally in Spark transparent to the user program
- Done lazily, saved to HDFS the first time it is computed


red_rdd.checkpoint()
HDFS file
Contents of red_rdd saved to a HDFS file
transparent to all child RDDs

Why is RDD checkpointing necessary?

Stateful DStream operators can have infinite lineages

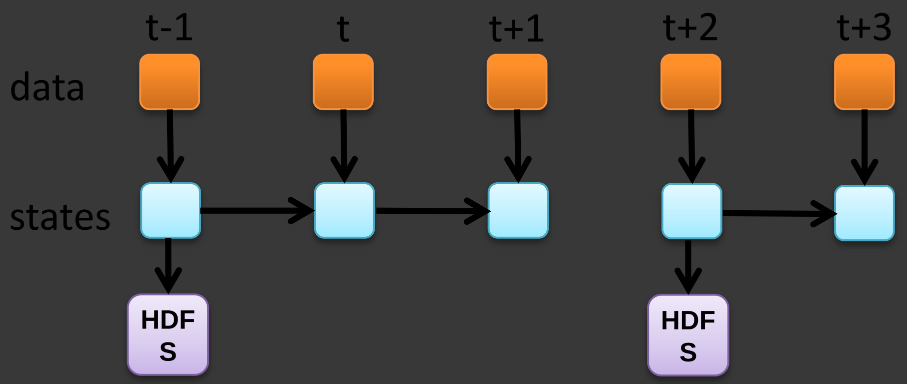


Large lineages lead to …

- Large closure of the RDD object → large task sizes → high task launch times
- High recovery times under failure

Why is RDD checkpointing necessary?

Stateful DStream operators can have infinite lineages



Periodic RDD checkpointing solves this

Useful for iterative Spark programs as well

RDD Checkpointing

- Periodicity of checkpoint determines a tradeoff
  - Checkpoint too frequent: HDFS writing will slow things down
  - Checkpoint too infrequent: Task launch times may increase
  - Default setting checkpoints at most once in 10 seconds
  - Try to checkpoint once in about 10 batches

CHECKPOINTING

Example:

```
val sc = new SparkContext(conf)  // Create a StreamingContext with a 1 sec batch
val ssc = new StreamingContext(sc, Seconds(1))
ssc.checkpoint(checkpointDir)
……
val ssc = StreamingContext.getOrCreate(checkpointDir, createStreamingContext _)
```

Performance Tuning

Step 1

Achieve a stable configuration that can sustain the streaming workload

Step 2

Optimize for lower latency

Step 1: Achieving Stable Configuration

How to identify whether a configuration is stable?

- Look for the following messages in the log

```
    Total delay: 0.01500 s for job 12 of time 1371512674000 …
```

- If the total delay is continuously increasing, then unstable as the system is unable to process data as fast as its receiving!

- If the total delay stays roughly constant and around 2x the configured batch duration, then stable

Step 1: Achieving Stable Configuration

How to figure out a good stable configuration?

- Start with a low data rate, small number of nodes, reasonably large batch duration (5 – 10 seconds)
- Increase the data rate, number of nodes, etc.
- Find the bottleneck in the job processing
  - Jobs are divided into stages
  - Find which stage is taking the most amount of time

Step 1: Achieving Stable Configuration

How to figure out a good stable configuration?

- If the first map stage on raw data is taking most time, then try …
  - Enabling delayed scheduling by setting property `spark.locality.wait`
  - Splitting your data source into multiple sub streams
  - Repartitioning the raw data into many partitions as first step
- If any of the subsequent stages are taking a lot of time, try…
  - Try increasing the level of parallelism (i.e., increase number of reducers)
  - Add more processors to the system

Step 2: Optimize for Lower Latency

- Reduce batch size and find a stable configuration again
  - Increase levels of parallelism, etc.

- Optimize serialization overheads
  - Consider using Kryo serialization instead of the default Java serialization for both data and tasks
  - For data, set property `spark.serializer=spark.KryoSerializer`
  - For tasks, set `spark.closure.serializer=spark.KryoSerializer`

- Use Spark stand-alone mode rather than Mesos

Step 2: Optimize for Lower Latency

- Using concurrent mark sweep GC `-XX:+UseConcMarkSweepGC` is recommended
  - Reduces throughput a little, but also reduces large GC pauses and may allow lower batch sizes by making processing time more consistent

- Try disabling serialization in DStream/RDD persistence levels
  - Increases memory consumption and randomness of GC related pauses, but may reduce latency by further reducing serialization overheads

- For a full list of guidelines for performance tuning
  - Spark Tuning Guide
  - Spark Streaming Tuning Guide

- ***Example***

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming._
import org.apache.spark.streaming.twitter.TwitterUtils
/**
 * @author ${user.name}
 */
object App {

  def main(args: Array[String]) {
    val conf = new SparkConf()
      .setAppName("The swankiest Spark app ever")
      .setMaster("local[2]")

    val sc = new SparkContext(conf)
    val ssc = new StreamingContext(sc, Seconds(10))
    val Array(consumerKey, consumerSecret, accessToken, accessTokenSecret) = args.take(4)
    val filters = args.takeRight(args.length - 4)
```

```
// Set the system properties so that Twitter4j library used by Twitter stream
    // can use them to generate OAuth credentials
    System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
    System.setProperty("twitter4j.oauth.consumerSecret", consumerSecret)
    System.setProperty("twitter4j.oauth.accessToken", accessToken)
    System.setProperty("twitter4j.oauth.accessTokenSecret", accessTokenSecret)
    val stream = TwitterUtils.createStream(ssc, None, filters)
    //Transformations on DStream.
    //flatmap
    val hashTags = stream.flatMap(status => status.getText.split(" ").filter(_.startsWith("#")))
    println("hashTags counts===>" + hashTags.print())
    //CountByValue Transformation.
    val tagCounts = hashTags.countByValue()
    println("tag counts===>" + tagCounts.print())
    val topCounts60 = hashTags.map((_, 1)).reduceByKeyAndWindow(_ + _, Seconds(60))
      .map { case (topic, count) => (count, topic) }
      .transform(_.sortByKey(false))
```

```
val topCounts10 = hashTags.map((_, 1)).reduceByKeyAndWindow(_ + _, Seconds(10))
      .map { case (topic, count) => (count, topic) }
      .transform(_.sortByKey(false))
    // Print popular hashtags
    topCounts60.foreachRDD(rdd => {
      val topList = rdd.take(10)
      println("\nPopular topics in last 60 seconds (%s total):".format(rdd.count()))
      topList.foreach { case (count, tag) => println("%s (%s tweets)".format(tag, count)) }
    })
    topCounts10.foreachRDD(rdd => {
      val topList = rdd.take(10)
      println("\nPopular topics in last 10 seconds (%s total):".format(rdd.count()))
      topList.foreach { case (count, tag) => println("%s (%s tweets)".format(tag, count)) }
    })
    ssc.start()
    ssc.awaitTermination()
  }
}
```

THANK YOU