UNIVERSITY of WASHINGTON

Welcome to BIGDATA 210: Introduction to Data Engineering

Week 5

Week 5

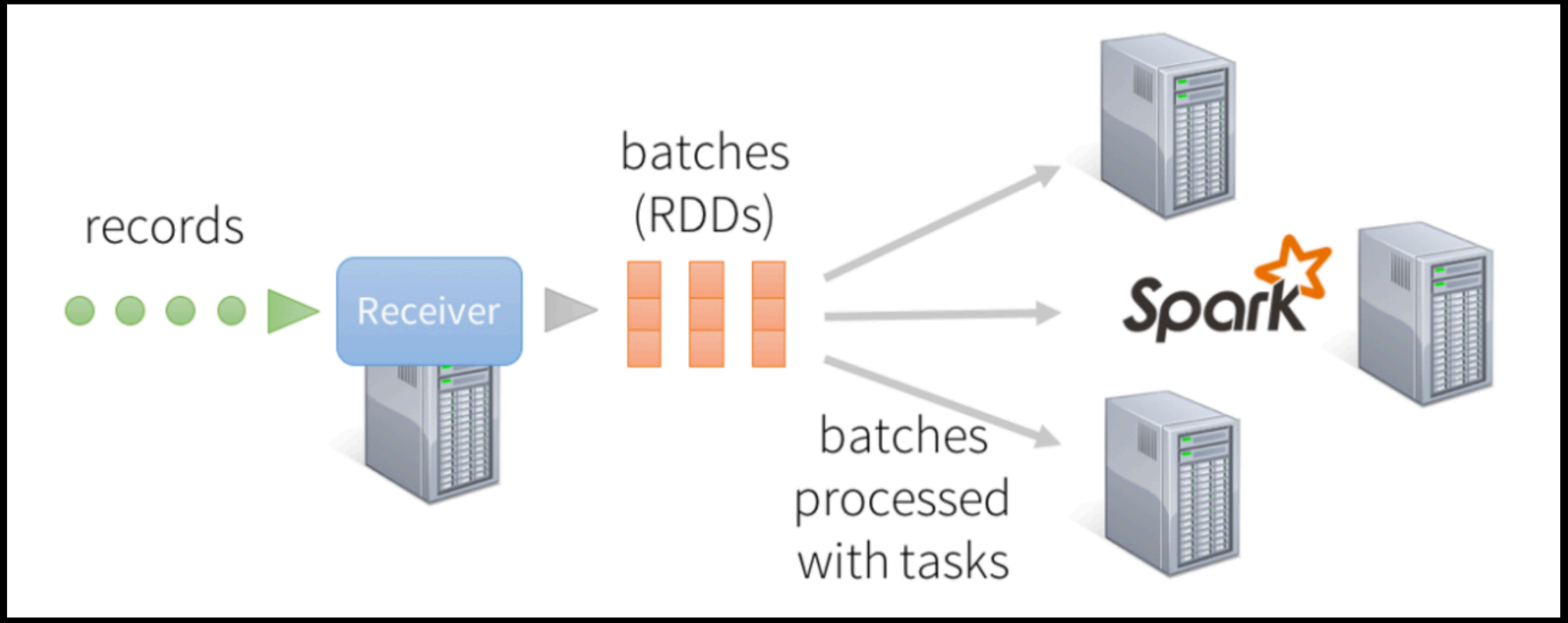
What we will do?

- Questions from last week
- Status of Projects
- Spark Streaming
  - Illustrate Spark Streaming Architecture
  - Describe Spark Streaming
  - List and Describe Input Sources/Receivers
  - List and Use Common Transformations and Output Operations
  - Effectively use Checkpointing
- Hands-On

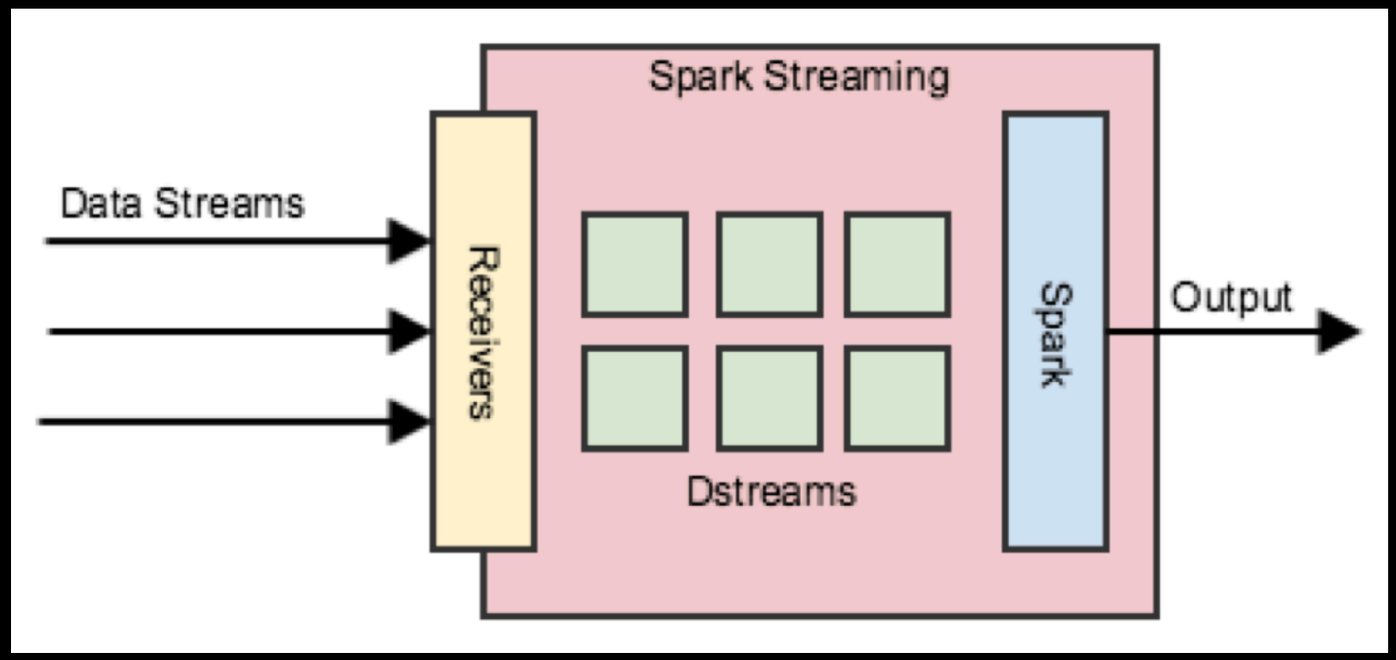
What is Spark Streaming

- Extends Spark for doing large scale stream processing
  - Scales to 100s of nodes and achieves second scale latencies

- Efficient and fault-tolerant stateful stream processing
  - Simple batch-like API for implementing complex algorithms

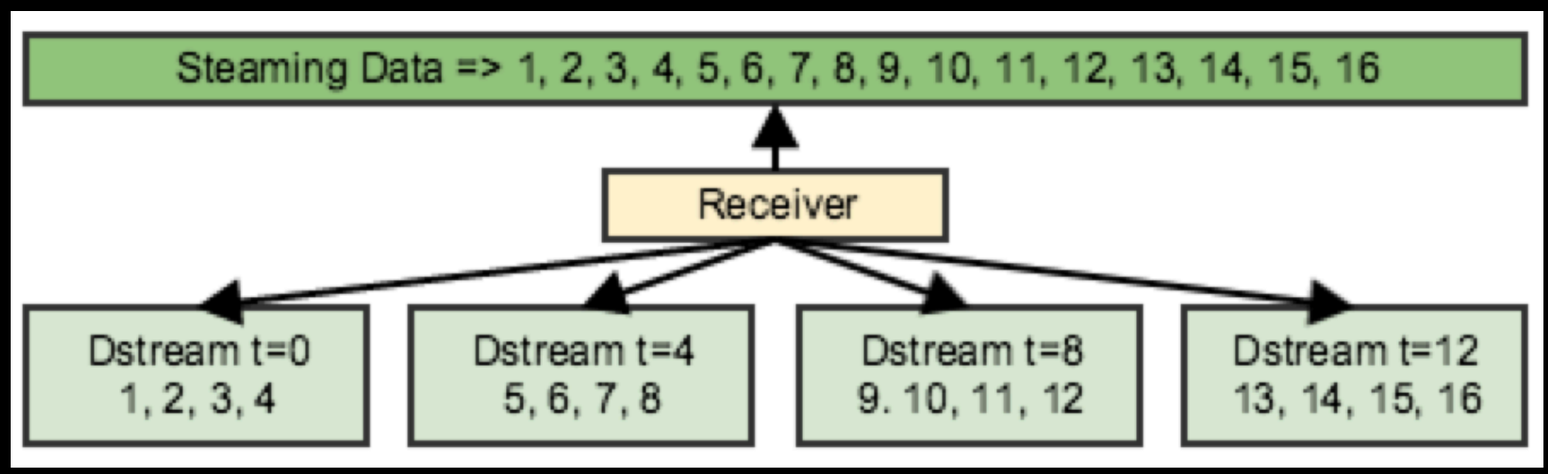


- Streaming Applications consist of the same components as a Core application, but add the concept of a receiver
- The receiver is a process running on an executor



Dstream Creation

- Batches are created at regular time intervals
  - After the batch duration completes, data is shipped off

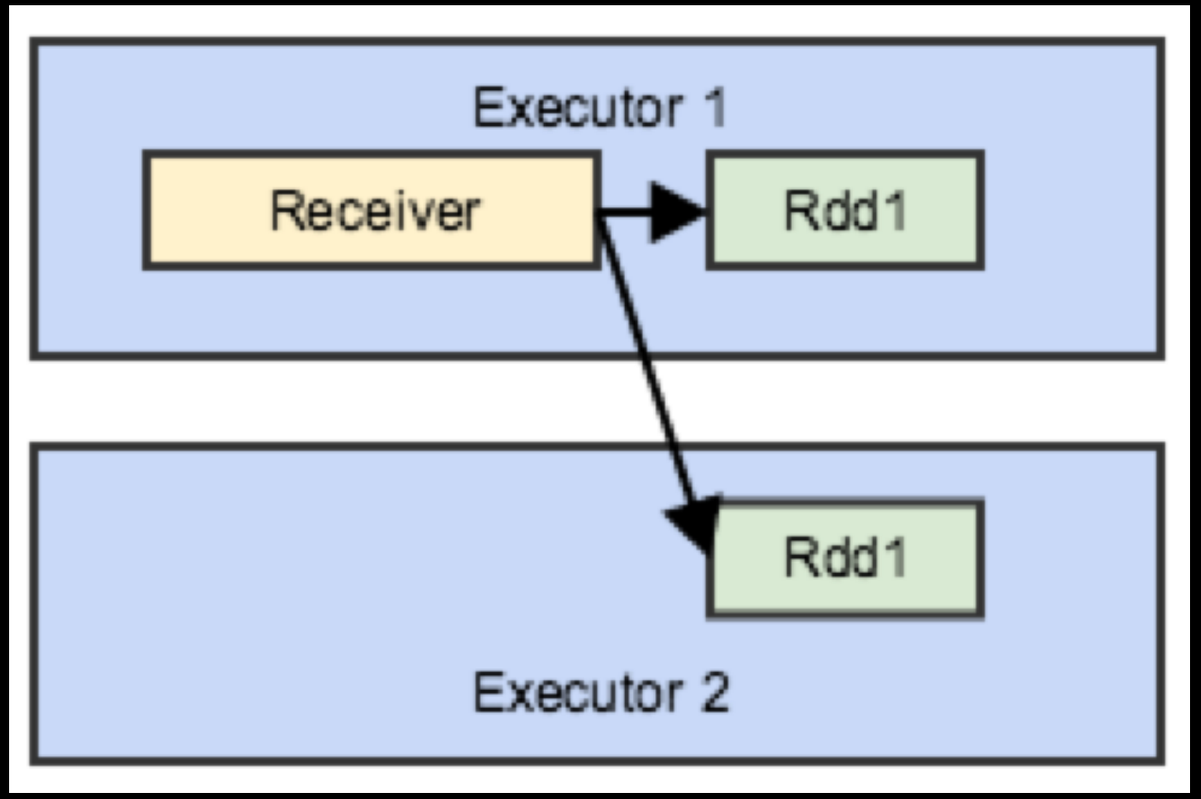


Discretiized Stream Processing

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs
- Processed results of the RDD operations are returned in batches

Distribution of Data

- Receiver duplicates data to two executors by default



Distribution of Data

- As soon as the data is done being processed by Spark, executors discarded
  - For stateful operations, the data is preserved in memory
- Checkpointing is required for stateful operations

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Ad impressions
- Distributed stream processing framework is required to
  - Scale to large clusters (100s of machines)
  - Achieve low latency (few seconds)

Integration with Batch Processing

- Many environments require processing same data in live streaming as well as batch post processing
- Existing framework cannot do both
  - Either do stream processing of 100s of MB/s with low latency
  - Or do batch processing of TBs / PBs of data with high latency
- Extremely painful to maintain two different stacks
  - Different programming models
  - Double the implementation effort
  - Double the number of bugs

Spark Streaming Context

- Main entry point for streaming application
- Created from a Spark Context and an interval time

```
ssc = StreamingContext(sparkContext, batchDuration)
```

- Creating a streaming context with 2 working threads, named StreamingApp, and a batch duration of 1 second

```
from pyspark.streaming import StreamingContext
sc = SparkContext("local[2]","StreamingApp")
ssc = StreamingContext(sc, 1)
```

Receivers

- Data comes from a receiver running on an Executor
- The receiver breaks the data into batches of the duration specified in the StreamingContext
  - Converts the data from continuous to discrete
- Basic socketTextStream

```
inputDS = ssc.socketTextStream("localhost", 2222)
```

Working with Dstreams

- Dstreams heavily reuse the APIs from Spark Core
  - Some special APIs for stateful transformations
- Example of doing a Spark Streaming WordCount

```
wcDS = inputDS.flatMap(lambda line: line.split(" \
").map(lambda word: (word,1)).reduceByKey(lambda a,b: a+b)
wcDS.pprint()
```

Starting the Stream

Transformation/Outputs are defined before the Stream is started

The REPL can still be used to explore streaming, but is severely limited by this

Once the stream is started, the developer cannot make updates to the transformations

```
ssc.start()
ssc.awaitTermination()
```

Input Sources

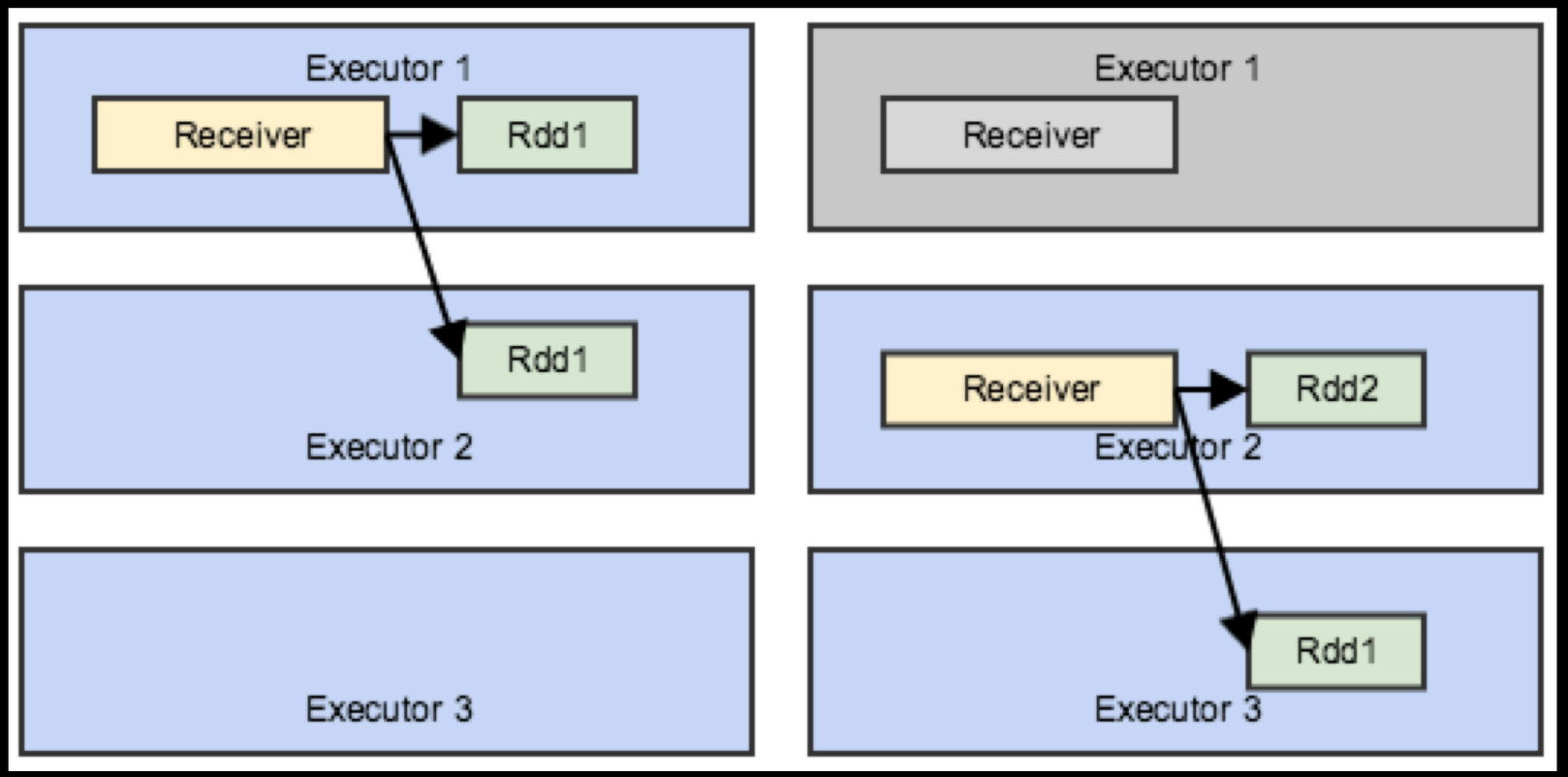
- Input sources are defined as receivers
- Every input Dstream is defined by a receiver, which is configured to a source

```
inputDS = ssc.socketTextStream(host, port)
```

- As of Spark 1.6.1 only a few receivers are supported by python
  - Kafka
  - Kinesis
  - Flume
  - MQTT
  - textFileStream(someDir)
  - textSocketStream(host, port)

Receiver Reliability

- If a node with a receiver goes down the receiver is started on another node



Receiver Reliability

Two types of receivers

- Reliable Receiver
  - Sends acknowledgment to a reliable source when the data has been received and stored in Spark with replication.
- Unreliable Receiver
  - Does not send acknowledgment to a source.
  - Used for sources that do not support acknowledgment, or when acknowledgement is not required
- To implement a reliable receiver, the user will have to create a custom receiver and ensure the source has some method for accepting a "handshake"

Fault-tolerance: Worker

- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- All transformed data is fault-tolerant, and exactly-once transformations

Fault-tolerance: Master

- Master saves the state of the DStreams to a checkpoint file
  - Checkpoint file saved to HDFS periodically
- If master fails, it can be restarted using the checkpoint file

Stateless Transformations

- Data does not depend on any data in a previous batch is processed using stateless transformation
- Stateless transformations are just like Core transformations
- Examples of transformations that are stateless
  - map
  - flatMap
  - filter
  - reduceByKey
  - repartition
- These transformations are applied to the current Dstream, and not across previous Dstreams

Stateless Transformation Example

```
inputDS = ssc.socketTextStream("localhost", 2222)

wcDS = inputDS.flatMap(lambda line: line.split(" \ ")\
  .map(lambda word: (word,1)).reduceByKey(lambda a,b: a+b)
wcDS.print()
```

Stateless Transformation

- Can combine data from multiple receivers using join, union, etc
  - This works if the receivers have the same batch duration
  - Again, this only is applied for the current point in time
- Dstreams are not RDDs, but conceptually similar
  - Dstreams can be converted to RDDs using the transform() api
    - A common use of the transform() api is to reuse code written for batch processing
    - Not all RDD operations are exposed as Dstream operations, using transform is an acceptable work around

```
rdd.transform(lambda rdd: somefunction(rdd))
```

Combining batch and streaming processing

- Using the transform API, Spark applications combine data streams with static datasets

```
dataset = sc.textFile("somefile.txt")

inputDS.transform(lambda Dstream: Dstream.join(dataset).map(…))
```

Output Operations

- Output operations allow DStream's data to be pushed out external systems
- Output operations trigger the actual execution of all the DStream transformations (similar to actions for RDDs)
- Examples of Output operations:
  - pprint()
  - saveAsTextFile
  - foreachRDD

Arbitrary Combinations of Batch and Streaming Computations

Inter-mix RDD and DStream operations!

- Example: Join incoming tweets with a spam HDFS file to filter out bad tweets

```
tweets = ssc.twitterStream()
spamHDFSFile = sc.textFile("spamHDFSFile.txt")

tweets.transform(lambda tweetsRDD: /
  ( tweetsRDD.join(spamHDFSFile).filter(…)
))
```

Unifying Batch and Stream Processing Models

➔ Spark Batch program on Twitter log file using RDDs

```
tweets = sc.textFile("hdfs://...")
hashTags = tweets.flatMap (lambda status: getTags(status))
hashTags.saveAsTextFile("hdfs://...")
```

➔ Spark Streaming program on Twitter stream using DStreams

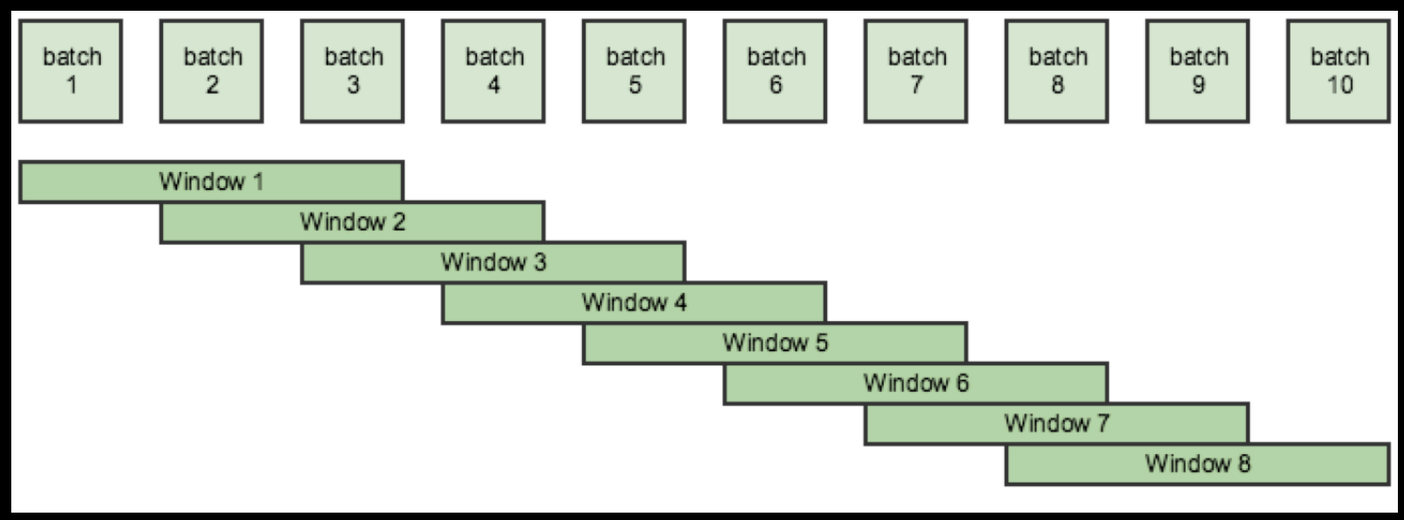
```
tweets = ssc.twitterStream()
hashTags = tweets.flatMap (lambda status: getTags(status))
hashTags.saveAsTextFile("hdfs://...")
```

Stateful Transformations

- Operations that track data across multiple dstreams
- Two main types of stateful transformations:
  - Window
  - State
- Stateful transformations require checkpointing to be enabled for state transformations
  - "Source" data is discarded once the next batch comes in

Stateful Transformations - Window functions

- Window functions operate of a set of dstreams to combine results from multiple Dstreams



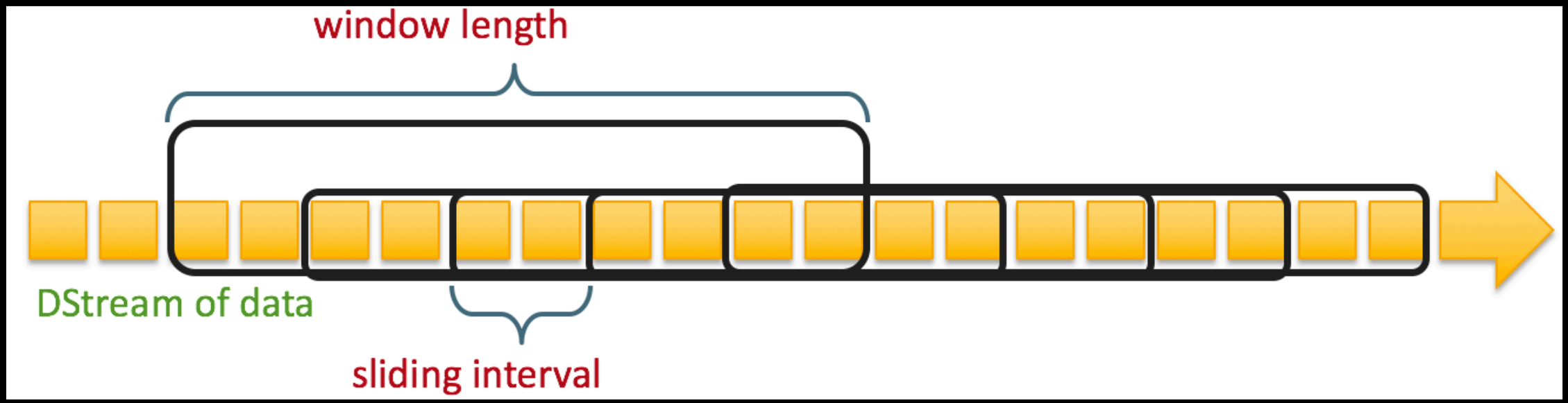
Stateful Transformations - Window functions

Window operations take two parameters:

- Window Duration
  - Multiple of the Dstream duration
  - Defines how many batches of data are needed
- Sliding Duration
  - Multiple of the Dstream duration
  - Defines how often the window moves

Window-based Transformations

```
tweets = ssc.twitterStream()
hashTags = tweets.flatMap (lambda status: getTags(status))
tagCounts = hashTags.window(60,12).countByValue()
```



Creating a Window

```
ssc = StreamingContext(sc, 1)

inputDS = ssc.socketTextStream("localhost", 2222)

windowDS = inputDS.window(30, 10).map(lambda line: \
 line.split(" ")).map(lambda line:(line,1))
windowDs.reduceByKey(lambda a,b: a+b)).pprint()
```

a window is built on top of inputDS with a length of 30s, and a sliding interval of 10 seconds

Window Transformations

Special window operations that can be executed on Dstreams

Examples:

- countByWindow(windowDuration, slideInterval)
- reduceByKeyAndWindow(func, windowDuration, slideInterval)
- countByValueAndWindow(windowDuration, slideInterval)

```
inputDS.reduceByKeyAndWindow(lambda a,b: a+b, 30, 10)
```

Stateful Transformations: updateStateByKey

- Some applications required to keep data from the beginning of when a streaming application is started
- updateStateByKey allows the user to do this

Example:

```
def updateCount(newValues, runningCount):
  if runningCount is None:
    runningCount=0
    return sum(newValues + runningCount)
inputDS.updateStateByKey(updateCount)
```

Configuring Checkpointing

- Required for applications that use state transformations

```
def createContext():
  sc = SparkContext()
  ssc = StreamingContext()
  ssc.checkpoint(checkpointDirector)
  return ssc
```

- Set up a function to create the context, this is important in the next step
  - Creating this function allows the user to restart the application from checkpoint directory if needed

Configuring Checkpoint

- Streaming applications are long living by nature
- For failures, the DevOps team will want to restart the application from a previous state
- The code below will create a streaming context, and start it from the previous checkpointDirectory, otherwise it will create one if the checkpointDirectory is null

```
ssc = StreamingContext.getOrCreate(checkpointDirectory, \
 createContext)
ssc.start()
ssc.awaitTermination()
```

Monitoring Applications

- The Spark UI has a Streaming tab for streaming applications
- There are a couple very important metrics to monitor:
  - Processing Time - The time to process each batch of data
  - Scheduler Delay - The time a batch waits in a queue for the processing of previous batches to finish
- If the batch processing time is consistently more than the batch interval and/or the queueing delay keeps increasing
  - it indicates that the system is not able to process the batches as fast they are being generated and is falling behind
  - consider reducing the batch interval
  - consider repartitioning to more partitions and increasing the number of cores available