Spark Streaming

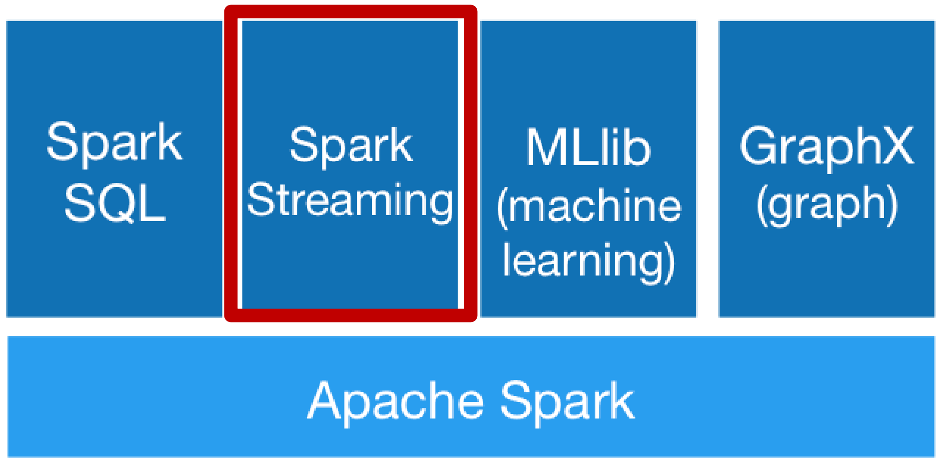
http://spark.apache.org/docs/latest/streaming-programming-guide.html

Nagiza F. Samatova, samatova@csc.ncsu.edu
Professor, Department of Computer Science
North Carolina State University

Senior Scientist, Computer Science & Mathematics Division
Oak Ridge National Laboratory

Spark Streaming

- Streaming applications benefit from acting on data as soon as it arrives by:
  - tracking statistics about page views in real time,
  - training a machine learning model, or
  - automatically detecting anomalies
- **Spark Streaming** is Spark's module for such applications:
  - lets users write streaming applications using a very similar API to batch jobs, and
  - thus reuse a lot of the skills and even code they built for those.
- **DStreams**: Spark's abstraction for discretized streams



DStreams

- **DStream** is **a sequence of RDDs** arriving at each time step (hence, the name "discretized").
- DStreams can be created from various input sources, such as Flume, Kafka, or HDFS.
- Operations on DStreams:
  - *Transformations*: yield a new DStream
  - *Actions*: operate on RDDs of the DStreams
  - *Sliding Window* operations
  - *Output operations*: write data to an external system
- 24/7 Operation: Unlike batch programs, Spark Streaming applications operate 24/7:
  - via **Checkpointing** mechanism for storing data in a reliable file system such as HDFS
  - via **Restarting** (manually or automatically) applications on failure

Example: Step 1: Import Modules

<u>Example</u>: Receive a stream of newline-delimited lines of text from a server running at port 77777, filter only the lines that contain the word "error," and print them

Step 1. Import the required modules

```
from __future__ import division
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
import sys
import random
```

Step 2: Create StreamingContext

- Create a **StreamingContext**, key entry point for streaming functionality
  - set up an underlying SparkContext to process the data
- StreamingContext takes as input a **batch interval** specifying how often to process new data (e.g., every second)

```
# Step 2. Create StreamingContext

conf = (SparkConf()
         .setMaster("local[4]")
         .setAppName("SentimentAnalysis")
         .set("spark.executor.memory", "2g"))

sc = SparkContext(conf=conf)

ssc = StreamingContext(sc, 1)    # Create a streaming context with
batch interval of 1 sec
ssc.checkpoint("checkpoint")
```

Step 3: Create DStream w/ socketTextStream

- Use **socketTextStream()** to <u>create</u> **DStream** based on text data received on port 7777 of the local machine
- <u>Transform</u> the DStream with **filter()** to get only lines that contain “error”
- Apply the <u>output operation</u> *p*print() [pretty print] to print some of the filtered lines.

Step 3. Create DStream

```
# Streaming filter for printing lines containing "error" in Python
lines = ssc.socketTextStream("localhost", 7777)
error_lines = lines.filter(lambda x: "error" in x)
error_lines.pprint()

# Start our streaming context and wait for it to "finish"
ssc.start()
# Wait for the job to finish
ssc.awaitTermination()
```

Step 4: Test

- Start the Spark job, provide its input via a socket using ncat:
- Install: `# sudo apt-get install nmap`
- Start the socket on Ubuntu: `$ nc -lk 7777`
- Type in the lines, the lines that contain "error" will be printed within a respective interval on the terminal where this Spark program is running

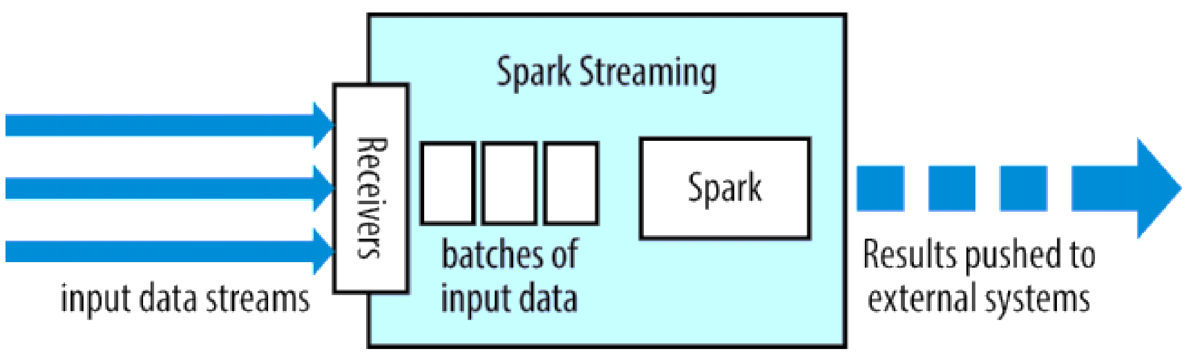
```
tolik@tolik-VirtualBox:~$ nc -lk 7777
error
another error
```

```
16/01/04 01:00:35 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.sca
93, took 5.084788 s
-------------------------------------------
Time: 2016-01-04 01:00:30
-------------------------------------------
another error

16/01/04 01:00:35 INFO JobScheduler: Finished job streaming job 145188723000
.0 from job set of time 1451887230000 ms
```

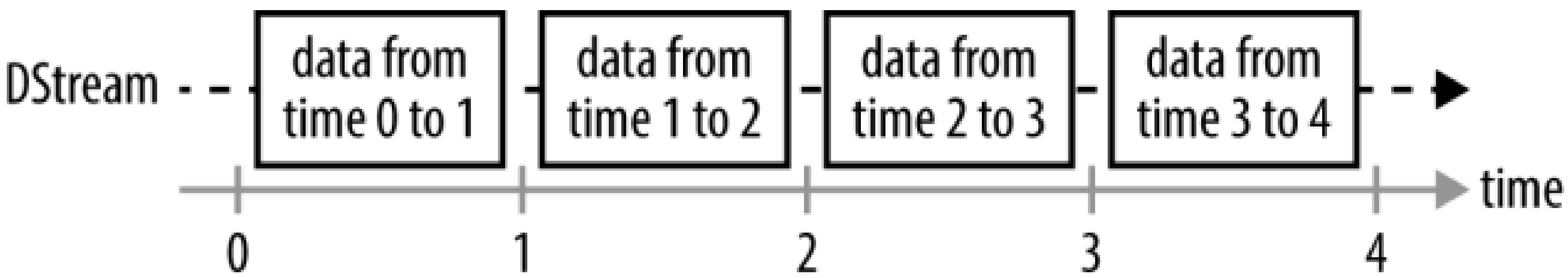
Micro-Batching Architecture

- **Spark Streaming uses a "micro-batch" architecture:**
  - streaming computation 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 called the batch interval:
    - The batch interval is typically between **500 milliseconds and several seconds**, as configured by application developer



DStream as a Continuous Series of RDDs

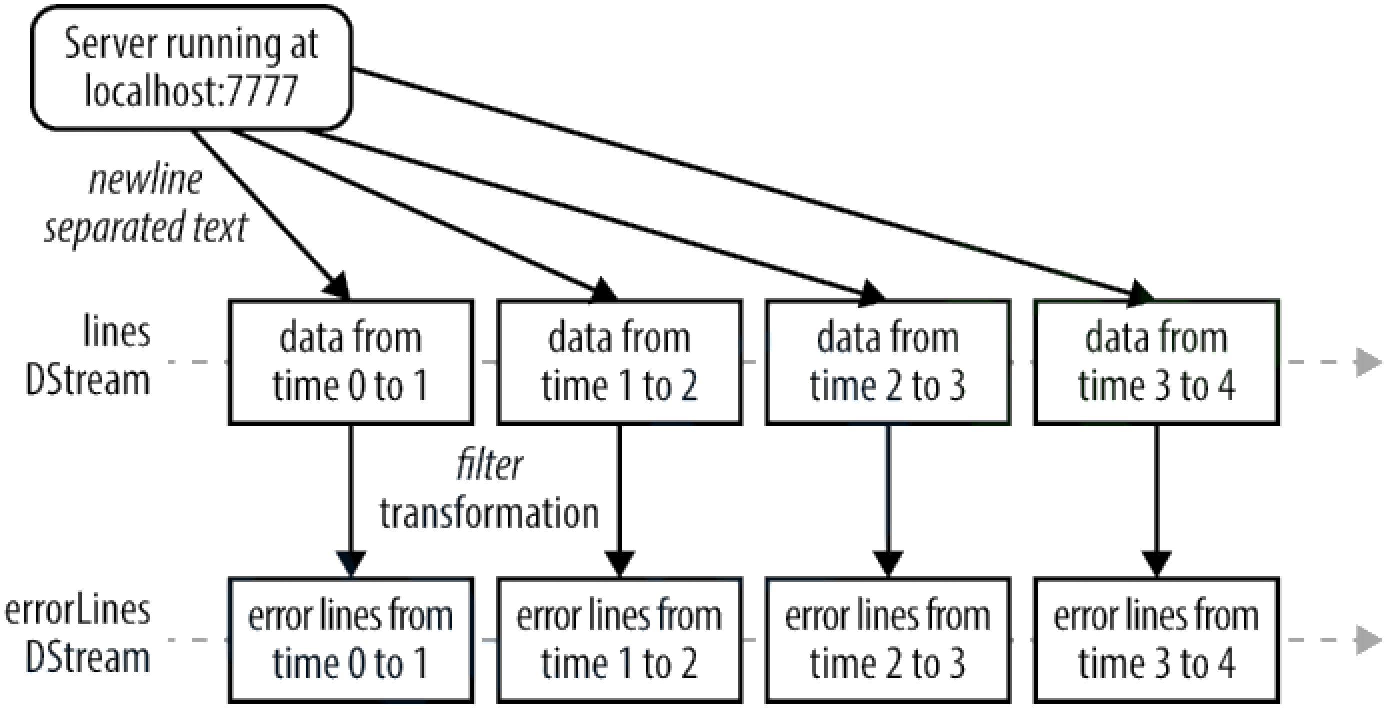
- **DStream is a discretized stream, the programming abstraction in Spark Streaming**
- **DStream is a sequence of RDDs, where each RDD has one time slice of the data in the stream**



- **Create DStreams:**
  - either from external input sources, or
  - by applying transformations to other DStreams
- **DStreams support many of the transformations on RDDs**
- **DStreams have stateful transformations to aggregate data over time**

DStreams and Transformations Example

- **In our example, we created a DStream from data received through a socket, and then applied a filter() transformation to it.**
- **This internally creates RDDs:**

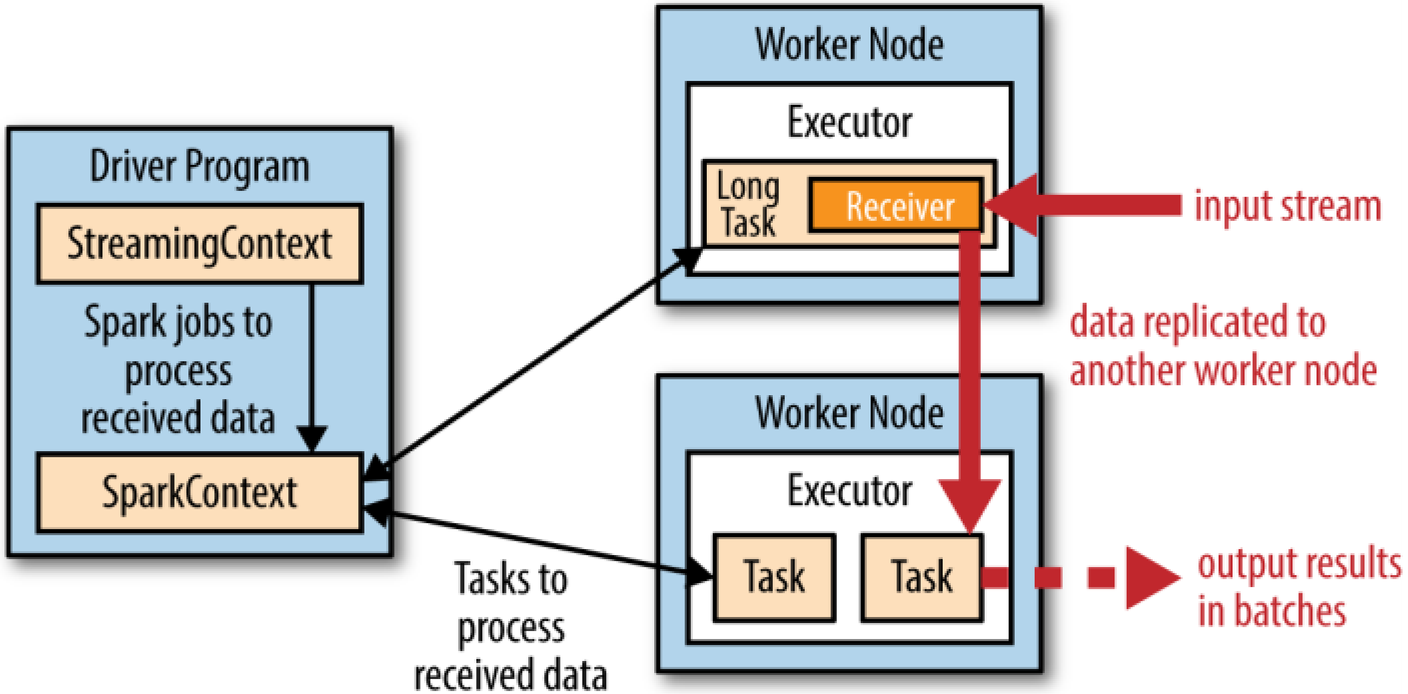


Output Operations

- **DStreams support output operations such as pprint()**
- **Output operations are similar to RDD actions in that they write data to an external system, but in Spark Streaming they run periodically on each time step, producing output in batches**

Execution of Spark Streaming

- **The execution of Spark Streaming within Spark's driver-worker components**



- **For each input source, Spark Streaming launches receivers, which are tasks running within the application's executors that collect data from the input source and save it as RDDs**

Fault Tolerance in Spark Streaming

- The input data is replicated (by default) to another executor for **fault tolerance:**
  - data is stored in-memory of the executors as cached RDDs.
- StreamingContext in the driver program:
  - periodically runs Spark jobs to process this data and
  - combines it with RDDs from previous time steps
- Spark Streaming fault-tolerance for DStreams:
  - as long as a copy of the input data is available, it re-computes any state derived from it using the lineage of the RDDs
    - i.e. by rerunning the operations used to process it
- Spark Streaming also includes a mechanism called **checkpointing** that saves state periodically to a reliable filesystem (HDFS or S3)
  - when recovering lost data Spark Streaming needs only to go to the last checkpoint (typically 5-10 batches)

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**
    - E.g., map(), filter(), and reduceByKey()
  - Although they look operating on the whole stream, internally each DStream is composed of multiple RDDs (batches), and each stateless transformation is applied separately **to each RDD**.
    - E.g., reduceByKey() will reduce data within each time step, but not across time steps.
  - **Stateful transformations use data or intermediate results from previous batches to compute the results of the current batch**
    - Transformations based on **sliding windows** and **tracking state** across time

Stateless Transformations

Function name	Purpose	Scala example	Signature of user-supplied function on `DStream[T]`
`map()`	Apply a function to each element in the DStream and return a DStream of the result.	`ds.map(x => x + 1)`	`f: (T) → U`
`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(" "))`	`f: T → Iterable[U]`
`filter()`	Return a DStream consisting of only elements that pass the condition passed to filter.	`ds.filter(x => x != 1)`	`f: T → Boolean`
`repartition()`	Change the number of partitions of the DStream.	`ds.repartition(10)`	N/A
`reduceByKey()`	Combine values with the same key in each batch.	`ds.reduceByKey((x, y) => x + y)`	`f: T, T → T`
`groupByKey()`	Group values with the same key in each batch.	`ds.groupByKey()`	N/A

Stateless Example

Example: log processing program that uses map() and reduceByKey() to count log events by IP address in each time step

```
# map() and reduceByKey() on DStream
# Run this example, and then copy the file to
directory = sys.argv[1]
print(directory)

# create DStream from text file
# Note: the spark streaming checks for any updates to this directory.
# So first, start this program, and then copy the log
# file logs/access_log.log to 'directory' location
log_data = ssc.textFileStream(directory)

# Parse each line using a utility class
access_log_dstream = log_data.map(ApacheAccessLog.parse_from_log_line)
    .filter(lambda parsed_line: parsed_line is not None)

# map each ip with value 1. So the stream becomes (ip, 1)
ip_dstream = access_log_dstream.map(lambda parsed_line: (parsed_line.ip, 1))

ip_count = ip_dstream.reduceByKey(lambda x,y: x+y)
ip_count.pprint(num = 30)
```

Stateless Transformation over Multiple DStreams

- **Stateless transformations can also combine data from multiple DStreams within each time step**
  - **For example, key/value DStreams have the same join-related transformations as RDDs**
    - cogroup(), join(), leftOuterJoin()

Join between two DStreams Example

- **Given data keyed by IP address, join the request count against the bytes transferred**

```
# Join two Dstreams

ip_bytes_dstream = access_log_dstream.map(lambda parsed_line:
    (parsed_line.ip, parsed_line.content_size))
ip_bytes_sum_dstream = ip_bytes_dstream.reduceByKey(lambda x,y: x+y)
ip_bytes_request_count_dstream = ip_count.join(ip_bytes_sum_dstream)
ip_bytes_request_count_dstream.pprint(num = 30)
```

- **Or merge the contents of two different DStreams using the union() operator as in regular Spark, or using StreamingContext.union() for multiple streams**

transform()

- **transform(): DStreams provide an advanced operator that lets you operate directly on the RDDs inside them:**
  - lets you provide any arbitrary RDD-to-RDD function to act on the DStream
- **Example application:**
  - to reuse batch processing code written for RDDs
  - For example, extractOutliers() function that acted on an RDD of log lines to produce an RDD of outliers (perhaps after running some statistics on the messages) can be reused within a transform()

```
def extractOutliers(rdd):
    """ Currently, no logic implemented. But you can specify any rdd logic here"""
    return rdd

transformed_access_log_dstream = access_log_dstream.transform(extractOutliers)
transformed_access_log_dstream.pprint()
```

Stateful Transformations

- **Stateful tranformations are operations on DStreams that track data across time**
  - some data from pervious batches is used to generate the results for a new batch
- **The two main types are:**
  - **windowed operations: act over sliding window of time periods**
  - **updateStateByKey(): track state across events for each key**
    - e.g. to build up an object representing each user session
- **Stateful tranformations require checkpointing to be enabled in your StreamingContext for fault tolerance**

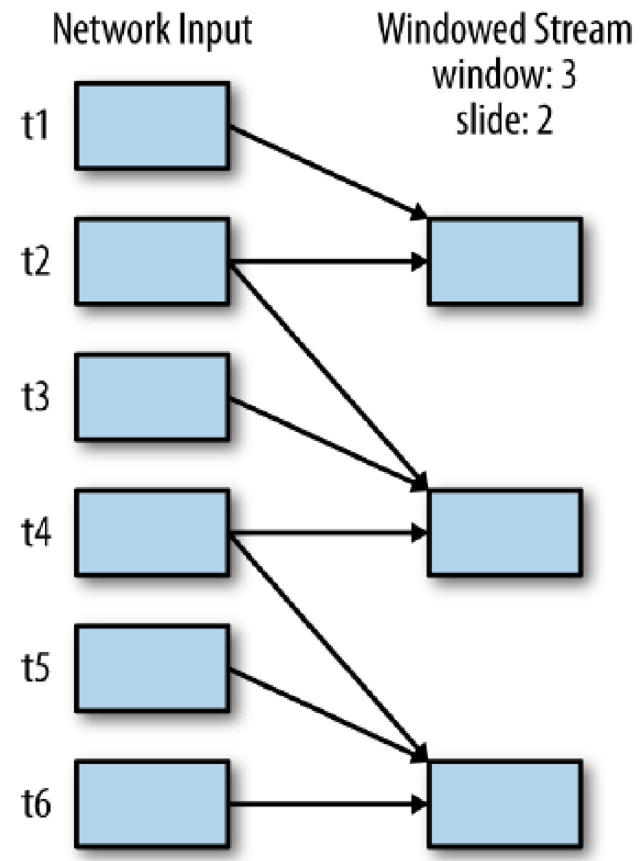
```
ssc.checkpoint("hdfs://...")
```

Windowed Transformations

- **Windowed operations compute the results in a longer time period than the StreamingContext's batch interval, by combining results from multiple batches.**
  - Used to keep track of the most common response codes, content sizes, and clients in a web server access log
  - Need two parameters, window duration and sliding duration, each is a multiple of the StreamingContext's batch interval

Windowing Parameters

- **The window duration controls how many previous batches are considered, namely windowDuration/batchInterval**
  - Eg: For a DStream with a batch interval of 10 seconds and a sliding window of the last 30 seconds (or last 3 batches), set the windowDuration to 30 seconds
- **The sliding duration, defaults to the batch interval, controls how frequently the new DStream computes results.**
  - For a DStream with a batch interval of 10 seconds, to compute out window only on every second batch, set the sliding interval to 20 seconds



every two time steps, compute a result over the 3 previous time steps

Request a window with window()

- **window(): returns a new DStream with the data for the requested window**
  - each RDD in the DStream resulting from window() will contain data from multiple batches that can be processed with count(), transform(), and so on.

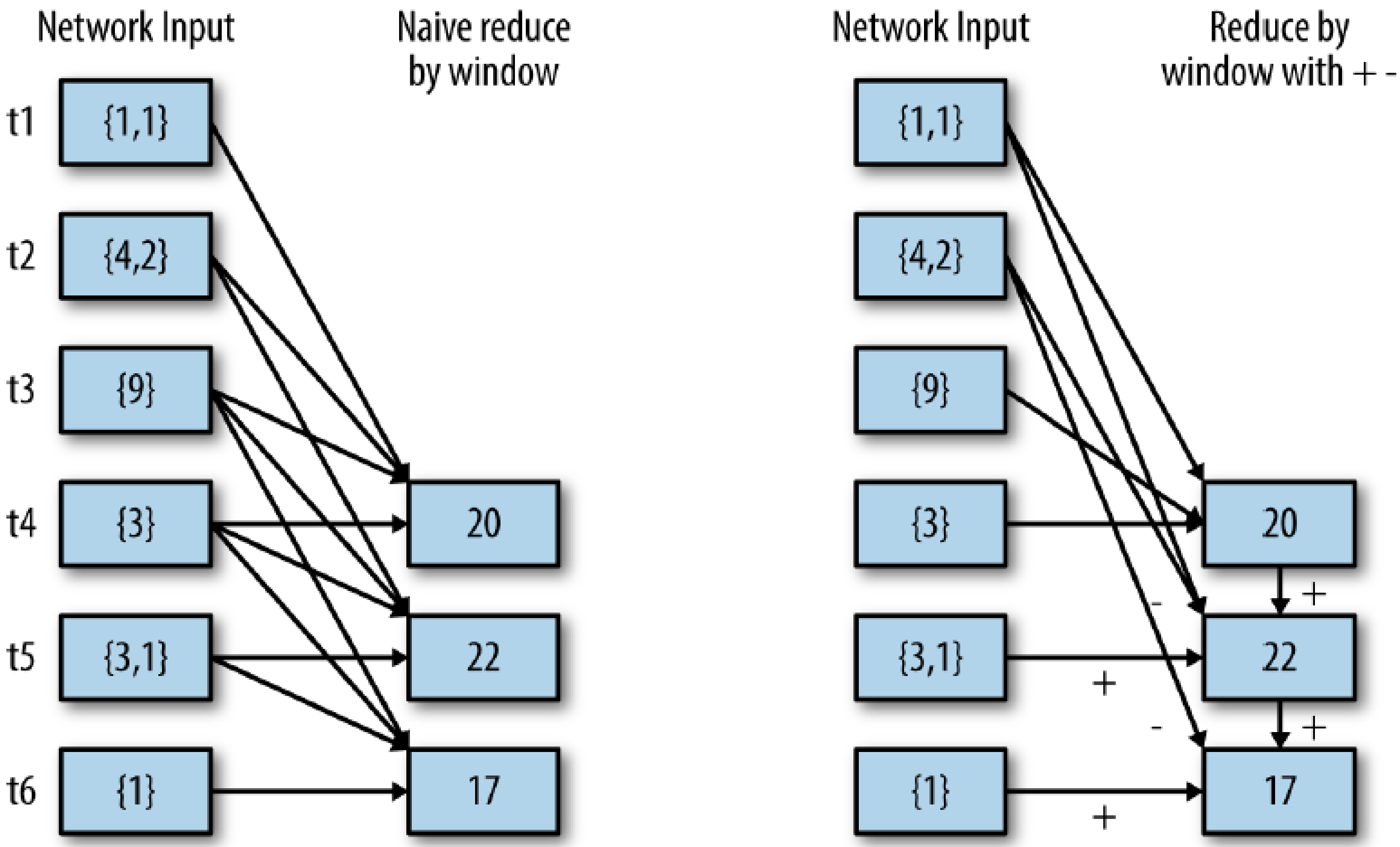
```
# How to use window()to count data over a window
access_logs_window = access_log_dstream.window(windowDuration = 6,
    slideDuration=4)
window_counts = access_logs_window.count()
print( " Window count: ")
window_counts.pprint()
```

Aggregate with reduceByWindow()

- **reduceByWindow() and reduceByKeyAndWindow() allow to perform each window more efficiently**
  - take a single reduce function (e.g., +) to run on the whole window
  - allow Spark to compute the reduction incrementally, by considering only data coming into the window.

Naïve vs Incremental reduceByWindow()

- **Difference between naïve reduceByWindow() and incremental reduceByWindow(), using an inverse function**



Example: reduceByWindow()

Example: Use reduceByWindow() and reduceByKeyAndWindow() to count visits by each IP address more efficiently

```
# Ip counts per window
ip_count_dstream = ip_dstream.reduceByKeyAndWindow(func = lambda x,y:
    x+y, invFunc = lambda x,y: x-y, windowDuration = 6, slideDuration=4)
ip_count_dstream.pprint(num=30)
```

Counting Data in DStreams

- **countByWindow()**: gives a DStream representing the number of elements in each window
- **countByValueAndWindow()**: gives a DStream with the counts for each value

```
# Windowed count operation
ip_dstream = access_log_dstream.map(lambda entry: entry.ip)
ip_address_request_count =
    ip_dstream.countByValueAndWindow(windowDuration = 6, slideDuration=4)
ip_address_request_count.pprint()
request_count =
    access_log_dstream.countByWindow(windowDuration = 6, slideDuration=4)
request_count.pprint()
```

updateStateByKey()

- **updateStateByKey() enables to maintain state across the batches in a DStream by providing access to a state variable for DStreams of key/value pairs:**
  - e.g. to track sessions as users visit a site
- **Given a DStream of (key, event) pairs, it lets you construct a new DStream of (key, state) pairs by taking a function that specifies how to update the state for each key given new events:**
  - e.g. in a web server log, events might be visits to the site, where the key is the user ID
  - Using updateStateByKey() we could track the last 10 pages each user visited. This list would be the "state" object, and will be updated as each event arrives

How to Use updateStateByKey()?

- Provide a function update(events, oldState) that takes in the events that have arrived for a key and its previous state, and returns a newState to store for it.
  - events is a list of events that arrived in the current batch (may be empty)
  - oldState is an optional state object, stored within an Option; it might be missing if there was no previous state for the key
  - newState returned by the function is also an Option; we can return an empty Option to specify that we want to delete the state
- The result of updateStateByKey() will be a new DStream that contains an RDD of (key, state) pairs on each time step

Example: updateStateByKey()

- **updateStateByKey() is used to keep a running count of the number of log messages with each HTTP response code.**
  - keys are the response codes
  - state is an integer representing each count
  - events are page views
- **Unlike window examples it keeps "infinitely growing" count since the beginning of the program**

```
# Running count of response codes using updateStateByKey() in Scala
# This basically maintains a running sum , rather than sum in windows

def state_full_sum(new_values, global_sum):
    return sum(new_values) + (global_sum or 0)

response_code_dstream = access_log_dstream.map(lambda entry: (entry.response_code, 1))
response_code_count_dstream = response_code_dstream.updateStateByKey(state_full_sum)
response_code_count_dstream.pprint()
```

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
- Much like lazy evaluation in RDDs, if no output operation is applied on a DStream and any of its descendants, then those DStreams will not be evaluated.
- If there are no output operations set in a StreamingContext, then the context will not start

Output Example

- **Spark Streaming has similar to save() operations for DStreams, each of which takes a directory to save files into and an optional suffix.**
- **The results of each batch are saved as subdirectories in the given directory, with the time and the suffix in the filename**

```
ip_address_request_count.saveAsTextFiles(prefix = "outputDir", suffix = "txt")
```

Acknowledgement

- Nirmesh Khandelwal, NCSU
- Anatoli Melechko, NCSU
- Learning Spark by Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia (http://shop.oreilly.com/product/0636920028512.do)