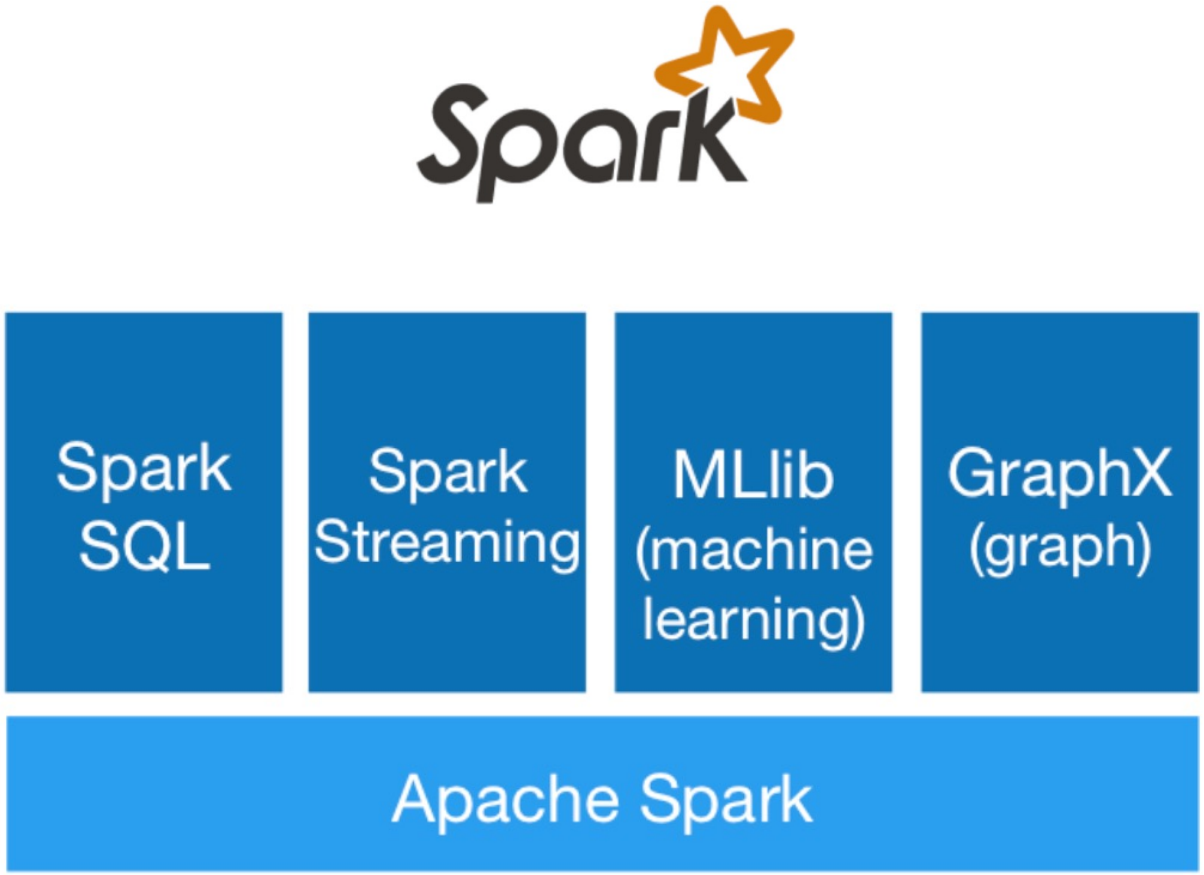
Spark Streaming Introduction

Spark Streaming makes it easy to build scalable fault-tolerant streaming applications



Spark Streaming makes it easy to build scalable fault-tolerant streaming applications

Ease of Use

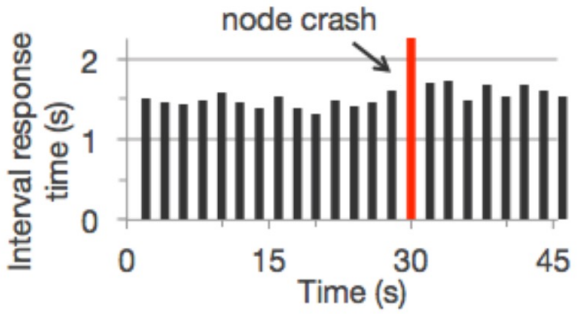
Build applications through high-level operators.

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

Counting tweets on a sliding window

Fault Tolerance

Stateful exactly-once semantics out of the box.



Spark Integration

Combine streaming with batch and interactive queries.

```
stream.join(historicCounts).filter {
  case (word, (curCount, oldCount)) =>
    curCount > oldCount
}
```

Find words with higher frequency than historic data

Alternatives

Process unbounded streams of data in real-time manner and used for real-time analytics, online machine learning, continuous computation, distributed RPC, ETL, and more.

Alternatives

Apache Flink® is an open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications.

Spark Streaming overview

- Spark Streaming can read data from HDFS, Flume, Kafka, Twitter, ZeroMQ and TCP sockets.
- You can also define your own custom data sources.

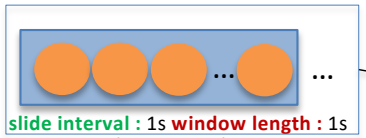
Spark Streaming API Programming model

Spark Streaming API for Python: **pyspark.streaming**

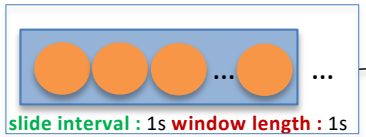
DStrem object represents the streaming data

DStrem type provides **transformation** and **actions** like RDD

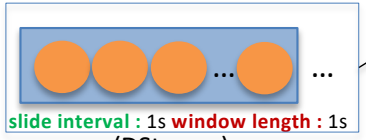
.window(4, 1)



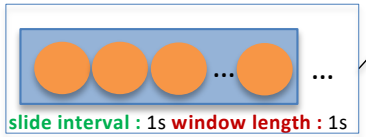
(DStream)



(DStream)



(DStream)



(DStream)

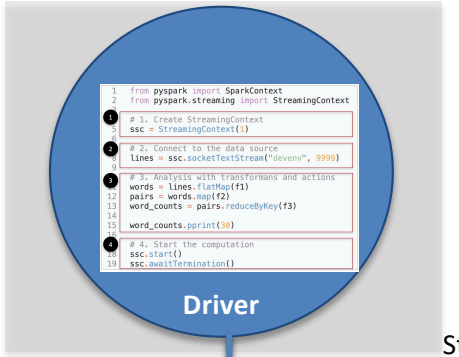
```
1  from pyspark import SparkContext
2  from pyspark.streaming import StreamingContext
3
1  # 1. Create StreamingContext
   ssc = StreamingContext(1)
6
2  # 2. Connect to the data source
8  lines = ssc.socketTextStream("devenv", 9999)
9
3  # 3. Analysis with transformans and actions
11 words = lines.flatMap(f1)
12 pairs = words.map(f2)
13 word_counts = pairs.reduceByKey(f3)
14
15 word_counts.pprint(30)
16
4  # 4. Start the computation
18 ssc.start()
19 ssc.awaitTermination()
```

Spark Streaming Run-time Execution Flow

Spark Streaming API for Python: **pyspark.sql**

```
1  from pyspark import SparkContext
2  from pyspark.streaming import StreamingContext
3
1  # 1. Create StreamingContext
5  ssc = StreamingContext(1)
6
2  # 2. Connect to the data source
8  lines = ssc.socketTextStream("devenv", 9999)
9
3  # 3. Analysis with transformans and actions
11 words = lines.flatMap(f1)
12 pairs = words.map(f2)
13 word_counts = pairs.reduceByKey(f3)
14
15 word_counts.pprint(30)
16
4  # 4. Start the computation
18 ssc.start()
19 ssc.awaitTermination()
```

spark-submit –master … network_wordcount.py



Standalone (/YARN/Mesos)

M

S

S

S

Executor

Executor

Executor

Data Source

T

Receiving and **store** data and **generate RDD**

(RDD)

(RDD)

T T T ..... T

Processing RDDs

Output Destination

Each rdd contain data of 1 second period

batch interval = 1 S

Processing data ona a micro match basis

slide interval = 1 s

window length = 1 s

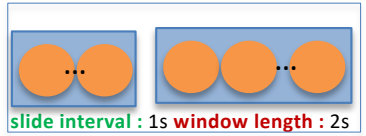
tN … t9 t8 t7 t6 t5 t4 t3 t2 t1 t0

Spark Streaming API Programming model

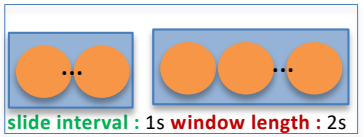
Spark Streaming API for Python: **pyspark.streaming**

DStrem object represents the streaming data

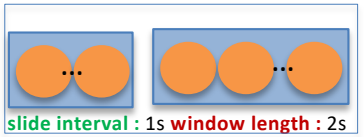
DStrem type provides **transformation** and **actions** like RDD



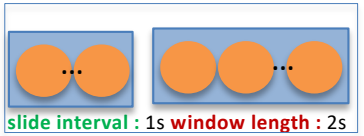
(DStream)



(DStream)



(DStream)



(DStream)

```
1  from pyspark import SparkContext
2  from pyspark.streaming import StreamingContext
3
   # 1. Create StreamingContext
   ssc = StreamingContext(1)
6
   # 2. Connect to the data source
   lines = ssc.socketTextStream("devenv", 9999).window(2, 1)
9
   # 3. Analysis with transformans and actions
   words = lines.flatMap(f1)
12 pairs = words.map(f2)
13 word_counts = pairs.reduceByKey(f3)
14
15 word_counts.pprint(30)
16
   # 4. Start the computation
18 ssc.start()
19 ssc.awaitTermination()
```

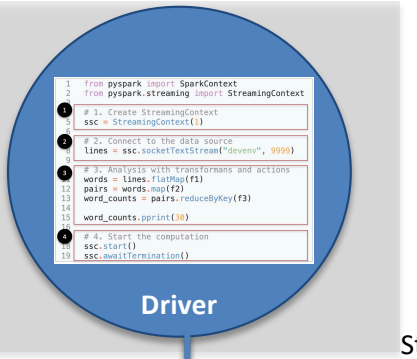
1
2
3
4

Spark Streaming Run-time Execution Flow

Spark Streaming API for Python: **pyspark.sql**

```
1  from pyspark import SparkContext
2  from pyspark.streaming import StreamingContext
3
1  # 1. Create StreamingContext
5  ssc = StreamingContext(1)
6
2  # 2. Connect to the data source
8  lines = ssc.socketTextStream("devenv", 9999)
9
3  # 3. Analysis with transformans and actions
11 words = lines.flatMap(f1)
12 pairs = words.map(f2)
13 word_counts = pairs.reduceByKey(f3)
14
15 word_counts.pprint(30)
16
4  # 4. Start the computation
18 ssc.start()
19 ssc.awaitTermination()
```

spark-submit –master … network_wordcount.py



Standalone (/YARN/Mesos)

M

S

S

S

Executor

Executor

Executor

Data Source

T

Receiving and **store** data and **generate RDD**

(RDD)

(RDD)

T T T ..... T

Processing RDDs

Output Destination

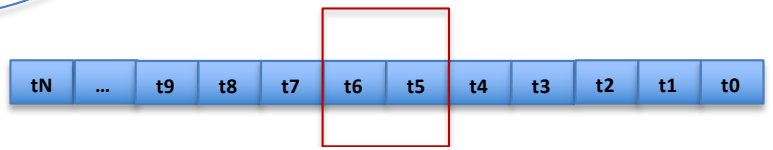
Each rdd contain data of 1 second period

batch interval = 1 S

Processing data ona a micro match basis

slide interval = 1 s

window length = 2 s



Dive-in Demo: Network Wordcount

```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

if __name__ == "__main__":
    # Create a local StreamingContext with two working thread and batch interval of 1 second
    sc = SparkContext()
    ssc = StreamingContext(sc, 1)

    # Create a DStream that will connect to hostname:port, like localhost:9999
    lines = ssc.socketTextStream("localhost", 9999)

    # Split each line into words
    words = lines.flatMap(lambda line: line.split(" "))

    # Count each word in each batch
    pairs = words.map(lambda word: (word, 1))
    word_counts = pairs.reduceByKey(lambda x, y: x + y)

    # Print the first ten elements of each RDD generated in this DStream to the console
    word_counts.pprint()

    ssc.start()             # Start the computation
    ssc.awaitTermination()  # Wait for the computation to terminate
```

Dive-in Demo: Network Wordcount

- Launch NC server

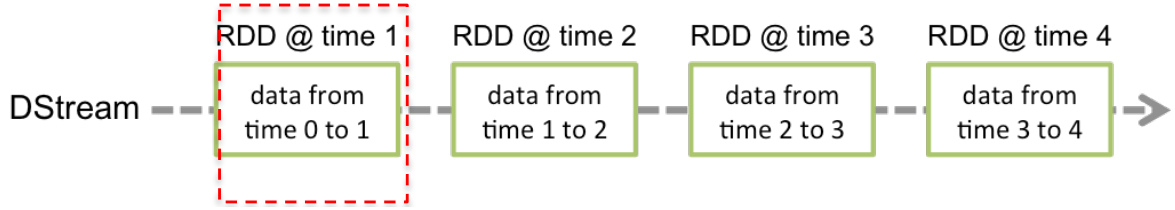
```
[cloudera@quickstart Desktop]$ nc -lk 9999
a b c
a b
```

- Launch the spark streaming job

StreamingContext

- The main entry point of all Spark Streaming functionality.
- Batch interval
  - Batch interval decides the batch computation interval and the size the each RDD in the input DStreams.
  - Should be based on the latency requirements and available cluster resources

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext(master, appName)
ssc = StreamingContext(sc, 1)
```

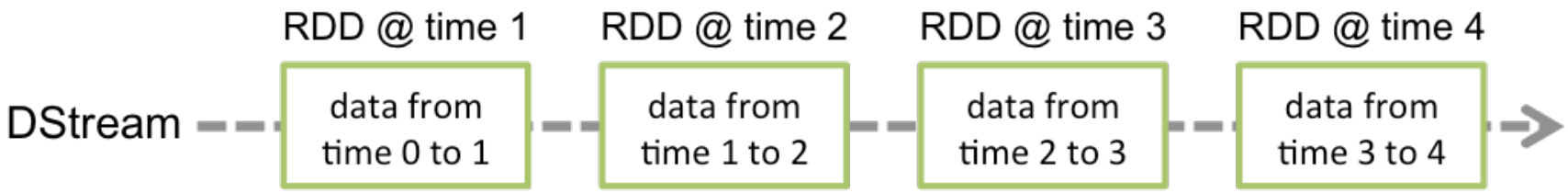


StreamingContext

- StreammingContext is also used when input DSteams are created.
  - Basic sources
    - Sources directly available in the StreamingContext API. e.g. file systems, socket connections, …
  - Advanced sources
    - Sources like Kafka, Flume, Kinesis, Twitter, etc. are available through extra utility classes.

DStream

- A Dstream (Discretized Stream) represents a continuous stream of data.
- Internally, a DStream is represented by a continuous series of RDDs.



Batch interval = 1s

DStreams and Receivers

- Every input DStream is associated with a Receiver object. (except file stream)
  - Input DStreams are DStreams representing the stream of input data received from streaming sources.
- The receiver receives the data from a source and stores it in Spark's memory for processing.

Figures and tables from the book Learning Spark

DStreams and Receivers

- Allocate enough cores to both run the receivers and process the received data.
  - i.e. the # 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.
- Creating multiple input Dstreams create multiple receivers.

Figures and tables from the book Learning Spark

File Streams

- File Streams
  - for reading data from files on any file system compatible with the HDFS API (HDFS, S3,…)

(for general hadoop files)

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

```
streamingContext.textFileStream(dataDirectory)
```
(for text)

  - It monitors the directory and process any files created in that directory
    - Files written in nested directories not supported
    - The files must have the same data format.
    - The files must be created by atomically *moving* or *renaming* them into the data directory. (e.g. hadoop fs -mv)
    - Once moved, the files must not be changed.

WordCount from a text folder

```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

if __name__ == "__main__":
    # Create a local StreamingContext with two working thread and batch interval of 1 second
    sc = SparkContext()
    ssc = StreamingContext(sc, 1)

    # Create a DStream that will monitor the textfolder
    lines = ssc.textFileStream("hdfs://localhost/user/cloudera/spark_streaming_101/textfolder")

    # Split each line into words
    words = lines.flatMap(lambda line: line.split(" "))

    # Count each word in each batch
    pairs = words.map(lambda word: (word, 1))
    word_counts = pairs.reduceByKey(lambda x, y: x + y)

    # Print the first ten elements of each RDD generated in this DStream to the console
    word_counts.pprint()

    ssc.start()             # Start the computation
    ssc.awaitTermination()  # Wait for the computation to terminate
```

Spark Streaming + Kafka Integration

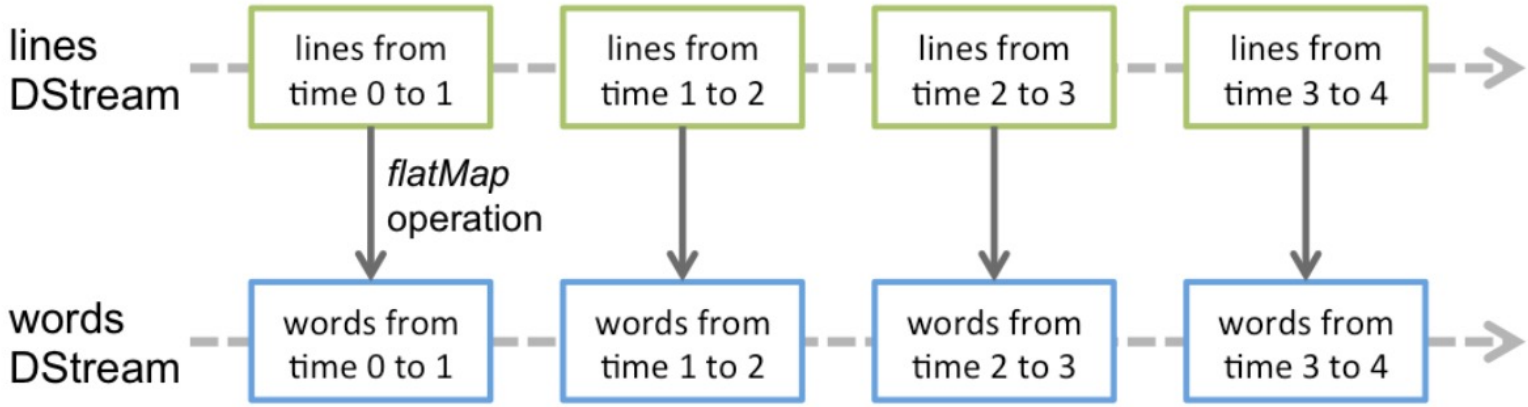
Receiver-based Approach

```
# Create a DStream from Kafaka "page_views" topic and apply window to the stream
page_view_stream = KafkaUtils.createStream(ssc, "localhost:2181", "consumer-group", {"page_views_logs_stream": 1})
lines = page_view_stream.map(lambda (k,v): v).window(60,10)
```

- Use a Receiver to receive the data. The Receiver is implemented using the Kafka high-level consumer API.
- Require external Kafka receiver libraries to use the API.

Dstream operations

- Any operation applied on a DStream translates to operations on the underlying RDDs



These underlying RDD transformations are computed by the Spark engine.

Common Transformation Methods on DStream

map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. **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.

Common Transformation Methods on DStream

groupByKey(*numPartitions=None*) Return a new DStream by applying groupByKey on each RDD.	
mapValues(*f*) Return a new DStream by applying a map function to the value of each key-value pairs in this DStream without changing the 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.

Common Transformation Methods on DStream

window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.
countByWindow(*windowLength*, *slideInterval*)	Return a sliding window count of elements in the stream.
reduceByWindow(*func*, *windowLength*, *slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative and commutative so that it can be computed correctly in parallel.
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.
reduceByKeyAndWindow(*func*, *invFunc*, *windowLength*, *slideInterval*, [*numTasks*])	A more efficient version of the above `reduceByKeyAndWindow()` where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enters the sliding window, and "inverse reducing" the old data that leaves the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable only to "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter *invFunc*). Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument. Note that checkpointing must be enabled for using this operation.
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.

Common Transformation Methods on DStream

`groupByKeyAndWindow`(*windowDuration*, *slideDuration*, *numPartitions=None*)

Return a new DStream by applying *groupByKey* over a sliding window. Similar to *DStream.groupByKey()*, but applies it over a sliding window.

Parameters:
- **windowDuration** – width of the window; must be a multiple of this DStream's batching interval
- **slideDuration** – sliding interval of the window (i.e., the interval after which the new DStream will generate RDDs); must be a multiple of this DStream's batching interval
- **numPartitions** – Number of partitions of each RDD in the new DStream.

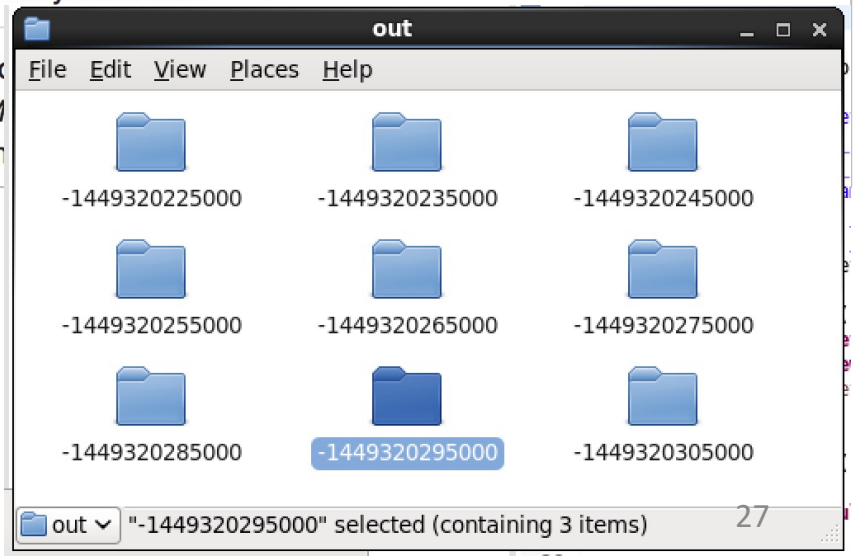
Window Network WordCount

```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

if __name__ == "__main__":
    # Create a local StreamingContext with two working thread and batch interval of 1 second
    sc = SparkContext()
    ssc = StreamingContext(sc, 1)

    # Create a DStream that will connect to hostname:port, like localhost:9999
    lines = ssc.socketTextStream("localhost", 9999)
    lines_windowed = lines.window(20,10) # process last 20 seconds of data, every 10 seconds

    # Split each line into words
    words = lines_windowed.flatMap(lambda line: line.split(" "))

    # Count each word in each batch
    pairs = words.map(lambda word: (word, 1))
    word_counts = pairs.reduceByKey(lambda x, y: x + y)

    # Print the first ten elements of each RDD generated in this DStream to the console
    word_counts.pprint()

    ssc.start()             # Start the computation
    ssc.awaitTermination()  # Wait for the computation to terminate
```

Output Operations on DStreams

Output operations trigger the actual execution of all the DStream transformations (similar to actions for RDDs)

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. **Python API** This is called **pprint()** in the Python API.
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]*".
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]*". **Python API** This is not available in the Python API.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as Hado based on *prefix* and *suffix*: "*prefix-TIM* **Python API** This is not available in th



Output Operations on Dstreams

foreachRDD(*func*) allows data to be sent out to external systems

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. Note that the function *func* is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.

Design Patterns for using foreachRDD

This does not work!

```
def sendRecord(rdd):
    connection = createNewConnection()  # executed at the driver
    rdd.foreach(lambda record: connection.send(record))
    connection.close()

dstream.foreachRDD(sendRecord)
```

Poor solution

```
def sendRecord(record):
    connection = createNewConnection()
    connection.send(record)
    connection.close()

dstream.foreachRDD(lambda rdd: rdd.foreach(sendRecord))
```

Design Patterns for using foreachRDD

Better solution

```
def sendPartition(iter):
    connection = createNewConnection()
    for record in iter:
        connection.send(record)
    connection.close()

dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
```

Persistence

- Use persist() method on a DStream will automatically persist every RDD of that DStream in memory.
  - This is useful if the data in the DStream will be computed multiple times (e.g., multiple operations on the same data)
- DStreams generated by window-based and updateStateByKey operations are automatically persisted in memory, without calling persist()

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

Custom Receiver

- If none suits you, you can create your own ones.
- A custom receiver must extend this abstract class *Receiver* by implementing two methods

```
class CustomReceiver(host: String, port: Int)
  extends Receiver[String](StorageLevel.MEMORY_AND_DISK_2) with Logging {

  def onStart() {
    // Start the thread that receives data over a connection
    new Thread("Socket Receiver") {
      override def run() { receive() }
    }.start()
  }

  def onStop() {
    // There is nothing much to do as the thread calling receive()
    // is designed to stop by itself if isStopped() returns false
  }
```

Custom Receiver (Reference)

- You can create your own ones if none suits your custom receiver.
- A custom receiver must extend this abstract class *Receiver* by implementing two methods.
  - onStart():
    - Things to do to start receiving data like creating receiving threads.
    - onStart() must not block indefinitely.
    - Inside the implementation, once the data is received, call Receiver's *store(data)* method to save the data to Spark.
  - onStop():
    - Things to do to stop receiving data.

A custom receiver example

A custom receiver that receives a stream of text over a socket. It treats \n delimited lines in the text stream as records and stores them with Spark

```
class CustomReceiver(host: String, port: Int)
  extends Receiver[String](StorageLevel.MEMORY_AND_DISK_2) with Logging {

  def onStart() {
    // Start the thread that receives data over a connection
    new Thread("Socket Receiver") {
      override def run() { receive() }
    }.start()
  }

  def onStop() {
    // There is nothing much to do as the thread calling receive()
    // is designed to stop by itself if isStopped() returns false
  }
```

A custom receiver example

```scala
/** Create a socket connection and receive data until receiver is stopped */
private def receive() {
  var socket: Socket = null
  var userInput: String = null
  try {
    // Connect to host:port
    socket = new Socket(host, port)

    // Until stopped or connection broken continue reading
    val reader = new BufferedReader(
      new InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8))
    userInput = reader.readLine()
    while(!isStopped && userInput != null) {
      store(userInput)
      userInput = reader.readLine()
    }
    reader.close()
    socket.close()

    // Restart in an attempt to connect again when server is active again
    restart("Trying to connect again")
  } catch {
    case e: java.net.ConnectException =>
      // restart if could not connect to server
      restart("Error connecting to " + host + ":" + port, e)
    case t: Throwable =>
      // restart if there is any other error
      restart("Error receiving data", t)
  }
}
```

Data can be stored inside Spark by calling store(data)

A custom receiver example

- Using the custom receiver in a Spark Streaming application

```
// Assuming ssc is the StreamingContext
val customReceiverStream = ssc.receiverStream(new CustomReceiver(host, port))
val words = lines.flatMap(_.split(" "))
...
```