

# SWE404/DMT413

## BIG DATA ANALYTICS

Lecture 12: Spark Streaming

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

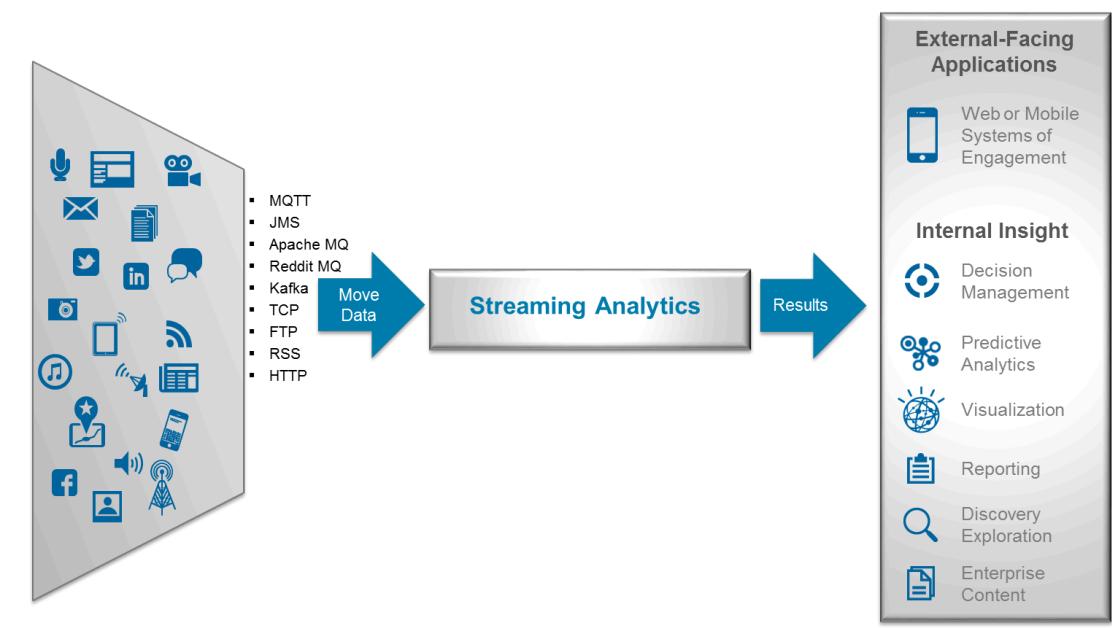
Office hour: 2pm-4pm Mon & Thur

# Outlines

- Data Stream
- Spark Streaming
- DStream
- Transformations on DStreams

# Data Stream

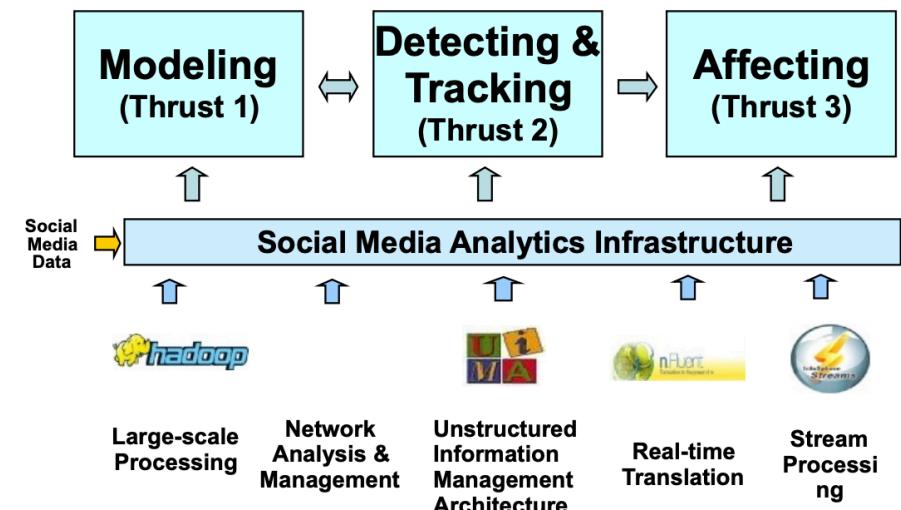
- A data stream is an unbounded sequence of data arriving continuously.
- Streaming divides continuously flowing input data into discrete units for further processing.
- Stream processing typically requires:
  - High volume data processing ability.
  - Real-time data processing with low latency.
  - Ability to efficiently recover from failures.



# Example: Social Media Stream Monitoring

					
<b>Live Monitoring</b> Monitoring real-time tweets on keyword: <input type="text" value="keyword"/>	<b>Trend Monitoring</b> Analyzing trend of conversations based on hashtags <a href="#">View trends »</a>	<b>Multimedia Monitoring</b> Recognizing visual content and analyzing visual sentiments <a href="#">View multimedia »</a>	<b>Geo Monitoring</b> Monitoring the places that people are sending out tweets <a href="#">View places »</a>	<b>Scope Identification</b> Define user-specified sets of keywords for monitoring and analytics <a href="#">Define scopes »</a>	<b>Concept Analytics</b> Analyzing statistics of groups based on time, topics, etc <a href="#">Concept searches »</a>
					
<b>Link Exploration</b> Visualizing relationships, discussion sequences and graphs <a href="#">View relationships »</a>	<b>Impact Prediction</b> Analyzing conversations and predicting their impact to business <a href="#">View impacts »</a>	<b>Story Detection</b> Detecting live developing stories on social media and their evolution <a href="#">View stories »</a>	<b>Person Analytics</b> Analyzing a person's personality, trustworthiness, etc. <a href="#">View person »</a>	<b>Target Discovery</b> Inspecting potential users for bot detection, marketing, or influencing <a href="#">Inspect targets »</a>	<b>Forensic Analytics</b> Analyzing retweet sequences and displaying anomalies <a href="#">View anomalies »</a>

**Approach: Modeling, Tracking and Affecting Information Dissemination in Context**



# Limitation of Stream Processing on Hadoop and Storm

- Using Hadoop (only) is suitable for processing batch data, but not quite suitable for processing stream data.
  - Reason: High latency.
- Using Storm+Hadoop can tremendously reduce the latency (up to millisecond level). However, there are other problems:
  - Tends to loose “state” in data processing if a node running Storm goes down.
  - Increases code size.
  - Other issues.
- Apache Spark Streaming can overcome these limitations.
  - But Storm still has lower latency than Spark.

# SPARK STREAMING

# Spark Streaming

- Spark Streaming is an extension of the core Spark API that enables *scalable, high-throughput, fault-tolerant* stream processing of live data streams.
- Input: Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets.
- Output: Processed data can be pushed out to filesystems, databases, and live dashboards.



# Spark Streaming

- Input data streams are divided into batches based on time intervals (of a few seconds or sub-second).
- Each batch of data as RDDs and processes them using RDD operations.
- Processed results are pushed out in batches.



# Streams and Batches

- Spark has provided a *unified engine* that natively supports both batch and streaming workloads.
- This 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.

# Goals of Spark Streaming

- Dynamic load balancing (small sized RDDs in DStreams).
- Fast failure recovery (“checkpointing” mechanism).
- Unification of batch, streaming and interactive analytics.
- Advanced analytics like machine learning and interactive SQL.
- Performance.

## Example: Network WordCount

- First, we import StreamingContext, which is the main entry point for all streaming functionality.
- We create a local StreamingContext with two execution threads, and batch interval of 1 second.

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

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```



## Example: Network WordCount

- Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).

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

- This lines DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space into words.

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



## Example: Network WordCount

- The words DStream is further mapped to a DStream of  $(\text{word}, 1)$  pairs, which is then reduced to get the frequency of words in each batch of data.
- Finally, `wordCounts.pprint()` will print a few of the counts generated every second.

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

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

## Example: Network WordCount

- Note that when these lines are executed, Spark Streaming only sets up the computation it will perform when it is started, and no real processing has started yet.
- To start the processing after all the transformations have been setup, we finally call

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



# Example: Network WordCount

- You will first need to run Netcat (a small utility found in most Unix-like systems) as a data server by call nc in terminal.
- Then, any lines typed in the terminal running the Netcat server will be counted and printed on screen every second.

```
# TERMINAL 1:  
# Running Net  
cat  
  
$ nc -lk 9999  
  
hello world  
  
...
```

```
# TERMINAL 2: RUNNING network_wordcount.py  
  
$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py local  
host 9999  
...  
-----  
Time: 2014-10-14 15:25:21  
-----  
(hello,1)  
(world,1)  
...
```

Results in Terminal

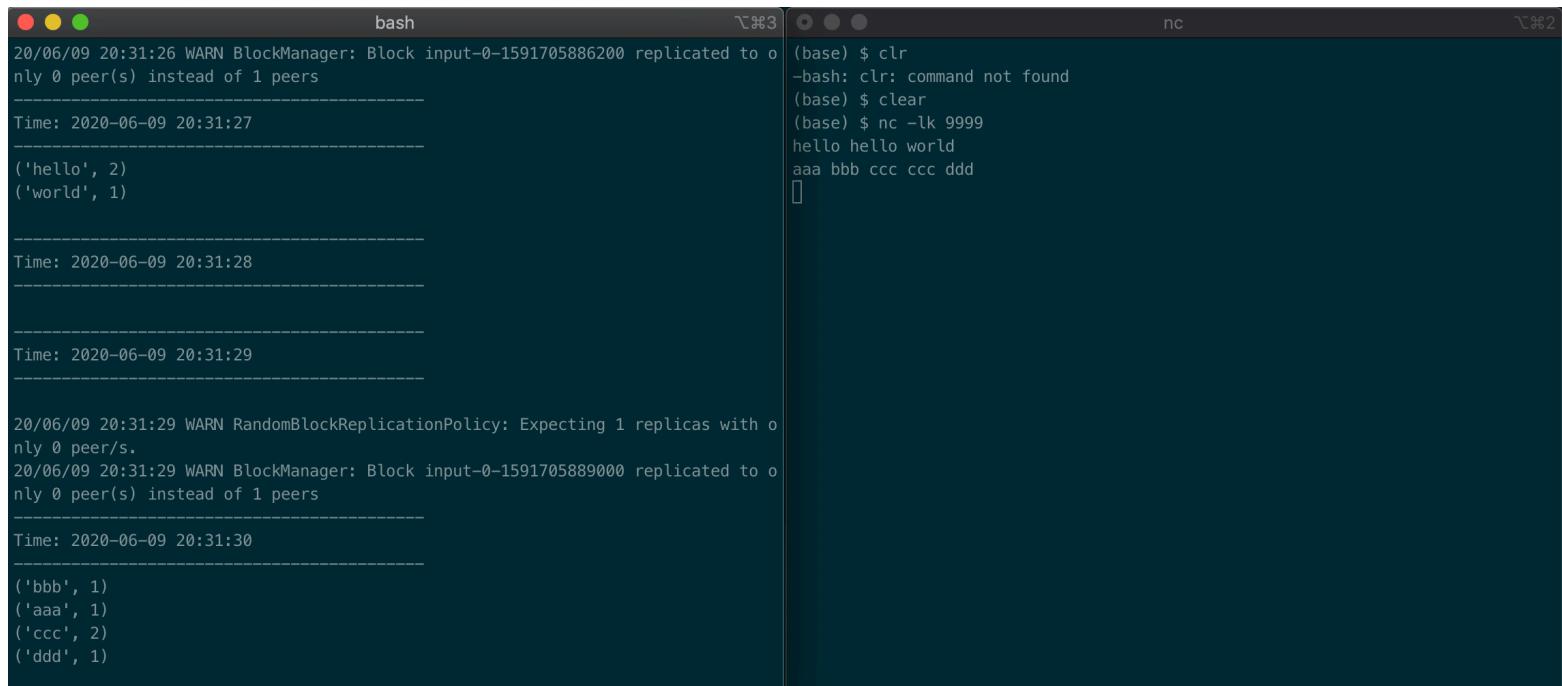
```
import sys  
  
from pyspark import SparkContext  
from pyspark.streaming import StreamingContext  
  
if __name__ == "__main__":  
    if len(sys.argv) != 3:  
        print("Usage: network_wordcount.py <hostname> <port>", file=sys.stderr)  
        sys.exit(-1)  
    sc = SparkContext(appName="PythonStreamingNetworkWordCount")  
    ssc = StreamingContext(sc, 1)  
  
    lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))  
    counts = lines.flatMap(lambda line: line.split(" "))\  
        .map(lambda word: (word, 1))\  
        .reduceByKey(lambda a, b: a+b)  
    counts.pprint()  
  
    ssc.start()  
    ssc.awaitTermination()
```

Python script of network wordcount

# Example: Network WordCount

- Jupyter Notebook doesn't support to run this example.
- Call spark-submit to run the python script on terminal.
  - If you linked PySpark with Jupyter Notebook previously, you should run

```
$unset PYSPARK_DRIVER_PYTHON
```



```
bash
20/06/09 20:31:26 WARN BlockManager: Block input-0-1591705886200 replicated to o
nly 0 peer(s) instead of 1 peers
-----
Time: 2020-06-09 20:31:27
-----
('hello', 2)
('world', 1)

-----
Time: 2020-06-09 20:31:28
-----
Time: 2020-06-09 20:31:29
-----
20/06/09 20:31:29 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with o
nly 0 peer/s.
20/06/09 20:31:29 WARN BlockManager: Block input-0-1591705889000 replicated to o
nly 0 peer(s) instead of 1 peers
-----
Time: 2020-06-09 20:31:30
-----
('bbb', 1)
('aaa', 1)
('ccc', 2)
('ddd', 1)
```

```
nc
(base) $ clr
-bash: clr: command not found
(base) $ clear
(base) $ nc -lk 9999
hello hello world
aaa bbb ccc ccc ddd
[
```

# DSTREAM

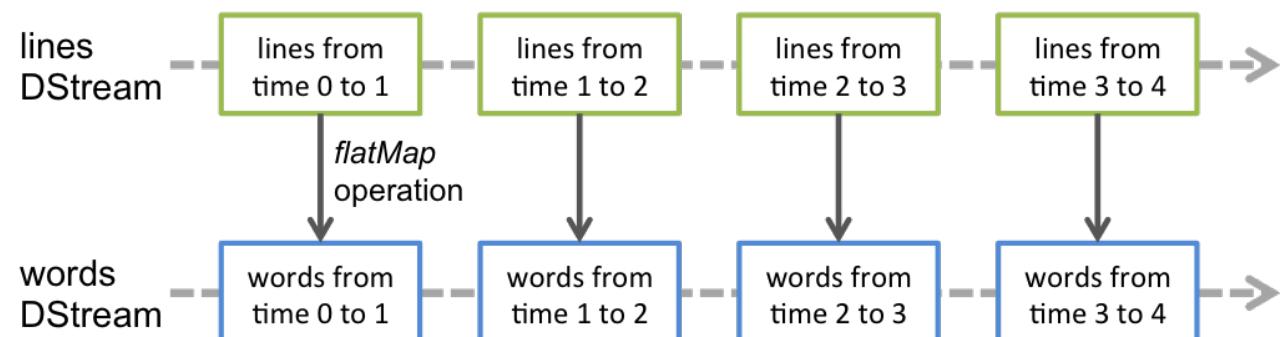
# DStream

- *Discretized Stream (DStream)* is the basic abstraction provided by Spark Streaming.
- It represents a continuous stream of data:
  - either the input data stream received from source,
  - or the processed data stream generated by transforming the input stream.
- Internally, a DStream is represented by a continuous series of RDDs.
- Each RDD in a DStream contains data from a certain interval.



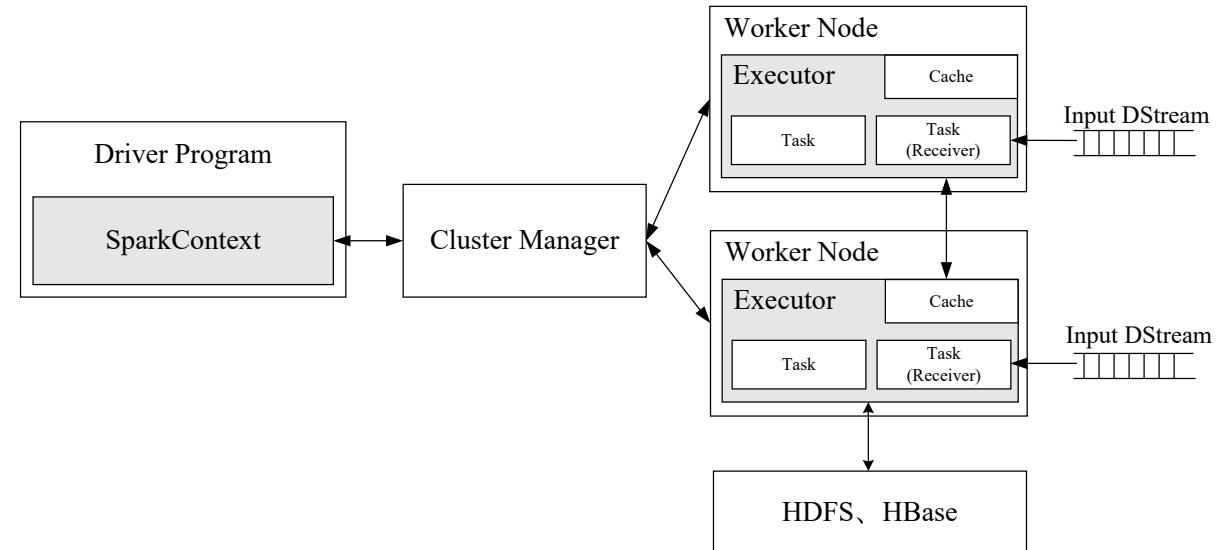
# DStream

- Any operation applied on a DStream translates to operations on the underlying RDDs.
- For example, the `flatMap` operation is applied on each RDD in the lines DStream to generate the RDDs of the words DStream.
- These underlying RDD transformations are computed by the Spark engine.
  - The DStream operations hide most of these details and provide the developer with a higher-level API for convenience.
  - Again, you don't need to care about how transformations are applied to streaming data.



# Input DStream

- Input DStream is a DStream representing the stream of input data from streaming source.
- A receiver object is associated with every input DStream object.
- Receivers receive the data from a source and stores it in Spark's memory for processing.
- Two types of built-in streaming sources:
  - Basic sources (file systems, and socket connections).
  - Advanced sources (Kafka, Flume, Kinesis).



# TRANSFORMATIONS ON DSTREAMS

# Transformations on DStreams

- Stateless transformations.
- Stateful transformations.
  - `updateStateByKey()` operation.
  - Window operations.

# Stateless Transformations

- Stateless transformations are similar to that of RDDs applied on every batch (meaning every RDD in a DStream).
  - Common RDD transformations: `map()`, `filter()`, `reduceByKey()` etc.
  - Key-Value RDD transformations: `cogroup()`, `join()`, `leftOuterJoin()` etc.
- Performing these operations on DStreams is equivalent to performing underlying RDD operations on each batch.
  - The only difference is that it is applied to a DStream or a DStream pair.

# Stateless Transformations

<b>map(func)</b>	Return a new DStream by passing each element of the source DStream through a function <i>func</i> .
<b>flatMap(func)</b>	Similar to map, but each input item can be mapped to 0 or more output items.
<b>filter(func)</b>	Return a new DStream by selecting only the records of the source DStream on which <i>func</i> returns true.
<b>repartition(<i>numPartitions</i>)</b>	Changes the level of parallelism in this DStream by creating more or fewer partitions.
<b>union(<i>otherStream</i>)</b>	Return a new DStream that contains the union of the elements in the source DStream and <i>otherDStream</i> .
<b>count()</b>	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
<b>reduce(func)</b>	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function <i>func</i> (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.

# Stateless Transformations

<b>countByValue()</b>	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.
<b>reduceByKey(func, [numTasks])</b>	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. <b>Note:</b> 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 <code>spark.default.parallelism</code> ) to do the grouping. You can pass an optional <code>numTasks</code> argument to set a different number of tasks.
<b>join(otherStream, [numTasks])</b>	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.
<b>cogroup(otherStream, [numTasks])</b>	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
<b>transform(func)</b>	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.

# Transform Operation

- The `transform` operation allows arbitrary RDD-to-RDD functions to be applied on a DStream.
- It can be used to apply any RDD operation that is not exposed in the DStream API.
  - For example, the functionality of joining every batch in a data stream with another dataset is not directly exposed in the DStream API.

```
spamInfoRDD = sc.pickleFile(...) # RDD containing spam information

# join data stream with spam information to do data cleaning
cleanedDStream = wordCounts.transform(lambda rdd: rdd.join(spamInfoRDD).filter(...))
```



# Stateful Transformations

- Stateful transformations are operations on DStreams that *track data across time*.
- Thus it makes use of some data from previous batches to generate the results for a new batch.
- Two main types:
  - `updateStateByKey()` operation.
  - Windowed operations.

## updateStateByKey( ) Operation

- The `updateStateByKey( )` operation allows you to maintain arbitrary state while continuously updating it with new information.
- To use this, you 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.

## updateStateByKey() Operation

- Let's illustrate this with an example. Say you want to maintain a running count of each word seen in a text data stream. Here, the running count is the state and it is an integer. We define the update function as:

```
def updateFunction(newValues, runningCount):  
    if runningCount is None:  
        runningCount = 0  
    return sum(newValues, runningCount) # add the new values with the previous running count to get the new count
```

- This is applied on a DStream containing words (say, the pairs DStream containing (word, 1) pairs in the earlier example).

```
runningCounts = pairs.updateStateByKey(updateFunction)
```

# Example: Stateful Network WordCount

Note that using `updateStateByKey()` requires the checkpoint directory to be configured.

```
sc = SparkContext(appName="PythonStreamingStatefulNetworkWordCount")
ssc = StreamingContext(sc, 1)
ssc.checkpoint("checkpoint")

# RDD with initial state (key, value) pairs
initialStateRDD = sc.parallelize([(u'hello', 1), (u'world', 1)])

def updateFunc(new_values, last_sum):
    return sum(new_values) + (last_sum or 0)

lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))
running_counts = lines.flatMap(lambda line: line.split(" "))\
    .map(lambda word: (word, 1))\
    .updateStateByKey(updateFunc, initialRDD=initialStateRDD)

running_counts.pprint()
```



# Example: Stateful Network WordCount

The image shows two terminal windows side-by-side. The left window, titled 'bash', displays the output of a Spark application. It shows three distinct time points: 2020-06-09 22:03:45, 2020-06-09 22:03:50, and 2020-06-09 22:03:55. At each time point, it prints tuples representing word counts: ('hello', 1), ('world', 1) at the first two points, and ('hello', 2), ('aaa', 1), ('world', 1) at the third. The right window, titled 'nc', shows the command 'nc -lk 9999' being run, which listens for incoming connections on port 9999. It then receives three lines of input: 'hello', 'aaa', and 'world'.

```
bash
20/06/09 22:03:39 WARN Utils: Service 'SparkUI' could not bind on port 4040. Att
empting port 4041.

Time: 2020-06-09 22:03:45
-----
('hello', 1)
('world', 1)

Time: 2020-06-09 22:03:50
-----
('hello', 2)
('world', 1)

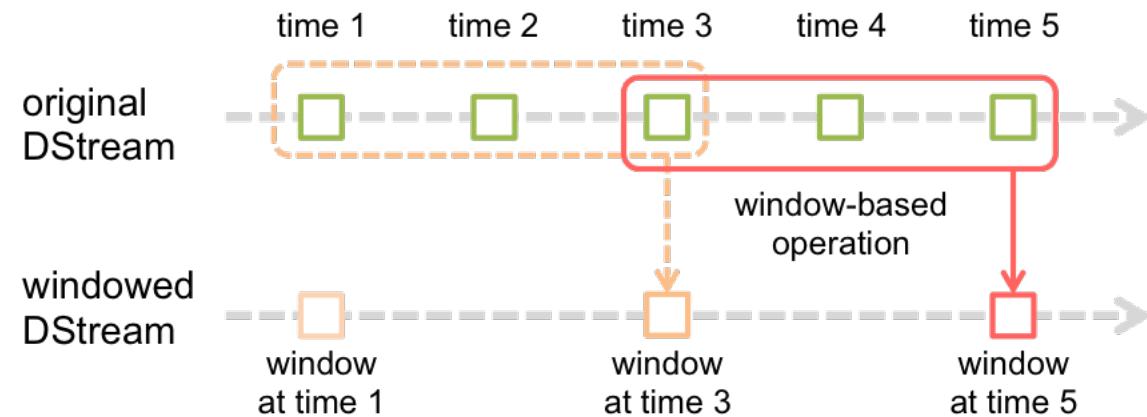
Time: 2020-06-09 22:03:55
-----
('hello', 2)
('aaa', 1)
('world', 1)

Time: 2020-06-09 22:04:00
-----
('hello', 2)
('aaa', 1)
('world', 2)

nc
(base) $ nc -lk 9999
hello
aaa
world
```

# Windowed Operations

- Windowed operations apply transformations over a sliding window of data.
- It is useful when you want to track a period (e.g. Tweeter topics in the latest 24 hours).
- Two parameters must be included:
  - **Window length:** The duration of the window (3 in the figure).
  - **Sliding interval:** The interval at which the window operation is performed (2 in the figure).



# Windowed Operations

- Now, we only want to keep the word counts over the last 30 seconds of data, in every 10 seconds period.
- This is done using the operation `reduceByKeyAndWindow()`.
  - It applies the `reduceByKey()` operation on the pairs DStream of `(word, 1)` pairs over the last 30 seconds of data.

```
# Reduce last 30 seconds of data, every 10 seconds
```

```
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)
```



# Windowed Operations

- Some of the common window operations are as follows.
- All of these operations take the said two parameters - `windowLength` and `slideInterval`.

<code>window(windowLength, slideInterval)</code>	Return a new DStream which is computed based on windowed batches of the source DStream.
<code>countByWindow(windowLength, slideInterval)</code>	Return a sliding window count of elements in the stream.
<code>reduceByWindow(func, windowLength, slideInterval)</code>	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using <code>func</code> . The function should be associative and commutative so that it can be computed correctly in parallel.
<code>reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])</code>	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 <code>func</code> over batches in a sliding window. <b>Note:</b> 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 <code>spark.default.parallelism</code> ) to do the grouping. You can pass an optional <code>numTasks</code> argument to set a different number of tasks.

# Windowed Operations

- Some of the common window operations are as follows.
- All of these operations take the said two parameters - `windowLength` and `slideInterval`.

---

**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.



# Example: Windowed Network WordCount

Refresh for  
every 5 seconds.

Only accumulate  
for the latest 10  
seconds.

```
emptying port 4041.  
-----  
Time: 2020-06-10 16:38:59  
  
-----  
Time: 2020-06-10 16:39:04  
('hello', 1)  
  
-----  
Time: 2020-06-10 16:39:09  
('hello', 1)  
('world', 1)  
  
-----  
Time: 2020-06-10 16:39:14  
('world', 1)  
  
-----  
Time: 2020-06-10 16:39:19  
  
-----  
Time: 2020-06-10 16:39:24  
  
-----
```

```
(base) $ nc -lk 9999  
hello  
world  
^C  
(base) $  
(base) $
```

windowLength=10 and slideInterval=5

# Spark Streaming Programming Model

- Create input DStream.
- Define operations (transformations and output) on DStreams.
- Use `streamingContext.start()` to start accepting and processing data.
- Use `streamingContext.awaitTermination()` to waiting for termination (manually or by incidents).
- You may use `streamingContext.stop()` to manually stop the data processing.

# Output Operations

- Output operations allow DStream's data to be pushed out to external systems like a database or a file systems.

Output Operation	Meaning
<code>print()</code>	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. <b>Python API</b> This is called <code>pprint()</code> in the Python API.
<code>saveAsTextFiles(prefix, [suffix])</code>	Save this DStream's contents as text files. The file name at each batch interval is generated based on <code>prefix</code> and <code>suffix</code> : " <code>prefix-TIME_IN_MS[.suffix]</code> ".
<code>saveAsObjectFiles(prefix, [suffix])</code>	Save this DStream's contents as SequenceFiles of serialized Java objects. The file name at each batch interval is generated based on <code>prefix</code> and <code>suffix</code> : " <code>prefix-TIME_IN_MS[.suffix]</code> ". <b>Python API</b> This is not available in the Python API.
<code>saveAsHadoopFiles(prefix, [suffix])</code>	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on <code>prefix</code> and <code>suffix</code> : " <code>prefix-TIME_IN_MS[.suffix]</code> ". <b>Python API</b> This is not available in the Python API.
<code>foreachRDD(func)</code>	The most generic output operator that applies a function, <code>func</code> , 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 <code>func</code> 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.

# Checkpointing

- A streaming application must **operate 24/7** and hence must be resilient to failures unrelated to the application logic.
- For this to be possible, Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures.
- There are two types of data that are checkpointed:
  - Metadata checkpointing: Saving of the information defining the streaming computation to fault-tolerant storage like HDFS. It is primarily needed for recovery from driver failures.
  - Data checkpointing: Saving of the generated RDDs to reliable storage. It is necessary for basic functioning if stateful transformations are used.

# When to Enable Checkpointing

- Checkpointing must be enabled for applications with any of the following requirements:
  - **Usage of stateful transformations:** If either `updateStateByKey` or `reduceByKeyAndWindow` is used in the application, then the checkpoint directory must be provided to allow for periodic RDD checkpointing.
  - **Recovering from failures of the driver running the application:** Metadata checkpoints are used to recover with progress information.
- Note that simple streaming applications without the aforementioned stateful transformations can be run without enabling checkpointing.
  - This is often acceptable and many run Spark Streaming applications in this way.
  - You can think that stateless transformations are less important to make it fault tolerant.

# How to Configure Checkpointing

- Checkpointing can be enabled by setting a directory in a fault-tolerant, reliable file system (e.g., HDFS) to save the checkpoint information. This is done by ——————
  - When the program is being started for the first time, it will create a new StreamingContext, set up all the streams and then call `start()`.
  - When the program is being restarted after failure, it will re-create a StreamingContext from the checkpoint data in the checkpoint directory.

```
# Function to create and setup a new StreamingContext
def functionToCreateContext():
    sc = SparkContext(...) # new context
    ssc = StreamingContext(...)
    lines = ssc.socketTextStream(...) # create DStreams
    ...
    ssc.checkpoint(checkpointDirectory) # set checkpoint directory
    return ssc

# Get StreamingContext from checkpoint data or create a new one
context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext)

# Do additional setup on context that needs to be done,
# irrespective of whether it is being started or restarted
context. ...

# Start the context
context.start()
context.awaitTermination()
```

# Conclusion

After this lecture, you should know:

- What is a data stream.
- What analytics solution can be made from data stream.
- How does Spark Streaming handle data stream.
- What is DStream.
- What is the difference between stateless and stateful transformations.

# Thank you!

## Reference:

- Spark Streaming Official Guide: <http://spark.apache.org/docs/latest/streaming-programming-guide.html>.

Acknowledgement: Thankfully acknowledge slide contents shared by Dr. Ye Luo.