Overview

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like `map`, `reduce`, `join` and `window`. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.

Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

This guide shows you how to start writing Spark Streaming programs with DStreams. You can write Spark Streaming programs in Scala, Java or Python (introduced in Spark 1.2), all of which are presented in this guide. You will find tabs throughout this guide that let you choose between code snippets of different languages.

Note: There are a few APIs that are either different or not available in Python. Throughout this guide, you will find the tag Python API highlighting these differences.

A Quick Example

Before we go into the details of how to write your own Spark Streaming program, let's take a quick look at what a simple Spark Streaming program looks like. Let's say we want to count the number of words in text data received from a data server listening on a TCP socket. All you need to do is as follows.

Scala | Java | Python

First, we import the names of the Spark Streaming classes and some implicit conversions from StreamingContext into our environment in order to add useful methods to other classes we need (like DStream). StreamingContext is the main entry point for all streaming functionality. We create a local StreamingContext with two execution threads, and a batch interval of 1 second.

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3

// Create a local StreamingContext with two working thread and batch interval of 1 second.
// The master requires 2 cores to prevent a starvation scenario.

val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

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
val 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 characters into words.

```
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

`flatMap` is a one-to-many DStream operation that creates a new DStream by generating multiple new records from each record in the source DStream. In this case, each line will be split into multiple words and the stream of words is represented as the `words` DStream. Next, we want to count these words.

```scala
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

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

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

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

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

The complete code can be found in the Spark Streaming example NetworkWordCount.

If you have already downloaded and built Spark, you can run this example as follows. You will first need to run Netcat (a small utility found in most Unix-like systems) as a data server by using

```
$ nc -lk 9999
```

Then, in a different terminal, you can start the example by using

Scala | **Java** | **Python**

```
$ ./bin/run-example streaming.NetworkWordCount localhost 9999
```

Then, any lines typed in the terminal running the netcat server will be counted and printed on screen every second. It will look something like the following.

Scala | **Java** | **Python**

```
# TERMINAL 1:
# Running Netcat

$ nc -lk 9999

hello world



...
```

```
# TERMINAL 2: RUNNING NetworkWordCount

$ ./bin/run-example streaming.NetworkWordCount localhost 9999
...
-------------------------------------------
Time: 1357008430000 ms
-------------------------------------------
(hello,1)
(world,1)
...
```

Basic Concepts

Next, we move beyond the simple example and elaborate on the basics of Spark Streaming.

Linking

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

Maven | **SBT**

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.12</artifactId>
    <version>2.4.6</version>
    <scope>provided</scope>
</dependency>
```

For ingesting data from sources like Kafka, Flume, and Kinesis that are not present in the Spark Streaming core API, you will have to add the corresponding artifact `spark-streaming-xyz_2.12` to the dependencies. For example, some of the common ones are as follows.

Source	Artifact
Kafka	spark-streaming-kafka-0-10_2.12
Flume	spark-streaming-flume_2.12
Kinesis	spark-streaming-kinesis-asl_2.12 [Amazon Software License]

For an up-to-date list, please refer to the Maven repository for the full list of supported sources and artifacts.

Initializing StreamingContext

To initialize a Spark Streaming program, a **StreamingContext** object has to be created which is the main entry point of all Spark Streaming functionality.

Scala | **Java** | **Python**

A StreamingContext object can be created from a SparkConf object.

```scala
import org.apache.spark._
import org.apache.spark.streaming._

val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

The `appName` parameter is a name for your application to show on the cluster UI. `master` is a Spark, Mesos, Kubernetes or YARN cluster URL, or a special **"local[*]"** string to run in local mode. In practice, when running on a cluster, you will not want to hardcode `master` in the program, but rather launch the application with `spark-submit` and receive it there. However, for local testing and unit tests, you can pass "local[*]" to run Spark Streaming in-process (detects the number of cores in the local system). Note that this internally creates a SparkContext (starting point of all Spark functionality) which can be accessed as `ssc.sparkContext`.

The batch interval must be set based on the latency requirements of your application and available cluster resources. See the Performance Tuning section for more details.

A `StreamingContext` object can also be created from an existing `SparkContext` object.

```
import org.apache.spark.streaming._

val sc = ...                // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

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

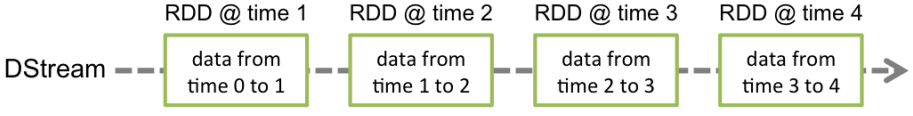
1. Define the input sources by creating input DStreams.
2. Define the streaming computations by applying transformation and output operations to DStreams.
3. Start receiving data and processing it using `streamingContext.start()`.
4. Wait for the processing to be stopped (manually or due to any error) using `streamingContext.awaitTermination()`.
5. The processing can be manually stopped using `streamingContext.stop()`.

Points to remember:

- 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.

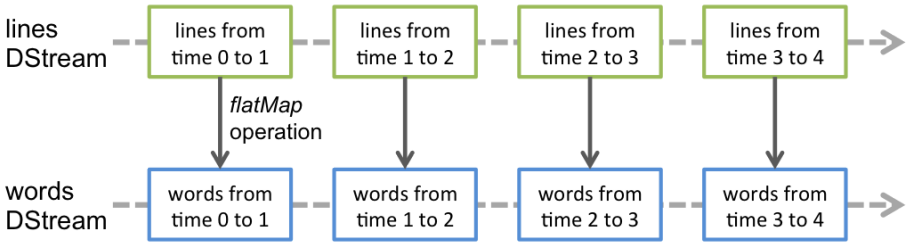
Discretized Streams (DStreams)

Discretized Stream or **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, which is Spark's abstraction of an immutable, distributed dataset (see Spark Programming Guide for more details). Each RDD in a DStream contains data from a certain interval, as shown in the following figure.



Any operation applied on a DStream translates to operations on the underlying RDDs. For example, in the earlier example of converting a stream of lines to words, the `flatMap` operation is applied on each RDD in the `lines` DStream to generate

the RDDs of the `words` DStream. This is shown in the following figure.



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. These operations are discussed in detail in later sections.

Input DStreams and 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 (except file stream, discussed later in this section) is associated with a **Receiver** (Scala doc, Java doc) 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, and socket connections.
- *Advanced sources*: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies as discussed in the linking section.

this one for kafka integration

We are going to discuss some of the sources present in each category later in this section.

Note that, if you want to receive multiple streams of data in parallel in your streaming application, you can create multiple input DStreams (discussed further in the Performance Tuning section). This will create multiple receivers which will simultaneously receive multiple data streams. But note that a Spark worker/executor is a long-running task, hence it occupies one of the cores allocated to the Spark Streaming application. Therefore, it is important to remember that a Spark Streaming application needs to be allocated enough cores (or threads, if running locally) to process the received data, as well as to run the receiver(s).

Points to remember

- 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 an 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 (see Spark Properties for information on how to set the master).

allocate enough core

- 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.

Basic Sources

We have already taken a look at the `ssc.socketTextStream(...)` in the quick example which creates a DStream from text data received over a TCP socket connection. Besides sockets, the StreamingContext API provides methods for creating

DStreams from files as input sources.

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 via `StreamingContext.fileStream[KeyClass, ValueClass, InputFormatClass]`.

File streams do not require running a receiver so there is no need to allocate any cores for receiving file data.

For simple text files, the easiest method is `StreamingContext.textFileStream(dataDirectory)`.

Scala | **Java** | **Python**

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

For text files

```
streamingContext.textFileStream(dataDirectory)
```

How Directories are Monitored

Spark Streaming will monitor the directory `dataDirectory` and process any files created in that directory.

- A simple directory can be monitored, such as `"hdfs://namenode:8040/logs/"`. All files directly under such a path will be processed as they are discovered.
- A POSIX glob pattern can be supplied, such as `"hdfs://namenode:8040/logs/2017/*"`. Here, the DStream will consist of all files in the directories matching the pattern. That is: it is a pattern of directories, not of files in directories.
- All files must be in the same data format.
- A file is considered part of a time period based on its modification time, not its creation time.
- Once processed, changes to a file within the current window will not cause the file to be reread. That is: *updates are ignored*.
- The more files under a directory, the longer it will take to scan for changes — even if no files have been modified.
- If a wildcard is used to identify directories, such as `"hdfs://namenode:8040/logs/2016-*"`, renaming an entire directory to match the path will add the directory to the list of monitored directories. Only the files in the directory whose modification time is within the current window will be included in the stream.
- Calling `FileSystem.setTimes()` to fix the timestamp is a way to have the file picked up in a later window, even if its contents have not changed.

Using Object Stores as a source of data

"Full" Filesystems such as HDFS tend to set the modification time on their files as soon as the output stream is created. When a file is opened, even before data has been completely written, it may be included in the `DStream` - after which updates to the file within the same window will be ignored. That is: changes may be missed, and data omitted from the stream.

To guarantee that changes are picked up in a window, write the file to an unmonitored directory, then, immediately after the output stream is closed, rename it into the destination directory. Provided the renamed file appears in the scanned destination directory during the window of its creation, the new data will be picked up.

In contrast, Object Stores such as Amazon S3 and Azure Storage usually have slow rename operations, as the data is actually copied. Furthermore, renamed object may have the time of the `rename()` operation as its modification time, so may not be considered part of the window which the original create time implied they were.

Careful testing is needed against the target object store to verify that the timestamp behavior of the store is consistent with that expected by Spark Streaming. It may be that writing directly into a destination directory is the appropriate strategy for streaming data via the chosen object store.

For more details on this topic, consult the Hadoop Filesystem Specification.

Streams based on Custom Receivers

DStreams can be created with data streams received through custom receivers. See the Custom Receiver Guide for more details.

Queue of RDDs as a Stream

For testing a Spark Streaming application with test data, one can also create a DStream based on a queue of RDDs, using `streamingContext.queueStream(queueOfRDDs)`. Each RDD pushed into the queue will be treated as a batch of data in the DStream, and processed like a stream.

For more details on streams from sockets and files, see the API documentations of the relevant functions in StreamingContext for Scala, JavaStreamingContext for Java, and StreamingContext for Python.

Advanced Sources

Python API As of Spark 2.4.6, out of these sources, Kafka, Kinesis and Flume are available in the Python API.

This category of sources require interfacing with external non-Spark libraries, some of them with complex dependencies (e.g., Kafka and Flume). Hence, 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.

Note that these advanced sources are not available in the Spark shell, hence applications based on these advanced sources cannot be tested in the shell. If you really want to use them in the Spark shell you will have to download the corresponding Maven artifact's JAR along with its dependencies and add it to the classpath.

Some of these advanced sources are as follows.

- **Kafka:** Spark Streaming 2.4.6 is compatible with Kafka broker versions 0.8.2.1 or higher. See the Kafka Integration Guide for more details.
- **Flume:** Spark Streaming 2.4.6 is compatible with Flume 1.6.0. See the Flume Integration Guide for more details.
- **Kinesis:** Spark Streaming 2.4.6 is compatible with Kinesis Client Library 1.2.1. See the Kinesis Integration Guide for more details.

Custom Sources

Python API This is not yet supported in Python.

Input DStreams can also be created out of custom data sources. All you have to do is implement a user-defined **receiver** (see next section to understand what that is) that can receive data from the custom sources and push it into Spark. See the Custom Receiver Guide for details.

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:

1. *Reliable Receiver* - A *reliable receiver* correctly sends acknowledgment to a reliable source when the data has been received and stored in Spark with replication.

2. *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.

The details of how to write a reliable receiver are discussed in the Custom Receiver Guide.

Transformations on DStreams

Similar to that of RDDs, transformations allow the data from the input DStream to be modified. DStreams support many of the transformations available on normal Spark RDD's. Some of the common ones are as follows.

Transformation	Meaning
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.
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.
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.
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.
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.

A few of these transformations are worth discussing in more detail.

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.

1. Define the state - The state can be an arbitrary data type.
2. 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.

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:

Scala | Java | Python

```scala
def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int] = {
    val newCount = ...  // add the new values with the previous running count to get the new count
    Some(newCount)
}
```

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

```scala
val runningCounts = pairs.updateStateByKey[Int](updateFunction _)
```

The update function will be called for each word, with `newValues` having a sequence of 1's (from the `(word, 1)` pairs) and the `runningCount` having the previous count.

Note that using `updateStateByKey` requires the checkpoint directory to be configured, which is discussed in detail in the checkpointing section.

Transform Operation

The `transform` operation (along with its variations like `transformWith`) 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. However, you can easily use `transform` to do this. This enables very powerful possibilities. For example, one can do real-time data cleaning by joining the input data stream with precomputed spam information (maybe generated with Spark as well) and then filtering based on it.

Scala | Java | Python

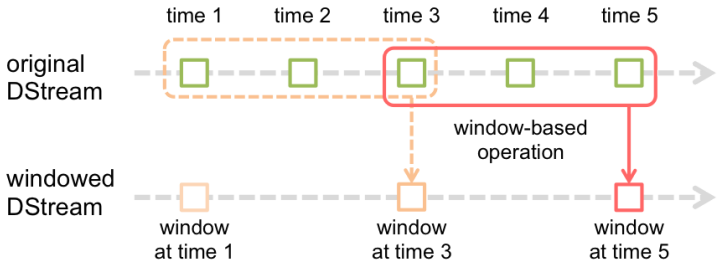
```scala
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...) // RDD containing spam information
```

```
val cleanedDStream = wordCounts.transform { rdd =>
  rdd.join(spamInfoRDD).filter(...) // join data stream with spam information to do data cleaning
  ...
}
```

Note that the supplied function gets called in every batch interval. This allows you to do time-varying RDD operations, that is, RDD operations, number of partitions, broadcast variables, etc. can be changed between batches.

Window Operations

Spark Streaming also provides *windowed computations*, which allow you to apply transformations over a sliding window of data. The following figure illustrates this sliding window.



As shown in the figure, every time the window *slides* over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream. In this specific case, the operation is applied over the last 3 time units of data, and slides by 2 time units. This shows that any window operation needs to specify two parameters.

- *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).

These two parameters must be multiples of the batch interval of the source DStream (1 in the figure).

Let's illustrate the window operations with an example. Say, you want to extend the earlier example by generating word counts over the last 30 seconds of data, every 10 seconds. To do this, we have to apply the `reduceByKey` operation on the `pairs` DStream of `(word, 1)` pairs over the last 30 seconds of data. This is done using the operation `reduceByKeyAndWindow`.

Scala | **Java** | **Python**

```
// Reduce last 30 seconds of data, every 10 seconds
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b), Seconds(30), Seconds(10))
```

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

Transformation	Meaning
window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.
countByWindow(*windowLength*,	Return a sliding window count of elements in the stream.

slideInterval)

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.

Join Operations

Finally, its worth highlighting how easily you can perform different kinds of joins in Spark Streaming.

Stream-stream joins

Streams can be very easily joined with other streams.

Scala | Java | Python

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)
```

Here, in each batch interval, the RDD generated by `stream1` will be joined with the RDD generated by `stream2`. You can also do `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin`. Furthermore, it is often very useful to do joins over windows of the streams. That is pretty easy as well.

Scala | Java | Python

```scala
val windowedStream1 = stream1.window(Seconds(20))
val windowedStream2 = stream2.window(Minutes(1))
val joinedStream = windowedStream1.join(windowedStream2)
```

Stream-dataset joins

This has already been shown earlier while explain `DStream.transform` operation. Here is yet another example of joining a windowed stream with a dataset.

Scala | Java | Python

```scala
val dataset: RDD[String, String] = ...
val windowedStream = stream.window(Seconds(20))...
val joinedStream = windowedStream.transform { rdd => rdd.join(dataset) }
```

In fact, you can also dynamically change the dataset you want to join against. The function provided to `transform` is evaluated every batch interval and therefore will use the current dataset that `dataset` reference points to.

The complete list of DStream transformations is available in the API documentation. For the Scala API, see DStream and PairDStreamFunctions. For the Java API, see JavaDStream and JavaPairDStream. For the Python API, see DStream.

Output Operations on DStreams

Output operations allow DStream's data to be pushed out to external systems like a database or a file systems. Since the output operations actually allow the transformed data to be consumed by external systems, they trigger the actual execution of all the DStream transformations (similar to actions for RDDs). Currently, the following output operations are defined:

Output Operation	Meaning
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 Hadoop files. 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.
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

`dstream.foreachRDD` is a powerful primitive that allows data to be sent out to external systems. However, it is important to understand how to use this primitive correctly and efficiently. Some of the common mistakes to avoid are as follows.

Often writing data to external system requires creating a connection object (e.g. TCP connection to a remote server) and using it to send data to a remote system. For this purpose, a developer may inadvertently try creating a connection object at the Spark driver, and then try to use it in a Spark worker to save records in the RDDs. For example (in Scala),

Scala | **Java** | **Python**

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection()  // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}
```

This is incorrect as this requires the connection object to be serialized and sent from the driver to the worker. Such connection objects are rarely transferable across machines. This error may manifest as serialization errors (connection object not serializable), initialization errors (connection object needs to be initialized at the workers), etc. The correct solution is to create the connection object at the worker.

However, this can lead to another common mistake - creating a new connection for every record. For example,

Scala | **Java** | **Python**

```
dstream.foreachRDD { rdd =>
  rdd.foreach { record =>
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  }
}
```

Typically, creating a connection object has time and resource overheads. Therefore, creating and destroying a connection object for each record can incur unnecessarily high overheads and can significantly reduce the overall throughput of the system. A better solution is to use `rdd.foreachPartition` - create a single connection object and send all the records in a RDD partition using that connection.

Scala | **Java** | **Python**

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    val connection = createNewConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    connection.close()
  }
}
```

This amortizes the connection creation overheads over many records.

Finally, this can be further optimized by reusing connection objects across multiple RDDs/batches. One can maintain a static pool of connection objects than can be reused as RDDs of multiple batches are pushed to the external system, thus further reducing the overheads.

Scala | **Java** | **Python**

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    // ConnectionPool is a static, lazily initialized pool of connections
    val connection = ConnectionPool.getConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    ConnectionPool.returnConnection(connection)  // return to the pool for future reuse
  }
}
```

Note that the connections in the pool should be lazily created on demand and timed out if not used for a while. This achieves the most efficient sending of data to external systems.

Other points to remember:

- DStreams are executed lazily by the output operations, just like RDDs are lazily executed by RDD actions. Specifically, RDD actions inside the DStream output operations force the processing of the received data. Hence, if your application does not have any output operation, or has output operations like `dstream.foreachRDD()` without any RDD action inside them, then nothing will get executed. The system will simply receive the data and discard it.
- By default, output operations are executed one-at-a-time. And they are executed in the order they are defined in the application.

DataFrame and SQL Operations

You can easily use DataFrames and SQL operations on streaming data. You have to create a SparkSession using the SparkContext that the StreamingContext is using. Furthermore, this has to done such that it can be restarted on driver failures. This is done by creating a lazily instantiated singleton instance of SparkSession. This is shown in the following example. It modifies the earlier word count example to generate word counts using DataFrames and SQL. Each RDD is converted to a DataFrame, registered as a temporary table and then queried using SQL.

Scala | **Java** | **Python**

```scala
/** DataFrame operations inside your streaming program */

val words: DStream[String] = ...

words.foreachRDD { rdd =>

  // Get the singleton instance of SparkSession
  val spark = SparkSession.builder.config(rdd.sparkContext.getConf).getOrCreate()
  import spark.implicits._

  // Convert RDD[String] to DataFrame
  val wordsDataFrame = rdd.toDF("word")

  // Create a temporary view
  wordsDataFrame.createOrReplaceTempView("words")

  // Do word count on DataFrame using SQL and print it
  val wordCountsDataFrame =
    spark.sql("select word, count(*) as total from words group by word")
  wordCountsDataFrame.show()
}
```

See the full source code.

You can also run SQL queries on tables defined on streaming data from a different thread (that is, asynchronous to the running StreamingContext). Just make sure that you set the StreamingContext to remember a sufficient amount of streaming data such that the query can run. Otherwise the StreamingContext, which is unaware of the any asynchronous SQL queries, will delete off old streaming data before the query can complete. For example, if you want to query the last batch, but your query can take 5 minutes to run, then call `streamingContext.remember(Minutes(5))` (in Scala, or equivalent in other languages).

See the DataFrames and SQL guide to learn more about DataFrames.

MLlib Operations

You can also easily use machine learning algorithms provided by MLlib. First of all, there are streaming machine learning algorithms (e.g. Streaming Linear Regression, Streaming KMeans, etc.) which can simultaneously learn from the streaming data as well as apply the model on the streaming data. Beyond these, for a much larger class of machine learning algorithms, you can learn a learning model offline (i.e. using historical data) and then apply the model online on streaming data. See the MLlib guide for more details.

Caching / Persistence

Similar to RDDs, DStreams also allow developers to persist the stream's data in memory. That is, using the `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). For window-based operations like `reduceByWindow` and `reduceByKeyAndWindow` and state-based operations like `updateStateByKey`, this is implicitly true. Hence, DStreams generated by window-based operations are automatically persisted in memory, without the developer calling `persist()`.

For input streams that receive data over the network (such as, Kafka, Flume, sockets, etc.), the default persistence level is set to replicate the data to two nodes for fault-tolerance.

Note that, unlike RDDs, the default persistence level of DStreams keeps the data serialized in memory. This is further discussed in the Performance Tuning section. More information on different persistence levels can be found in the Spark Programming Guide.

Checkpointing

A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.). 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. This is used to recover from failure of the node running the driver of the streaming application (discussed in detail later). Metadata includes:
  - *Configuration* - The configuration that was used to create the streaming application.
  - *DStream operations* - The set of DStream operations that define the streaming application.
  - *Incomplete batches* - Batches whose jobs are queued but have not completed yet.
- *Data checkpointing* - Saving of the generated RDDs to reliable storage. This is necessary in some *stateful* transformations that combine data across multiple batches. In such transformations, the generated RDDs depend on RDDs of previous batches, which causes the length of the dependency chain to keep increasing with time. To avoid such unbounded increases in recovery time (proportional to dependency chain), intermediate RDDs of stateful transformations are periodically *checkpointed* to reliable storage (e.g. HDFS) to cut off the dependency chains.

To summarize, metadata checkpointing is primarily needed for recovery from driver failures, whereas data or RDD checkpointing is necessary even 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` (with inverse function) 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. The recovery from driver failures will also be partial in that case (some received but unprocessed data may be lost). This is often acceptable and many run Spark Streaming applications in this way. Support for non-Hadoop environments is expected to improve in the future.

How to configure Checkpointing

Checkpointing can be enabled by setting a directory in a fault-tolerant, reliable file system (e.g., HDFS, S3, etc.) to which the checkpoint information will be saved. This is done by using `streamingContext.checkpoint(checkpointDirectory)`. This will allow you to use the aforementioned stateful transformations. Additionally, if you want to make the application recover from driver failures, you should rewrite your streaming application to have the following behavior.

- 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.

Scala **Java** **Python**

This behavior is made simple by using `StreamingContext.getOrCreate`. This is used as follows.

```
// Function to create and setup a new StreamingContext
def functionToCreateContext(): StreamingContext = {
  val ssc = new StreamingContext(...)   // new context
  val lines = ssc.socketTextStream(...) // create DStreams
  ...
  ssc.checkpoint(checkpointDirectory)   // set checkpoint directory
  ssc
}

// Get StreamingContext from checkpoint data or create a new one
val 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()
```

If the `checkpointDirectory` exists, then the context will be recreated from the checkpoint data. If the directory does not exist (i.e., running for the first time), then the function `functionToCreateContext` will be called to create a new context and set up the DStreams. See the Scala example RecoverableNetworkWordCount. This example appends the word counts of network data into a file.

In addition to using `getOrCreate` one also needs to ensure that the driver process gets restarted automatically on failure. This can only be done by the deployment infrastructure that is used to run the application. This is further discussed in the Deployment section.

Note that checkpointing of RDDs incurs the cost of saving to reliable storage. This may cause an increase in the processing time of those batches where RDDs get checkpointed. Hence, the interval of checkpointing needs to be set carefully. At small batch sizes (say 1 second), checkpointing every batch may significantly reduce operation throughput. Conversely, checkpointing too infrequently causes the lineage and task sizes to grow, which may have detrimental effects. For stateful transformations that require RDD checkpointing, the default interval is a multiple of the batch interval that is at least 10 seconds. It can be set by using `dstream.checkpoint(checkpointInterval)`. Typically, a checkpoint interval of 5 - 10 sliding intervals of a DStream is a good setting to try.

Accumulators, Broadcast Variables, and Checkpoints

Accumulators and Broadcast variables cannot be recovered from checkpoint in Spark Streaming. If you enable checkpointing and use Accumulators or Broadcast variables as well, you'll have to create lazily instantiated singleton instances for Accumulators and Broadcast variables so that they can be re-instantiated after the driver restarts on failure. This is shown in the following example.

Scala **Java** **Python**

```scala
object WordBlacklist {

  @volatile private var instance: Broadcast[Seq[String]] = null

  def getInstance(sc: SparkContext): Broadcast[Seq[String]] = {
    if (instance == null) {
      synchronized {
        if (instance == null) {
          val wordBlacklist = Seq("a", "b", "c")
          instance = sc.broadcast(wordBlacklist)
        }
      }
    }
    instance
  }
}

object DroppedWordsCounter {

  @volatile private var instance: LongAccumulator = null

  def getInstance(sc: SparkContext): LongAccumulator = {
    if (instance == null) {
      synchronized {
        if (instance == null) {
          instance = sc.longAccumulator("WordsInBlacklistCounter")
        }
      }
    }
    instance
  }
}

wordCounts.foreachRDD { (rdd: RDD[(String, Int)], time: Time) =>
  // Get or register the blacklist Broadcast
  val blacklist = WordBlacklist.getInstance(rdd.sparkContext)
  // Get or register the droppedWordsCounter Accumulator
  val droppedWordsCounter = DroppedWordsCounter.getInstance(rdd.sparkContext)
  // Use blacklist to drop words and use droppedWordsCounter to count them
  val counts = rdd.filter { case (word, count) =>
    if (blacklist.value.contains(word)) {
      droppedWordsCounter.add(count)
      false
    } else {
      true
    }
  }.collect().mkString("[", ", ", "]")
  val output = "Counts at time " + time + " " + counts
})
```

See the full source code.

Deploying Applications

This section discusses the steps to deploy a Spark Streaming application.

Requirements

To run a Spark Streaming applications, you need to have the following.

- *Cluster with a cluster manager* - This is the general requirement of any Spark application, and discussed in detail in the deployment guide.
- *Package the application JAR* - You have to compile your streaming application into a JAR. If you are using `spark-submit` to start the application, then you will not need to provide Spark and Spark Streaming in the JAR. However, if your application uses advanced sources (e.g. Kafka, Flume), then you will have to package the extra artifact they link to, along with their dependencies, in the JAR that is used to deploy the application. For example, an application using `KafkaUtils` will have to include `spark-streaming-kafka-0-10_2.12` and all its transitive dependencies in the application JAR.
- *Configuring sufficient memory for the executors* - Since the received data must be stored in memory, the executors must be configured with sufficient memory to hold the received data. Note that if you are doing 10 minute window operations, the system has to keep at least last 10 minutes of data in memory. So the memory requirements for the application depends on the operations used in it.
- *Configuring checkpointing* - If the stream application requires it, then a directory in the Hadoop API compatible fault-tolerant storage (e.g. HDFS, S3, etc.) must be configured as the checkpoint directory and the streaming application written in a way that checkpoint information can be used for failure recovery. See the checkpointing section for more details.
- *Configuring automatic restart of the application driver* - To automatically recover from a driver failure, the deployment infrastructure that is used to run the streaming application must monitor the driver process and relaunch the driver if it fails. Different cluster managers have different tools to achieve this.
  - *Spark Standalone* - A Spark application driver can be submitted to run within the Spark Standalone cluster (see cluster deploy mode), that is, the application driver itself runs on one of the worker nodes. Furthermore, the Standalone cluster manager can be instructed to *supervise* the driver, and relaunch it if the driver fails either due to non-zero exit code, or due to failure of the node running the driver. See *cluster mode* and *supervise* in the Spark Standalone guide for more details.
  - *YARN* - Yarn supports a similar mechanism for automatically restarting an application. Please refer to YARN documentation for more details.
  - *Mesos* - Marathon has been used to achieve this with Mesos.
- *Configuring write-ahead logs* - Since Spark 1.2, we have introduced *write-ahead logs* for achieving strong fault-tolerance guarantees. If enabled, all the data received from a receiver gets written into a write-ahead log in the configuration checkpoint directory. This prevents data loss on driver recovery, thus ensuring zero data loss (discussed in detail in the Fault-tolerance Semantics section). This can be enabled by setting the configuration parameter `spark.streaming.receiver.writeAheadLog.enable` to `true`. However, these stronger semantics may come at the cost of the receiving throughput of individual receivers. This can be corrected by running more receivers in parallel to increase aggregate throughput. Additionally, it is recommended that the replication of the received data within Spark be disabled when the write-ahead log is enabled as the log is already stored in a replicated storage system. This can be done by setting the storage level for the input stream to `StorageLevel.MEMORY_AND_DISK_SER`. While using S3 (or any file system that does not support flushing) for *write-ahead logs*, please remember to enable `spark.streaming.driver.writeAheadLog.closeFileAfterWrite` and `spark.streaming.receiver.writeAheadLog.closeFileAfterWrite`. See Spark Streaming Configuration for more details. Note that Spark will not encrypt data written to the write-ahead log when I/O encryption is enabled. If encryption of the write-ahead log data is desired, it should be stored in a file system that supports encryption natively.