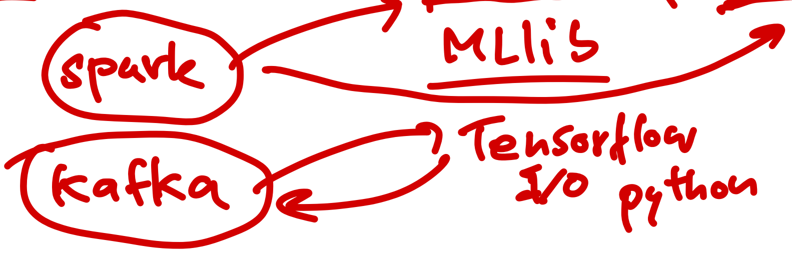
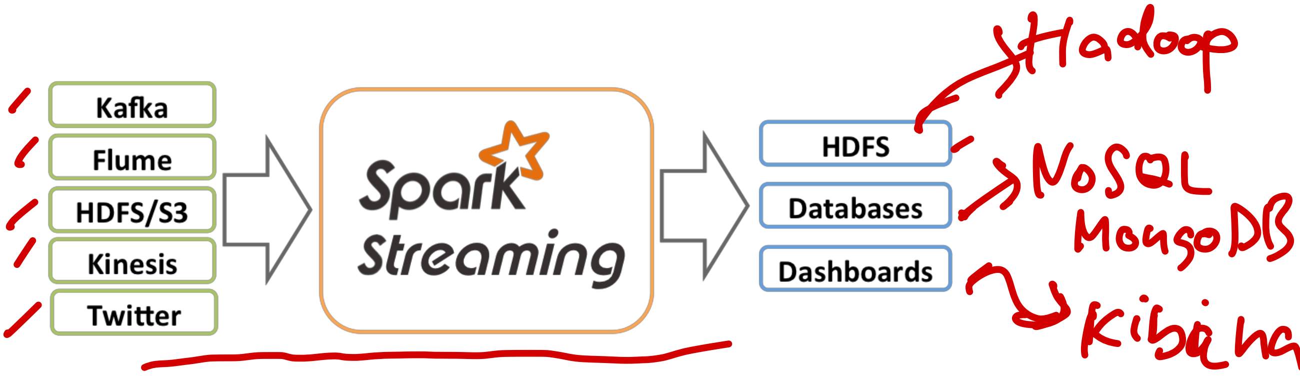
Spark Streaming Programming Guide

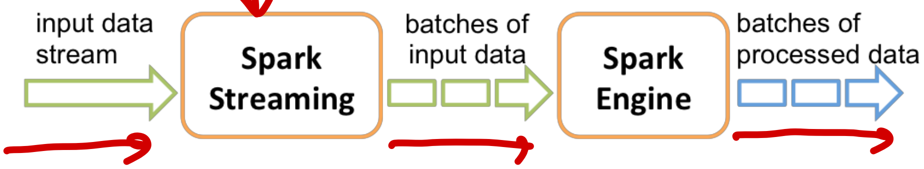
Overview

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, 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, 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 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)
```

① ②

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)
```

lines

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(" "))
```

par | body)

DS "a a b"

`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.

```
# 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 c
onsole
wordCounts.pprint()
```

(a,5)
DS (a,1)
DSt (a,1)
DSt (b,1)
(a,2)
(b,1)

flatmap
map
reducebyKey

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.pprint()` 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

```
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/spark-submit examples/src/main/python/streaming/network_wordcount.py
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.

```
#
TE
RM
IN
AL
1:
#
Ru
nn
in
g
Ne
```

Scala **Java** **Python**

```
tcat

$ nc -lk 9999

hello world



...
```

```
# TERMINAL 2: RUNNING network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py localhost 9999
...
-------------------------------------------
Time: 2014-10-14 15:25:21
-------------------------------------------
(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>3.0.1</version>
    <scope>provided</scope>
</dependency>
```

For ingesting data from sources like Kafka 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
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 SparkContext object.

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

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

The `appName` parameter is a name for your application to show on the cluster UI. `master` is a Spark, Mesos 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).

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.

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

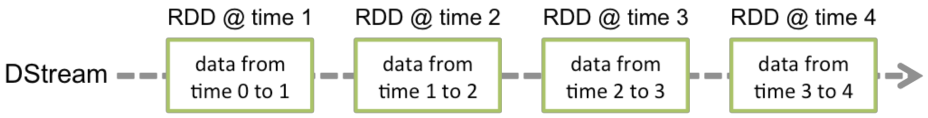
1. Define the input sources by creating input DStreams.
2. Define the streaming computations by applying transformation and output operations to DStreams.
3. Start receiving data and processing it using `streamingContext.start()`.
4. Wait for the processing to be stopped (manually or due to any error)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

Points to remember:

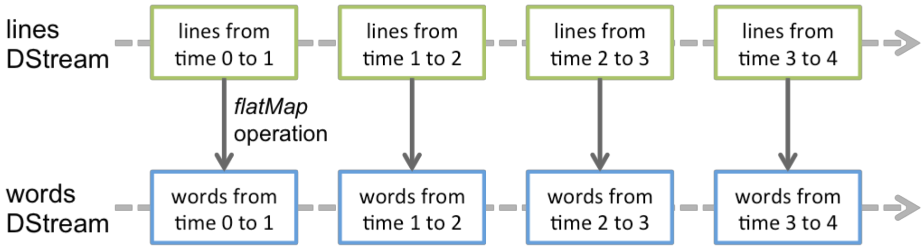
- Once a context has been started, no new streaming computations can be set up or added to it.
- Once a context has been stopped, it cannot be restarted.
- Only one StreamingContext can be active in a JVM at the same time.
- stop() on StreamingContext also stops the SparkContext. To stop only the StreamingContext, 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, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies as discussed in the linking section.

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, 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).
- 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**

`fileStream` is not available in the Python API; only `textFileStream` is available.

```
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 3.0.1, out of these sources, Kafka and Kinesis are available in the Python API.

This category of sources requires interfacing with external non-Spark libraries, some of them with complex dependencies (e.g., Kafka). 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 3.0.1 is compatible with Kafka broker versions 0.10 or higher. See the Kafka Integration Guide for more details.
- **Kinesis:** Spark Streaming 3.0.1 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) 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.

7.1 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**

```
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)
```

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. For the complete Python code, take a look at the example stateful_network_wordcount.py.

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

7.1 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**

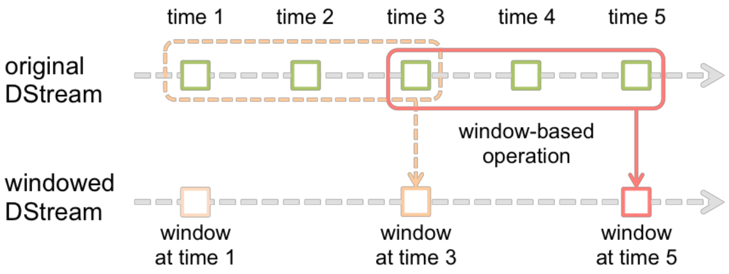
```
spamInfoRDD = sc.pickleFile(...)  # RDD containing spam information

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

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.

7.1 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
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x,
y: x - y, 30, 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*, *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.

7.1 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**

```
stream1 = ...
stream2 = ...
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**

```
windowedStream1 = stream1.window(20)
windowedStream2 = stream2.window(60)
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

```
dataset = ... # some RDD
windowedStream = stream.window(20)
joinedStream = windowedStream.transform(lambda 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

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

dstream.foreachRDD(sendRecord)
```

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**

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

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

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**

7.1

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

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

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**

```
def sendPartition(iter):
    # ConnectionPool is a static, lazily initialized pool of connections
    connection = ConnectionPool.getConnection()
    for record in iter:
        connection.send(record)
    # return to the pool for future reuse
    ConnectionPool.returnConnection(connection)

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

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**

```
# Lazily instantiated global instance of SparkSession
def getSparkSessionInstance(sparkConf):
    if ("sparkSessionSingletonInstance" not in globals()):
        globals()["sparkSessionSingletonInstance"] = SparkSession \
            .builder \
            .config(conf=sparkConf) \
            .getOrCreate()
    return globals()["sparkSessionSingletonInstance"]

...

# DataFrame operations inside your streaming program

words = ... # DStream of strings

def process(time, rdd):
    print("========= %s =========" % str(time))
    try:
        # Get the singleton instance of SparkSession
        spark = getSparkSessionInstance(rdd.context.getConf())

        # Convert RDD[String] to RDD[Row] to DataFrame
        rowRdd = rdd.map(lambda w: Row(word=w))
        wordsDataFrame = spark.createDataFrame(rowRdd)

        # Creates a temporary view using the DataFrame
        wordsDataFrame.createOrReplaceTempView("words")

        # Do word count on table using SQL and print it
        wordCountsDataFrame = spark.sql("select word, count(*) as total from words group by word")
        wordCountsDataFrame.show()
    except:
        pass

words.foreachRDD(process)
```

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.

7.1

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, 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.

7.1 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.

```python
# 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()
```

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 Python example recoverable_network_wordcount.py. This example appends the word counts of network data into a file.

You can also explicitly create a `StreamingContext` from the checkpoint data and start the computation by using `StreamingContext.getOrCreate(checkpointDirectory, None)`.

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**

```
def getWordBlacklist(sparkContext):
    if ("wordBlacklist" not in globals()):
        globals()["wordBlacklist"] = sparkContext.broadcast(["a", "b", "c"])
    return globals()["wordBlacklist"]

def getDroppedWordsCounter(sparkContext):
    if ("droppedWordsCounter" not in globals()):
        globals()["droppedWordsCounter"] = sparkContext.accumulator(0)
    return globals()["droppedWordsCounter"]

def echo(time, rdd):
    # Get or register the blacklist Broadcast
    blacklist = getWordBlacklist(rdd.context)
    # Get or register the droppedWordsCounter Accumulator
    droppedWordsCounter = getDroppedWordsCounter(rdd.context)

    # Use blacklist to drop words and use droppedWordsCounter to count them
    def filterFunc(wordCount):
        if wordCount[0] in blacklist.value:
            droppedWordsCounter.add(wordCount[1])
            False
        else:
            True

    counts = "Counts at time %s %s" % (time, rdd.filter(filterFunc).collect()
)

wordCounts.foreachRDD(echo)
```

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), 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.

- *Setting the max receiving rate* - If the cluster resources is not large enough for the streaming application to process data as fast as it is being received, the receivers can be rate limited by setting a maximum rate limit in terms of records / sec. See the configuration parameters `spark.streaming.receiver.maxRate` for receivers and `spark.streaming.kafka.maxRatePerPartition` for Direct Kafka approach. In Spark 1.5, we have introduced a feature called *backpressure* that eliminate the need to set this rate limit, as Spark Streaming automatically figures out the rate limits and dynamically adjusts them if the processing conditions change. This backpressure can be enabled by setting the configuration parameter `spark.streaming.backpressure.enabled` to `true`.

Upgrading Application Code

If a running Spark Streaming application needs to be upgraded with new application code, then there are two possible mechanisms.

7.1

- The upgraded Spark Streaming application is started and run in parallel to the existing application. Once the new one (receiving the same data as the old one) has been warmed up and is ready for prime time, the old one be can be brought down. Note that this can be done for data sources that support sending the data to two destinations (i.e., the earlier and upgraded applications).
- The existing application is shutdown gracefully (see `StreamingContext.stop(...)` or `JavaStreamingContext.stop(...)` for graceful shutdown options) which ensure data that has been received is completely processed before shutdown. Then the upgraded application can be started, which will start processing from the same point where the earlier application left off. Note that this can be done only with input sources that support source-side buffering (like Kafka) as data needs to be buffered while the previous application was down and the upgraded application is not yet up. And restarting from earlier checkpoint information of pre-upgrade code cannot be done. The checkpoint information essentially contains serialized Scala/Java/Python objects and trying to deserialize objects with new, modified classes may lead to errors. In this case, either start the upgraded app with a different checkpoint directory, or delete the previous checkpoint directory.

Monitoring Applications

7.1

Beyond Spark's monitoring capabilities, there are additional capabilities specific to Spark Streaming. When a StreamingContext is used, the Spark web UI shows an additional `Streaming` tab which shows statistics about running receivers (whether receivers are active, number of records received, receiver error, etc.) and completed batches (batch processing times, queueing delays, etc.). This can be used to monitor the progress of the streaming application.

The following two metrics in web UI are particularly important:

- *Processing Time* - The time to process each batch of data.
- *Scheduling Delay* - the time a batch waits in a queue for the processing of previous batches to finish.

If the batch processing time is consistently more than the batch interval and/or the queueing delay keeps increasing, then it indicates that the system is not able to process the batches as fast they are being generated and is falling behind. In that case, consider reducing the batch processing time.

The progress of a Spark Streaming program can also be monitored using the StreamingListener interface, which allows you to get receiver status and processing times. Note that this is a developer API and it is likely to be improved upon (i.e., more information reported) in the future.

Performance Tuning

Getting the best performance out of a Spark Streaming application on a cluster requires a bit of tuning. This section explains a number of the parameters and configurations that can be tuned to improve the performance of you application. At a high level, you need to consider two things:

1. Reducing the processing time of each batch of data by efficiently using cluster resources.
2. Setting the right batch size such that the batches of data can be processed as fast as they are received (that is, data processing keeps up with the data ingestion).

Reducing the Batch Processing Times

There are a number of optimizations that can be done in Spark to minimize the processing time of each batch. These have been discussed in detail in the Tuning Guide. This section highlights some of the most important ones.

Level of Parallelism in Data Receiving

Receiving data over the network (like Kafka, socket, etc.) requires the data to be deserialized and stored in Spark. If the data receiving becomes a bottleneck in the system, then consider parallelizing the data receiving. Note that each input DStream creates a single receiver (running on a worker machine) that receives a single stream of data. Receiving multiple data streams can therefore be achieved by creating multiple input DStreams and configuring them to receive different partitions of the data stream from the source(s). For example, a single Kafka input DStream receiving two topics of data can be split into two Kafka input streams, each receiving only one topic. This would run two receivers, allowing data to be received in parallel, thus increasing overall throughput. These multiple DStreams can be unioned together to create a single DStream. Then the transformations that were being applied on a single input DStream can be applied on the unified stream. This is done as follows.

Scala **Java** **Python**

```
numStreams = 5
kafkaStreams = [KafkaUtils.createStream(...) for _ in range (numStreams)]
unifiedStream = streamingContext.union(*kafkaStreams)
unifiedStream.pprint()
```

Another parameter that should be considered is the receiver's block interval, which is determined by the configuration parameter `spark.streaming.blockInterval`. For most receivers, the received data is coalesced together into blocks of data before storing inside Spark's memory. The number of blocks in each batch determines the number of tasks that will be used to process the received data in a map-like transformation. The number of tasks per receiver per batch will be approximately (batch interval / block interval). For example, block interval of 200 ms will create 10 tasks per 2 second batches. If the number of tasks is too low (that is, less than the number of cores per machine), then it will be inefficient as all available cores will not be used to process the data. To increase the number of tasks for a given batch interval, reduce the block interval. However, the recommended minimum value of block interval is about 50 ms, below which the task launching overheads may be a problem.

An alternative to receiving data with multiple input streams / receivers is to explicitly repartition the input data stream (using `inputStream.repartition(<number of partitions>)`). This distributes the received batches of data across the specified number of machines in the cluster before further processing.

For direct stream, please refer to Spark Streaming + Kafka Integration Guide

Level of Parallelism in Data Processing

Cluster resources can be under-utilized if the number of parallel tasks used in any stage of the computation is not high enough. For example, for distributed reduce operations like `reduceByKey` and `reduceByKeyAndWindow`, the default number of parallel tasks is controlled by the `spark.default.parallelism` configuration property. You can pass the level of parallelism as an argument (see `PairDStreamFunctions` documentation), or set the `spark.default.parallelism` configuration property to change the default.

Data Serialization

The overheads of data serialization can be reduced by tuning the serialization formats. In the case of streaming, there are two types of data that are being serialized.

- **Input data**: By default, the input data received through Receivers is stored in the executors' memory with StorageLevel.MEMORY_AND_DISK_SER_2. That is, the data is serialized into bytes to reduce GC overheads, and replicated for tolerating executor failures. Also, the data is kept first in memory, and spilled over to disk only if the memory is insufficient to hold all of the input data necessary for the streaming computation. This serialization obviously has overheads – the receiver must deserialize the received data and re-serialize it using Spark's serialization format.
- **Persisted RDDs generated by Streaming Operations**: RDDs generated by streaming computations may be persisted in memory. For example, window operations persist data in memory as they would be processed multiple times. However, unlike the Spark Core default of StorageLevel.MEMORY_ONLY, persisted RDDs generated by streaming computations are persisted with StorageLevel.MEMORY_ONLY_SER (i.e. serialized) by default to minimize GC overheads.

In both cases, using Kryo serialization can reduce both CPU and memory overheads. See the Spark Tuning Guide for more details. For Kryo, consider registering custom classes, and disabling object reference tracking (see Kryo-related configurations in the Configuration Guide).

In specific cases where the amount of data that needs to be retained for the streaming application is not large, it may be feasible to persist data (both types) as deserialized objects without incurring excessive GC overheads. For example, if you are using batch intervals of a few seconds and no window operations, then you can try disabling serialization in persisted data by explicitly setting the storage level accordingly. This would reduce the CPU overheads due to serialization, potentially improving performance without too much GC overheads.

Task Launching Overheads

If the number of tasks launched per second is high (say, 50 or more per second), then the overhead of sending out tasks to the slaves may be significant and will make it hard to achieve sub-second latencies. The overhead can be reduced by the following changes:

- **Execution mode**: Running Spark in Standalone mode or coarse-grained Mesos mode leads to better task launch times than the fine-grained Mesos mode. Please refer to the Running on Mesos guide for more details.

These changes may reduce batch processing time by 100s of milliseconds, thus allowing sub-second batch size to be viable.

Setting the Right Batch Interval

For a Spark Streaming application running on a cluster to be stable, the system should be able to process data as fast as it is being received. In other words, batches of data should be processed as fast as they are being generated. Whether this is true for an application can be found by monitoring the processing times in the streaming web UI, where the batch processing time should be less than the batch interval.

Depending on the nature of the streaming computation, the batch interval used may have significant impact on the data rates that can be sustained by the application on a fixed set of cluster resources. For example, let us consider the earlier WordCountNetwork example. For a particular data rate, the system may be able to keep up with reporting word counts every 2 seconds (i.e., batch interval of 2 seconds), but not every 500 milliseconds. So the batch interval needs to be set such that the expected data rate in production can be sustained.

A good approach to figure out the right batch size for your application is to test it with a conservative batch interval (say, 5-10 seconds) and a low data rate. To verify whether the system is able to keep up with the data rate, you can check the value of the end-to-end delay experienced by each processed batch (either look for “Total delay” in Spark driver log4j logs, or use the StreamingListener interface). If the delay is maintained to be comparable to the batch size, then system is stable. Otherwise, if the delay is continuously increasing, it means that the system is unable to keep up and it therefore unstable. Once you have an idea of a stable configuration, you can try increasing the data rate and/or reducing the batch size. Note that a momentary increase in the delay due to temporary data rate increases may be fine as long as the delay reduces back to a low value (i.e., less than batch size).

Memory Tuning

Tuning the memory usage and GC behavior of Spark applications has been discussed in great detail in the Tuning Guide. It is strongly recommended that you read that. In this section, we discuss a few tuning parameters specifically in the context of Spark Streaming applications.

The amount of cluster memory required by a Spark Streaming application depends heavily on the type of transformations used. For example, if you want to use a window operation on the last 10 minutes of data, then your cluster should have sufficient memory to hold 10 minutes worth of data in memory. Or if you want to use `updateStateByKey` with a large number of keys, then the necessary memory will be high. On the contrary, if you want to do a simple map-filter-store operation, then the necessary memory will be low.

In general, since the data received through receivers is stored with StorageLevel.MEMORY_AND_DISK_SER_2, the data that does not fit in memory will spill over to the disk. This may reduce the performance of the streaming application, and hence it is advised to provide sufficient memory as required by your streaming application. Its best to try and see the memory usage on a small scale and estimate accordingly.

Another aspect of memory tuning is garbage collection. For a streaming application that requires low latency, it is undesirable to have large pauses caused by JVM Garbage Collection.

There are a few parameters that can help you tune the memory usage and GC overheads:

- **Persistence Level of DStreams**: As mentioned earlier in the Data Serialization section, the input data and RDDs are by default persisted as serialized bytes. This reduces both the memory usage and GC overheads, compared to deserialized persistence. Enabling Kryo serialization further reduces serialized sizes and memory usage. Further reduction in memory usage can be achieved with compression (see the Spark configuration `spark.rdd.compress`), at the cost of CPU time.
- **Clearing old data**: By default, all input data and persisted RDDs generated by DStream transformations are automatically cleared. Spark Streaming decides when to clear the data based on the transformations that are used. For example, if you are using a window operation of 10 minutes, then Spark Streaming will keep around the last 10 minutes of data, and actively throw away older data. Data can be retained for a longer duration (e.g. interactively querying older data) by setting `streamingContext.remember`.
- **CMS Garbage Collector**: Use of the concurrent mark-and-sweep GC is strongly recommended for keeping GC-related pauses consistently low. Even though concurrent GC is known to reduce the overall processing throughput of the system, its use is still recommended to achieve more consistent batch processing times. Make sure you set the CMS GC on both the driver (using `--driver-java-options` in `spark-submit`) and the executors (using Spark configuration `spark.executor.extraJavaOptions`).
- **Other tips**: To further reduce GC overheads, here are some more tips to try.
    - Persist RDDs using the `OFF_HEAP` storage level. See more detail in the Spark Programming Guide.
    - Use more executors with smaller heap sizes. This will reduce the GC pressure within each JVM heap.

Important points to remember:

- A DStream is associated with a single receiver. For attaining read parallelism multiple receivers i.e.

multiple DStreams need to be created. A receiver is run within an executor. It occupies one core. Ensure that there are enough cores for processing after receiver slots are booked i.e. `spark.cores.max` should take the receiver slots into account. The receivers are allocated to executors in a round robin fashion.

- When data is received from a stream source, receiver creates blocks of data. A new block of data is generated every blockInterval milliseconds. N blocks of data are created during the batchInterval where N = batchInterval/blockInterval. These blocks are distributed by the BlockManager of the current executor to the block managers of other executors. After that, the Network Input Tracker running on the driver is informed about the block locations for further processing.
- An RDD is created on the driver for the blocks created during the batchInterval. The blocks generated during the batchInterval are partitions of the RDD. Each partition is a task in spark. blockInterval== batchinterval would mean that a single partition is created and probably it is processed locally.
- The map tasks on the blocks are processed in the executors (one that received the block, and another where the block was replicated) that has the blocks irrespective of block interval, unless non-local scheduling kicks in. Having bigger blockinterval means bigger blocks. A high value of `spark.locality.wait` increases the chance of processing a block on the local node. A balance needs to be found out between these two parameters to ensure that the bigger blocks are processed locally.
- Instead of relying on batchInterval and blockInterval, you can define the number of partitions by calling `inputDstream.repartition(n)`. This reshuffles the data in RDD randomly to create n number of partitions. Yes, for greater parallelism. Though comes at the cost of a shuffle. An RDD's processing is scheduled by driver's jobscheduler as a job. At a given point of time only one job is active. So, if one job is executing the other jobs are queued.
- If you have two dstreams there will be two RDDs formed and there will be two jobs created which will be scheduled one after the another. To avoid this, you can union two dstreams. This will ensure that a single unionRDD is formed for the two RDDs of the dstreams. This unionRDD is then considered as a single job. However, the partitioning of the RDDs is not impacted.
- If the batch processing time is more than batchinterval then obviously the receiver's memory will start filling up and will end up in throwing exceptions (most probably BlockNotFoundException). Currently, there is no way to pause the receiver. Using SparkConf configuration `spark.streaming.receiver.maxRate`, rate of receiver can be limited.

Fault-tolerance Semantics

In this section, we will discuss the behavior of Spark Streaming applications in the event of failures.

Background

To understand the semantics provided by Spark Streaming, let us remember the basic fault-tolerance semantics of Spark's RDDs.

1. An RDD is an immutable, deterministically re-computable, distributed dataset. Each RDD remembers the lineage of deterministic operations that were used on a fault-tolerant input dataset to create it.
2. If any partition of an RDD is lost due to a worker node failure, then that partition can be re-computed from the original fault-tolerant dataset using the lineage of operations.
3. Assuming that all of the RDD transformations are deterministic, the data in the final transformed RDD will always be the same irrespective of failures in the Spark cluster.

Spark operates on data in fault-tolerant file systems like HDFS or S3. Hence, all of the RDDs generated from the fault-tolerant data are also fault-tolerant. However, this is not the case for Spark Streaming as the data in most cases is received over the network (except when `fileStream` is used). To achieve the same fault-tolerance properties for all of the generated RDDs, the received data is replicated among multiple Spark executors in worker nodes in the cluster (default replication factor is 2). This leads to two kinds of data in the system that need to recovered in the event of failures:

1. *Data received and replicated* - This data survives failure of a single worker node as a copy of it exists on one of the other nodes.
2. *Data received but buffered for replication* - Since this is not replicated, the only way to recover this data is to get it again from the source.

Furthermore, there are two kinds of failures that we should be concerned about:

1. *Failure of a Worker Node* - Any of the worker nodes running executors can fail, and all in-memory data on those nodes will be lost. If any receivers were running on failed nodes, then their buffered data will be lost.
2. *Failure of the Driver Node* - If the driver node running the Spark Streaming application fails, then obviously the SparkContext is lost, and all executors with their in-memory data are lost.

With this basic knowledge, let us understand the fault-tolerance semantics of Spark Streaming.

Definitions

The semantics of streaming systems are often captured in terms of how many times each record can be processed by the system. There are three types of guarantees that a system can provide under all possible operating conditions (despite failures, etc.)

1. *At most once*: Each record will be either processed once or not processed at all.
2. *At least once*: Each record will be processed one or more times. This is stronger than *at-most once*

as it ensure that no data will be lost. But there may be duplicates.

3. *Exactly once*: Each record will be processed exactly once - no data will be lost and no data will be processed multiple times. This is obviously the strongest guarantee of the three.

Basic Semantics

In any stream processing system, broadly speaking, there are three steps in processing the data.

1. *Receiving the data*: The data is received from sources using Receivers or otherwise.
2. *Transforming the data*: The received data is transformed using DStream and RDD transformations.
3. *Pushing out the data*: The final transformed data is pushed out to external systems like file systems, databases, dashboards, etc.

If a streaming application has to achieve end-to-end exactly-once guarantees, then each step has to provide an exactly-once guarantee. That is, each record must be received exactly once, transformed exactly once, and pushed to downstream systems exactly once. Let's understand the semantics of these steps in the context of Spark Streaming.

1. *Receiving the data*: Different input sources provide different guarantees. This is discussed in detail in the next subsection.
2. *Transforming the data*: All data that has been received will be processed *exactly once*, thanks to the guarantees that RDDs provide. Even if there are failures, as long as the received input data is accessible, the final transformed RDDs will always have the same contents.
3. *Pushing out the data*: Output operations by default ensure *at-least once* semantics because it depends on the type of output operation (idempotent, or not) and the semantics of the downstream system (supports transactions or not). But users can implement their own transaction mechanisms to achieve *exactly-once* semantics. This is discussed in more details later in the section.

Semantics of Received Data

Different input sources provide different guarantees, ranging from *at-least once* to *exactly once*. Read for more details.

With Files

If all of the input data is already present in a fault-tolerant file system like HDFS, Spark Streaming can always recover from any failure and process all of the data. This gives *exactly-once* semantics, meaning all of the data will be processed exactly once no matter what fails.

With Receiver-based Sources

For input sources based on receivers, the fault-tolerance semantics depend on both the failure scenario and the type of receiver. As we discussed earlier, there are two types of receivers:

1. *Reliable Receiver* - These receivers acknowledge reliable sources only after ensuring that the received data has been replicated. If such a receiver fails, the source will not receive acknowledgment for the buffered (unreplicated) data. Therefore, if the receiver is restarted, the source will resend the data, and no data will be lost due to the failure.
2. *Unreliable Receiver* - Such receivers do *not* send acknowledgment and therefore *can* lose data when they fail due to worker or driver failures.

Depending on what type of receivers are used we achieve the following semantics. If a worker node fails, then there is no data loss with reliable receivers. With unreliable receivers, data received but not replicated can get lost. If the driver node fails, then besides these losses, all of the past data that was received and replicated in memory will be lost. This will affect the results of the stateful transformations.

To avoid this loss of past received data, Spark 1.2 introduced *write ahead logs* which save the received data to fault-tolerant storage. With the write-ahead logs enabled and reliable receivers, there is zero data loss. In terms of semantics, it provides an at-least once guarantee.

The following table summarizes the semantics under failures:

Deployment Scenario	**Worker Failure**	**Driver Failure**
Spark 1.1 or earlier, OR *Spark 1.2 or later without write-ahead logs*	Buffered data lost with unreliable receivers Zero data loss with reliable receivers At-least once semantics	Buffered data lost with unreliable receivers Past data lost with all receivers Undefined semantics
Spark 1.2 or later with write-ahead logs	Zero data loss with reliable receivers At-least once semantics	Zero data loss with reliable receivers and files At-least once semantics

With Kafka Direct API

In Spark 1.3, we have introduced a new Kafka Direct API, which can ensure that all the Kafka data is received by Spark Streaming exactly once. Along with this, if you implement exactly-once output operation, you can achieve end-to-end exactly-once guarantees. This approach is further discussed in the Kafka Integration Guide.

Semantics of output operations

Output operations (like `foreachRDD`) have *at-least once* semantics, that is, the transformed data may get written to an external entity more than once in the event of a worker failure. While this is acceptable for saving to file systems using the `saveAs***Files` operations (as the file will simply get overwritten with the same data), additional effort may be necessary to achieve exactly-once semantics. There are two approaches.

- *Idempotent updates*: Multiple attempts always write the same data. For example, `saveAs***Files` always writes the same data to the generated files.
- *Transactional updates*: All updates are made transactionally so that updates are made exactly once atomically. One way to do this would be the following.
    - Use the batch time (available in `foreachRDD`) and the partition index of the RDD to create an identifier. This identifier uniquely identifies a blob data in the streaming application.
    - Update external system with this blob transactionally (that is, exactly once, atomically) using the identifier. That is, if the identifier is not already committed, commit the partition data and the identifier atomically. Else, if this was already committed, skip the update.

    ```
    dstream.foreachRDD { (rdd, time) =>
      rdd.foreachPartition { partitionIterator =>
        val partitionId = TaskContext.get.partitionId()
        val uniqueId = generateUniqueId(time.milliseconds, partitionId)
        // use this uniqueId to transactionally commit the data in partitionIterator
      }
    }
    ```

Where to Go from Here

- Additional guides
    - Kafka Integration Guide
    - Kinesis Integration Guide
    - Custom Receiver Guide
- Third-party DStream data sources can be found in Third Party Projects

- API documentation
    - Scala docs
        - StreamingContext and DStream
        - KafkaUtils, KinesisUtils,
    - Java docs
        - JavaStreamingContext, JavaDStream and JavaPairDStream
        - KafkaUtils, KinesisUtils
    - Python docs
        - StreamingContext and DStream
        - KafkaUtils
- More examples in Scala and Java and Python
- Paper and video describing Spark Streaming.

Streaming ML

Ex. E-commerce / Financial Fraud detection

- Millions of daily operations
- Detect fraud fast & correctly
- Storing data is impractical / unfeasible
- Long training time may result in missing potential fraud.

Data Streams /

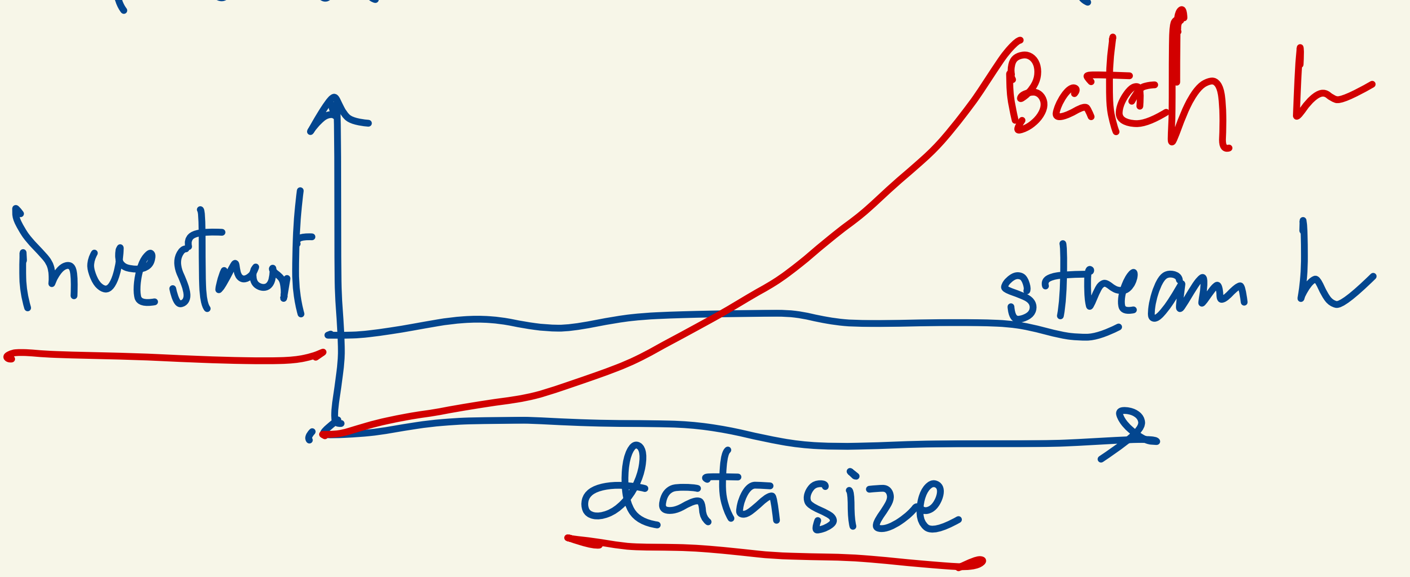
- Data is assumed infinite
- Maintain models in an online fashion.
  - Incorporate data on the fly
  - Unbounded training sets
  - Resource efficient
  - Detect changes and adapt
  - Dynamic models. /

Requirement

- Process one sample at a time, and inspect it only once.
- Use a limited amount of memory
- Work in a limit amount of time.
- Always ready to predict.

Batch learning VS Stream learning

investment Vs data size

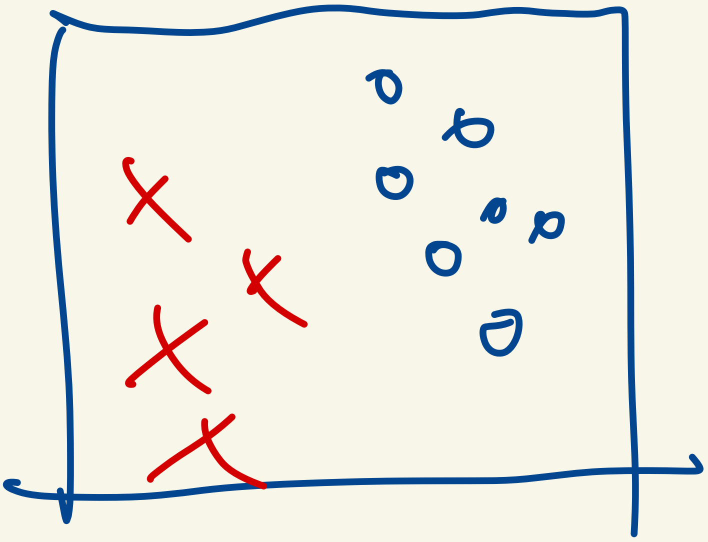


Intro. to stream learniy Algo.

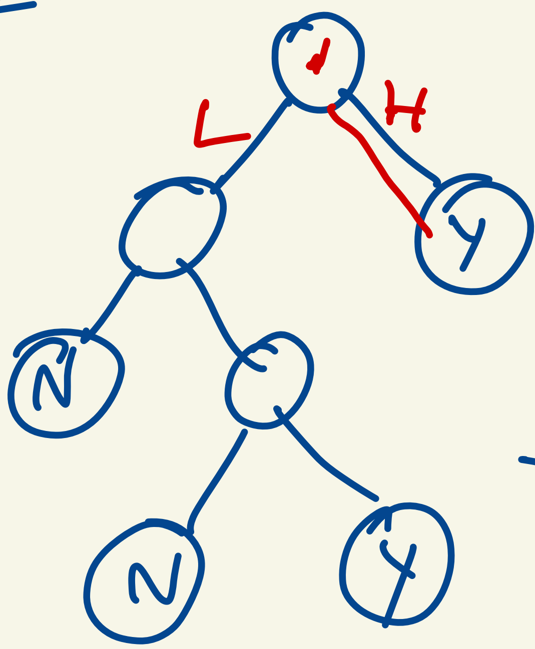
Classification problem

Given a set of training examples (x, y) belonging to n_c different classes, a classifier algorithm builds a model that predicts for every unlabeled instance x the class C to which it belongs.

Ex., Email spam filter
Twitter sentimental analyzer



Decision tree



- Each node test a feature.
- Each branch represent values.
- Each leaf assigns a class.
- Greedy recursive induction

issue - We need all data to train the tree.

- Not suitable for stream learning.

Very Fast decision tree (Hoeffding tree)

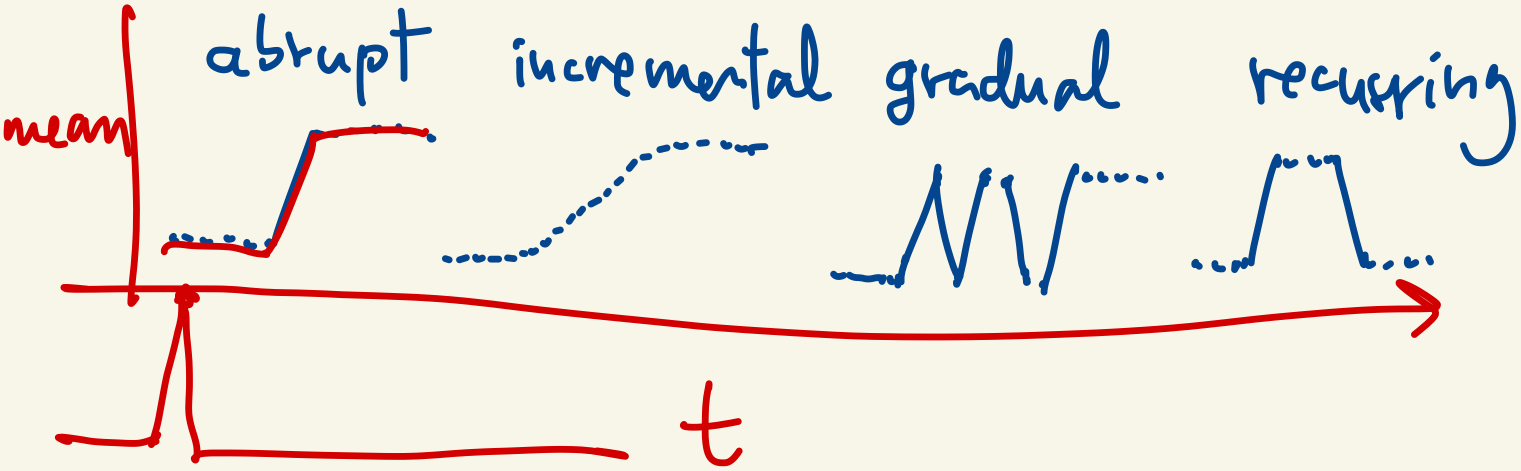
- An incremental decision tree learner for large data streams.
- HT exploit the fact that a small sample can often be enough to choose an optimal splitting attribute.
- output is close to batch Learner

Challenge

Concept drift *

Definition In dynamic & non-stationary environments, the data distribution can change over time.

Concept drift patterns



challenge → outlier (noise)

Evaluation

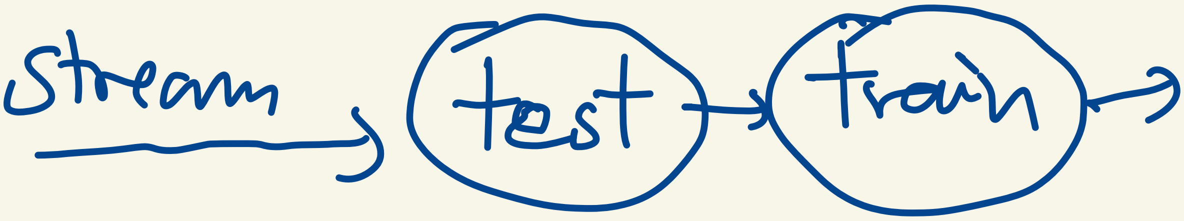
① Holdout an independent test set

Popular method for both batch & stream.

(Predictive + sequential)

② Prequential. (interleaved test-then-train evolution)

- Stream
- Test then train.


stream
test
train