Table of Contents

Spark Streaming

Welcome to Spark Streaming!

I'm Jacek Laskowski, an **independent consultant** who is passionate about **Apache Spark**, Apache Kafka, Scala, sbt (with some flavour of Apache Mesos, Hadoop YARN, and DC/OS). I lead Warsaw Scala Enthusiasts and Warsaw Spark meetups in Warsaw, Poland.

Contact me at jacek@japila.pl or @jaceklaskowski to discuss Apache Spark opportunities, e.g. courses, workshops, mentoring or application development services.

If you like the Apache Spark notes you should seriously consider participating in my own, very hands-on Spark Workshops.

This collections of notes (what some may rashly call a "book") serves as the ultimate place of mine to collect all the nuts and bolts of using Apache Spark. The notes aim to help me designing and developing better products with Apache Spark. It is also a viable proof of my understanding of Apache Spark. I do eventually want to reach the highest level of mastery in Apache Spark.

The collection of notes serves as **the study material** for my trainings, workshops, videos and courses about Apache Spark. Follow me on twitter @jaceklaskowski to know it early. You will also learn about the upcoming events about Apache Spark.

Expect text and code snippets from Spark's mailing lists, the official documentation of Apache Spark, StackOverflow, blog posts, books from O'Reilly, press releases, YouTube/Vimeo videos, Quora, the source code of Apache Spark, etc. Attribution follows.

Spark Streaming — Streaming RDDs

Spark Streaming is the incremental **micro-batching stream processing framework** for Spark.

Spark Streaming offers the data abstraction called DStream that hides the complexity of dealing with a continuous data stream and makes it as easy for programmers as using one single RDD at a time.

That is why Spark Streaming is also called a **micro-batching streaming framework** as a batch is one RDD at a time.

Note	I think Spark Streaming shines on performing the **T** stage well, i.e. the transformation stage, while leaving the **E** and **L** stages for more specialized tools like Apache Kafka or frameworks like Akka.

For a software developer, a `DStream` is similar to work with as a `RDD` with the DStream API to match RDD API. Interestingly, you can reuse your RDD-based code and apply it to `DStream` - a stream of RDDs - with no changes at all (through foreachRDD).

It runs streaming jobs every batch duration to pull and process data (often called *records*) from one or many input streams.

Each batch computes (*generates*) a RDD for data in input streams for a given batch and submits a Spark job to compute the result. It does this over and over again until the streaming context is stopped (and the owning streaming application terminated).

To avoid losing records in case of failure, Spark Streaming supports checkpointing that writes received records to a highly-available HDFS-compatible storage and allows to recover from temporary downtimes.

Spark Streaming allows for integration with real-time data sources ranging from such basic ones like a HDFS-compatible file system or socket connection to more advanced ones like Apache Kafka or Apache Flume.

Checkpointing is also the foundation of stateful and windowed operations.

About Spark Streaming from the official documentation (that pretty much nails what it offers):

> 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, Twitter, ZeroMQ, 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.

Essential concepts in Spark Streaming:

- StreamingContext
- Stream Operators
- Batch, Batch time, and JobSet
- Streaming Job
- Discretized Streams (DStreams)
- Receivers

Other concepts often used in Spark Streaming:

- **ingestion** = the act of processing streaming data.

Micro Batch

Micro Batch is a collection of input records as collected by Spark Streaming that is later represented as an RDD.

A **batch** is internally represented as a JobSet.

Batch Interval (aka batchDuration)

Batch Interval is a property of a Streaming application that describes how often an RDD of input records is generated. It is the time to collect input records before they become a micro-batch.

Streaming Job

A streaming `Job` represents a Spark computation with one or many Spark jobs.

It is identified (in the logs) as `streaming job [time].[outputOpId]` with `outputOpId` being the position in the sequence of jobs in a JobSet.

When executed, it runs the computation (the input `func` function).

Note	A collection of streaming jobs is generated for a batch using DStreamGraph.generateJobs(time: Time).

Internal Registries

- `nextInputStreamId` - the current InputStream id

`StreamingContext` — The Entry Point to Spark Streaming

`StreamingContext` is the entry point for all Spark Streaming functionality. Whatever you do in Spark Streaming has to start from creating an instance of StreamingContext.

```
import org.apache.spark.streaming._
val sc = SparkContext.getOrCreate
val ssc = new StreamingContext(sc, Seconds(5))
```

Note	`StreamingContext` belongs to `org.apache.spark.streaming` package.

With an instance of `StreamingContext` in your hands, you can create ReceiverInputDStreams or set the checkpoint directory.

Once streaming pipelines are developed, you start StreamingContext to set the stream transformations in motion. You stop the instance when you are done.

Creating Instance

You can create a new instance of `StreamingContext` using the following constructors. You can group them by whether a StreamingContext constructor creates it from scratch or it is recreated from a checkpoint directory (follow the links for their extensive coverage).

- Creating StreamingContext from scratch:
  - `StreamingContext(conf: SparkConf, batchDuration: Duration)`
  - `StreamingContext(master: String, appName: String, batchDuration: Duration, sparkHome: String, jars: Seq[String], environment: Map[String,String])`
  - `StreamingContext(sparkContext: SparkContext, batchDuration: Duration)`
- Recreating StreamingContext from a checkpoint file (where `path` is the checkpoint directory):
  - `StreamingContext(path: String)`
  - `StreamingContext(path: String, hadoopConf: Configuration)`
  - `StreamingContext(path: String, sparkContext: SparkContext)`

Note	`StreamingContext(path: String)` uses SparkHadoopUtil.get.conf.

Note	When a StreamingContext is created and spark.streaming.checkpoint.directory setting is set, the value gets passed on to checkpoint method.

Creating StreamingContext from Scratch

When you create a new instance of `StreamingContext`, it first checks whether a SparkContext or the checkpoint directory are given (but not both!)

Tip	`StreamingContext` will warn you when you use `local` or `local[1]` master URLs: `WARN StreamingContext: spark.master should be set as local[n], n > 1 in local mode if you have receivers to get data, otherwise Spark jobs will not get resources to process the received data.`

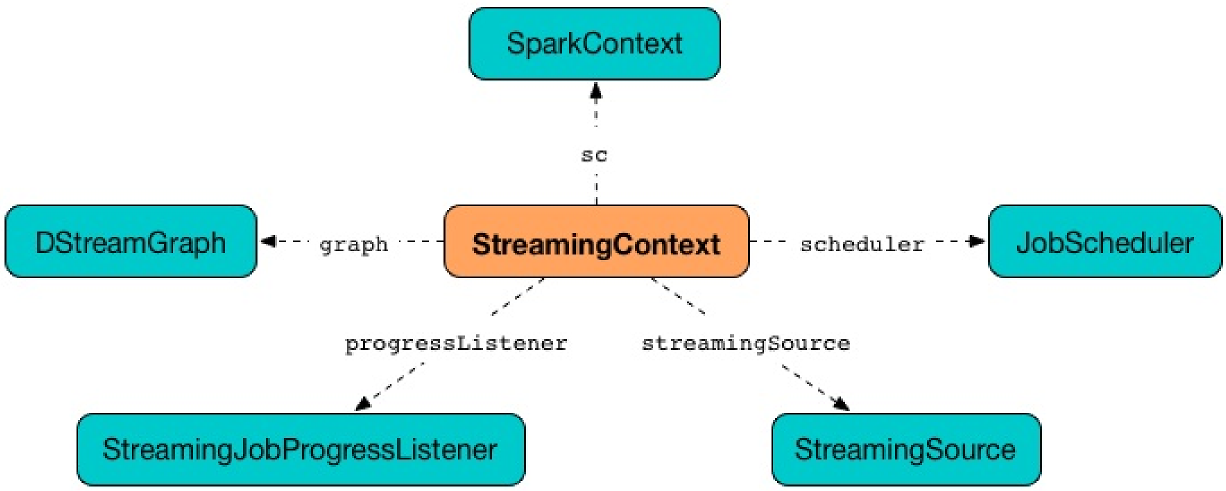

Figure 1. StreamingContext and Dependencies

A DStreamGraph is created.

A JobScheduler is created.

A StreamingJobProgressListener is created.

Streaming tab in web UI is created (when spark.ui.enabled is enabled).

A StreamingSource is instantiated.

At this point, `StreamingContext` enters INITIALIZED state.

Creating ReceiverInputDStreams

`StreamingContext` offers the following methods to create ReceiverInputDStreams:

- receiverStream(receiver: Receiver[T])
- `actorStream[T](props: Props, name: String, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2, supervisorStrategy: SupervisorStrategy = ActorSupervisorStrategy.defaultStrategy): ReceiverInputDStream[T]`
- `socketTextStream(hostname: String, port: Int, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[String]`
- `socketStream[T](hostname: String, port: Int, converter: (InputStream) ⇒ Iterator[T], storageLevel: StorageLevel): ReceiverInputDStream[T]`
- `rawSocketStream[T](hostname: String, port: Int, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[T]`

`StreamingContext` offers the following methods to create InputDStreams:

- `queueStream[T](queue: Queue[RDD[T]], oneAtATime: Boolean = true): InputDStream[T]`
- `queueStream[T](queue: Queue[RDD[T]], oneAtATime: Boolean, defaultRDD: RDD[T]): InputDStream[T]`

You can also use two additional methods in `StreamingContext` to build (or better called *compose*) a custom DStream:

- `union[T](streams: Seq[DStream[T]]): DStream[T]`
- transform(dstreams, transformFunc): DStream[T]

`receiverStream` method

```
receiverStream[T: ClassTag](receiver: Receiver[T]): ReceiverInputDStream[T]
```

You can register a custom input dstream using `receiverStream` method. It accepts a Receiver.

Note	You can find an example of a custom `Receiver` in Custom Receiver.

`transform` method

```
transform[T](dstreams: Seq[DStream[_]], transformFunc: (Seq[RDD[_]], Time) => RDD[T]): DStream[T]
```

`transform` Example

```
import org.apache.spark.rdd.RDD
def union(rdds: Seq[RDD[_]], time: Time) = {
  rdds.head.context.union(rdds.map(_.asInstanceOf[RDD[Int]]))
}
ssc.transform(Seq(cis), union)
```

`remember` method

```
remember(duration: Duration): Unit
```

`remember` method sets the remember interval (for the graph of output dstreams). It simply calls DStreamGraph.remember method and exits.

Caution	FIXME figure

Checkpoint Interval

The **checkpoint interval** is an internal property of `StreamingContext` and corresponds to batch interval or checkpoint interval of the checkpoint (when checkpoint was present).

Note	The checkpoint interval property is also called **graph checkpointing interval**.

checkpoint interval is mandatory when checkpoint directory is defined (i.e. not `null`).

Checkpoint Directory

A **checkpoint directory** is a HDFS-compatible directory where checkpoints are written to.

Note	*"A HDFS-compatible directory"* means that it is Hadoop's Path class to handle all file system-related operations.

Its initial value depends on whether the StreamingContext was (re)created from a checkpoint or not, and is the checkpoint directory if so. Otherwise, it is not set (i.e. `null`).

You can set the checkpoint directory when a StreamingContext is created or later using checkpoint method.

Internally, a checkpoint directory is tracked as `checkpointDir`.

Tip	Refer to Checkpointing for more detailed coverage.

Initial Checkpoint

Initial checkpoint is the checkpoint (file) this StreamingContext has been recreated from.

The initial checkpoint is specified when a StreamingContext is created.

```
val ssc = new StreamingContext("_checkpoint")
```

Marking StreamingContext As Recreated from Checkpoint — `isCheckpointPresent` method

`isCheckpointPresent` internal method behaves like a flag that remembers whether the `StreamingContext` instance was created from a checkpoint or not so the other internal parts of a streaming application can make decisions how to initialize themselves (or just be initialized).

`isCheckpointPresent` checks the existence of the initial checkpoint that gave birth to the StreamingContext.

Setting Checkpoint Directory — `checkpoint` method

```
checkpoint(directory: String): Unit
```

You use `checkpoint` method to set `directory` as the current checkpoint directory.

Note	Spark creates the directory unless it exists already.

`checkpoint` uses SparkContext.hadoopConfiguration to get the file system and create `directory` on. The full path of the directory is passed on to SparkContext.setCheckpointDir method.

Note	Calling `checkpoint` with `null` as `directory` clears the checkpoint directory that effectively disables checkpointing.

Note	When StreamingContext is created and spark.streaming.checkpoint.directory setting is set, the value gets passed on to `checkpoint` method.

Starting `StreamingContext` — `start` method

```
start(): Unit
```

`start()` starts stream processing. It acts differently per state of StreamingContext and only INITIALIZED state makes for a proper startup.

Note	Consult States section in this document to learn about the states of StreamingContext.

Starting in INITIALIZED state

Right after StreamingContext has been instantiated, it enters `INITIALIZED` state in which `start` first checks whether another `StreamingContext` instance has already been started in the JVM. It throws `IllegalStateException` exception if it was and exits.

```
java.lang.IllegalStateException: Only one StreamingContext may
be started in this JVM. Currently running StreamingContext was
started at [startSite]
```

If no other StreamingContext exists, it performs setup validation and starts `JobScheduler` (in a separate dedicated daemon thread called **streaming-start**).

Figure 2. When started, StreamingContext starts JobScheduler

It enters ACTIVE state.

It then register the shutdown hook stopOnShutdown and streaming metrics source. If web UI is enabled, it attaches the Streaming tab.

Given all the above has have finished properly, it is assumed that the StreamingContext started fine and so you should see the following INFO message in the logs:

```
INFO StreamingContext: StreamingContext started
```

Starting in ACTIVE state

When in `ACTIVE` state, i.e. after it has been started, executing `start` merely leads to the following WARN message in the logs:

```
WARN StreamingContext: StreamingContext has already been started
```

Starting in STOPPED state

Attempting to start `StreamingContext` in STOPPED state, i.e. after it has been stopped, leads to the `IllegalStateException` exception:

```
java.lang.IllegalStateException: StreamingContext has already been stopped
```

Stopping StreamingContext — `stop` methods

You stop `StreamingContext` using one of the three variants of `stop` method:

- `stop(stopSparkContext: Boolean = true)`
- `stop(stopSparkContext: Boolean, stopGracefully: Boolean)`

Note	The first `stop` method uses spark.streaming.stopSparkContextByDefault configuration setting that controls `stopSparkContext` input parameter.

`stop` methods stop the execution of the streams immediately (`stopGracefully` is `false`) or wait for the processing of all received data to be completed (`stopGracefully` is `true`).

`stop` reacts appropriately per the state of `StreamingContext` , but the end state is always STOPPED state with shutdown hook removed.

If a user requested to stop the underlying SparkContext (when `stopSparkContext` flag is enabled, i.e. `true`), it is now attempted to be stopped.

Stopping in ACTIVE state

It is only in ACTIVE state when `stop` does more than printing out WARN messages to the logs.

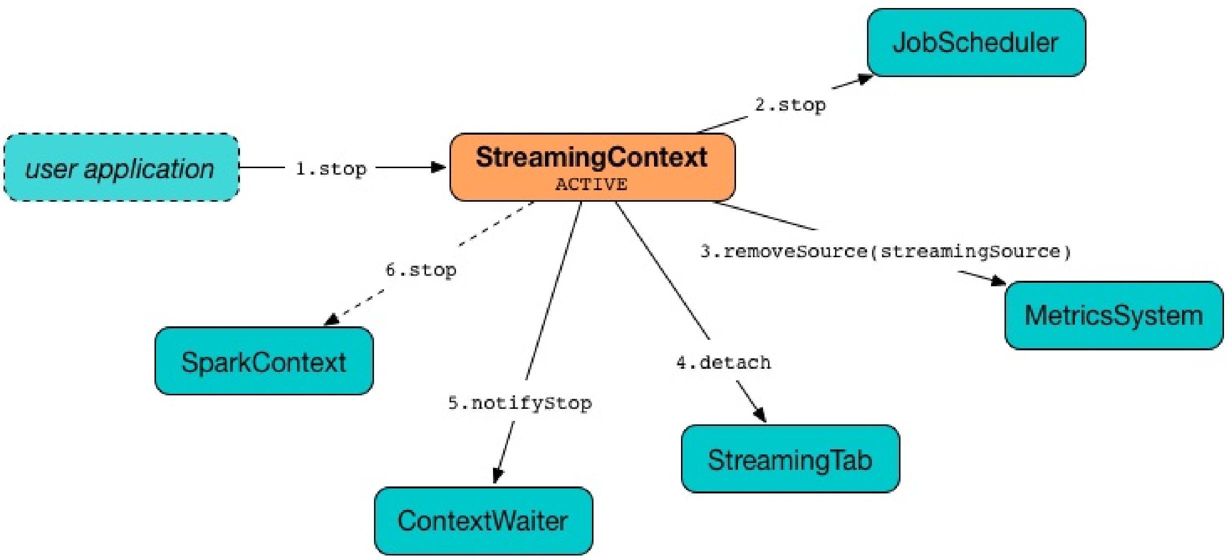

Figure 3. StreamingContext Stop Procedure

It does the following (in order):

1. JobScheduler is stopped.

2. StreamingSource is removed from MetricsSystem (using `MetricsSystem.removeSource`)
3. Streaming tab is detached (using `StreamingTab.detach`).
4. `ContextWaiter` is `notifyStop()`
5. `shutdownHookRef` is cleared.

At that point, you should see the following INFO message in the logs:

```
INFO StreamingContext: StreamingContext stopped successfully
```

`StreamingContext` enters STOPPED state.

Stopping in INITIALIZED state

When in INITIALIZED state, you should see the following WARN message in the logs:

```
WARN StreamingContext: StreamingContext has not been started yet
```

`StreamingContext` enters STOPPED state.

Stopping in STOPPED state

When in STOPPED state, it prints the WARN message to the logs:

```
WARN StreamingContext: StreamingContext has already been stopped
```

`StreamingContext` enters STOPPED state.

`stopOnShutdown` Shutdown Hook

`stopOnShutdown` is a JVM shutdown hook to clean up after `StreamingContext` when the JVM shuts down, e.g. all non-daemon thread exited, `System.exit` was called or `^C` was typed.

Note	It is registered to ShutdownHookManager when StreamingContext starts.

Note	`ShutdownHookManager` uses `org.apache.hadoop.util.ShutdownHookManager` for its work.

When executed, it first reads spark.streaming.stopGracefullyOnShutdown setting that controls whether to stop StreamingContext gracefully or not. You should see the following INFO message in the logs:

```
INFO Invoking stop(stopGracefully=[stopGracefully]) from shutdown hook
```

With the setting it stops StreamingContext without stopping the accompanying `SparkContext` (i.e. `stopSparkContext` parameter is disabled).

Setup Validation — `validate` method

```
validate(): Unit
```

`validate()` method validates configuration of `StreamingContext`.

Note	The method is executed when `StreamingContext` is started.

It first asserts that `DStreamGraph` has been assigned (i.e. `graph` field is not `null`) and triggers validation of DStreamGraph.

Caution	It appears that `graph` could never be `null`, though.

If checkpointing is enabled, it ensures that checkpoint interval is set and checks whether the current streaming runtime environment can be safely serialized by serializing a checkpoint for fictitious batch time 0 (not zero time).

If dynamic allocation is enabled, it prints the following WARN message to the logs:

```
WARN StreamingContext: Dynamic Allocation is enabled for this
application. Enabling Dynamic allocation for Spark Streaming
applications can cause data loss if Write Ahead Log is not
enabled for non-replayable sources like Flume. See the
programming guide for details on how to enable the Write Ahead
Log
```

Registering Streaming Listeners — `addStreamingListener` method

Caution	FIXME

Streaming Metrics Source — `streamingSource` Property

Caution	FIXME

States

`StreamingContext` can be in three states:

- `INITIALIZED` , i.e. after it was instantiated.
- `ACTIVE` , i.e. after it was started.
- `STOPPED` , i.e. after it has been stopped

Stream Operators

You use **stream operators** to apply **transformations** to the elements received (often called **records**) from input streams and ultimately trigger computations using **output operators**.

Transformations are **stateless**, but Spark Streaming comes with an *experimental* support for stateful operators (e.g. mapWithState or updateStateByKey). It also offers windowed operators that can work across batches.

Note	You may use RDDs from other (non-streaming) data sources to build more advanced pipelines.

There are two main types of operators:

- **transformations** that transform elements in input data RDDs
- **output operators** that register input streams as output streams so the execution can start.

Every Discretized Stream (DStream) offers the following operators:

- (output operator) `print` to print 10 elements only or the more general version `print(num: Int)` to print up to `num` elements. See print operation in this document.
- slice
- window
- reduceByWindow
- reduce
- map
- (output operator) foreachRDD
- glom
- (output operator) saveAsObjectFiles
- (output operator) saveAsTextFiles
- transform
- transformWith
- `flatMap`

- `filter`
- `repartition`
- `mapPartitions`
- `count`
- `countByValue`
- `countByWindow`
- `countByValueAndWindow`
- `union`

Note	`DStream` companion object offers a Scala implicit to convert `DStream[(K, V)]` to `PairDStreamFunctions` with methods on DStreams of key-value pairs, e.g. mapWithState or updateStateByKey.

Most streaming operators come with their own custom `DStream` to offer the service. It however very often boils down to overriding the compute method and applying corresponding RDD operator on a generated RDD.

print Operator

`print(num: Int)` operator prints `num` first elements of each RDD in the input stream.

`print` uses `print(num: Int)` with `num` being `10`.

It is a **output operator** (that returns `Unit`).

For each batch, `print` operator prints the following header to the standard output (regardless of the number of elements to be printed out):

```
-------------------------------------------
Time: [time] ms
-------------------------------------------
```

Internally, it calls `RDD.take(num + 1)` (see take action) on each RDD in the stream to print `num` elements. It then prints `…` if there are more elements in the RDD (that would otherwise exceed `num` elements being requested to print).

It creates a ForEachDStream stream and registers it as an output stream.

foreachRDD Operators

```
foreachRDD(foreachFunc: RDD[T] => Unit): Unit
foreachRDD(foreachFunc: (RDD[T], Time) => Unit): Unit
```

`foreachRDD` operator applies `foreachFunc` function to every RDD in the stream.

It creates a ForEachDStream stream and registers it as an output stream.

foreachRDD Example

```
val clicks: InputDStream[(String, String)] = messages
// println every single data received in clicks input stream
clicks.foreachRDD(rdd => rdd.foreach(println))
```

glom Operator

```
glom(): DStream[Array[T]]
```

`glom` operator creates a new stream in which RDDs in the source stream are RDD.glom over, i.e. it coalesces all elements in RDDs within each partition into an array.

reduce Operator

```
reduce(reduceFunc: (T, T) => T): DStream[T]
```

`reduce` operator creates a new stream of RDDs of a single element that is a result of applying `reduceFunc` to the data received.

Internally, it uses map and reduceByKey operators.

reduce Example

```
val clicks: InputDStream[(String, String)] = messages
type T = (String, String)
val reduceFunc: (T, T) => T = {
  case in @ ((k1, v1), (k2, v2)) =>
    println(s">>> input: $in")
    (k2, s"$v1 + $v2")
}
val reduceClicks: DStream[(String, String)] = clicks.reduce(reduceFunc)
reduceClicks.print
```

map Operator

```
map[U](mapFunc: T => U): DStream[U]
```

`map` operator creates a new stream with the source elements being mapped over using `mapFunc` function.

It creates `MappedDStream` stream that, when requested to compute a RDD, uses RDD.map operator.

map Example

```
val clicks: DStream[...] = ...
val mappedClicks: ... = clicks.map(...)
```

reduceByKey Operator

```
reduceByKey(reduceFunc: (V, V) => V): DStream[(K, V)]
reduceByKey(reduceFunc: (V, V) => V, numPartitions: Int): DStream[(K, V)]
reduceByKey(reduceFunc: (V, V) => V, partitioner: Partitioner): DStream[(K, V)]
```

transform Operators

```
transform(transformFunc: RDD[T] => RDD[U]): DStream[U]
transform(transformFunc: (RDD[T], Time) => RDD[U]): DStream[U]
```

`transform` operator applies `transformFunc` function to the generated RDD for a batch.

It creates a TransformedDStream stream.

Note	It asserts that one and exactly one RDD has been generated for a batch before calling the `transformFunc`.

Note	It is not allowed to return `null` from `transformFunc` or a `SparkException` is reported. See TransformedDStream.

transform Example

```
import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

val rdd = sc.parallelize(0 to 9)
import org.apache.spark.streaming.dstream.ConstantInputDStream
val clicks = new ConstantInputDStream(ssc, rdd)

import org.apache.spark.rdd.RDD
val transformFunc: RDD[Int] => RDD[Int] = { inputRDD =>
  println(s">>> inputRDD: $inputRDD")

  // Use SparkSQL's DataFrame to manipulate the input records
  import spark.implicits._
  inputRDD.toDF("num").show

  inputRDD
}
clicks.transform(transformFunc).print
```

transformWith Operators

```
transformWith(other: DStream[U], transformFunc: (RDD[T], RDD[U]) => RDD[V]): DStream[V
]
transformWith(other: DStream[U], transformFunc: (RDD[T], RDD[U], Time) => RDD[V]): DSt
ream[V]
```

`transformWith` operators apply the `transformFunc` function to two generated RDD for a batch.

It creates a TransformedDStream stream.

Note	It asserts that two and exactly two RDDs have been generated for a batch before calling the `transformFunc` .

Note	It is not allowed to return `null` from `transformFunc` or a `SparkException` is reported. See TransformedDStream.

transformWith Example

```
import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

val ns = sc.parallelize(0 to 2)
import org.apache.spark.streaming.dstream.ConstantInputDStream
val nums = new ConstantInputDStream(ssc, ns)

val ws = sc.parallelize(Seq("zero", "one", "two"))
import org.apache.spark.streaming.dstream.ConstantInputDStream
val words = new ConstantInputDStream(ssc, ws)

import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.Time
val transformFunc: (RDD[Int], RDD[String], Time) => RDD[(Int, String)] = { case (ns, w
s, time) =>
  println(s">>> ns: $ns")
  println(s">>> ws: $ws")
  println(s">>> batch: $time")

  ns.zip(ws)
}
nums.transformWith(words, transformFunc).print
```

Windowed Operators

Note	Go to Window Operations to read the official documentation. This document aims at presenting the *internals* of window operators with examples.

In short, **windowed operators** allow you to apply transformations over a **sliding window** of data, i.e. build a *stateful computation* across multiple batches.

Note	Windowed operators, windowed operations, and window-based operations are all the same concept.

By default, you apply transformations using different stream operators to a single RDD that represents a dataset that has been built out of data received from one or many input streams. The transformations know nothing about the past (datasets received and already processed). The computations are hence *stateless*.

You can however build datasets based upon the past ones, and that is when windowed operators enter the stage. Using them allows you to cross the boundary of a single dataset (per batch) and have a series of datasets in your hands (as if the data they hold arrived in a single batch interval).

Table 1. Streaming Windowed Operators

Operator	Description
slice	
window	
reduceByWindow	

`slice` Operators

```
slice(interval: Interval): Seq[RDD[T]]
slice(fromTime: Time, toTime: Time): Seq[RDD[T]]
```

`slice` operators return a collection of RDDs that were generated during time interval inclusive, given as `Interval` or a pair of `Time` ends.

Both `Time` ends have to be a multiple of this stream's slide duration. Otherwise, they are aligned using `Time.floor` method.

When used, you should see the following INFO message in the logs:

```
INFO Slicing from [fromTime] to [toTime] (aligned to [alignedFromTime] and [alignedToT
ime])
```

For every batch in the slicing interval, a RDD is computed.

`window` Operators

```
window(windowDuration: Duration): DStream[T]
window(windowDuration: Duration, slideDuration: Duration): DStream[T]
```

`window` operator creates a new stream that generates RDDs containing all the elements received during `windowDuration` with `slideDuration` slide duration.

Note	`windowDuration` must be a multiple of the slide duration of the source stream.

`window(windowDuration: Duration): DStream[T]` operator uses `window(windowDuration: Duration, slideDuration: Duration)` with the source stream's slide duration.

```
messages.window(Seconds(10))
```

It creates WindowedDStream stream and register it as an output stream.

Note	`window` operator is used by `reduceByWindow` , reduceByKeyAndWindow and `groupByKeyAndWindow` operators.

`reduceByWindow` Operator

```
reduceByWindow(
  reduceFunc: (T, T) => T,
  windowDuration: Duration,
  slideDuration: Duration): DStream[T]

reduceByWindow(
  reduceFunc: (T, T) => T,
  invReduceFunc: (T, T) => T,
  windowDuration: Duration,
  slideDuration: Duration): DStream[T]
```

`reduceByWindow` creates a new stream of RDDs of one element only that was computed using `reduceFunc` function over the data received during batch duration that later was *again* applied to a collection of the reduced elements from the past being window duration `windowDuration` sliding `slideDuration` forward.

Internally, `reduceByWindow` is exactly reduce operator (with `reduceFunc`) followed by window (of `windowDuration` and `slideDuration`) that ultimately gets `reduce` d (again) with `reduceFunc` .

```
// batchDuration = Seconds(5)

val clicks: InputDStream[(String, String)] = messages
type T = (String, String)
val reduceFn: (T, T) => T = {
  case in @ ((k1, v1), (k2, v2)) =>
    println(s">>> input: $in")
    (k2, s"$v1 + $v2")
}
val windowedClicks: DStream[(String, String)] =
  clicks.reduceByWindow(reduceFn, windowDuration = Seconds(10), slideDuration = Seconds
(5))

windowedClicks.print
```

SaveAs Operators

There are two **saveAs operators** in DStream:

- `saveAsObjectFiles`
- `saveAsTextFiles`

They are output operators that return nothing as they save each RDD in a batch to a storage.

Their full signature is as follows:

```
saveAsObjectFiles(prefix: String, suffix: String = ""): Unit
saveAsTextFiles(prefix: String, suffix: String = ""): Unit
```

Note	SaveAs operators use foreachRDD output operator.

`saveAsObjectFiles` uses RDD.saveAsObjectFile while `saveAsTextFiles` uses RDD.saveAsTextFile.

The file name is based on mandatory `prefix` and batch `time` with optional `suffix`. It is in the format of `[prefix]-[time in milliseconds].[suffix]`.

Example

```
val clicks: InputDStream[(String, String)] = messages
clicks.saveAsTextFiles("clicks", "txt")
```

Working with State using Stateful Operators

> Building Stateful Stream Processing Pipelines using Spark (Streaming)

Stateful operators (like mapWithState or updateStateByKey) are part of the set of additional operators available on DStreams of key-value pairs, i.e. instances of `DStream[(K, V)]`. They allow you to build **stateful stream processing pipelines** and are also called cumulative calculations.

The motivation for the stateful operators is that by design streaming operators are stateless and know nothing about the previous records and hence a state. If you'd like to react to new records appropriately given the previous records you would have to resort to using persistent storages outside Spark Streaming.

Note	These additional operators are available automatically on pair DStreams through the Scala implicit conversion `DStream.toPairDStreamFunctions`.

mapWithState Operator

```
mapWithState(spec: StateSpec[K, V, ST, MT]): MapWithStateDStream[K, V, ST, MT]
```

You create StateSpec instances for `mapWithState` operator using the factory methods StateSpec.function.

`mapWithState` creates a MapWithStateDStream dstream.

mapWithState Example

```
import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

// checkpointing is mandatory
ssc.checkpoint("_checkpoints")

val rdd = sc.parallelize(0 to 9).map(n => (n, n % 2 toString))
import org.apache.spark.streaming.dstream.ConstantInputDStream
val sessions = new ConstantInputDStream(ssc, rdd)

import org.apache.spark.streaming.{State, StateSpec, Time}
val updateState = (batchTime: Time, key: Int, value: Option[String], state: State[Int]
) => {
  println(s">>> batchTime = $batchTime")
  println(s">>> key       = $key")
  println(s">>> value     = $value")
  println(s">>> state     = $state")
  val sum = value.getOrElse("").size + state.getOption.getOrElse(0)
  state.update(sum)
  Some((key, value, sum)) // mapped value
}
val spec = StateSpec.function(updateState)
val mappedStatefulStream = sessions.mapWithState(spec)

mappedStatefulStream.print()
```

StateSpec - Specification of mapWithState

`StateSpec` is a state specification of mapWithState and describes how the corresponding state RDD should work (RDD-wise) and maintain a state (streaming-wise).

Note	`StateSpec` is a Scala `sealed abstract class` and hence all the implementations are in the same compilation unit, i.e. source file.

It requires the following:

- `initialState` which is the initial state of the transformation, i.e. paired `RDD[(KeyType, StateType)`.
- `numPartitions` which is the number of partitions of the state RDD. It uses HashPartitioner with the given number of partitions.
- `partitioner` which is the partitioner of the state RDD.
- `timeout` that sets the idle duration after which the state of an *idle* key will be removed. A key and its state is considered *idle* if it has not received any data for at least the given idle duration.

StateSpec.function Factory Methods

You create `StateSpec` instances using the factory methods `StateSpec.function` (that differ in whether or not you want to access a batch time and return an optional mapped value):

```
// batch time and optional mapped return value
StateSpec.function(f: (Time, K, Option[V], State[S]) => Option[M]): StateSpec[K, V, S,
M]

// no batch time and mandatory mapped value
StateSpec.function(f: (K, Option[V], State[S]) => M): StateSpec[K, V, S, M]
```

Internally, the `StateSpec.function` executes `ClosureCleaner.clean` to clean up the input function `f` and makes sure that `f` can be serialized and sent over the wire (cf. Closure Cleaning (clean method)). It will throw an exception when the input function cannot be serialized.

updateStateByKey Operator

```
updateStateByKey(updateFn: (Seq[V], Option[S]) => Option[S]): DStream[(K, S)] (1)
updateStateByKey(updateFn: (Seq[V], Option[S]) => Option[S],
      numPartitions: Int): DStream[(K, S)]  (2)
updateStateByKey(updateFn: (Seq[V], Option[S]) => Option[S],
      partitioner: Partitioner): DStream[(K, S)]  (3)
updateStateByKey(updateFn: (Iterator[(K, Seq[V], Option[S])]) => Iterator[(K, S)],
      partitioner: Partitioner,
      rememberPartitioner: Boolean): DStream[(K, S)]  (4)
updateStateByKey(updateFn: (Seq[V], Option[S]) => Option[S],
      partitioner: Partitioner,
      initialRDD: RDD[(K, S)]): DStream[(K, S)]
updateStateByKey(updateFn: (Iterator[(K, Seq[V], Option[S])]) => Iterator[(K, S)],
      partitioner: Partitioner,
      rememberPartitioner: Boolean,
      initialRDD: RDD[(K, S)]): DStream[(K, S)]
```

1. When not specified explicitly, the partitioner used is HashPartitioner with the number of partitions being the default level of parallelism of a Task Scheduler.
2. You may however specify the number of partitions explicitly for HashPartitioner to use.
3. This is the "canonical" `updateStateByKey` the other two variants (without a partitioner or the number of partitions) use that allows specifying a partitioner explicitly. It then executes the "last" `updateStateByKey` with `rememberPartitioner` enabled.
4. The "last" `updateStateByKey`

`updateStateByKey` stateful operator allows for maintaining per-key state and updating it using `updateFn`. The `updateFn` is called for each key, and uses new data and existing state of the key, to generate an updated state.

Tip	You should use mapWithState operator instead as a much performance effective alternative.

Note	Please consult SPARK-2629 Improved state management for Spark Streaming for performance-related changes to the operator.

The state update function `updateFn` scans every key and generates a new state for every key given a collection of values per key in a batch and the current state for the key (if exists).

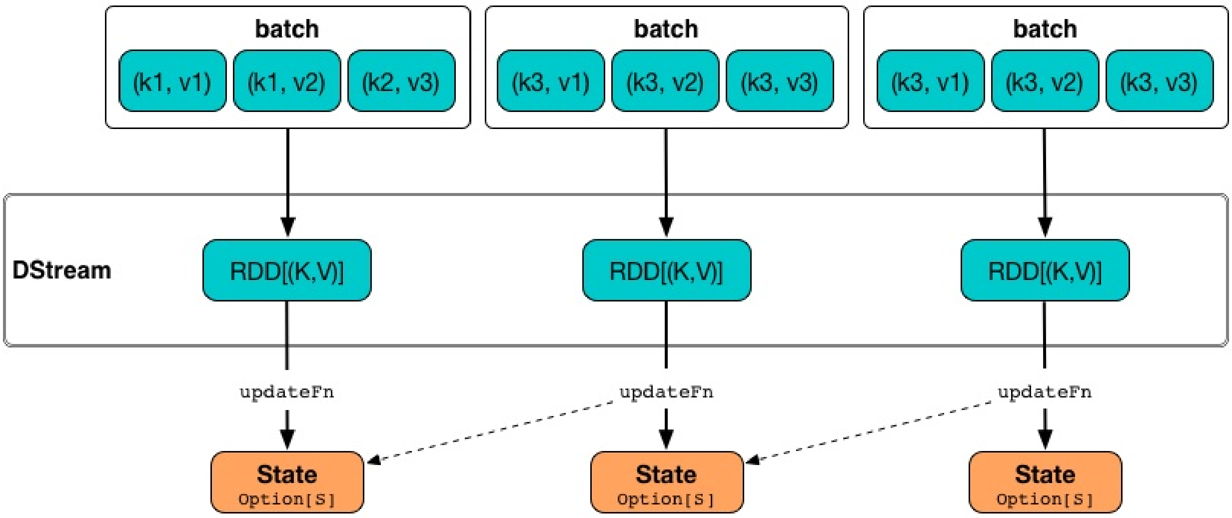

Figure 1. updateStateByKey in motion

Internally, `updateStateByKey` executes SparkContext.clean on the input function `updateFn`.

Note	The operator does not offer any timeout of idle data.

`updateStateByKey` creates a StateDStream stream.

updateStateByKey Example

```
import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

// checkpointing is mandatory
ssc.checkpoint("_checkpoints")

val rdd = sc.parallelize(0 to 9).map(n => (n, n % 2 toString))
import org.apache.spark.streaming.dstream.ConstantInputDStream
val clicks = new ConstantInputDStream(ssc, rdd)

// helper functions
val inc = (n: Int) => n + 1
def buildState: Option[Int] = {
  println(s">>> >>> Initial execution to build state or state is deliberately uninitia
lized yet")
  println(s">>> >>> Building the state being the number of calls to update state funct
ion, i.e. the number of batches")
  Some(1)
}

// the state update function
val updateFn: (Seq[String], Option[Int]) => Option[Int] = { case (vs, state) =>
  println(s">>> update state function with values only, i.e. no keys")
  println(s">>> vs    = $vs")
  println(s">>> state = $state")
  state.map(inc).orElse(buildState)
}
val statefulStream = clicks.updateStateByKey(updateFn)
statefulStream.print()
```

PairDStreamFunctions

`PairDStreamFunctions` is a collection of operators available on DStreams of `(key, value)` pairs (through an implicit conversion).

Table 1. Streaming `PairDStreamFunctions` Operators

Operator	Description
reduceByKeyAndWindow	

`reduceByKeyAndWindow` Operators

```
reduceByKeyAndWindow(
  reduceFunc: (V, V) => V,
  windowDuration: Duration): DStream[(K, V)]

reduceByKeyAndWindow(
  reduceFunc: (V, V) => V,
  windowDuration: Duration,
  slideDuration: Duration): DStream[(K, V)]

reduceByKeyAndWindow(
  reduceFunc: (V, V) => V,
  windowDuration: Duration,
  slideDuration: Duration,
  numPartitions: Int): DStream[(K, V)]

reduceByKeyAndWindow(
  reduceFunc: (V, V) => V,
  windowDuration: Duration,
  slideDuration: Duration,
  partitioner: Partitioner): DStream[(K, V)]

reduceByKeyAndWindow(
  reduceFunc: (V, V) => V,
  invReduceFunc: (V, V) => V,
  windowDuration: Duration,
  slideDuration: Duration = self.slideDuration,
  numPartitions: Int = ssc.sc.defaultParallelism,
  filterFunc: ((K, V)) => Boolean = null): DStream[(K, V)]

reduceByKeyAndWindow(
  reduceFunc: (V, V) => V,
  invReduceFunc: (V, V) => V,
  windowDuration: Duration,
  slideDuration: Duration,
  partitioner: Partitioner,
  filterFunc: ((K, V)) => Boolean): DStream[(K, V)]
```

`reduceByKeyAndWindow` returns a `ReducedWindowedDStream` with the input `reduceFunc` , `invReduceFunc` and `filterFunc` functions cleaned up.

Tip	Enable `DEBUG` logging level for `org.apache.spark.streaming.dstream.ReducedWindowedDStream` to see the times for window, slide, zero with current and previous windows in the logs.

web UI and Streaming Statistics Page

When you start a Spark Streaming application, you can use web UI to monitor streaming statistics in **Streaming** tab (aka *page*).

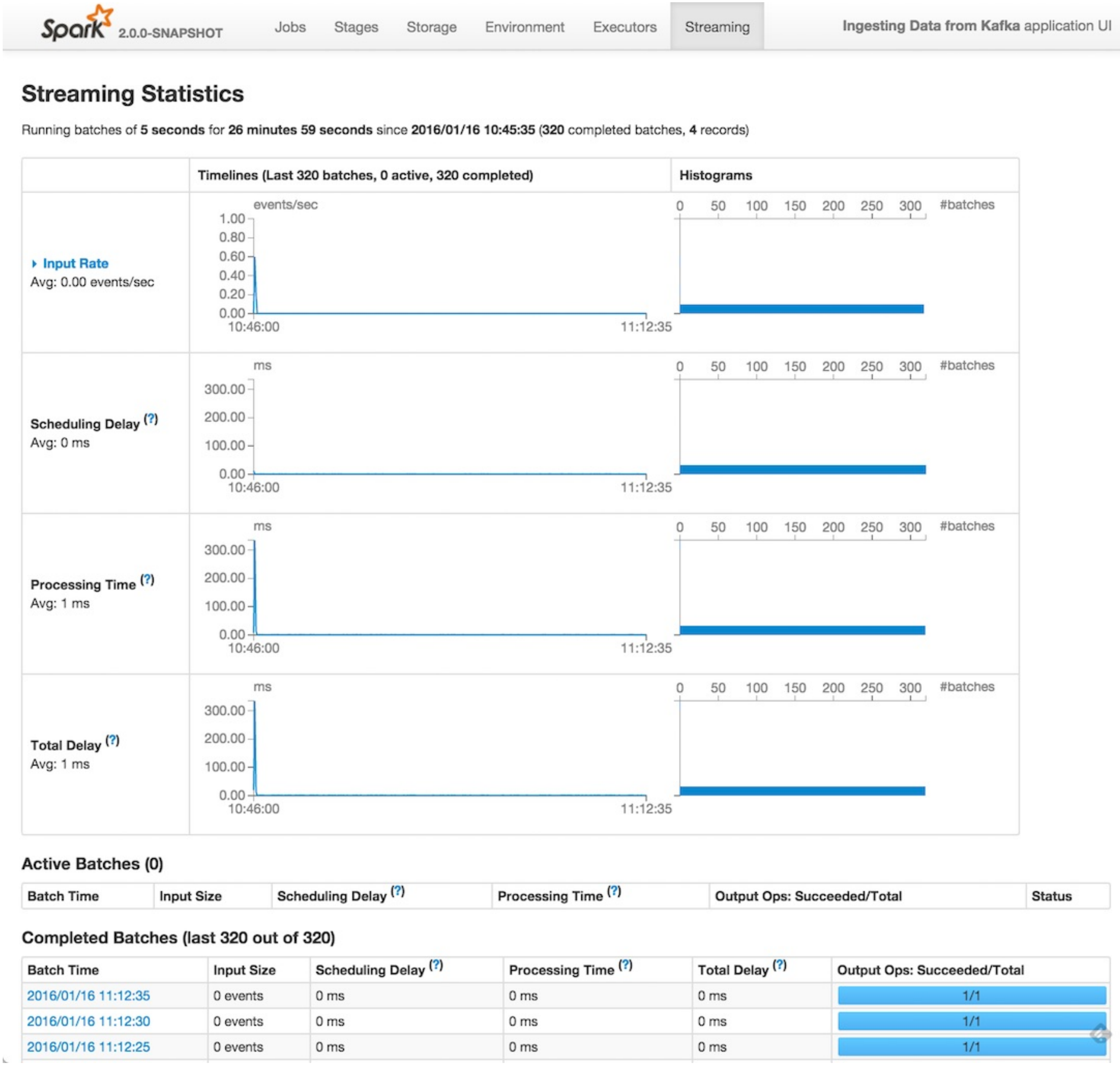

Figure 1. Streaming Tab in web UI

Note	The number of completed batches to retain to compute statistics upon is controlled by spark.streaming.ui.retainedBatches (and defaults to `1000`).

The page is made up of three sections (aka *tables*) - the unnamed, top-level one with basic information about the streaming application (right below the title **Streaming Statistics**), Active Batches and Completed Batches.

Note	The Streaming page uses StreamingJobProgressListener for most of the information displayed.

Basic Information

Basic Information section is the top-level section in the Streaming page that offers basic information about the streaming application.

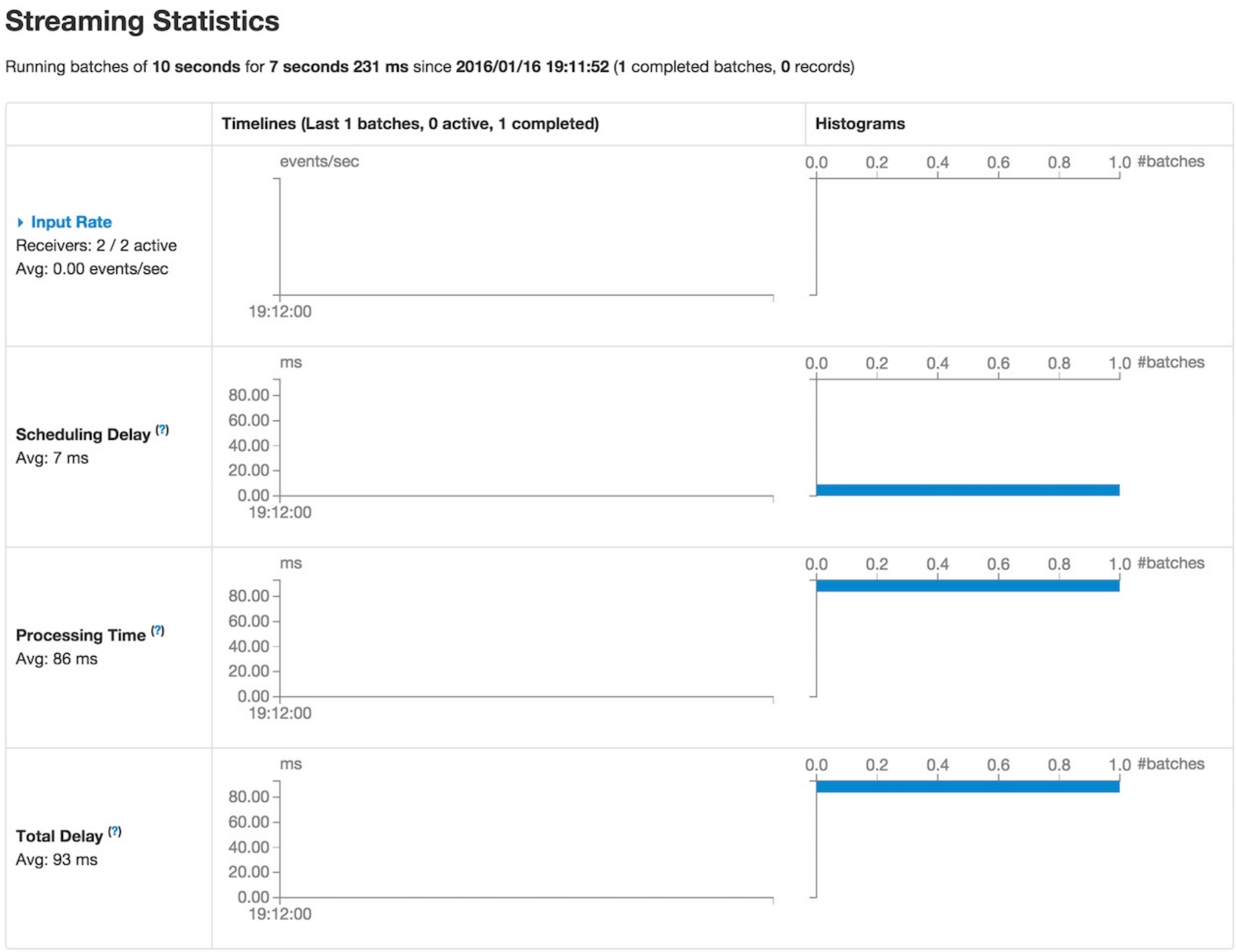

Figure 2. Basic Information section in Streaming Page (with Receivers)

The section shows the batch duration (in *Running batches of [batch duration]*), and the time it runs for and since StreamingContext was created (*not* when this streaming application has been started!).

It shows the number of all **completed batches** (for the entire period since the StreamingContext was started) and **received records** (in parenthesis). These information are later displayed in detail in Active Batches and Completed Batches sections.

Below is the table for retained batches (i.e. waiting, running, and completed batches).

In **Input Rate** row, you can show and hide details of each input stream.

If there are input streams with receivers, the numbers of all the receivers and active ones are displayed (as depicted in the Figure 2 above).

The average event rate for all registered streams is displayed (as *Avg: [avg] events/sec*).

Scheduling Delay

Scheduling Delay is the time spent from when the collection of streaming jobs for a batch was submitted to when the first streaming job (out of possibly many streaming jobs in the collection) was started.

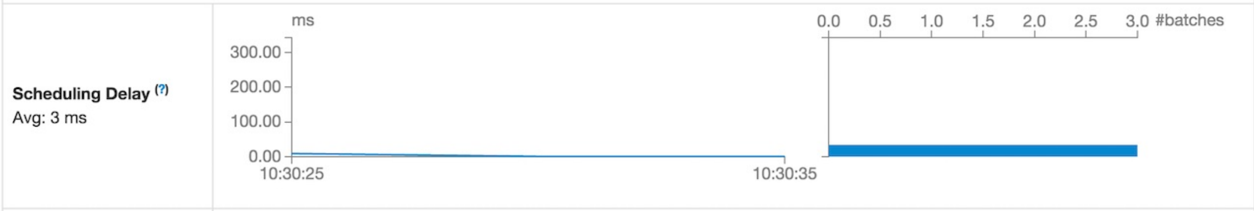

Figure 3. Scheduling Delay in Streaming Page

It should be as low as possible meaning that the streaming jobs in batches are scheduled almost instantly.

Note	The values in the timeline (the first column) depict the time between the events StreamingListenerBatchSubmitted and StreamingListenerBatchStarted (with minor yet additional delays to deliver the events).

You may see increase in scheduling delay in the timeline when streaming jobs are queued up as in the following example:

```
// batch duration = 5 seconds
val messages: InputDStream[(String, String)] = ...
messages.foreachRDD { rdd =>
  println(">>> Taking a 15-second sleep")
  rdd.foreach(println)
  java.util.concurrent.TimeUnit.SECONDS.sleep(15)
}
```

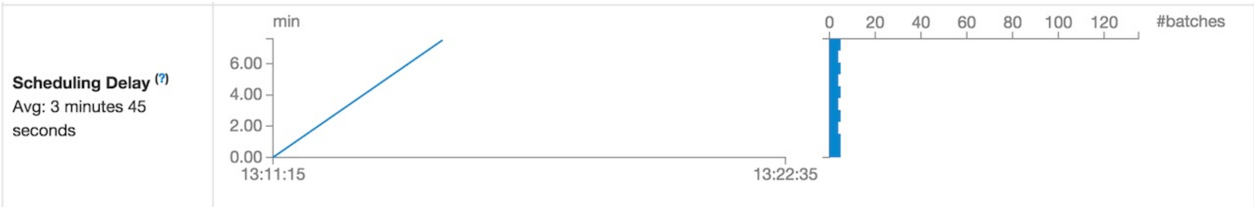

Figure 4. Scheduling Delay Increased in Streaming Page

Processing Time

Processing Time is the time spent to complete all the streaming jobs of a batch.

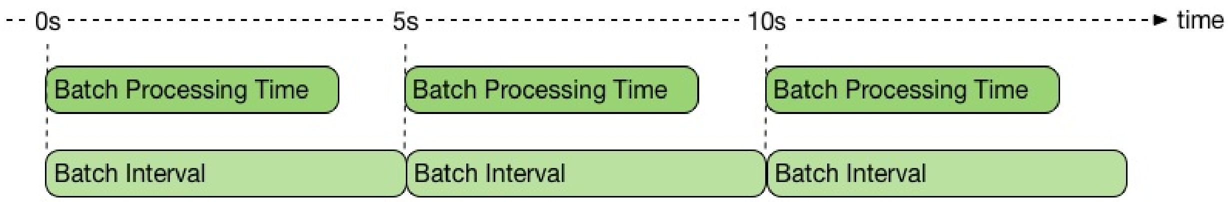

Figure 5. Batch Processing Time and Batch Intervals

Total Delay

Total Delay is the time spent from submitting to complete all jobs of a batch.

Active Batches

Active Batches section presents `waitingBatches` and `runningBatches` together.

Completed Batches

Completed Batches section presents retained completed batches (using `completedBatchUIData`).

Note	The number of retained batches is controlled by spark.streaming.ui.retainedBatches.

Completed Batches (last 5 out of 42)

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)	Output Ops: Succeeded/Total
2016/01/19 21:34:00	0 events	1 ms	0 ms	1 ms	1/1
2016/01/19 21:33:55	0 events	0 ms	1 ms	1 ms	1/1
2016/01/19 21:33:50	0 events	0 ms	0 ms	0 ms	1/1
2016/01/19 21:33:45	0 events	1 ms	0 ms	1 ms	1/1
2016/01/19 21:33:40	0 events	1 ms	0 ms	1 ms	1/1

Figure 6. Completed Batches (limited to 5 elements only)

Example - Kafka Direct Stream in web UI

2016/01/16 10:46:10	1 events	0 ms	13 ms	13 ms	1/1
2016/01/16 10:46:05	3 events	0 ms	0.3 s	0.3 s	1/1
2016/01/16 10:46:00	0 events	12 ms	7 ms	19 ms	1/1

Figure 7. Two Batches with Incoming Data inside for Kafka Direct Stream in web UI (Streaming tab)

Spark 2.0.0-SNAPSHOT Jobs Stages Storage Environment Executors Streaming Ingesting Data from Kafka application UI

Spark Jobs (?)

Total Uptime: 5.2 min
Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	Streaming job from [output operation 0, batch time 10:46:10] print at <console>:35	2016/01/16 10:46:10	10 ms	1/1	1/1
0	Streaming job from [output operation 0, batch time 10:46:05] print at <console>:35	2016/01/16 10:46:05	0.3 s	1/1	1/1

Figure 8. Two Jobs for Kafka Direct Stream in web UI (Jobs tab)

Streaming Listeners

Streaming listeners are Spark listeners interested in streaming events like batch submitted, started or completed.

Streaming listeners implement org.apache.spark.streaming.scheduler.StreamingListener listener interface and process StreamingListenerEvent events.

The following streaming listeners are available in Spark Streaming:

- StreamingJobProgressListener
- RateController

StreamingListenerEvent Events

- `StreamingListenerBatchSubmitted` is posted when streaming jobs are submitted for execution and triggers `StreamingListener.onBatchSubmitted` (see StreamingJobProgressListener.onBatchSubmitted).
- `StreamingListenerBatchStarted` triggers `StreamingListener.onBatchStarted`
- `StreamingListenerBatchCompleted` is posted to inform that a collection of streaming jobs has completed, i.e. all the streaming jobs in JobSet have stopped their execution.

StreamingJobProgressListener

`StreamingJobProgressListener` is a streaming listener that collects information for StreamingSource and Streaming page in web UI.

Note	A `StreamingJobProgressListener` is created while `StreamingContext` is created and later registered as a `StreamingListener` and SparkListener when Streaming tab is created.

onBatchSubmitted

For `StreamingListenerBatchSubmitted(batchInfo: BatchInfo)` events, it stores `batchInfo` batch information in the internal `waitingBatchUIData` registry per batch time.

The number of entries in `waitingBatchUIData` registry contributes to `numUnprocessedBatches` (together with `runningBatchUIData`), `waitingBatches` , and `retainedBatches` . It is also used to look up the batch data for a batch time (in `getBatchUIData`).

`numUnprocessedBatches`, `waitingBatches` are used in StreamingSource.

Note	`waitingBatches` and `runningBatches` are displayed together in Active Batches in Streaming tab in web UI.

onBatchStarted

Caution	FIXME

onBatchCompleted

Caution	FIXME

Retained Batches

`retainedBatches` are waiting, running, and completed batches that web UI uses to display streaming statistics.

The number of retained batches is controlled by spark.streaming.ui.retainedBatches.

Checkpointing

Checkpointing is a process of writing received records (by means of input dstreams) at checkpoint intervals to a highly-available HDFS-compatible storage. It allows creating **fault-tolerant stream processing pipelines** so when a failure occurs input dstreams can restore the before-failure streaming state and continue stream processing (as if nothing had happened).

DStreams can checkpoint input data at specified time intervals.

Marking StreamingContext as Checkpointed

You use StreamingContext.checkpoint method to set up a HDFS-compatible **checkpoint directory** where checkpoint data will be persisted, as follows:

```
ssc.checkpoint("_checkpoint")
```

Checkpoint Interval and Checkpointing DStreams

You can set up periodic checkpointing of a dstream every **checkpoint interval** using DStream.checkpoint method.

```
val ssc: StreamingContext = ...
// set the checkpoint directory
ssc.checkpoint("_checkpoint")
val ds: DStream[Int] = ...
val cds: DStream[Int] = ds.checkpoint(Seconds(5))
// do something with the input dstream
cds.print
```

Recreating StreamingContext from Checkpoint

You can create a StreamingContext from a checkpoint directory, i.e. recreate a fully-working StreamingContext as recorded in the last valid checkpoint file that was written to the checkpoint directory.

Note	You can also create a brand new StreamingContext (and putting checkpoints aside).

Warning	You must not create input dstreams using a StreamingContext that has been recreated from checkpoint. Otherwise, you will not start the StreamingContext at all.

When you use `StreamingContext(path: String)` constructor (or the variants thereof), it uses Hadoop configuration to access `path` directory on a Hadoop-supported file system.

Effectively, the two variants use `StreamingContext(path: String, hadoopConf: Configuration)` constructor that reads the latest valid checkpoint file (and hence enables)

Note	`SparkContext` and batch interval are set to their corresponding values using the checkpoint file.

Example: Recreating StreamingContext from Checkpoint

The following Scala code demonstrates how to use the checkpoint directory `_checkpoint` to (re)create the StreamingContext or create one from scratch.

```
val appName = "Recreating StreamingContext from Checkpoint"
val sc = new SparkContext("local[*]", appName, new SparkConf())

val checkpointDir = "_checkpoint"

def createSC(): StreamingContext = {
  val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

  // NOTE: You have to create dstreams inside the method
  // See http://stackoverflow.com/q/35090180/1305344

  // Create constant input dstream with the RDD
  val rdd = sc.parallelize(0 to 9)
  import org.apache.spark.streaming.dstream.ConstantInputDStream
  val cis = new ConstantInputDStream(ssc, rdd)

  // Sample stream computation
  cis.print

  ssc.checkpoint(checkpointDir)
  ssc
}
val ssc = StreamingContext.getOrCreate(checkpointDir, createSC)

// Start streaming processing
ssc.start
```

DStreamCheckpointData

`DStreamCheckpointData` works with a single dstream. An instance of `DStreamCheckpointData` is created when a dstream is.

It tracks checkpoint data in the internal `data` registry that records batch time and the checkpoint data at that time. The internal checkpoint data can be anything that a dstream wants to checkpoint. `DStreamCheckpointData` returns the registry when `currentCheckpointFiles` method is called.

Note	By default, `DStreamCheckpointData` records the checkpoint files to which the generated RDDs of the DStream has been saved.

<table>
<tr><td>Tip</td><td>
Enable <code>DEBUG</code> logging level for <code>org.apache.spark.streaming.dstream.DStreamCheckpointData</code> logger to see what happens inside.
Add the following line to <code>conf/log4j.properties</code> :
<pre>log4j.logger.org.apache.spark.streaming.dstream.DStreamCheckpointData=DEBUG</pre>
Refer to Logging.
</td></tr>
</table>

Updating Collection of Batches and Checkpoint Directories (update method)

```
update(time: Time): Unit
```

`update` collects batches and the directory names where the corresponding RDDs were checkpointed (filtering the dstream's internal generatedRDDs mapping).

You should see the following DEBUG message in the logs:

```
DEBUG Current checkpoint files:
[checkpointFile per line]
```

The collection of the batches and their checkpointed RDDs is recorded in an internal field for serialization (i.e. it becomes the current value of the internal field `currentCheckpointFiles` that is serialized when requested).

The collection is also added to an internal *transient* (non-serializable) mapping `timeToCheckpointFile` and the oldest checkpoint (given batch times) is recorded in an internal *transient* mapping for the current `time` .

Note	It is called by DStream.updateCheckpointData(currentTime: Time).

Deleting Old Checkpoint Files (cleanup method)

```
cleanup(time: Time): Unit
```

`cleanup` deletes checkpoint files older than the oldest batch for the input `time` .

It first gets the oldest batch time for the input `time` (see Updating Collection of Batches and Checkpoint Directories (update method)).

If the (batch) time has been found, all the checkpoint files older are deleted (as tracked in the internal `timeToCheckpointFile` mapping).

You should see the following DEBUG message in the logs:

```
DEBUG Files to delete:
[comma-separated files to delete]
```

For each checkpoint file successfully deleted, you should see the following INFO message in the logs:

```
INFO Deleted checkpoint file '[file]' for time [time]
```

Errors in checkpoint deletion are reported as WARN messages in the logs:

```
WARN Error deleting old checkpoint file '[file]' for time [time]
```

Otherwise, when no (batch) time has been found for the given input `time` , you should see the following DEBUG message in the logs:

```
DEBUG Nothing to delete
```

Note	It is called by DStream.clearCheckpointData(time: Time).

Restoring Generated RDDs from Checkpoint Files (restore method)

```
restore(): Unit
```

`restore` restores the dstream's generatedRDDs given persistent internal `data` mapping with batch times and corresponding checkpoint files.

`restore` takes the current checkpoint files and restores checkpointed RDDs from each checkpoint file (using `SparkContext.checkpointFile`).

You should see the following INFO message in the logs per checkpoint file:

```
INFO Restoring checkpointed RDD for time [time] from file '[file]'
```

Note	It is called by DStream.restoreCheckpointData().

Checkpoint

`Checkpoint` class requires a StreamingContext and `checkpointTime` time to be instantiated. The internal property `checkpointTime` corresponds to the batch time it represents.

Note	`Checkpoint` class is written to a persistent storage (aka *serialized*) using CheckpointWriter.write method and read back (aka *deserialize*) using Checkpoint.deserialize.

Note	Initial checkpoint is the checkpoint a StreamingContext was started with.

It is merely a collection of the settings of the current streaming runtime environment that is supposed to recreate the environment after it goes down due to a failure or when the streaming context is stopped immediately.

It collects the settings from the input `StreamingContext` (and indirectly from the corresponding JobScheduler and SparkContext):

- The master URL from SparkContext as `master` .
- The mandatory application name from SparkContext as `framework` .
- The jars to distribute to workers from SparkContext as `jars` .
- The DStreamGraph as `graph`
- The checkpoint directory as `checkpointDir`
- The checkpoint interval as `checkpointDuration`
- The collection of pending batches to process as `pendingTimes`
- The Spark configuration (aka SparkConf) as `sparkConfPairs`

Tip	Enable `INFO` logging level for `org.apache.spark.streaming.Checkpoint` logger to see what happens inside. Add the following line to `conf/log4j.properties` : `log4j.logger.org.apache.spark.streaming.Checkpoint=INFO` Refer to Logging.

Serializing Checkpoint (serialize method)

```
serialize(checkpoint: Checkpoint, conf: SparkConf): Array[Byte]
```

`serialize` serializes the `checkpoint` object. It does so by creating a compression codec to write the input `checkpoint` object with and returns the result as a collection of bytes.

Caution	FIXME Describe compression codecs in Spark.

Deserializing Checkpoint (deserialize method)

```
deserialize(inputStream: InputStream, conf: SparkConf): Checkpoint
```

`deserialize` reconstructs a Checkpoint object from the input `inputStream` . It uses a compression codec and once read the just-built Checkpoint object is validated and returned back.

Note	`deserialize` is called when reading the latest valid checkpoint file.

Validating Checkpoint (validate method)

```
validate(): Unit
```

`validate` validates the Checkpoint. It ensures that `master` , `framework` , `graph` , and `checkpointTime` are defined, i.e. not `null` .

Note	`validate` is called when a checkpoint is deserialized from an input stream.

You should see the following INFO message in the logs when the object passes the validation:

```
INFO Checkpoint: Checkpoint for time [checkpointTime] ms validated
```

Get Collection of Checkpoint Files from Directory (getCheckpointFiles method)

```
getCheckpointFiles(checkpointDir: String, fsOption: Option[FileSystem] = None): Seq[Pa
th]
```

`getCheckpointFiles` method returns a collection of checkpoint files from the given checkpoint directory `checkpointDir`.

The method sorts the checkpoint files by time with a temporary `.bk` checkpoint file first (given a pair of a checkpoint file and its backup file).

CheckpointWriter

An instance of `CheckpointWriter` is created (lazily) when `JobGenerator` is, but only when JobGenerator is configured for checkpointing.

It uses the internal single-thread thread pool executor to execute checkpoint writes asynchronously and does so until it is stopped.

Writing Checkpoint for Batch Time (write method)

```
write(checkpoint: Checkpoint, clearCheckpointDataLater: Boolean): Unit
```

`write` method serializes the checkpoint object and passes the serialized form to CheckpointWriteHandler to write asynchronously (i.e. on a separate thread) using single-thread thread pool executor.

Note	It is called when JobGenerator receives DoCheckpoint event and the batch time is eligible for checkpointing.

You should see the following INFO message in the logs:

```
INFO CheckpointWriter: Submitted checkpoint of time [checkpoint.checkpointTime] ms wri
ter queue
```

If the asynchronous checkpoint write fails, you should see the following ERROR in the logs:

```
ERROR Could not submit checkpoint task to the thread pool executor
```

Stopping CheckpointWriter (using stop method)

```
stop(): Unit
```

`CheckpointWriter` uses the internal `stopped` flag to mark whether it is stopped or not.

Note	`stopped` flag is disabled, i.e. `false`, when `CheckpointWriter` is created.

`stop` method checks the internal `stopped` flag and returns if it says it is stopped already.

If not, it orderly shuts down the internal single-thread thread pool executor and awaits termination for 10 seconds. During that time, any asynchronous checkpoint writes can be safely finished, but no new tasks will be accepted.

Note	The wait time before `executor` stops is fixed, i.e. not configurable, and is set to 10 seconds.

After 10 seconds, when the thread pool did not terminate, `stop` stops it forcefully.

You should see the following INFO message in the logs:

```
INFO CheckpointWriter: CheckpointWriter executor terminated? [terminated], waited for
[time] ms.
```

`CheckpointWriter` is marked as stopped, i.e. `stopped` flag is set to `true`.

Single-Thread Thread Pool Executor

`executor` is an internal single-thread thread pool executor for executing asynchronous checkpoint writes using CheckpointWriteHandler.

It shuts down when CheckpointWriter is stopped (with a 10-second graceful period before it terminated forcefully).

CheckpointWriteHandler — Asynchronous Checkpoint Writes

`CheckpointWriteHandler` is an (internal) thread of execution that does checkpoint writes. It is instantiated with `checkpointTime`, the serialized form of the checkpoint, and whether or not to clean checkpoint data later flag (as `clearCheckpointDataLater`).

Note	It is only used by CheckpointWriter to queue a checkpoint write for a batch time.

It records the current checkpoint time (in `latestCheckpointTime`) and calculates the name of the checkpoint file.

Note	The name of the checkpoint file is `checkpoint-[checkpointTime.milliseconds]` .

It uses a backup file to do atomic write, i.e. it writes to the checkpoint backup file first and renames the result file to the final checkpoint file name.

Note	The name of the checkpoint backup file is `checkpoint-[checkpointTime.milliseconds].bk` .

Note	`CheckpointWriteHandler` does 3 write attempts at the maximum. The value is not configurable.

When attempting to write, you should see the following INFO message in the logs:

```
INFO CheckpointWriter: Saving checkpoint for time [checkpointTime] ms to file '[checkp
ointFile]'
```

Note	It deletes any checkpoint backup files that may exist from the previous attempts.

It then deletes checkpoint files when there are more than 10.

Note	The number of checkpoint files when the deletion happens, i.e. **10**, is fixed and not configurable.

You should see the following INFO message in the logs:

```
INFO CheckpointWriter: Deleting [file]
```

If all went fine, you should see the following INFO message in the logs:

```
INFO CheckpointWriter: Checkpoint for time [checkpointTime] ms saved to file '[checkpo
intFile]', took [bytes] bytes and [time] ms
```

JobGenerator is informed that the checkpoint write completed (with `checkpointTime` and `clearCheckpointDataLater` flag).

In case of write failures, you can see the following WARN message in the logs:

```
WARN CheckpointWriter: Error in attempt [attempts] of writing checkpoint to [checkpoin
tFile]
```

If the number of write attempts exceeded (the fixed) 10 or CheckpointWriter was stopped before any successful checkpoint write, you should see the following WARN message in the logs:

```
WARN CheckpointWriter: Could not write checkpoint for time [checkpointTime] to file [c
heckpointFile]'
```

CheckpointReader

`CheckpointReader` is a `private[streaming]` helper class to read the latest valid checkpoint file to recreate StreamingContext from (given the checkpoint directory).

Reading Latest Valid Checkpoint File

```
read(checkpointDir: String): Option[Checkpoint]
read(checkpointDir: String, conf: SparkConf,
     hadoopConf: Configuration, ignoreReadError: Boolean = false): Option[Checkpoint]
```

`read` methods read the latest valid checkpoint file from the checkpoint directory `checkpointDir`. They differ in whether Spark configuration `conf` and Hadoop configuration `hadoopConf` are given or created in place.

Note	The 4-parameter `read` method is used by StreamingContext to recreate itself from a checkpoint file.

The first `read` throws no `SparkException` when no checkpoint file could be read.

Note	It appears that no part of Spark Streaming uses the simplified version of `read`.

`read` uses Apache Hadoop's Path and Configuration to get the checkpoint files (using Checkpoint.getCheckpointFiles) in reverse order.

If there is no checkpoint file in the checkpoint directory, it returns None.

You should see the following INFO message in the logs:

```
INFO CheckpointReader: Checkpoint files found: [checkpointFiles]
```

The method reads all the checkpoints (from the youngest to the oldest) until one is successfully loaded, i.e. deserialized.

You should see the following INFO message in the logs just before deserializing a checkpoint `file` :

```
INFO CheckpointReader: Attempting to load checkpoint from file [file]
```

If the checkpoint file was loaded, you should see the following INFO messages in the logs:

```
INFO CheckpointReader: Checkpoint successfully loaded from file [file]
INFO CheckpointReader: Checkpoint was generated at time [checkpointTime]
```

In case of any issues while loading a checkpoint file, you should see the following WARN in the logs and the corresponding exception:

```
WARN CheckpointReader: Error reading checkpoint from file [file]
```

Unless `ignoreReadError` flag is disabled, when no checkpoint file could be read, `SparkException` is thrown with the following message:

```
Failed to read checkpoint from directory [checkpointPath]
```

`None` is returned at this point and the method finishes.

JobScheduler

Streaming scheduler (`JobScheduler`) schedules streaming jobs to be run as Spark jobs. It is created as part of creating a StreamingContext and starts with it.

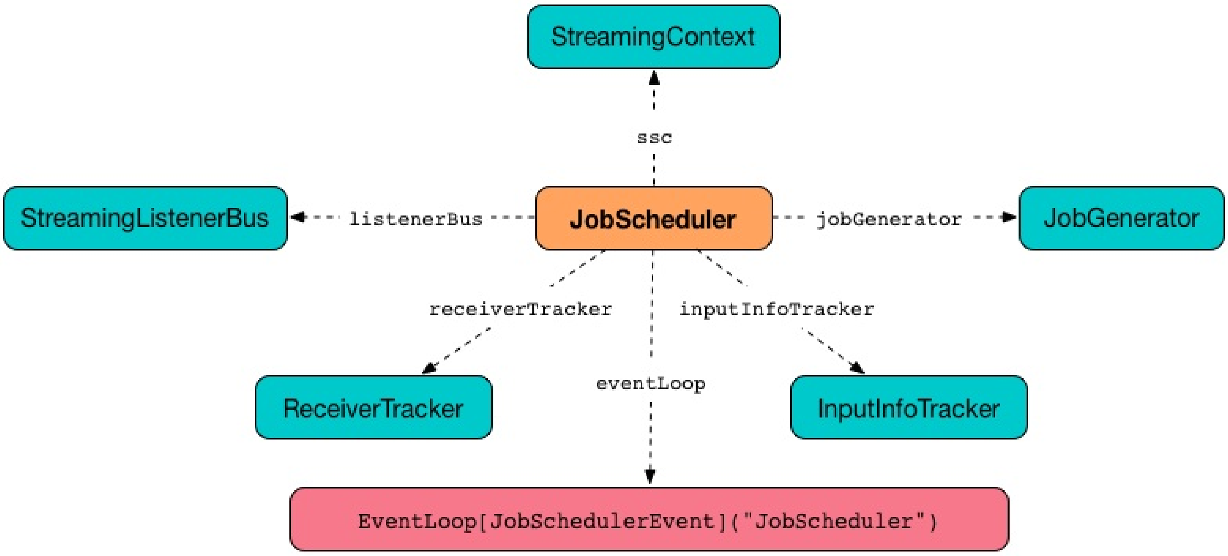

Figure 1. JobScheduler and Dependencies

It tracks jobs submitted for execution (as JobSets via submitJobSet method) in jobSets internal map.

Note	JobSets are submitted by JobGenerator.

It uses a **streaming scheduler queue** for streaming jobs to be executed.

Tip	Enable `DEBUG` logging level for `org.apache.spark.streaming.scheduler.JobScheduler` logger to see what happens in JobScheduler. Add the following line to `conf/log4j.properties` : `log4j.logger.org.apache.spark.streaming.scheduler.JobScheduler=DEBUG` Refer to Logging.

Starting JobScheduler (start method)

```
start(): Unit
```

When `JobScheduler` starts (i.e. when `start` is called), you should see the following DEBUG message in the logs:

```
DEBUG JobScheduler: Starting JobScheduler
```

It then goes over all the dependent services and starts them one by one as depicted in the figure.

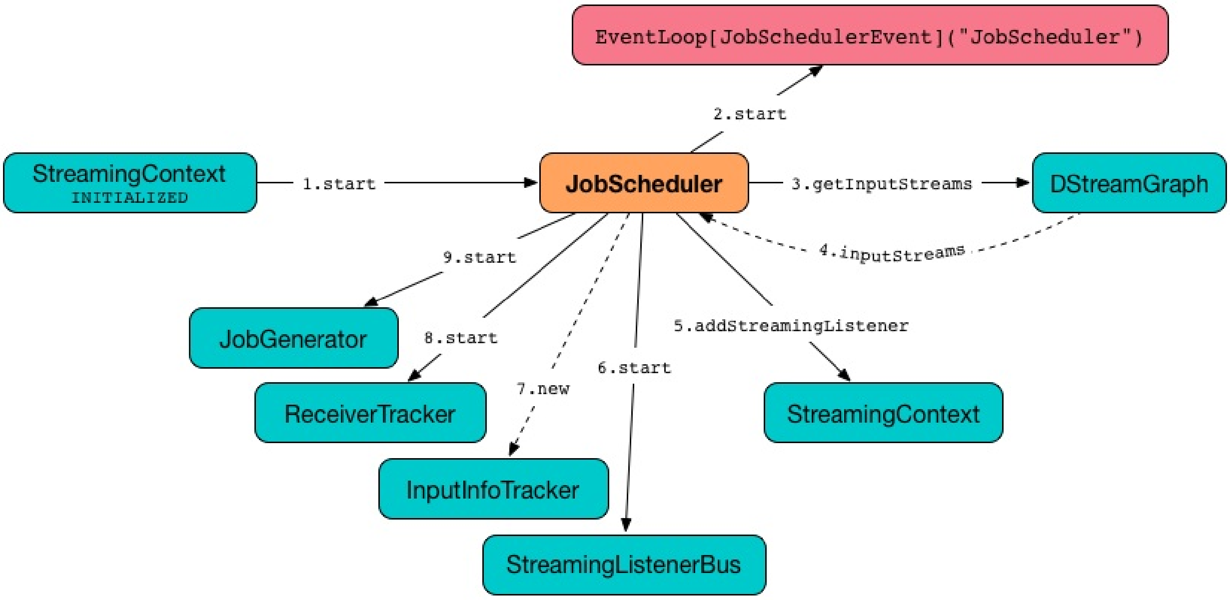

Figure 2. JobScheduler Start procedure

It first starts JobSchedulerEvent Handler.

It asks DStreamGraph for input dstreams and registers their RateControllers (if defined) as streaming listeners. It starts StreamingListenerBus afterwards.

It creates ReceiverTracker and InputInfoTracker. It then starts the `ReceiverTracker`.

It starts JobGenerator.

Just before `start` finishes, you should see the following INFO message in the logs:

```
INFO JobScheduler: Started JobScheduler
```

Pending Batches to Process (getPendingTimes method)

Caution	FIXME

Stopping JobScheduler (stop method)

```
stop(processAllReceivedData: Boolean): Unit
```

`stop` stops `JobScheduler`.

Note	It is called when StreamingContext is being stopped.

You should see the following DEBUG message in the logs:

```
DEBUG JobScheduler: Stopping JobScheduler
```

ReceiverTracker is stopped.

Note	ReceiverTracker is only assigned (and started) while JobScheduler is starting.

It stops generating jobs.

You should see the following DEBUG message in the logs:

```
DEBUG JobScheduler: Stopping job executor
```

jobExecutor Thread Pool is shut down (using `jobExecutor.shutdown()`).

If the stop should wait for all received data to be processed (the input parameter `processAllReceivedData` is `true`), `stop` awaits termination of jobExecutor Thread Pool for **1 hour** (it is assumed that it is enough and is not configurable). Otherwise, it waits for **2 seconds**.

jobExecutor Thread Pool is forcefully shut down (using `jobExecutor.shutdownNow()`) unless it has terminated already.

You should see the following DEBUG message in the logs:

```
DEBUG JobScheduler: Stopped job executor
```

StreamingListenerBus and eventLoop - JobSchedulerEvent Handler are stopped.

You should see the following INFO message in the logs:

```
INFO JobScheduler: Stopped JobScheduler
```

Submitting Collection of Jobs for Execution — `submitJobSet` method

When `submitJobSet(jobSet: JobSet)` is called, it reacts appropriately per `jobSet` JobSet given.

Note	The method is called by JobGenerator only (as part of JobGenerator.generateJobs and JobGenerator.restart).

When no streaming jobs are inside the `jobSet`, you should see the following INFO in the logs:

```
INFO JobScheduler: No jobs added for time [jobSet.time]
```

Otherwise, when there is at least one streaming job inside the `jobSet`, StreamingListenerBatchSubmitted (with data statistics of every registered input stream for which the streaming jobs were generated) is posted to StreamingListenerBus.

The JobSet is added to the internal jobSets registry.

It then goes over every streaming job in the `jobSet` and executes a JobHandler (on jobExecutor Thread Pool).

At the end, you should see the following INFO message in the logs:

```
INFO JobScheduler: Added jobs for time [jobSet.time] ms
```

JobHandler

`JobHandler` is a thread of execution for a streaming job (that simply calls `Job.run`).

Note	It is called when a new JobSet is submitted (see submitJobSet in this document).

When started, it prepares the environment (so the streaming job can be nicely displayed in the web UI under `/streaming/batch/?id=[milliseconds]`) and posts `JobStarted` event to JobSchedulerEvent event loop.

It runs the streaming job that executes the job function as defined while generating a streaming job for an output stream.

Note	This is when Spark is requested to run a Spark job.

You may see similar-looking INFO messages in the logs (it depends on the operators you use):

```
INFO SparkContext: Starting job: print at <console>:39
INFO DAGScheduler: Got job 0 (print at <console>:39) with 1 output partitions
...
INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 0 (KafkaRDD[2] at creat
eDirectStream at <console>:36)
...
INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 987 bytes result sent to driver
...
INFO DAGScheduler: Job 0 finished: print at <console>:39, took 0.178689 s
```

It posts `JobCompleted` event to JobSchedulerEvent event loop.

jobExecutor Thread Pool

While `JobScheduler` is instantiated, the daemon thread pool `streaming-job-executor-ID` with spark.streaming.concurrentJobs threads is created.

It is used to execute JobHandler for jobs in JobSet (see submitJobSet in this document).

It shuts down when StreamingContext stops.

eventLoop - JobSchedulerEvent Handler

JobScheduler uses `EventLoop` for `JobSchedulerEvent` events. It accepts JobStarted and JobCompleted events. It also processes `ErrorReported` events.

JobStarted and JobScheduler.handleJobStart

When `JobStarted` event is received, `JobScheduler.handleJobStart` is called.

Note	It is JobHandler to post `JobStarted` .

`handleJobStart(job: Job, startTime: Long)` takes a `JobSet` (from `jobSets`) and checks whether it has already been started.

It posts `StreamingListenerBatchStarted` to StreamingListenerBus when the JobSet is about to start.

It posts `StreamingListenerOutputOperationStarted` to StreamingListenerBus.

You should see the following INFO message in the logs:

```
INFO JobScheduler: Starting job [job.id] from job set of time [jobSet.time] ms
```

JobCompleted and JobScheduler.handleJobCompletion

When `JobCompleted` event is received, it triggers `JobScheduler.handleJobCompletion(job: Job, completedTime: Long)`.

Note	JobHandler posts `JobCompleted` events when it finishes running a streaming job.

`handleJobCompletion` looks the JobSet up (from the jobSets internal registry) and calls JobSet.handleJobCompletion(job) (that marks the `JobSet` as completed when no more streaming jobs are incomplete). It also calls `Job.setEndTime(completedTime)`.

It posts `StreamingListenerOutputOperationCompleted` to StreamingListenerBus.

You should see the following INFO message in the logs:

```
INFO JobScheduler: Finished job [job.id] from job set of time [jobSet.time] ms
```

If the entire JobSet is completed, it removes it from jobSets, and calls JobGenerator.onBatchCompletion.

You should see the following INFO message in the logs:

```
INFO JobScheduler: Total delay: [totalDelay] s for time [time] ms (execution: [process
ingDelay] s)
```

It posts `StreamingListenerBatchCompleted` to StreamingListenerBus.

It reports an error if the job's result is a failure.

StreamingListenerBus and StreamingListenerEvents

`StreamingListenerBus` is a asynchronous listener bus to post `StreamingListenerEvent` events to streaming listeners.

Internal Registries

`JobScheduler` maintains the following information in internal registries:

- `jobSets` - a mapping between time and JobSets. See JobSet.

JobSet

A `JobSet` represents a collection of streaming jobs that were created at (batch) `time` for output streams (that have ultimately produced a streaming job as they may opt out).

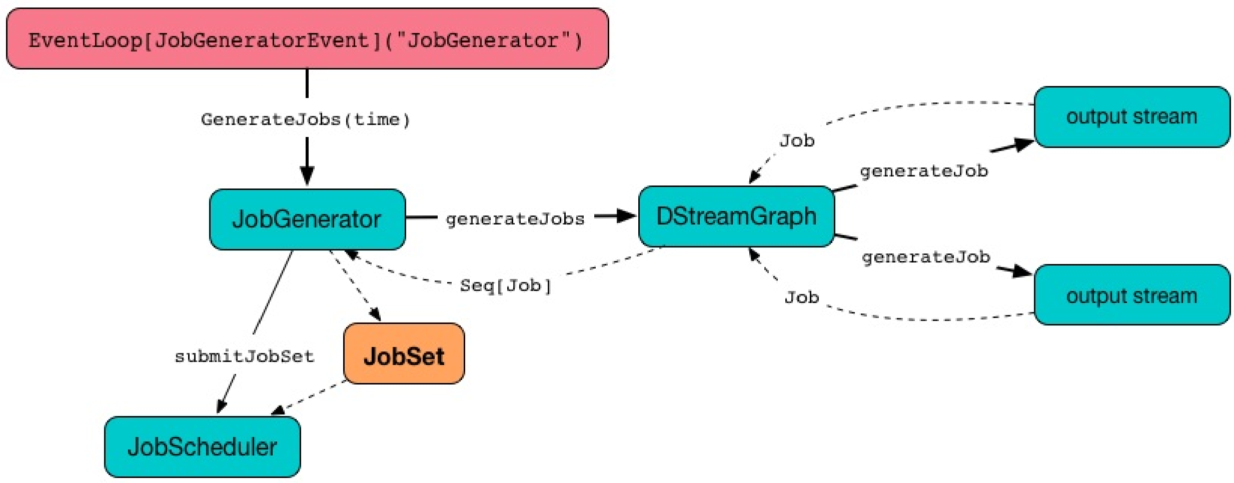

Figure 3. JobSet Created and Submitted to JobScheduler

`JobSet` tracks what streaming jobs are in incomplete state (in `incompleteJobs` internal registry).

Note	At the beginning (when `JobSet` is created) all streaming jobs are incomplete.

Caution	FIXME There is a duplication in how streaming jobs are tracked as completed since a `Job` knows about its `_endTime`. Is this a optimization? How much time does it buy us?

A `JobSet` tracks the following moments in its lifecycle:

- `submissionTime` being the time when the instance was created.
- `processingStartTime` being the time when the first streaming job in the collection was started.
- `processingEndTime` being the time when the last streaming job in the collection finished processing.

A `JobSet` changes state over time. It can be in the following states:

- **Created** after a `JobSet` was created. `submissionTime` is set.
- **Started** after `JobSet.handleJobStart` was called. `processingStartTime` is set.
- **Completed** after `JobSet.handleJobCompletion` and no more jobs are incomplete (in `incompleteJobs` internal registry). `processingEndTime` is set.

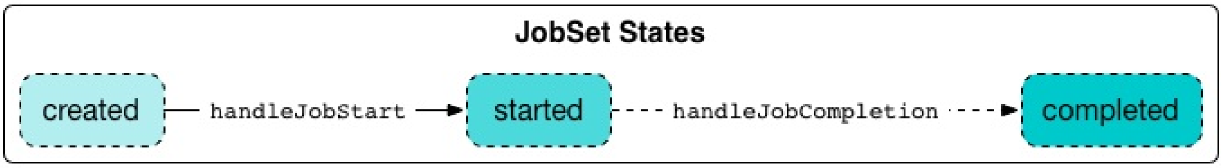

Figure 4. JobSet States

Given the states a `JobSet` has **delays**:

- **Processing delay** is the time spent for processing all the streaming jobs in a `JobSet` from the time the very first job was started, i.e. the time between started and completed states.
- **Total delay** is the time from the batch time until the `JobSet` was completed.

Note	Total delay is always longer than processing delay.

You can map a `JobSet` to a `BatchInfo` using `toBatchInfo` method.

Note	`BatchInfo` is used to create and post StreamingListenerBatchSubmitted, StreamingListenerBatchStarted, and StreamingListenerBatchCompleted events.

`JobSet` is used (created or processed) in:

- JobGenerator.generateJobs
- JobScheduler.submitJobSet(jobSet: JobSet)
- JobGenerator.restart
- JobScheduler.handleJobStart(job: Job, startTime: Long)
- JobScheduler.handleJobCompletion(job: Job, completedTime: Long)

InputInfoTracker

`InputInfoTracker` tracks batch times and input record statistics for all registered input dstreams. It is used when `JobGenerator` submits streaming jobs for a batch interval and in turn propagated to streaming listeners (as StreamingListenerBatchSubmitted events).

Note	`InputInfoTracker` is managed by JobScheduler, i.e. it is created when JobScheduler starts and is stopped alongside.

`InputInfoTracker` uses internal registry batchTimeToInputInfos to maintain the mapping of batch times and input dstreams (i.e. another mapping between input stream ids and StreamInputInfo).

`InputInfoTracker` accumulates batch statistics for every batch when input streams are computing RDDs (and call reportInfo).

Note	It is up to input dstreams to have these batch statistics collected (and requires calling reportInfo method explicitly). The following input streams report information: • DirectKafkaInputDStream • ReceiverInputDStreams — Input Streams with Receivers • FileInputDStream

Tip	Enable `INFO` logging level for `org.apache.spark.streaming.scheduler.InputInfoTracker` logger to see what happens inside. Add the following line to `conf/log4j.properties` : `log4j.logger.org.apache.spark.streaming.scheduler.InputInfoTracker=INFO` Refer to Logging.

Batch Intervals and Input DStream Statistics — `batchTimeToInputInfos` Registry

```
batchTimeToInputInfos: HashMap[Time, HashMap[Int, StreamInputInfo]]
```

`batchTimeToInputInfos` keeps track of batches (`Time`) with input dstreams (`Int`) that reported their statistics per batch.

Reporting Input DStream Statistics for Batch — `reportInfo` Method

```
reportInfo(batchTime: Time, inputInfo: StreamInputInfo): Unit
```

`reportInfo` adds the input `inputInfo` for the `batchTime` to batchTimeToInputInfos.

Internally, `reportInfo` accesses the input dstream reports for `batchTime` using the internal `batchTimeToInputInfos` registry (creating a new empty one if `batchTime` has not been registered yet).

`reportInfo` then makes sure that the `inputInfo` input dstream has not been registered already for the input `batchTime` and throws a `IllegalStateException` otherwise.

```
Input stream [inputStreamId] for batch [batchTime] is already added into InputInfoTrac
ker, this is an illegal state
```

Ultimatelly, `reportInfo` adds the input report to `batchTimeToInputInfos` .

Requesting Statistics For Input DStreams For Batch — `getInfo` Method

```
getInfo(batchTime: Time): Map[Int, StreamInputInfo]
```

`getInfo` returns all the reported input dstream statistics for `batchTime` . It returns an empty collection if there are no reports for a batch.

Note	`getInfo` is used when `JobGenerator` has successfully generated streaming jobs (and submits the jobs to `JobScheduler`).

Removing Batch Statistics — `cleanup` Method

```
cleanup(batchThreshTime: Time): Unit
```

`cleanup` removes statistics for batches older than `batchThreshTime` . It removes the batches from batchTimeToInputInfos registry.

When executed, you should see the following INFO message (akin to *garbage collection*):

```
INFO InputInfoTracker: remove old batch metadata: [timesToCleanup]
```

`StreamInputInfo` — Input Record Statistics

`StreamInputInfo` is used by input dstreams to report their statistics with `InputInfoTracker`.

`StreamInputInfo` contains:

1. The id of the input dstream
2. The number of records in a batch
3. A metadata (with `Description`)

Note	`Description` is used in `BatchPage` (Details of batch) in web UI for Streaming under `Input Metadata`.

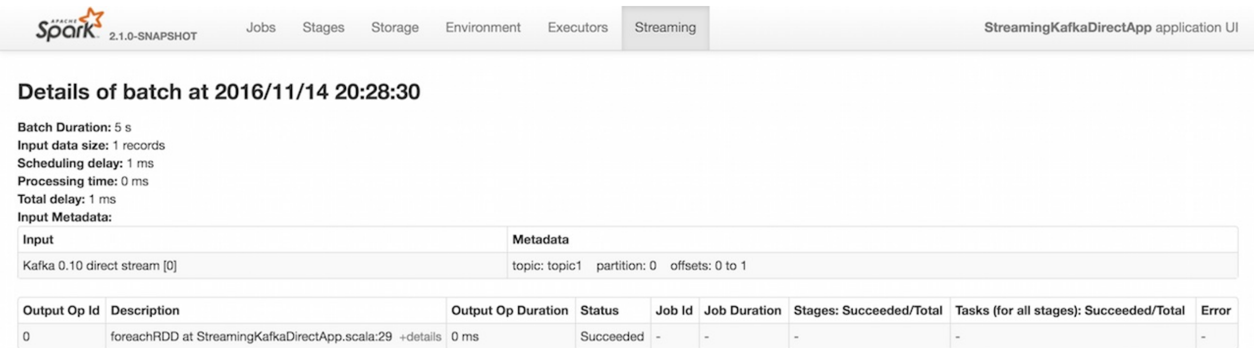

Figure 1. Details of batch in web UI for Kafka 0.10 direct stream with Metadata

JobGenerator

`JobGenerator` asynchronously generates streaming jobs every batch interval (using recurring timer) that may or may not be checkpointed afterwards. It also periodically requests clearing up metadata and checkpoint data for each input dstream.

Note	`JobGenerator` is completely owned and managed by JobScheduler, i.e. `JobScheduler` creates an instance of JobGenerator and starts it (while being started itself).

Tip	Enable `INFO` or `DEBUG` logging level for `org.apache.spark.streaming.scheduler.JobGenerator` logger to see what happens inside. Add the following line to `conf/log4j.properties` : `log4j.logger.org.apache.spark.streaming.scheduler.JobGenerator=DEBUG` Refer to Logging.

Starting JobGenerator (start method)

```
start(): Unit
```

`start` method creates and starts the internal JobGeneratorEvent handler.

Note	`start` is called when JobScheduler starts.

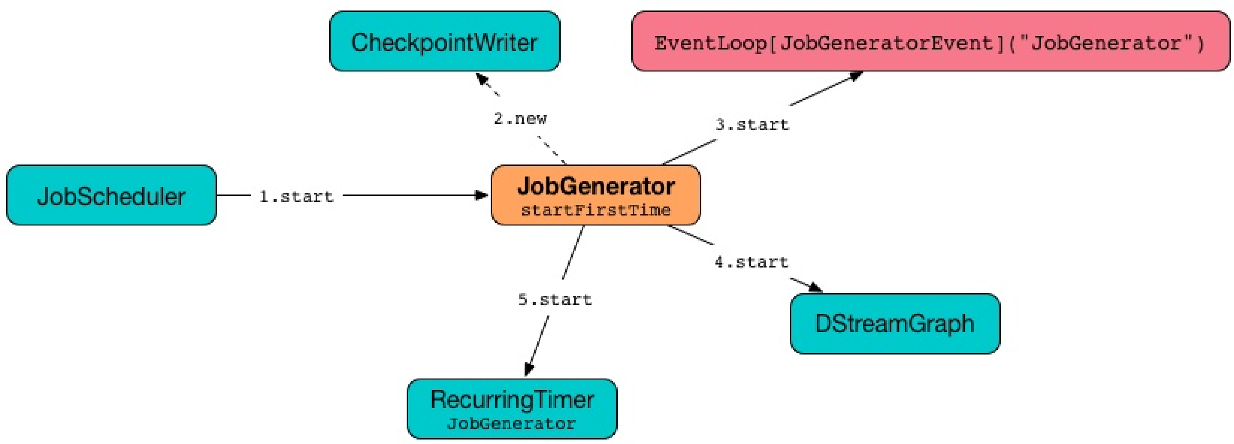

Figure 1. JobGenerator Start (First Time) procedure (tip: follow the numbers)

It first checks whether or not the internal event loop has already been created which is the way to know that the JobScheduler was started. If so, it does nothing and exits.

Only if checkpointing is enabled, it creates CheckpointWriter.

It then creates and starts the internal JobGeneratorEvent handler.

Depending on whether checkpoint directory is available or not it restarts itself or starts, respectively.

Start Time and startFirstTime Method

```
startFirstTime(): Unit
```

`startFirstTime` starts DStreamGraph and the timer.

Note	`startFirstTime` is called when JobGenerator starts (and no checkpoint directory is available).

It first requests timer for the **start time** and passes the start time along to DStreamGraph.start and RecurringTimer.start.

Note	The start time has the property of being a multiple of batch interval and after the current system time. It is in the hands of recurring timer to calculate a time with the property given a batch interval.

Note	Because of the property of the start time, DStreamGraph.start is passed the time of one batch interval before the calculated start time.

Note	When recurring timer starts for `JobGenerator`, you should see the following INFO message in the logs: `INFO RecurringTimer: Started timer for JobGenerator at time [nextTime]`

Right before the method finishes, you should see the following INFO message in the logs:

```
INFO JobGenerator: Started JobGenerator at [startTime] ms
```

Stopping JobGenerator (stop method)

```
stop(processReceivedData: Boolean): Unit
```

`stop` stops a `JobGenerator`. The `processReceivedData` flag tells whether to stop `JobGenerator` gracefully, i.e. after having processed all received data and pending streaming jobs, or not.

Note	`JobGenerator` is stopped as JobScheduler stops. `processReceivedData` flag in `JobGenerator` corresponds to the value of `processAllReceivedData` in `JobScheduler`.

It first checks whether `eventLoop` internal event loop was ever started (through checking `null`).

Warning	It doesn't set `eventLoop` to `null` (but it is assumed to be the marker).

When `JobGenerator` should stop immediately, i.e. ignoring unprocessed data and pending streaming jobs (`processReceivedData` flag is disabled), you should see the following INFO message in the logs:

```
INFO JobGenerator: Stopping JobGenerator immediately
```

It requests the timer to stop forcefully (`interruptTimer` is enabled) and stops the graph.

Otherwise, when `JobGenerator` should stop gracefully, i.e. `processReceivedData` flag is enabled, you should see the following INFO message in the logs:

```
INFO JobGenerator: Stopping JobGenerator gracefully
```

You should immediately see the following INFO message in the logs:

```
INFO JobGenerator: Waiting for all received blocks to be consumed for job generation
```

`JobGenerator` waits spark.streaming.gracefulStopTimeout milliseconds or until ReceiverTracker has any blocks left to be processed (whatever is shorter) before continuing.

Note	Poll (sleeping) time is `100` milliseconds and is not configurable.

When a timeout occurs, you should see the WARN message in the logs:

```
WARN JobGenerator: Timed out while stopping the job generator (timeout = [stopTimeoutM
s])
```

After the waiting is over, you should see the following INFO message in the logs:

```
INFO JobGenerator: Waited for all received blocks to be consumed for job generation
```

It requests timer to stop generating streaming jobs (`interruptTimer` flag is disabled) and stops the graph.

You should see the following INFO message in the logs:

```
INFO JobGenerator: Stopped generation timer
```

You should immediately see the following INFO message in the logs:

```
INFO JobGenerator: Waiting for jobs to be processed and checkpoints to be written
```

`JobGenerator` waits spark.streaming.gracefulStopTimeout milliseconds or until all the batches have been processed (whatever is shorter) before continuing. It waits for batches to complete using last processed batch internal property that should eventually be exactly the time when the timer was stopped (it returns the last time for which the streaming job was generated).

Note	spark.streaming.gracefulStopTimeout is ten times the batch interval by default.

After the waiting is over, you should see the following INFO message in the logs:

```
INFO JobGenerator: Waited for jobs to be processed and checkpoints to be written
```

Regardless of `processReceivedData` flag, if checkpointing was enabled, it stops CheckpointWriter.

It then stops the event loop.

As the last step, when `JobGenerator` is assumed to be stopped completely, you should see the following INFO message in the logs:

```
INFO JobGenerator: Stopped JobGenerator
```

Starting from Checkpoint (restart method)

```
restart(): Unit
```

`restart` starts `JobGenerator` from checkpoint. It basically reconstructs the runtime environment of the past execution that may have stopped immediately, i.e. without waiting for all the streaming jobs to complete when checkpoint was enabled, or due to a abrupt shutdown (a unrecoverable failure or similar).

Note	`restart` is called when JobGenerator starts and checkpoint is present.

`restart` first calculates the batches that may have been missed while `JobGenerator` was down, i.e. batch times between the current restart time and the time of initial checkpoint.

Warning	`restart` doesn't check whether the initial checkpoint exists or not that may lead to NPE.

You should see the following INFO message in the logs:

```
INFO JobGenerator: Batches during down time ([size] batches): [downTimes]
```

It then ask the initial checkpoint for pending batches, i.e. the times of streaming job sets.

Caution	FIXME What are the pending batches? Why would they ever exist?

You should see the following INFO message in the logs:

```
INFO JobGenerator: Batches pending processing ([size] batches): [pendingTimes]
```

It then computes the batches to reschedule, i.e. pending and down time batches that are before restart time.

You should see the following INFO message in the logs:

```
INFO JobGenerator: Batches to reschedule ([size] batches): [timesToReschedule]
```

For each batch to reschedule, `restart` requests ReceiverTracker to allocate blocks to batch and submits streaming job sets for execution.

Note	`restart` mimics generateJobs method.

It restarts the timer (by using `restartTime` as `startTime`).

You should see the following INFO message in the logs:

```
INFO JobGenerator: Restarted JobGenerator at [restartTime]
```

Last Processed Batch (aka lastProcessedBatch)

JobGenerator tracks the last batch time for which the batch was completed and cleanups performed as `lastProcessedBatch` internal property.

The only purpose of the `lastProcessedBatch` property is to allow for stopping the streaming context gracefully, i.e. to wait until all generated streaming jobs are completed.

Note	It is set to the batch time after ClearMetadata Event is processed (when checkpointing is disabled).

JobGenerator eventLoop and JobGeneratorEvent Handler

`JobGenerator` uses the internal `EventLoop` event loop to process `JobGeneratorEvent` events asynchronously (one event at a time) on a separate dedicated *single* thread.

Note	`EventLoop` uses unbounded java.util.concurrent.LinkedBlockingDeque.

For every `JobGeneratorEvent` event, you should see the following DEBUG message in the logs:

```
DEBUG JobGenerator: Got event [event]
```

There are 4 `JobGeneratorEvent` event types:

- GenerateJobs
- DoCheckpoint
- ClearMetadata
- ClearCheckpointData

See below in the document for the extensive coverage of the supported `JobGeneratorEvent` event types.

`GenerateJobs` Event and `generateJobs` method

Note	`GenerateJobs` events are posted regularly by the internal `timer` RecurringTimer every batch interval. The `time` parameter is exactly the current batch time.

When `GenerateJobs(time: Time)` event is received the internal `generateJobs` method is called that submits a collection of streaming jobs for execution.

```
generateJobs(time: Time)
```

It first calls ReceiverTracker.allocateBlocksToBatch (it does nothing when there are no receiver input streams in use), and then requests DStreamGraph for streaming jobs for a given batch time.

If the above two calls have finished successfully, `InputInfoTracker` is requested for record statistics of every registered input dstream for the given batch `time` that, together with the collection of streaming jobs (from DStreamGraph), is then passed on to JobScheduler.submitJobSet (as a JobSet).

In case of failure, `JobScheduler.reportError` is called.

Ultimately, DoCheckpoint event is posted (with `clearCheckpointDataLater` being disabled, i.e. `false`).

DoCheckpoint Event and doCheckpoint method

Note	`DoCheckpoint` events are posted by JobGenerator itself as part of generating streaming jobs (with `clearCheckpointDataLater` being disabled, i.e. `false`) and clearing metadata (with `clearCheckpointDataLater` being enabled, i.e. `true`).

`DoCheckpoint` events trigger execution of `doCheckpoint` method.

```
doCheckpoint(time: Time, clearCheckpointDataLater: Boolean)
```

If checkpointing is disabled or the current batch `time` is not eligible for checkpointing, the method does nothing and exits.

Note	A current batch is **eligible for checkpointing** when the time interval between current batch `time` and zero time is a multiple of checkpoint interval.

Caution	FIXME Who checks and when whether checkpoint interval is greater than batch interval or not? What about checking whether a checkpoint interval is a multiple of batch time?

Caution	FIXME What happens when you start a StreamingContext with a checkpoint directory that was used before?

Otherwise, when checkpointing should be performed, you should see the following INFO message in the logs:

```
INFO JobGenerator: Checkpointing graph for time [time] ms
```

It requests DStreamGraph for updating checkpoint data and CheckpointWriter for writing a new checkpoint. Both are given the current batch `time` .

ClearMetadata Event and clearMetadata method

Note	`ClearMetadata` are posted after a micro-batch for a batch time has completed.

It removes old RDDs that have been generated and collected so far by output streams (managed by DStreamGraph). It is a sort of *garbage collector*.

When `ClearMetadata(time)` arrives, it first asks DStreamGraph to clear metadata for the given time.

If checkpointing is enabled, it posts a DoCheckpoint event (with `clearCheckpointDataLater` being enabled, i.e. `true`) and exits.

Otherwise, when checkpointing is disabled, it asks DStreamGraph for the maximum remember duration across all the input streams and requests ReceiverTracker and the InputInfoTracker to do their cleanups.

Caution	FIXME Describe cleanups of ReceiverTracker.

Eventually, it marks the batch as fully processed, i.e. that the batch completed as well as checkpointing or metadata cleanups, using the internal lastProcessedBatch marker.

ClearCheckpointData Event and clearCheckpointData method

Note	`ClearCheckpointData` event is posted after checkpoint is saved and checkpoint cleanup is requested.

`ClearCheckpointData` events trigger execution of `clearCheckpointData` method.

```
clearCheckpointData(time: Time)
```

In short, `clearCheckpointData` requests the DStreamGraph, ReceiverTracker, and InputInfoTracker to do their cleaning and marks the current batch `time` as fully processed.

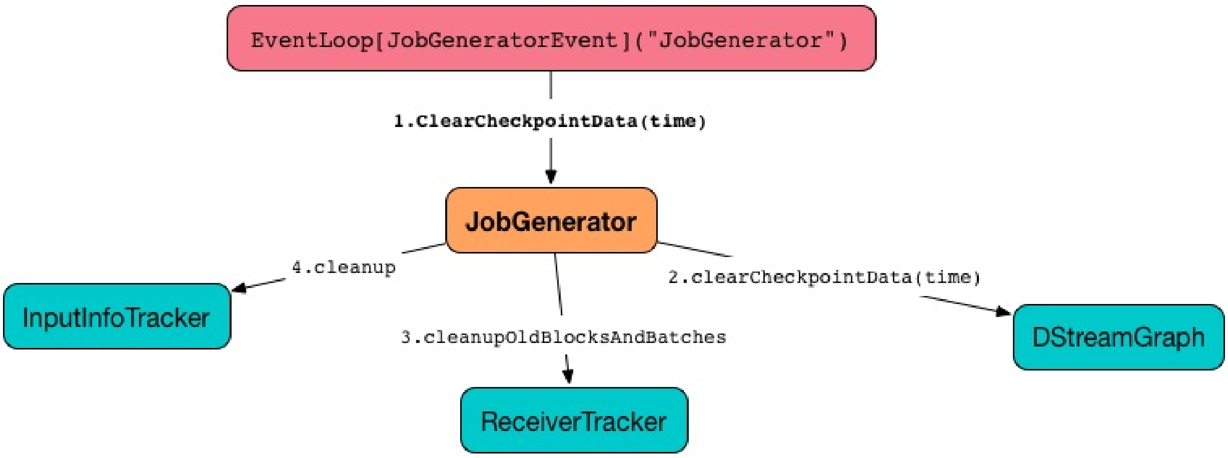

Figure 2. JobGenerator and ClearCheckpointData event

When executed, `clearCheckpointData` first requests DStreamGraph to clear checkpoint data for the given batch time.

It then asks `DStreamGraph` for the maximum remember interval. Given the maximum remember interval `JobGenerator` requests `ReceiverTracker` to cleanup old blocks and batches and `InputInfoTracker` to do cleanup for data accumulated before the maximum remember interval (from `time`).

Having done that, the current batch `time` is marked as fully processed.

Whether or Not to Checkpoint (aka shouldCheckpoint)

`shouldCheckpoint` flag is used to control a CheckpointWriter as well as whether to post DoCheckpoint in clearMetadata or not.

`shouldCheckpoint` flag is enabled (i.e. `true`) when checkpoint interval and checkpoint directory are defined (i.e. not `null`) in StreamingContext.

Note	However the flag is completely based on the properties of StreamingContext, these dependent properties are used by JobScheduler only. *Really?*

Caution	FIXME Report an issue When and what for are they set? Can one of `ssc.checkpointDuration` and `ssc.checkpointDir` be `null`? Do they all have to be set and is this checked somewhere? Answer: See Setup Validation.

Caution	Potential bug: Can `StreamingContext` have no checkpoint duration set? At least, the batch interval **must** be set. In other words, it's StreamingContext to say whether to checkpoint or not and there should be a method in StreamingContext *not* JobGenerator.

onCheckpointCompletion

Caution	FIXME

timer RecurringTimer

`timer` RecurringTimer (with the name being `JobGenerator`) is used to posts GenerateJobs events to the internal JobGeneratorEvent handler every batch interval.

Note	`timer` is created when `JobGenerator` is. It starts when JobGenerator starts (for the first time only).

DStreamGraph

`DStreamGraph` (is a final helper class that) manages **input** and **output dstreams**. It also holds zero time for the other components that marks the time when it was started.

`DStreamGraph` maintains the collections of InputDStream instances (as `inputStreams`) and output DStream instances (as `outputStreams`), but, more importantly, it generates streaming jobs for output streams for a batch (time).

`DStreamGraph` holds the batch interval for the other parts of a Streaming application.

<table>
<tr>
<td>Tip</td>
<td>
Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.streaming.DStreamGraph</code> logger to see what happens in <code>DStreamGraph</code> .
Add the following line to <code>conf/log4j.properties</code> :
<pre>log4j.logger.org.apache.spark.streaming.DStreamGraph=DEBUG</pre>
Refer to Logging.
</td>
</tr>
</table>

Zero Time (aka zeroTime)

Zero time (internally `zeroTime`) is the time when DStreamGraph has been started.

It is passed on down the output dstream graph so output dstreams can initialize themselves.

Start Time (aka startTime)

Start time (internally `startTime`) is the time when DStreamGraph has been started or restarted.

Note	At regular start start time is exactly zero time.

Batch Interval (aka batchDuration)

`DStreamGraph` holds the **batch interval** (as `batchDuration`) for the other parts of a Streaming application.

`setBatchDuration(duration: Duration)` is the method to set the batch interval.

It appears that it is *the* place for the value since it must be set before JobGenerator can be instantiated.

It *is* set while StreamingContext is being instantiated and is validated (using `validate()` method of `StreamingContext` and `DStreamGraph`) before `StreamingContext` is started.

Maximum Remember Interval — getMaxInputStreamRememberDuration Method

```
getMaxInputStreamRememberDuration(): Duration
```

Maximum Remember Interval is the maximum remember interval across all the input dstreams. It is calculated using `getMaxInputStreamRememberDuration` method.

Note	It is called when JobGenerator is requested to clear metadata and checkpoint data.

Input DStreams Registry

Caution	FIXME

Output DStreams Registry

`DStream` by design has no notion of being an output dstream. To mark a dstream as output you need to register a dstream (using DStream.register method) which happens for…FIXME

Starting `DStreamGraph`

```
start(time: Time): Unit
```

When `DStreamGraph` is started (using `start` method), it sets zero time and start time.

Note	`start` method is called when JobGenerator starts for the first time (not from a checkpoint).

Note	You can start `DStreamGraph` as many times until `time` is not `null` and zero time has been set.

(*output dstreams*) `start` then walks over the collection of output dstreams and for each output dstream, one at a time, calls their initialize(zeroTime), remember (with the current remember interval), and validateAtStart methods.

(*input dstreams*) When all the output streams are processed, it starts the input dstreams (in parallel) using `start` method.

Stopping `DStreamGraph`

```
stop(): Unit
```

Caution	FIXME

Restarting `DStreamGraph`

```
restart(time: Time): Unit
```

`restart` sets start time to be `time` input parameter.

Note	This is the only moment when zero time can be different than start time.

Caution	`restart` doesn't seem to be called ever.

Generating Streaming Jobs for Output DStreams for Batch Time — `generateJobs` Method

```
generateJobs(time: Time): Seq[Job]
```

`generateJobs` method generates a collection of streaming jobs for output streams for a given batch `time`. It walks over each registered output stream (in `outputStreams` internal registry) and requests each stream for a streaming job

Note	`generateJobs` is called by JobGenerator to generate jobs for a given batch time or when restarted from checkpoint.

When `generateJobs` method executes, you should see the following DEBUG message in the logs:

```
DEBUG DStreamGraph: Generating jobs for time [time] ms
```

`generateJobs` then walks over each registered output stream (in `outputStreams` internal registry) and requests the streams for a streaming job.

Right before the method finishes, you should see the following DEBUG message with the number of streaming jobs generated (as `jobs.length`):

```
DEBUG DStreamGraph: Generated [jobs.length] jobs for time [time] ms
```

Validation Check

`validate()` method checks whether batch duration and at least one output stream have been set. It will throw `java.lang.IllegalArgumentException` when either is not.

Note	It is called when StreamingContext starts.

Metadata Cleanup

Note	It is called when JobGenerator clears metadata.

When `clearMetadata(time: Time)` is called, you should see the following DEBUG message in the logs:

```
DEBUG DStreamGraph: Clearing metadata for time [time] ms
```

It merely walks over the collection of output streams and (synchronously, one by one) asks to do its own metadata cleaning.

When finishes, you should see the following DEBUG message in the logs:

```
DEBUG DStreamGraph: Cleared old metadata for time [time] ms
```

Restoring State for Output DStreams — `restoreCheckpointData` Method

```
restoreCheckpointData(): Unit
```

When `restoreCheckpointData()` is executed, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Restoring checkpoint data
```

Then, every output dstream is requested to restoreCheckpointData.

At the end, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Restored checkpoint data
```

Note	`restoreCheckpointData` is executed when StreamingContext is recreated from checkpoint.

Updating Checkpoint Data — `updateCheckpointData` Method

```
updateCheckpointData(time: Time): Unit
```

Note	`updateCheckpointData` is called when JobGenerator processes DoCheckpoint events.

When `updateCheckpointData` is called, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Updating checkpoint data for time [time] ms
```

It then walks over every output dstream and calls its updateCheckpointData(time).

When `updateCheckpointData` finishes it prints out the following INFO message to the logs:

```
INFO DStreamGraph: Updated checkpoint data for time [time] ms
```

Checkpoint Cleanup — `clearCheckpointData` Method

```
clearCheckpointData(time: Time)
```

Note	`clearCheckpointData` is called when JobGenerator clears checkpoint data.

When `clearCheckpointData` is called, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Clearing checkpoint data for time [time] ms
```

It merely walks through the collection of output streams and (synchronously, one by one) asks to do their own checkpoint data cleaning.

When finished, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Cleared checkpoint data for time [time] ms
```

Remember Interval

Remember interval is the time to remember (aka *cache*) the RDDs that have been generated by (output) dstreams in the context (before they are released and garbage collected).

It can be set using remember method.

`remember` Method

```
remember(duration: Duration): Unit
```

`remember` method simply sets remember interval and exits.

Note	It is called by StreamingContext.remember method.

It first checks whether or not it has been set already and if so, throws `java.lang.IllegalArgumentException` as follows:

```
java.lang.IllegalArgumentException: requirement failed: Remember
duration already set as [rememberDuration] ms. Cannot set it
again.
  at scala.Predef$.require(Predef.scala:219)
  at
org.apache.spark.streaming.DStreamGraph.remember(DStreamGraph.sc
ala:79)
  at
org.apache.spark.streaming.StreamingContext.remember(StreamingCo
ntext.scala:222)
  ... 43 elided
```

Note	It only makes sense to call `remember` method before DStreamGraph is started, i.e. before StreamingContext is started, since the output dstreams are only given the remember interval when DStreamGraph starts.

`DStream` — Discretized Stream

Discretized Stream (DStream) is the fundamental concept of Spark Streaming. It is basically a stream of RDDs with elements being the data received from input streams for batch (possibly extended in scope by windowed or stateful operators).

There is no notion of input and output dstreams. DStreams are all instances of `DStream` abstract class (see DStream Contract in this document). You may however *correctly* assume that all dstreams are input. And it happens to be so until you register a dstream that marks it as output.

Table 1. `DStream` 's Internal Properties

Name	Initial Value	Description
`storageLevel`	`NONE`	StorageLevel of the RDDs in the `DStream` .
`restoredFromCheckpointData`	`false`	The flag to inform whether it was restored from checkpoint.
`graph`	`null`	The reference to DStreamGraph.

A `DStream` is represented as org.apache.spark.streaming.dstream.DStream abstract class.

> Tip
>
> Enable `INFO` or `DEBUG` logging level for `org.apache.spark.streaming.dstream.DStream` logger to see what happens inside a `DStream` .
>
> Add the following line to `conf/log4j.properties` :
>
> ```
> log4j.logger.org.apache.spark.streaming.dstream.DStream=DEBUG
> ```
>
> Refer to Logging.

DStream Contract

A `DStream` is defined by the following properties (with the names of the corresponding methods that subclasses have to implement):

- **dstream dependencies**, i.e. a collection of `DStreams` that this `DStream` depends on. They are often referred to as **parent dstreams**.

```
def dependencies: List[DStream[_]]
```

- **slide duration** (aka *slide interval*), i.e. a time interval after which the stream is requested to generate a RDD out of input data it consumes.

```
def slideDuration: Duration
```

- How to **compute** (*generate*) an optional RDD for the given batch if any. `validTime` is a point in time that marks the end boundary of slide duration.

```
def compute(validTime: Time): Option[RDD[T]]
```

Creating DStreams

You can create dstreams through the built-in input stream constructors using streaming context or more specialized add-ons for external input data sources, e.g. Apache Kafka.

Note	DStreams can only be created before `StreamingContext` is started.

Zero Time (aka zeroTime)

Zero time (internally `zeroTime`) is the time when a dstream was initialized.

It serves as the initialization marker (via `isInitialized` method) and helps calculating intervals for RDD checkpointing (when checkpoint interval is set and the current batch time is a multiple thereof), slicing, and the time validation for a batch (when a dstream generates a RDD).

Remember Interval (aka rememberDuration)

Remember interval (internally `rememberDuration`) is the time interval for how long a dstream is supposed to remember (aka *cache*) RDDs created. This is a mandatory attribute of every dstream which is validated at startup.

Note	It is used for metadata cleanup of a dstream.

Initially, when a dstream is created, the remember interval is not set (i.e. `null`), but is set when the dstream is initialized.

It can be set to a custom value using remember method.

Note	You may see the current value of remember interval when a dstream is validated at startup and the log level is INFO.

`generatedRDDs` - Internal Cache of Batch Times and Corresponding RDDs

`generatedRDDs` is an internal collection of pairs of batch times and the corresponding RDDs that were generated for the batch. It acts as a cache when a dstream is requested to compute a RDD for batch (i.e. `generatedRDDs` may already have the RDD or gets a new RDD added).

`generatedRDDs` is empty initially, i.e. when a dstream is created.

It is a *transient* data structure so it is not serialized when a dstream is. It is initialized to an empty collection when deserialized. You should see the following DEBUG message in the logs when it happens:

```
DEBUG [the simple class name of dstream].readObject used
```

As new RDDs are added, dstreams offer a way to clear the old metadata during which the old RDDs are removed from `generatedRDDs` collection.

If checkpointing is used, `generatedRDDs` collection can be recreated from a storage.

Initializing DStreams — `initialize` Method

```
initialize(time: Time): Unit
```

`initialize` method sets zero time and optionally checkpoint interval (if the dstream must checkpoint and the interval was not set already) and remember duration.

Note	`initialize` method is called for output dstreams only when DStreamGraph is started.

The zero time of a dstream can only be set once or be set again to the same zero time. Otherwise, it throws `SparkException` as follows:

```
ZeroTime is already initialized to [zeroTime], cannot initialize it again to [time]
```

It verifies that checkpoint interval is defined when mustCheckpoint was enabled.

Note	The internal `mustCheckpoint` flag is disabled by default. It is set by custom dstreams like StateDStreams.

If `mustCheckpoint` is enabled and the checkpoint interval was not set, it is automatically set to the slide interval or 10 seconds, whichever is longer. You should see the following INFO message in the logs when the checkpoint interval was set automatically:

```
INFO [DStreamType]: Checkpoint interval automatically set to [checkpointDuration]
```

It then ensures that remember interval is at least twice the checkpoint interval (only if defined) or the slide duration.

At the very end, it initializes the parent dstreams (available as dependencies) that recursively initializes the entire graph of dstreams.

`remember` Method

```
remember(duration: Duration): Unit
```

`remember` sets remember interval for the current dstream and the dstreams it depends on (see dependencies).

If the input `duration` is specified (i.e. not `null`), `remember` allows setting the remember interval (only when the current value was not set already) or extend it (when the current value is shorter).

You should see the following INFO message in the logs when the remember interval changes:

```
INFO Duration for remembering RDDs set to [rememberDuration] for [dstream]
```

At the end, `remember` always sets the current remember interval (whether it was set, extended or did not change).

Checkpointing DStreams — `checkpoint` Method

```
checkpoint(interval: Duration): DStream[T]
```

You use `checkpoint(interval: Duration)` method to set up a periodic checkpointing every (checkpoint) `interval`.

You can only enable checkpointing and set the checkpoint interval before StreamingContext is started or `UnsupportedOperationException` is thrown as follows:

```
java.lang.UnsupportedOperationException: Cannot change checkpoint interval of an DStre
am after streaming context has started
  at org.apache.spark.streaming.dstream.DStream.checkpoint(DStream.scala:177)
  ... 43 elided
```

Internally, `checkpoint` method calls persist (that sets the default `MEMORY_ONLY_SER` storage level).

If checkpoint interval is set, the checkpoint directory is mandatory. Spark validates it when StreamingContext starts and throws a `IllegalArgumentException` exception if not set.

```
java.lang.IllegalArgumentException: requirement failed: The checkpoint directory has n
ot been set. Please set it by StreamingContext.checkpoint().
```

You can see the value of the checkpoint interval for a dstream in the logs when it is validated:

```
INFO Checkpoint interval = [checkpointDuration]
```

Checkpointing

DStreams can checkpoint input data at specified time intervals.

The following settings are internal to a dstream and define how it checkpoints the input data if any.

- `mustCheckpoint` (default: `false`) is an internal private flag that marks a dstream as being checkpointed (`true`) or not (`false`). It is an implementation detail and the author of a `DStream` implementation sets it.

  Refer to Initializing DStreams (initialize method) to learn how it is used to set the checkpoint interval, i.e. `checkpointDuration` .

- `checkpointDuration` is a configurable property that says how often a dstream checkpoints data. It is often called **checkpoint interval**. If not set explicitly, but the dstream is checkpointed, it will be while initializing dstreams.

- `checkpointData` is an instance of DStreamCheckpointData.

- `restoredFromCheckpointData` (default: `false`) is an internal flag to describe the initial state of a dstream, i.e.. whether (`true`) or not (`false`) it was started by restoring state from checkpoint.

Validating Setup at Startup — `validateAtStart` Method

Caution	FIXME Describe me!

Registering Output Streams — `register` Method

```
register(): DStream[T]
```

`DStream` by design has no notion of being an output stream. It is DStreamGraph to know and be able to differentiate between input and output streams.

`DStream` comes with internal `register` method that registers a `DStream` as an output stream.

The internal private `foreachRDD` method uses `register` to register output streams to DStreamGraph. Whenever called, it creates ForEachDStream and calls `register` upon it. That is how streams become output streams.

Generating Streaming Job For Batch For Output DStream — `generateJob` Internal Method

```
generateJob(time: Time): Option[Job]
```

`generateJob` generates a streaming job for a `time` batch for a (output) dstream. It may or may not generate a streaming job for the requested batch `time` if there are RDDs to process.

Note	`generateJob` is called when `DStreamGraph` generates jobs for a batch time.

It computes an RDD for the batch and, if there is one, returns a streaming job for the batch `time` and a job function that will run a Spark job (with the generated RDD and the job function) when executed.

Note	The Spark job uses an empty function to calculate partitions of a RDD.

Caution	FIXME What happens when `SparkContext.runJob(rdd, emptyFunc)` is called with the empty function, i.e. `(iterator: Iterator[T]) ⇒ {}` ?

Computing RDD for Batch — `getOrCompute` Internal Method

```
getOrCompute(time: Time): Option[RDD[T]]
```

`getOrCompute` returns an optional `RDD` for a `time` batch.

Note	`getOrCompute` is `private[streaming] final` method.

`getOrCompute` uses generatedRDDs to return the RDD if it has already been generated for the `time`. If not, it generates one by computing the input stream (using `compute(validTime: Time)` method).

If there was anything to process in the input stream, i.e. computing the input stream returned a RDD, the RDD is first persisted (only if `storageLevel` for the input stream is different from `NONE` storage level).

You should see the following DEBUG message in the logs:

```
DEBUG Persisting RDD [id] for time [time] to [storageLevel]
```

The generated RDD is checkpointed if checkpointDuration is defined and the time interval between current and zero times is a multiple of checkpointDuration.

You should see the following DEBUG message in the logs:

```
DEBUG Marking RDD [id] for time [time] for checkpointing
```

The generated RDD is saved in the internal generatedRDDs registry.

Note	`getOrCompute` is used when a `DStream` is requested to generate a streaming job for a batch.

Caching and Persisting

Caution	FIXME

Checkpoint Cleanup

Caution	FIXME

`restoreCheckpointData`

```
restoreCheckpointData(): Unit
```

`restoreCheckpointData` does its work only when the internal *transient* `restoredFromCheckpointData` flag is disabled (i.e. `false`) and is so initially.

Note	`restoreCheckpointData` method is called when DStreamGraph is requested to restore state of output dstreams.

If `restoredFromCheckpointData` is disabled, you should see the following INFO message in the logs:

```
INFO ...DStream: Restoring checkpoint data
```

DStreamCheckpointData.restore() is executed. And then `restoreCheckpointData` method is executed for every dstream the current dstream depends on (see DStream Contract).

Once completed, the internal `restoredFromCheckpointData` flag is enabled (i.e. `true`) and you should see the following INFO message in the logs:

```
INFO Restored checkpoint data
```

Metadata Cleanup — `clearMetadata` Method

Note	It is called when DStreamGraph clears metadata for every output stream.

`clearMetadata(time: Time)` is called to remove old RDDs that have been generated so far (and collected in generatedRDDs). It is a sort of *garbage collector*.

When `clearMetadata(time: Time)` is called, it checks spark.streaming.unpersist flag (default enabled).

It collects generated RDDs (from generatedRDDs) that are older than rememberDuration.

You should see the following DEBUG message in the logs:

```
DEBUG Clearing references to old RDDs: [[time] -> [rddId], ...]
```

Regardless of spark.streaming.unpersist flag, all the collected RDDs are removed from generatedRDDs.

When spark.streaming.unpersist flag is set (it is by default), you should see the following DEBUG message in the logs:

```
DEBUG Unpersisting old RDDs: [id1, id2, ...]
```

For every RDD in the list, it unpersists them (without blocking) one by one and explicitly removes blocks for BlockRDDs. You should see the following INFO message in the logs:

```
INFO Removing blocks of RDD [blockRDD] of time [time]
```

After RDDs have been removed from generatedRDDs (and perhaps unpersisted), you should see the following DEBUG message in the logs:

```
DEBUG Cleared [size] RDDs that were older than [time]: [time1, time2, ...]
```

The stream passes the call to clear metadata to its dependencies.

`updateCheckpointData` Method

```
updateCheckpointData(currentTime: Time): Unit
```

Note	It is called when DStreamGraph is requested to do updateCheckpointData itself.

When `updateCheckpointData` is called, you should see the following DEBUG message in the logs:

```
DEBUG Updating checkpoint data for time [currentTime] ms
```

It then executes DStreamCheckpointData.update(currentTime) and calls `updateCheckpointData` method on each dstream the dstream depends on.

When `updateCheckpointData` finishes, you should see the following DEBUG message in the logs:

```
DEBUG Updated checkpoint data for time [currentTime]: [checkpointData]
```

Input DStreams

Input DStreams in Spark Streaming are the way to ingest data from external data sources. They are represented as `InputDStream` abstract class.

InputDStream Contract

`InputDStream` is the abstract base class for all input DStreams. It provides two abstract methods `start()` and `stop()` to start and stop ingesting data, respectively.

When instantiated, an `InputDStream` registers itself as an input stream (using DStreamGraph.addInputStream) and, while doing so, is told about its owning DStreamGraph.

It asks for its own unique identifier using `StreamingContext.getNewInputStreamId()`.

Note	It is StreamingContext to maintain the identifiers and how many input streams have already been created.

`InputDStream` has a human-readable `name` that is made up from a nicely-formatted part based on the class name and the unique identifier.

Tip	Name your custom `InputDStream` using the CamelCase notation with the suffix **InputDStream**, e.g. MyCustomInputDStream.

- `slideDuration` calls DStreamGraph.batchDuration.
- `dependencies` method returns an empty collection.

Note	`compute(validTime: Time): Option[RDD[T]]` abstract method from DStream abstract class is not defined.

Custom implementations of `InputDStream` can override (and actually provide!) the optional RateController. It is undefined by default.

Custom Input DStream

Here is an example of a custom input dstream that produces an RDD out of the input collection of elements (of type `T`).

Note	It is similar to ConstantInputDStreams, but this custom implementation does not use an external RDD, but generates its own.

```
package pl.japila.spark.streaming

import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.{ Time, StreamingContext }
import org.apache.spark.streaming.dstream.InputDStream

import scala.reflect.ClassTag

class CustomInputDStream[T: ClassTag](ssc: StreamingContext, seq: Seq[T])
  extends InputDStream[T](ssc) {
  override def compute(validTime: Time): Option[RDD[T]] = {
    Some(ssc.sparkContext.parallelize(seq))
  }
  override def start(): Unit = {}
  override def stop(): Unit = {}
}
```

Its use could be as simple as follows (compare it to the example of ConstantInputDStreams):

```
// sc is the SparkContext instance
import org.apache.spark.streaming.Seconds
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

// Create the collection of numbers
val nums = 0 to 9

// Create constant input dstream with the RDD
import pl.japila.spark.streaming.CustomInputDStream
val cis = new CustomInputDStream(ssc, nums)

// Sample stream computation
cis.print
```

Tip	Copy and paste it to `spark-shell` to run it.

ReceiverInputDStreams - Input Streams with Receivers

Receiver Input Streams (`ReceiverInputDStreams`) are specialized input streams that use receivers to receive data (and hence the name which stands for an `InputDStream` with a receiver).

Note	Receiver input streams run receivers as long-running tasks that occupy a core per stream.

`ReceiverInputDStream` abstract class defines the following abstract method that custom implementations use to create receivers:

```
def getReceiver(): Receiver[T]
```

The receiver is then sent to and run on workers (when ReceiverTracker is started).

Note	A fine example of a very minimalistic yet still useful implementation of `ReceiverInputDStream` class is the pluggable input stream `org.apache.spark.streaming.dstream.PluggableInputDStream` (the sources on GitHub). It requires a `Receiver` to be given (by a developer) and simply returns it in `getReceiver` . `PluggableInputDStream` is used by StreamingContext.receiverStream() method.

`ReceiverInputDStream` uses `ReceiverRateController` when spark.streaming.backpressure.enabled is enabled.

Note	Both, `start()` and `stop` methods are implemented in `ReceiverInputDStream` , but do nothing. `ReceiverInputDStream` management is left to ReceiverTracker. Read ReceiverTrackerEndpoint.startReceiver for more details.

The source code of `ReceiverInputDStream` is here at GitHub.

Generate RDDs for Batch Interval — `compute` Method

The abstract `compute(validTime: Time): Option[RDD[T]]` method (from DStream) uses start time of DStreamGraph, i.e. the start time of StreamingContext, to check whether `validTime` input parameter is really valid.

If the time to generate RDDs (`validTime`) is earlier than the start time of StreamingContext, an empty `BlockRDD` is generated.

Otherwise, ReceiverTracker is requested for all the blocks that have been allocated to this stream for this batch (using `ReceiverTracker.getBlocksOfBatch`).

The number of records received for the batch for the input stream (as StreamInputInfo) is registered with InputInfoTracker.

If all BlockIds have `WriteAheadLogRecordHandle` , a `WriteAheadLogBackedBlockRDD` is generated. Otherwise, a `BlockRDD` is.

Back Pressure

Caution	FIXME

Back pressure for input dstreams with receivers can be configured using spark.streaming.backpressure.enabled setting.

Note	Back pressure is disabled by default.

ConstantInputDStreams

`ConstantInputDStream` is an input stream that always returns the same mandatory input RDD at every batch `time` .

```
ConstantInputDStream[T](_ssc: StreamingContext, rdd: RDD[T])
```

`ConstantInputDStream` dstream belongs to `org.apache.spark.streaming.dstream` package.

The `compute` method returns the input `rdd` .

Note	`rdd` input parameter is mandatory.

The mandatory `start` and `stop` methods do nothing.

Example

```
val sc = new SparkContext("local[*]", "Constant Input DStream Demo", new SparkConf())
import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

// Create the RDD
val rdd = sc.parallelize(0 to 9)

// Create constant input dstream with the RDD
import org.apache.spark.streaming.dstream.ConstantInputDStream
val cis = new ConstantInputDStream(ssc, rdd)

// Sample stream computation
cis.print
```

ForEachDStreams

`ForEachDStream` is an internal DStream with dependency on the `parent` stream with the exact same `slideDuration`.

The `compute` method returns no RDD.

When `generateJob` is called, it returns a streaming job for a batch when `parent` stream does. And if so, it uses the "foreach" function (given as `foreachFunc`) to work on the RDDs generated.

Note	Although it may seem that `ForEachDStreams` are by design output streams they are not. You have to use DStreamGraph.addOutputStream to register a stream as output. You use stream operators that do the registration as part of their operation, like `print`.

WindowedDStreams

`WindowedDStream` (aka **windowed stream**) is an internal DStream with dependency on the `parent` stream.

Note	It is the result of window operators.

`windowDuration` has to be a multiple of the parent stream's slide duration.

`slideDuration` has to be a multiple of the parent stream's slide duration.

Note	When `windowDuration` or `slideDuration` are *not* multiples of the parent stream's slide duration, `Exception` is thrown.

The parent's RDDs are automatically changed to be persisted at `MEMORY_ONLY_SER` storage level (since they need to last longer than the parent's slide duration for this stream to generate its own RDDs).

Obviously, slide duration of the stream is given explicitly (and must be a multiple of the parent's slide duration).

`parentRememberDuration` is extended to cover the parent's `rememberDuration` and the window duration.

`compute` method always returns a RDD, either `PartitionerAwareUnionRDD` or `UnionRDD` , depending on the number of the Partitioner defined by the RDDs in the window. It uses slice operator on the parent stream (using the slice window of `[now - windowDuration + parent.slideDuration, now]`).

If only one partitioner is used across the RDDs in window, `PartitionerAwareUnionRDD` is created and you should see the following DEBUG message in the logs:

```
DEBUG WindowedDStream: Using partition aware union for windowing at [time]
```

Otherwise, when there are multiple different partitioners in use, `UnionRDD` is created and you should see the following DEBUG message in the logs:

```
DEBUG WindowedDStream: Using normal union for windowing at [time]
```

Tip	Enable `DEBUG` logging level for `org.apache.spark.streaming.dstream.WindowedDStream` logger to see what happens inside `WindowedDStream` . Add the following line to `conf/log4j.properties` : `log4j.logger.org.apache.spark.streaming.dstream.WindowedDStream=DEBUG`

MapWithStateDStream

`MapWithStateDStream` is the result of mapWithState stateful operator.

It extends DStream Contract with the following additional method:

```
def stateSnapshots(): DStream[(KeyType, StateType)]
```

Note	`MapWithStateDStream` is a Scala `sealed abstract class` (and hence all the available implementations are in the source file).

Note	MapWithStateDStreamImpl is the only implementation of `MapWithStateDStream` (see below in this document for more coverage).

MapWithStateDStreamImpl

`MapWithStateDStreamImpl` is an internal DStream with dependency on the parent `dataStream` key-value dstream. It uses a custom internal dstream called `internalStream` (of type InternalMapWithStateDStream).

`slideDuration` is exactly the slide duration of the internal stream `internalStream` .

`dependencies` returns a single-element collection with the internal stream `internalStream` .

The `compute` method may or may not return a `RDD[MappedType]` by `getOrCompute` on the internal stream and…TK

Caution	FIXME

InternalMapWithStateDStream

`InternalMapWithStateDStream` is an internal dstream to support MapWithStateDStreamImpl and uses `dataStream` (as `parent` of type `DStream[(K, V)]`) as well as `StateSpecImpl[K, V, S, E]` (as `spec`).

`InternalMapWithStateDStream` is a `DStream[MapWithStateRDDRecord[K, S, E]]` that uses `MEMORY_ONLY` storage level by default.

`InternalMapWithStateDStream` uses the `StateSpec` 's partitioner or HashPartitioner (with SparkContext's defaultParallelism).

`slideDuration` is the slide duration of `parent` .

`dependencies` is a single-element collection with the `parent` stream.

It forces checkpointing (i.e. `mustCheckpoint` flag is enabled).

When initialized, if checkpoint interval is *not* set, it sets it as ten times longer than the slide duration of the `parent` stream (the multiplier is not configurable and always `10`).

Computing a `RDD[MapWithStateRDDRecord[K, S, E]]` (i.e. `compute` method) first looks up a previous RDD for the last `slideDuration` .

If the RDD is found, it is returned as is given the partitioners of the RDD and the stream are equal. Otherwise, when the partitioners are different, the RDD is "repartitioned" using `MapWithStateRDD.createFromRDD` .

Caution	FIXME `MapWithStateRDD.createFromRDD`

StateDStream

`StateDStream` is the specialized DStream that is the result of updateStateByKey stateful operator. It is a wrapper around a `parent` key-value pair dstream to build stateful pipeline (by means of `updateStateByKey` operator) and as a stateful dstream enables checkpointing (and hence requires some additional setup).

It uses a `parent` key-value pair dstream, updateFunc update state function, a `partitioner` , a flag whether or not to `preservePartitioning` and an optional key-value pair `initialRDD` .

It works with `MEMORY_ONLY_SER` storage level enabled.

The only dependency of `StateDStream` is the input `parent` key-value pair dstream.

The slide duration is exactly the same as that in `parent` .

It forces checkpointing regardless of the current dstream configuration, i.e. the internal mustCheckpoint is enabled.

When requested to compute a RDD it first attempts to get the **state RDD** for the previous batch (using DStream.getOrCompute). If there is one, `parent` stream is requested for a RDD for the current batch (using DStream.getOrCompute). If `parent` has computed one, computeUsingPreviousRDD(parentRDD, prevStateRDD) is called.

Caution	FIXME When could `getOrCompute` **not** return an RDD? How does this apply to the StateDStream? What about the parent's `getOrCompute` ?

If however `parent` has not generated a RDD for the current batch but the state RDD existed, `updateFn` is called for every key of the state RDD to generate a new state per partition (using RDD.mapPartitions)

Note	No input data for already-running input stream triggers (re)computation of the state RDD (per partition).

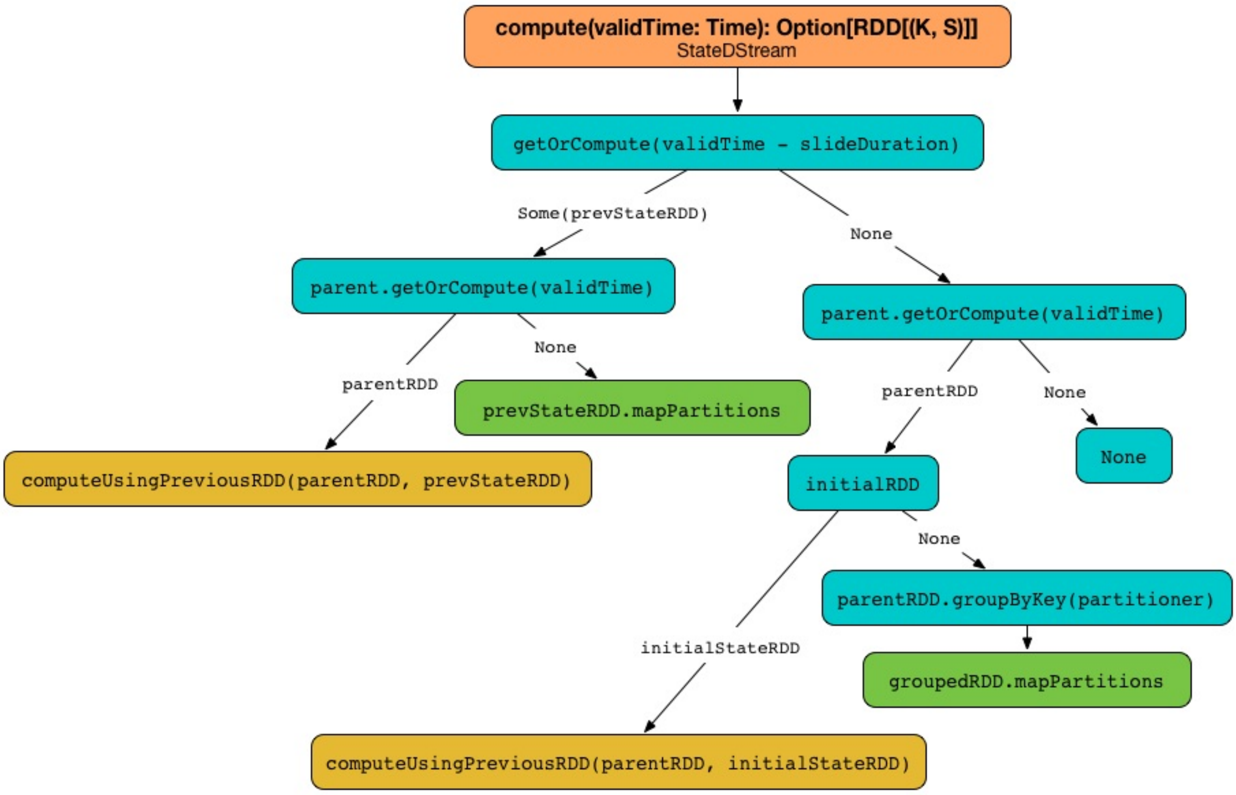

Figure 1. Computing stateful RDDs (StateDStream.compute)

If the state RDD has been found, which means that this is the first input data batch, `parent` stream is requested to getOrCompute the RDD for the current batch.

Otherwise, when no state RDD exists, `parent` stream is requested for a RDD for the current batch (using DStream.getOrCompute) and when no RDD was generated for the batch, no computation is triggered.

Note	When the stream processing starts, i.e. no state RDD exists, and there is no input data received, no computation is triggered.

Given no state RDD and with `parent` RDD computed, when `initialRDD` is `NONE`, the input data batch (as `parent` RDD) is grouped by key (using groupByKey with `partitioner`) and then the update state function `updateFunc` is applied to the partitioned input data (using RDD.mapPartitions) with `None` state. Otherwise, computeUsingPreviousRDD(parentRDD, initialStateRDD) is called.

updateFunc - State Update Function

The signature of `updateFunc` is as follows:

```
updateFunc: (Iterator[(K, Seq[V], Option[S])]) => Iterator[(K, S)]
```

It should be read as given a collection of triples of a key, new records for the key, and the current state for the key, generate a collection of keys and their state.

computeUsingPreviousRDD

```
computeUsingPreviousRDD(parentRDD: RDD[(K, V)], prevStateRDD: RDD[(K, S)]): Option[RDD
[(K, S)]]
```

The `computeUsingPreviousRDD` method uses `cogroup` and `mapPartitions` to build the final state RDD.

Note	Regardless of the return type `Option[RDD[(K, S)]]` that really allows no state, it will always return *some* state.

It first performs `cogroup` of `parentRDD` and `prevStateRDD` using the constructor's `partitioner` so it has a pair of iterators of elements of each RDDs per *every* key.

Note	It is acceptable to end up with keys that have no new records per batch, but these keys do have a state (since they were received previously when no state might have been built yet).

Note	The signature of `cogroup` is as follows and applies to key-value pair RDDs, i.e. `RI` `cogroup[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Iterable[V],`

It defines an internal update function `finalFunc` that maps over the collection of all the keys, new records per key, and at-most-one-element state per key to build new iterator that ensures that:

1. a state per key exists (it is `None` or the state built so far)
2. the *lazy* iterable of new records is transformed into an *eager* sequence.

Caution	FIXME Why is the transformation from an Iterable into a Seq so important? Why could not the constructor's updateFunc accept the former?

With every triple per every key, the internal update function calls the constructor's updateFunc.

The state RDD is a cogrouped RDD (on `parentRDD` and `prevStateRDD` using the constructor's `partitioner`) with every element per partition mapped over using the internal update function `finalFunc` and the constructor's `preservePartitioning` (through `mapPartitions`).

Caution	FIXME Why is `preservePartitioning` important? What happens when `mapPartitions` does not preserve partitioning (which by default it does **not**!)

TransformedDStream

`TransformedDStream` is the specialized DStream that is the result of transform operator.

It is constructed with a collection of `parents` dstreams and `transformFunc` transform function.

Note	When created, it asserts that the input collection of dstreams use the same StreamingContext and slide interval.

Note	It is acceptable to have more than one dependent dstream.

The dependencies is the input collection of dstreams.

The slide interval is exactly the same as that in the first dstream in `parents`.

When requested to compute a RDD, it goes over every dstream in `parents` and asks to getOrCompute a RDD.

Note	It may throw a `SparkException` when a dstream does not compute a RDD for a batch.

Caution	FIXME Prepare an example to face the exception.

It then calls `transformFunc` with the collection of RDDs.

If the transform function returns `null` a SparkException is thrown:

```
org.apache.spark.SparkException: Transform function must not
return null. Return SparkContext.emptyRDD() instead to represent
no element as the result of transformation.
        at
org.apache.spark.streaming.dstream.TransformedDStream.compute(Tr
ansformedDStream.scala:48)
```

The result of `transformFunc` is returned.

Receivers

Receivers run on workers to receive external data. They are created and belong to ReceiverInputDStreams.

Note	ReceiverTracker launches a receiver on a worker.

It is represented by abstract class Receiver that is parameterized by the type of the elements it processes as well as StorageLevel.

Note	You use StreamingContext.receiverStream method to register a custom `Receiver` to a streaming context.

The abstract `Receiver` class requires the following methods to be implemented (see Custom Receiver):

- `onStart()` that starts the receiver when the application starts.
- `onStop()` that stops the receiver.

A receiver is identified by the unique identifier `Receiver.streamId` (that corresponds to the unique identifier of the receiver input stream it is associated with).

Note	StorageLevel of a receiver is used to instantiate ReceivedBlockHandler in ReceiverSupervisorImpl.

A receiver uses `store` methods to store received data as data blocks into Spark's memory.

Note	Receivers must have ReceiverSupervisors attached before they can be started since `store` and management methods simply pass calls on to the respective methods in the ReceiverSupervisor.

A receiver can be in one of the three states: `Initialized`, `Started`, and `Stopped`.

Custom Receiver

```
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.receiver.Receiver

final class MyStringReceiver extends Receiver[String](StorageLevel.NONE) {

  def onStart() = {
    println("onStart called")
  }

  def onStop() = {
    println("onStop called")
  }
}

val ssc = new StreamingContext(sc, Seconds(5))
val strings = ssc.receiverStream(new MyStringReceiver)
strings.print

ssc.start

// MyStringReceiver will print "onStart called"

ssc.stop()

// MyStringReceiver will print "onStop called"
```

ReceiverTracker

Introduction

`ReceiverTracker` manages execution of all Receivers.

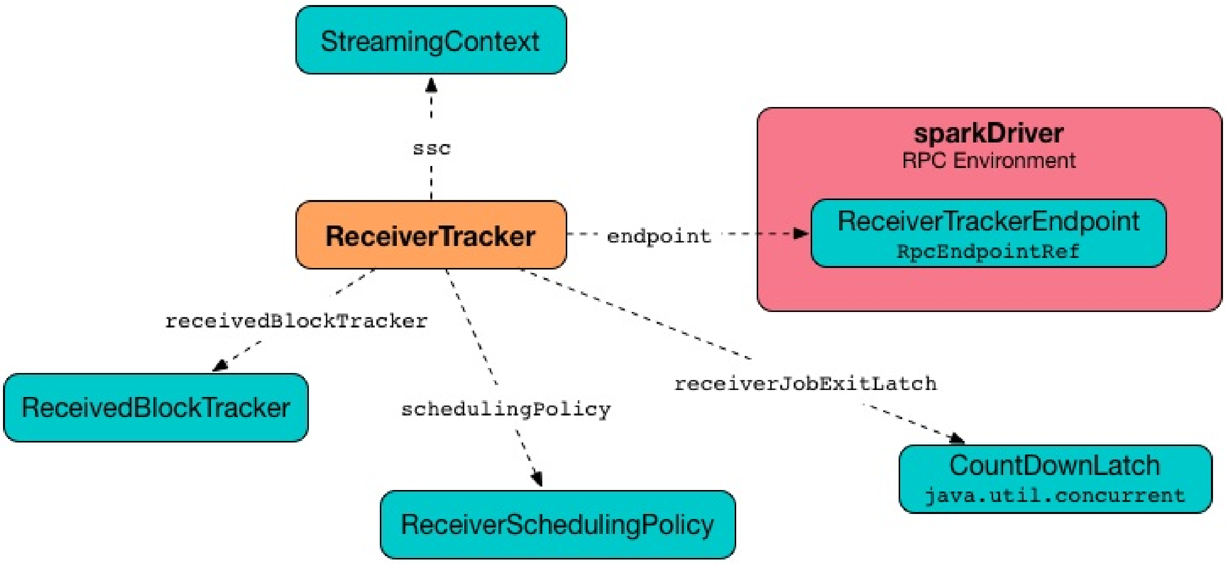

Figure 1. ReceiverTracker and Dependencies

It uses RPC environment for communication with ReceiverSupervisors.

Note	`ReceiverTracker` is started when JobScheduler starts.

It can only be started once and only when at least one input receiver has been registered.

`ReceiverTracker` can be in one of the following states:

- `Initialized` - it is in the state after having been instantiated.
- `Started` -
- `Stopping`
- `Stopped`

Starting ReceiverTracker (start method)

Note	You can only start `ReceiverTracker` once and multiple attempts lead to throwing `SparkException` exception.

Note	Starting `ReceiverTracker` when no ReceiverInputDStream has registered does nothing.

When `ReceiverTracker` starts, it first sets ReceiverTracker RPC endpoint up.

It then launches receivers, i.e. it collects receivers for all registered `ReceiverDStream` and posts them as StartAllReceivers to ReceiverTracker RPC endpoint.

In the meantime, receivers have their ids assigned that correspond to the unique identifier of their `ReceiverDStream`.

You should see the following INFO message in the logs:

```
INFO ReceiverTracker: Starting [receivers.length] receivers
```

A successful startup of `ReceiverTracker` finishes with the following INFO message in the logs:

```
INFO ReceiverTracker: ReceiverTracker started
```

`ReceiverTracker` enters `Started` state.

Cleanup Old Blocks And Batches (cleanupOldBlocksAndBatches method)

Caution	FIXME

hasUnallocatedBlocks

Caution	FIXME

ReceiverTracker RPC endpoint

Caution	FIXME

StartAllReceivers

`StartAllReceivers(receivers)` is a local message sent by ReceiverTracker when it starts (using `ReceiverTracker.launchReceivers()`).

It schedules receivers (using `ReceiverSchedulingPolicy.scheduleReceivers(receivers, getExecutors)`).

Caution	FIXME What does `ReceiverSchedulingPolicy.scheduleReceivers(receivers, getExecutors)` do?

It does *some* bookkeeping.

Caution	FIXME What is *the* bookkeeping?

It finally starts every receiver (using the helper method ReceiverTrackerEndpoint.startReceiver).

ReceiverTrackerEndpoint.startReceiver

Caution	FIXME When is the method called?

`ReceiverTrackerEndpoint.startReceiver(receiver: Receiver[_], scheduledLocations: Seq[TaskLocation])` starts a `receiver` Receiver at the given `Seq[TaskLocation]` locations.

Caution	FIXME When the scaladoc says *"along with the scheduled executors"*, does it mean that the executors are already started and waiting for the receiver?!

It defines an internal function (`startReceiverFunc`) to start `receiver` on a worker (in Spark cluster).

Namely, the internal `startReceiverFunc` function checks that the task attempt is `0` .

Tip	Read about `TaskContext` in TaskContext.

It then starts a ReceiverSupervisor for `receiver` and keeps awaiting termination, i.e. once the task is run it does so until *a termination message* comes from *some* other external source). The task is a long-running task for `receiver` .

Caution	FIXME When does `supervisor.awaitTermination()` finish?

Having the internal function, it creates `receiverRDD` - an instance of `RDD[Receiver[_]]` - that uses SparkContext.makeRDD with a one-element collection with the only element being `receiver` . When the collection of TaskLocation is empty, it uses exactly one partition. Otherwise, it distributes the one-element collection across the nodes (and potentially even executors) for `receiver` . The RDD has the name `Receiver [receiverId]` .

The Spark job's description is set to `Streaming job running receiver [receiverId]` .

Caution	FIXME What does `sparkContext.setJobDescription` actually do and how does this influence Spark jobs? It uses `ThreadLocal` so it assumes that a single thread will do a job?

Having done so, it submits a job (using SparkContext.submitJob) on the instance of `RDD[Receiver[_]]` with the function `startReceiverFunc` that runs `receiver` . It has SimpleFutureAction to monitor `receiver` .

Note	The method demonstrates how you could use Spark Core as the distributed computation platform to launch *any* process on clusters and let Spark handle the distribution. *Very clever indeed!*

When it completes (successfully or not), `onReceiverJobFinish(receiverId)` is called, but only for cases when the tracker is fully up and running, i.e. started. When the tracker is being stopped or has already stopped, the following INFO message appears in the logs:

```
INFO Restarting Receiver [receiverId]
```

And a `RestartReceiver(receiver)` message is sent.

When there was a failure submitting the job, you should also see the ERROR message in the logs:

```
ERROR Receiver has been stopped. Try to restart it.
```

Ultimately, right before the method exits, the following INFO message appears in the logs:

```
INFO Receiver [receiver.streamId] started
```

StopAllReceivers

Caution	FIXME

AllReceiverIds

Caution	FIXME

Stopping ReceiverTracker (stop method)

`ReceiverTracker.stop(graceful: Boolean)` stops `ReceiverTracker` only when it is in `Started` state. Otherwise, it does nothing and simply exits.

Note	The `stop` method is called while JobScheduler is being stopped.

The state of `ReceiverTracker` is marked `Stopping`.

It then sends the stop signal to all the receivers (i.e. posts StopAllReceivers to ReceiverTracker RPC endpoint) and waits **10 seconds** for all the receivers to quit gracefully (unless `graceful` flag is set).

Note	The 10-second wait time for graceful quit is not configurable.

You should see the following INFO messages if the `graceful` flag is enabled which means that the receivers quit in a graceful manner:

```
INFO ReceiverTracker: Waiting for receiver job to terminate gracefully
INFO ReceiverTracker: Waited for receiver job to terminate gracefully
```

It then checks whether all the receivers have been deregistered or not by posting AllReceiverIds to ReceiverTracker RPC endpoint.

You should see the following INFO message in the logs if they have:

```
INFO ReceiverTracker: All of the receivers have deregistered successfully
```

Otherwise, when there were receivers not having been deregistered properly, the following WARN message appears in the logs:

```
WARN ReceiverTracker: Not all of the receivers have deregistered, [receivers]
```

It stops ReceiverTracker RPC endpoint as well as ReceivedBlockTracker.

You should see the following INFO message in the logs:

```
INFO ReceiverTracker: ReceiverTracker stopped
```

The state of `ReceiverTracker` is marked `Stopped`.

Allocating Blocks To Batch (allocateBlocksToBatch method)

```
allocateBlocksToBatch(batchTime: Time): Unit
```

`allocateBlocksToBatch` simply passes all the calls on to ReceivedBlockTracker.allocateBlocksToBatch, but only when there *are* receiver input streams registered (in `receiverInputStreams` internal registry).

Note	When there are no receiver input streams in use, the method does nothing.

ReceivedBlockTracker

Caution	FIXME

You should see the following INFO message in the logs when `cleanupOldBatches` is called:

```
INFO ReceivedBlockTracker: Deleting batches [timesToCleanup]
```

allocateBlocksToBatch Method

```
allocateBlocksToBatch(batchTime: Time): Unit
```

`allocateBlocksToBatch` starts by checking whether the internal `lastAllocatedBatchTime` is younger than (after) the current batch time `batchTime`.

If so, it grabs all unallocated blocks per stream (using `getReceivedBlockQueue` method) and creates a map of stream ids and sequences of their `ReceivedBlockInfo`. It then writes the received blocks to **write-ahead log (WAL)** (using `writeToLog` method).

`allocateBlocksToBatch` stores the allocated blocks with the current batch time in `timeToAllocatedBlocks` internal registry. It also sets `lastAllocatedBatchTime` to the current batch time `batchTime`.

If there has been an error while writing to WAL or the batch time is older than `lastAllocatedBatchTime`, you should see the following INFO message in the logs:

```
INFO Possibly processed batch [batchTime] needs to be processed again in WAL recovery
```

ReceiverSupervisors

`ReceiverSupervisor` is an (abstract) handler object that is responsible for supervising a receiver (that runs on the worker). It assumes that implementations offer concrete methods to push received data to Spark.

Note	Receiver's `store` methods pass calls to respective `push` methods of ReceiverSupervisors.

Note	ReceiverTracker starts a ReceiverSupervisor per receiver.

`ReceiverSupervisor` can be started and stopped. When a supervisor is started, it calls (empty by default) `onStart()` and `startReceiver()` afterwards.

It attaches itself to the receiver it is a supervisor of (using `Receiver.attachSupervisor`). That is how a receiver knows about its supervisor (and can hence offer the `store` and management methods).

ReceiverSupervisor Contract

`ReceiverSupervisor` is a `private[streaming] abstract class` that assumes that concrete implementations offer the following **push methods**:

- `pushBytes`
- `pushIterator`
- `pushArrayBuffer`

There are the other methods required:

- `createBlockGenerator`
- `reportError`
- `onReceiverStart`

Starting Receivers

`startReceiver()` calls (abstract) `onReceiverStart()`. When `true` (it is unknown at this point to know when it is `true` or `false` since it is an abstract method - see ReceiverSupervisorImpl.onReceiverStart for the default implementation), it prints the following INFO message to the logs:

```
INFO Starting receiver
```

The receiver's `onStart()` is called and another INFO message appears in the logs:

```
INFO Called receiver onStart
```

If however `onReceiverStart()` returns `false`, the supervisor stops (using `stop`).

Stopping Receivers

`stop` method is called with a message and an optional cause of the stop (called `error`). It calls `stopReceiver` method that prints the INFO message and checks the state of the receiver to react appropriately.

When the receiver is in `Started` state, `stopReceiver` calls `Receiver.onStop()`, prints the following INFO message, and `onReceiverStop(message, error)`.

```
INFO Called receiver onStop
```

Restarting Receivers

A `ReceiverSupervisor` uses spark.streaming.receiverRestartDelay to restart the receiver with delay.

Note	Receivers can request to be restarted using `restart` methods.

When requested to restart a receiver, it uses a separate thread to perform it asynchronously. It prints the WARNING message to the logs:

```
WARNING Restarting receiver with delay [delay] ms: [message]
```

It then stops the receiver, sleeps for `delay` milliseconds and starts the receiver (using `startReceiver()`).

You should see the following messages in the logs:

```
DEBUG Sleeping for [delay]
INFO Starting receiver again
INFO Receiver started again
```

Caution	FIXME What is a backend data store?

Awaiting Termination

`awaitTermination` method blocks the current thread to wait for the receiver to be stopped.

Note	ReceiverTracker uses `awaitTermination` to wait for receivers to stop (see StartAllReceivers).

When called, you should see the following INFO message in the logs:

```
INFO Waiting for receiver to be stopped
```

If a receiver has terminated successfully, you should see the following INFO message in the logs:

```
INFO Stopped receiver without error
```

Otherwise, you should see the ERROR message in the logs:

```
ERROR Stopped receiver with error: [stoppingError]
```

`stoppingError` is the exception associated with the stopping of the receiver and is rethrown.

Note	Internally, ReceiverSupervisor uses java.util.concurrent.CountDownLatch with count `1` to await the termination.

Internals - How to count stopLatch down

`stopLatch` is decremented when ReceiverSupervisor's `stop` is called which is in the following cases:

- When a receiver itself calls `stop(message: String)` or `stop(message: String, error: Throwable)`
- When ReceiverSupervisor.onReceiverStart() returns `false` or `NonFatal` (less severe) exception is thrown in `ReceiverSupervisor.startReceiver`.
- When ReceiverTracker.stop is called that posts `StopAllReceivers` message to `ReceiverTrackerEndpoint`. It in turn sends `StopReceiver` to the `ReceiverSupervisorImpl` for every `ReceiverSupervisor` that calls `ReceiverSupervisorImpl.stop`.

Caution	FIXME Prepare exercises • for a receiver to call `stop(message: String)` when a custom "TERMINATE" message arrives • send `StopReceiver` to a ReceiverTracker

ReceiverSupervisorImpl

`ReceiverSupervisorImpl` is the implementation of ReceiverSupervisor contract.

Note	A dedicated `ReceiverSupervisorImpl` is started for every receiver when ReceiverTracker starts. See ReceiverTrackerEndpoint.startReceiver.

It communicates with ReceiverTracker that runs on the driver (by posting messages using the ReceiverTracker RPC endpoint).

Tip	Enable `DEBUG` logging level for `org.apache.spark.streaming.receiver.ReceiverSupervisorImpl` logger to see what happens in `ReceiverSupervisorImpl`. Add the following line to `conf/log4j.properties` : `log4j.logger.org.apache.spark.streaming.receiver.ReceiverSupervisorImpl=DEBUG`

push Methods

push methods, i.e. `pushArrayBuffer` , `pushIterator` , and `pushBytes` solely pass calls on to ReceiverSupervisorImpl.pushAndReportBlock.

ReceiverSupervisorImpl.onReceiverStart

`ReceiverSupervisorImpl.onReceiverStart` sends a blocking `RegisterReceiver` message to ReceiverTracker that responds with a boolean value.

Current Rate Limit

`getCurrentRateLimit` controls the current rate limit. It asks the `BlockGenerator` for the value (using `getCurrentLimit`).

ReceivedBlockHandler

`ReceiverSupervisorImpl` uses the internal field `receivedBlockHandler` for ReceivedBlockHandler to use.

It defaults to BlockManagerBasedBlockHandler, but could use WriteAheadLogBasedBlockHandler instead when spark.streaming.receiver.writeAheadLog.enable is `true`.

It uses `ReceivedBlockHandler` to `storeBlock` (see ReceivedBlockHandler Contract for more coverage and ReceiverSupervisorImpl.pushAndReportBlock in this document).

ReceiverSupervisorImpl.pushAndReportBlock

`ReceiverSupervisorImpl.pushAndReportBlock(receivedBlock: ReceivedBlock, metadataOption: Option[Any], blockIdOption: Option[StreamBlockId])` stores `receivedBlock` using `ReceivedBlockHandler.storeBlock` and reports it to the driver.

Note	`ReceiverSupervisorImpl.pushAndReportBlock` is only used by the push methods, i.e. `pushArrayBuffer`, `pushIterator`, and `pushBytes`. Calling the method is actually all they do.

When it calls `ReceivedBlockHandler.storeBlock`, you should see the following DEBUG message in the logs:

```
DEBUG Pushed block [blockId] in [time] ms
```

It then sends `AddBlock` (with `ReceivedBlockInfo` for `streamId`, `BlockStoreResult.numRecords`, `metadataOption`, and the result of `ReceivedBlockHandler.storeBlock`) to ReceiverTracker RPC endpoint (that runs on the driver).

When a response comes, you should see the following DEBUG message in the logs:

```
DEBUG Reported block [blockId]
```

ReceivedBlockHandlers

`ReceivedBlockHandler` represents how to handle the storage of blocks received by receivers.

Note	It is used by ReceiverSupervisorImpl (as the internal receivedBlockHandler).

ReceivedBlockHandler Contract

`ReceivedBlockHandler` is a `private[streaming] trait`. It comes with two methods:

- `storeBlock(blockId: StreamBlockId, receivedBlock: ReceivedBlock): ReceivedBlockStoreResult` to store a received block as `blockId`.
- `cleanupOldBlocks(threshTime: Long)` to clean up blocks older than `threshTime`.

Note	`cleanupOldBlocks` implies that there is a relation between blocks and the time they arrived.

Implementations of ReceivedBlockHandler Contract

There are two implementations of `ReceivedBlockHandler` contract:

- `BlockManagerBasedBlockHandler` that stores received blocks in Spark's BlockManager with the specified StorageLevel.

  Read BlockManagerBasedBlockHandler in this document.

- `WriteAheadLogBasedBlockHandler` that stores received blocks in a write ahead log and Spark's BlockManager. It is a more advanced option comparing to a simpler BlockManagerBasedBlockHandler.

  Read WriteAheadLogBasedBlockHandler in this document.

BlockManagerBasedBlockHandler

`BlockManagerBasedBlockHandler` is the default `ReceivedBlockHandler` in Spark Streaming.

It uses BlockManager and a receiver's StorageLevel.

`cleanupOldBlocks` is not used as blocks are cleared by *some other means* (FIXME)

`putResult` returns `BlockManagerBasedStoreResult`. It uses `BlockManager.putIterator` to store `ReceivedBlock`.

WriteAheadLogBasedBlockHandler

`WriteAheadLogBasedBlockHandler` is used when spark.streaming.receiver.writeAheadLog.enable is `true`.

It uses BlockManager, a receiver's `streamId` and StorageLevel, SparkConf for additional configuration settings, Hadoop Configuration, the checkpoint directory.

Ingesting Data from Apache Kafka

Spark Streaming comes with two built-in models of ingesting data from Apache Kafka:

- With no receivers
- Using receivers

There is yet another "middle-ground" approach (so-called unofficial since it is not available by default in Spark Streaming):

- Kafka Spark Consumer — a high-performance Kafka Consumer for Spark Streaming with support for Apache Kafka 0.10.

Data Ingestion with no Receivers

No-receivers approach supports the two following modes:

- Streaming mode (using KafkaUtils.createDirectStream) that uses a input dstream that polls for records from Kafka brokers on the driver every batch interval and passes the available topic offsets on to executors for processing.
- **Non-streaming mode** (using KafkaUtils.createRDD) which simply creates a KafkaRDD of key-value pairs, i.e. `RDD[(K, V)]` from the records in topics in Kafka.

Streaming mode

You create DirectKafkaInputDStream using KafkaUtils.createDirectStream.

Note	Define the types of keys and values in `KafkaUtils.createDirectStream` , e.g. `KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder]` , so proper decoders are used to decode messages from Kafka.

You have to specify `metadata.broker.list` or `bootstrap.servers` (in that order of precedence) for your Kafka environment. `metadata.broker.list` is a comma-separated list of Kafka's (seed) brokers in the format of `<host>:<port>` .

Note	You can start `DirectKafkaInputDStream` regardless of the status of Kafka brokers as it waits until at least one Kafka broker is available.

```
val conf = new SparkConf().setMaster("local[*]").setAppName("Ingesting Data from Kafka"
)
conf.set("spark.streaming.ui.retainedBatches", "5")

// Enable Back Pressure
conf.set("spark.streaming.backpressure.enabled", "true")

val ssc = new StreamingContext(conf, batchDuration = Seconds(5))

// Enable checkpointing
ssc.checkpoint("_checkpoint")

// You may or may not want to enable some additional DEBUG logging
import org.apache.log4j._
Logger.getLogger("org.apache.spark.streaming.dstream.DStream").setLevel(Level.DEBUG)
Logger.getLogger("org.apache.spark.streaming.dstream.WindowedDStream").setLevel(Level.
DEBUG)
Logger.getLogger("org.apache.spark.streaming.DStreamGraph").setLevel(Level.DEBUG)
Logger.getLogger("org.apache.spark.streaming.scheduler.JobGenerator").setLevel(Level.D
EBUG)

// Connect to Kafka
import org.apache.spark.streaming.kafka.KafkaUtils
import _root_.kafka.serializer.StringDecoder
val kafkaParams = Map("metadata.broker.list" -> "localhost:9092")
val kafkaTopics = Set("spark-topic")
val messages = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDeco
der](ssc, kafkaParams, kafkaTopics)

// print 10 last messages
messages.print()

// start streaming computation
ssc.start
```

If `zookeeper.connect` or `group.id` parameters are not set, they are added with their values being empty strings.

In this mode, you will only see jobs submitted (in the **Jobs** tab in web UI) when a message comes in.

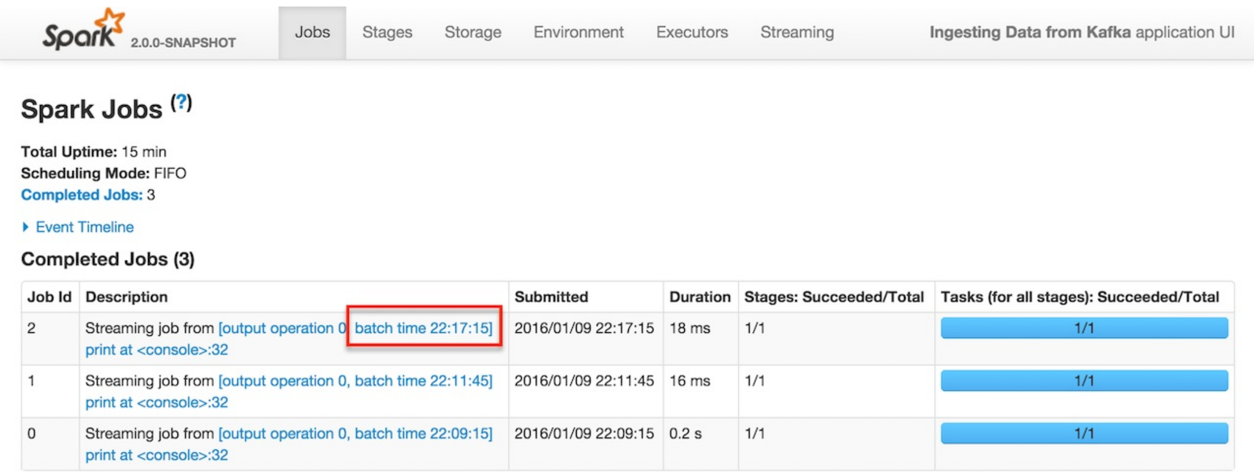

Figure 1. Complete Jobs in web UI for batch time 22:17:15

It corresponds to **Input size** larger than `0` in the **Streaming** tab in the web UI.

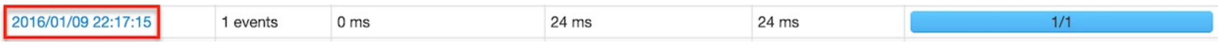

Figure 2. Completed Batch in web UI for batch time 22:17:15

Click the link in Completed Jobs for a batch and you see the details.

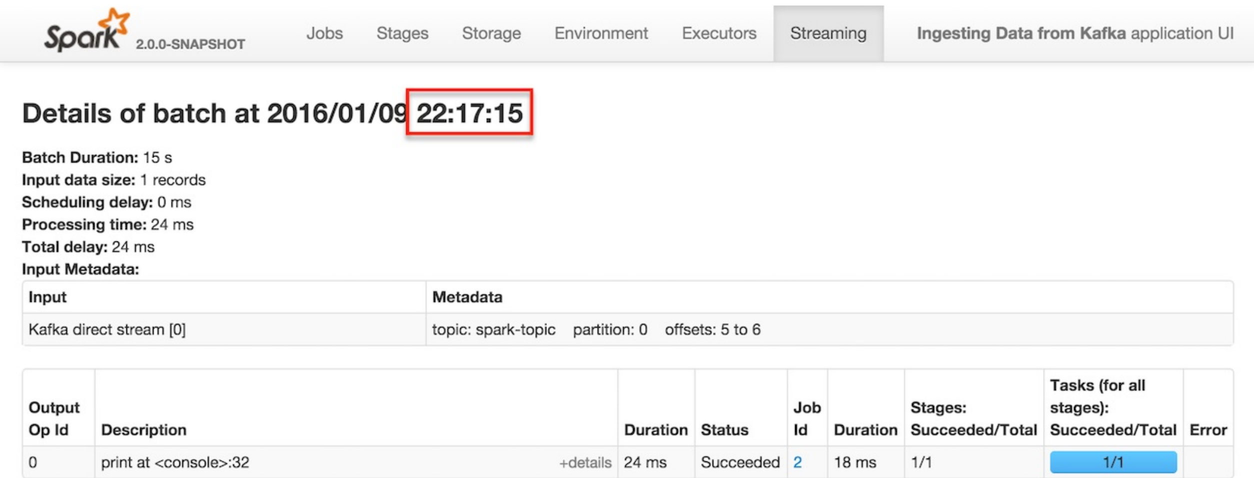

Figure 3. Details of batch in web UI for batch time 22:17:15

`spark-streaming-kafka-0-10` Library Dependency

The new API for both Kafka RDD and DStream is in the `spark-streaming-kafka` artifact. Add the following dependency to sbt project to use the streaming integration:

```
libraryDependencies += "org.apache.spark" %% "spark-streaming-kafka-0-10" % "2.0.1"
```

Tip	`spark-streaming-kafka-0-10` module is not included in the CLASSPATH of spark-she have to start it with `--packages` command-line option. `./bin/spark-shell --packages org.apache.spark:spark-streaming-kafka-0-10_2.11:2.`

Note	Replace `2.0.1` or `2.1.0-SNAPSHOT` with available version as found at The Central Repository's search.

LeaderOffset

`LeaderOffset` is an internal class to represent an offset on the topic partition on the broker that works on a host and a port.

Recommended Reading

- Exactly-once Spark Streaming from Apache Kafka

`KafkaUtils` — Creating Kafka DStreams and RDDs

`KafkaUtils` is the object with the factory methods to create input dstreams and RDDs from records in topics in Apache Kafka.

```
import org.apache.spark.streaming.kafka010.KafkaUtils
```

Tip	Use `spark-streaming-kafka-0-10` Library Dependency.

<table>
<tr>
<td>Tip</td>
<td>
Enable <code>WARN</code> logging level for <code>org.apache.spark.streaming.kafka010.KafkaUtils</code> logger to see what happens inside.
Add the following line to <code>conf/log4j.properties</code> :
<pre>log4j.logger.org.apache.spark.streaming.kafka010.KafkaUtils=WARN</pre>
Refer to Logging.
</td>
</tr>
</table>

Creating Kafka DStream — `createDirectStream` Method

```
createDirectStream[K, V](
  ssc: StreamingContext,
  locationStrategy: LocationStrategy,
  consumerStrategy: ConsumerStrategy[K, V]): InputDStream[ConsumerRecord[K, V]]
```

`createDirectStream` is a method that creates a DirectKafkaInputDStream from a StreamingContext, LocationStrategy, and ConsumerStrategy.

Tip	Enable `DEBUG` logging level for `org.apache.kafka.clients.consumer.KafkaConsumer` logger to see what happens inside the Kafka consumer that is used to communicate with Kafka broker(s).
	
	The following DEBUGs are from when a `DirectKafkaInputDStream` is started.
	
	`DEBUG KafkaConsumer: Starting the Kafka consumer`
	`DEBUG KafkaConsumer: Kafka consumer created`
	`DEBUG KafkaConsumer: Subscribed to topic(s): basic1, basic2, basic3`
	
	Add the following line to `conf/log4j.properties`:
	
	`log4j.logger.org.apache.kafka.clients.consumer.KafkaConsumer=DEBUG`
	
	Refer to Logging.

Using KafkaUtils.createDirectStream to Connect to Kafka Brokers

```
// Include org.apache.spark:spark-streaming-kafka-0-10_2.11:2.1.0-SNAPSHOT dependency
in the CLASSPATH, e.g.
// $ ./bin/spark-shell --packages org.apache.spark:spark-streaming-kafka-0-10_2.11:2.1
.0-SNAPSHOT

import org.apache.spark.streaming._
import org.apache.spark.SparkContext
val sc = SparkContext.getOrCreate
val ssc = new StreamingContext(sc, Seconds(5))

import org.apache.spark.streaming.kafka010._

val preferredHosts = LocationStrategies.PreferConsistent
val topics = List("topic1", "topic2", "topic3")
import org.apache.kafka.common.serialization.StringDeserializer
val kafkaParams = Map(
  "bootstrap.servers" -> "localhost:9092",
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> "spark-streaming-notes",
  "auto.offset.reset" -> "earliest"
)
import org.apache.kafka.common.TopicPartition
val offsets = Map(new TopicPartition("topic3", 0) -> 2L)

val dstream = KafkaUtils.createDirectStream[String, String](
  ssc,
  preferredHosts,
  ConsumerStrategies.Subscribe[String, String](topics, kafkaParams, offsets))

dstream.foreachRDD { rdd =>
  // Get the offset ranges in the RDD
  val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
  for (o <- offsetRanges) {
    println(s"${o.topic} ${o.partition} offsets: ${o.fromOffset} to ${o.untilOffset}")
  }
}

ssc.start

// the above code is printing out topic details every 5 seconds
// until you stop it.

ssc.stop(stopSparkContext = false)
```

Creating Kafka RDD — `createRDD` Method

```
def createRDD[K, V](
  sc: SparkContext,
  kafkaParams: java.util.Map[String, Object],
  offsetRanges: Array[OffsetRange],
  locationStrategy: LocationStrategy): RDD[ConsumerRecord[K, V]]
```

`createRDD` creates a KafkaRDD.

Caution	FIXME

`fixKafkaParams` Internal Method

```
fixKafkaParams(kafkaParams: ju.HashMap[String, Object]): Unit
```

`fixKafkaParams` fixes Kafka parameters to prevent any issues with communicating with Kafka on Spark executors.

Caution	FIXME

DirectKafkaInputDStream — Direct Kafka DStream

`DirectKafkaInputDStream` is an input dstream of KafkaRDD batches.

`DirectKafkaInputDStream` is also a `CanCommitOffsets` object.

As an input dstream, `DirectKafkaInputDStream` implements the mandatory abstract Methods (from DStream Contract and InputDStream Contract):

1. `dependencies` returns an empty collection, i.e. it has no dependencies on other streams (other than Kafka brokers to read data from).
2. `slideDuration` passes all calls on to DStreamGraph.batchDuration.
3. compute to create a `KafkaRDD` per batch.
4. start to start polling for messages from Kafka.
5. stop to close the Kafka consumer (and therefore polling for messages from Kafka).

The `name` of a `DirectKafkaInputDStream` is **Kafka 0.10 direct stream [id]** (that you can use to differentiate between the different implementations for Kafka 0.10+ and older releases).

Tip	You can find the name of a input dstream in the Streaming tab in web UI (in the details of a batch in **Input Metadata** section).

It uses spark.streaming.kafka.maxRetries setting while computing `latestLeaderOffsets` (i.e. a mapping of `kafka.common.TopicAndPartition` and LeaderOffset).

> **Tip**
>
> Enable `INFO` logging level for `org.apache.spark.streaming.kafka010.DirectKafkaInputDStream` logger to see what happens inside.
>
> Add the following line to `conf/log4j.properties`:
>
> ```
> log4j.logger.org.apache.spark.streaming.kafka010.DirectKafkaInputDStream=INFO
> ```
>
> Refer to Logging.

Creating `DirectKafkaInputDStream` Instance

You can create a `DirectKafkaInputDStream` instance using KafkaUtils.createDirectStream factory method.

```
import org.apache.spark.streaming.kafka010.KafkaUtils

// WARN: Incomplete to show only relevant parts
val dstream = KafkaUtils.createDirectStream[String, String](
  ssc = streamingContext,
  locationStrategy = hosts,
  consumerStrategy = ConsumerStrategies.Subscribe[String, String](topics, kafkaParams,
 offsets))
```

Internally, when a `DirectKafkaInputDStream` instance is created, it initializes the internal executorKafkaParams using the input `consumerStrategy` 's executorKafkaParams.

Tip	Use ConsumerStrategy for a Kafka Consumer configuration.

With WARN logging level enabled for the KafkaUtils logger, you may see the following WARN messages and one ERROR in the logs (the number of messages depends on how correct the Kafka Consumer configuration is):

```
WARN KafkaUtils: overriding enable.auto.commit to false for executor
WARN KafkaUtils: overriding auto.offset.reset to none for executor
ERROR KafkaUtils: group.id is null, you should probably set it
WARN KafkaUtils: overriding executor group.id to spark-executor-null
WARN KafkaUtils: overriding receive.buffer.bytes to 65536 see KAFKA-3135
```

Tip	You should always set `group.id` in Kafka parameters for `DirectKafkaInputDStream` . Refer to ConsumerStrategy — Kafka Consumers' Post-Configuration API.

It initializes the internal currentOffsets property.

It creates an instance of `DirectKafkaInputDStreamCheckpointData` as `checkpointData` .

It sets up `rateController` as `DirectKafkaRateController` when backpressure is enabled.

It sets up `maxRateLimitPerPartition` as spark.streaming.kafka.maxRatePerPartition.

It initializes commitQueue and commitCallback properties.

`currentOffsets` Property

```
currentOffsets: Map[TopicPartition, Long]
```

`currentOffsets` holds the latest (highest) available offsets for all the topic partitions the dstream is subscribed to (as set by latestOffsets and compute).

`currentOffsets` is initialized when `DirectKafkaInputDStream` is created afresh (it could also be re-created from a checkpoint).

The ConsumerStrategy (that was used to initialize `DirectKafkaInputDStream`) uses it to create a Kafka Consumer.

It is then set to the available offsets when `DirectKafkaInputDStream` is started.

`commitCallback` Property

```
commitCallback: AtomicReference[OffsetCommitCallback]
```

`commitCallback` is initialized when `DirectKafkaInputDStream` is created. It is set to a `OffsetCommitCallback` that is the input parameter of `commitAsync` when it is called (as part of the `CanCommitOffsets` contract that `DirectKafkaInputDStream` implements).

`commitQueue` Property

```
commitQueue: ConcurrentLinkedQueue[OffsetRange]
```

`commitQueue` is initialized when `DirectKafkaInputDStream` is created. It is used in `commitAsync` (that is part of the `CanCommitOffsets` contract that `DirectKafkaInputDStream` implements) to queue up offsets for commit to Kafka at a future time (i.e. when the internal commitAll is called).

Tip	Read java.util.concurrent.ConcurrentLinkedQueue javadoc.

`executorKafkaParams` Attribute

```
executorKafkaParams: HashMap[String, Object]
```

`executorKafkaParams` is a collection of …FIXME

When `DirectKafkaInputDStream` is created, it initializes `executorKafkaParams` with `executorKafkaParams` of the given `ConsumerStrategy` (that was used to create the `DirectKafkaInputDStream` instance).

`executorKafkaParams` is then reviewed and corrected where needed.

Note	`executorKafkaParams` is used when computing a `KafkaRDD` for a batch and restoring `KafkaRDD` s from checkpoint.

Starting `DirectKafkaInputDStream` — `start` Method

```
start(): Unit
```

`start` creates a Kafka consumer and fetches available records in the subscribed list of topics and partitions (using Kafka's Consumer.poll with `0` timeout that says to return immediately with any records that are available currently).

Note	`start` is part of the InputDStream Contract.

After the polling, `start` checks if the internal currentOffsets is empty, and if it is, it requests Kafka for topic (using Kafka's Consumer.assignment) and builds a map with topics and their offsets (using Kafka's Consumer.position).

Ultimately, `start` pauses all partitions (using Kafka's Consumer.pause with the internal collection of topics and their current offsets).

Generating KafkaRDD for Batch Interval — `compute` Method

```
compute(validTime: Time): Option[KafkaRDD[K, V]]
```

Note	`compute` is a part of the DStream Contract.

`compute` *always* computes a KafkaRDD (despite the return type that allows for no RDDs and irrespective the number of records inside). It is left to a `KafkaRDD` itself to decide what to do when no Kafka records exist in topic partitions to process for a given batch.

Note	It is `DStreamGraph` to request generating streaming jobs for batches.

When `compute` is called, it calls latestOffsets and clamp. The result topic partition offsets are then mapped to OffsetRanges with a topic, a partition, and current offset for the given partition and the result offset. That in turn is used to create KafkaRDD (with the current SparkContext, executorKafkaParams, the `OffsetRange` s, preferred hosts, and `useConsumerCache` enabled).

Caution	FIXME We all would appreciate if Jacek made the above less technical.

Caution	FIXME What's `useConsumerCache` ?

With that, `compute` informs `InputInfoTracker` about the state of an input stream (as StreamInputInfo with metadata with offsets and a human-friendly description).

In the end, `compute` sets the just-calculated offsets as current offsets, asynchronously commits all queued offsets (from commitQueue) and returns the newly-created `KafkaRDD`.

Committing Queued Offsets to Kafka — `commitAll` Method

```
commitAll(): Unit
```

`commitAll` commits all queued OffsetRanges in commitQueue (using Kafka's Consumer.commitAsync).

Note	`commitAll` is used for every batch interval (when compute is called to generate a `KafkaRDD`).

Internally, `commitAll` walks through `OffsetRange` s in commitQueue and calculates the offsets for every topic partition. It uses them to create a collection of Kafka's TopicPartition and OffsetAndMetadata pairs for Kafka's Consumer.commitAsync using the internal Kafka consumer reference.

`clamp` Method

```
clamp(offsets: Map[TopicPartition, Long]): Map[TopicPartition, Long]
```

`clamp` calls maxMessagesPerPartition on the input `offsets` collection (of topic partitions with their offsets)…

Caution	FIXME

`maxMessagesPerPartition` Method

Caution	FIXME

Creating Kafka Consumer — `consumer` Method

```
consumer(): Consumer[K, V]
```

`consumer` creates a Kafka `Consumer` with keys of type `K` and values of type `V` (specified when the `DirectKafkaInputDStream` is created).

`consumer` starts the ConsumerStrategy (that was used when the `DirectKafkaInputDStream` was created). It passes the internal collection of `TopicPartition` s and their offsets.

Caution	FIXME A note with What `ConsumerStrategy` is for?

Calculating Preferred Hosts Using `LocationStrategy` — `getPreferredHosts` Method

```
getPreferredHosts: java.util.Map[TopicPartition, String]
```

`getPreferredHosts` calculates preferred hosts per topic partition (that are later used to map KafkaRDD partitions to host leaders of topic partitions that Spark executors read records from).

`getPreferredHosts` relies exclusively on the LocationStrategy that was passed in when creating a `DirectKafkaInputDStream` instance.

Table 1. DirectKafkaInputDStream.getPreferredHosts and Location Strategies

Location Strategy	DirectKafkaInputDStream.getPreferredHosts
`PreferBrokers`	Calls Kafka broker(s) for topic partition assignments.
`PreferConsistent`	No host preference. Returns an empty collection of preferred hosts per topic partition. It does not call Kafka broker(s) for topic assignments.
`PreferFixed`	Returns the preferred hosts that were passed in when `PreferFixed` was created. It does not call Kafka broker(s) for topic assignments.

Note	`getPreferredHosts` is used when creating a KafkaRDD for a batch interval.

Requesting Partition Assignments from Kafka — `getBrokers` Method

```
getBrokers: ju.Map[TopicPartition, String]
```

`getBrokers` uses the internal Kafka Consumer instance to request Kafka broker(s) for partition assignments, i.e. the leader host per topic partition.

Note	`getBrokers` uses Kafka's Consumer.assignment().

Stopping DirectKafkaInputDStream — `stop` Method

```
stop(): Unit
```

`stop` closes the internal Kafka consumer.

Note	`stop` is a part of the InputDStream Contract.

Requesting Latest Offsets from Kafka Brokers — `latestOffsets` Method

```
latestOffsets(): Map[TopicPartition, Long]
```

`latestOffsets` uses the internal Kafka consumer to poll for the latest topic partition offsets, including partitions that have been added recently.

`latestOffsets` calculates the topic partitions that are new (comparing to current offsets) and adds them to `currentOffsets`.

Note	`latestOffsets` uses `poll(0)`, `assignment`, `position` (twice for every `TopicPartition`), `pause`, `seekToEnd` method calls. They *seem* quite performance-heavy. Are they?

The new partitions are `pause` d and the current offsets `seekToEnd` ed.

Caution	FIXME Why are new partitions paused? Make the description more user-friendly.

Note	`latestOffsets` is used when computing a KafkaRDD for batch intervals.

Back Pressure

Caution	FIXME

Back pressure for Direct Kafka input dstream can be configured using spark.streaming.backpressure.enabled setting.

Note	Back pressure is disabled by default.

`ConsumerStrategy` — Kafka Consumers' Post-Configuration API

`ConsumerStrategy` is a contract to create Kafka Consumers in a Spark Streaming application that allows for their custom configuration after the consumers have been created.

Note	Kafka consumers read records from topic partitions in a Kafka cluster.

`ConsumerStrategy[K, V]` is an abstract class with two methods, i.e. executorKafkaParams and onStart.

Table 1. ConsumerStrategy Contract and DirectKafkaInputDStream

Consumer Strategy	DirectKafkaInputDStream Usage
executorKafkaParams	Used when a `DirectKafkaInputDStream` is created to initialize internal state.
onStart	Used to create a Kafka consumer (in `DirectKafkaInputDStream`)

The following table are the Kafka Consumer strategies currently available in Spark 2.0.

Table 2. Kafka Consumer Strategies in Spark Streaming

Consumer Strategy	Description
Assign	
Subscribe	
SubscribePattern	

You can access the predefined `ConsumerStrategy` implementations using ConsumerStrategies factory object.

```
import org.apache.spark.streaming.kafka010.ConsumerStrategies

val topics = List("topic1")
import org.apache.kafka.common.serialization.StringDeserializer
val kafkaParams = Map(
  "bootstrap.servers" -> "localhost:9092",
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> "spark-streaming-notes",
  "auto.offset.reset" -> "earliest"
)
import org.apache.kafka.common.TopicPartition
val offsets = Map(new TopicPartition("topic3", 0) -> 2L)

val subscribeStrategy = ConsumerStrategies.Subscribe[String, String](topics, kafkaPara
ms, offsets)
```

ConsumerStrategy Contract

`executorKafkaParams` Method

```
executorKafkaParams: ju.Map[String, Object]
```

`onStart` Method

```
onStart(currentOffsets: ju.Map[TopicPartition, jl.Long]): Consumer[K, V]
```

Assign Strategy

```
class Assign[K, V](
  topicPartitions: java.util.Collection[TopicPartition],
  kafkaParams: java.util.Map[String, Object],
  offsets: java.util.Map[TopicPartition, java.util.Long]
) extends ConsumerStrategy[K, V]
```

`Assign` returns the input `kafkaParams` directly from executorKafkaParams method.

For `onStart`, `Assign` creates a `KafkaConsumer` (with `kafkaParams`) and explicitly assigns the list of partitions `topicPartitions` to this consumer (using Kafka's KafkaConsumer.assign method). It then overrides the fetch offsets that the consumer will use (on the next poll) to `onStart`'s input `currentOffsets` or `offsets` whatever is not empty (using Kafka's KafkaConsumer.seek method).

Subscribe Strategy

```
class Subscribe[K, V](
  topics: java.util.Collection[jl.String],
  kafkaParams: java.util.Map[String, Object],
  offsets: java.util.Map[TopicPartition, java.util.Long]
) extends ConsumerStrategy[K, V]
```

`Subscribe` returns the input `kafkaParams` directly from executorKafkaParams method.

For `onStart`, `Subscribe` creates a `KafkaConsumer` (with `kafkaParams`) and subscribes to `topics` (using Kafka's KafkaConsumer.subscribe method). For non-empty `currentOffsets` or `offsets` (whatever is not empty in that order), `onStart` polls data for topics or partitions (using Kafka's KafkaConsumer.poll method). It then overrides the fetch offsets that the consumer will use (on the next poll) to `onStart` 's input `currentOffsets` or `offsets` whatever is not empty (using Kafka's KafkaConsumer.seek method).

Tip	You can suppress Kafka's `NoOffsetForPartitionException` with Kafka's `auto.offset.reset` setting set to `NONE` in `kafkaParams`.

In case of Kafka's `NoOffsetForPartitionException` with exception suppression enabled, you can see the following WARN message in the logs:

```
WARN Catching NoOffsetForPartitionException since auto.offset.reset is none.  See KAFK
A-3370
```

Tip	Read through KAFKA-3370: Add options to auto.offset.reset to reset offsets upon initialization only

```
??? FIXME Example with the WARN above
```

SubscribePattern Strategy

```
class SubscribePattern[K, V](
    pattern: java.util.regex.Pattern,
    kafkaParams: java.util.Map[String, Object],
    offsets: java.util.Map[TopicPartition, java.util.Long]
) extends ConsumerStrategy[K, V]
```

`SubscribePattern` returns the input `kafkaParams` directly from executorKafkaParams method.

For `onStart`, `SubscribePattern` creates a `KafkaConsumer` (with `kafkaParams`) and subscribes to `pattern` topics with Kafka's internal `NoOpConsumerRebalanceListener` (using Kafka's KafkaConsumer.subscribe method).

Note	The only difference between SubscribePattern and Subscribe Consumer strategies is the use of Kafka's KafkaConsumer.subscribe(Collection, ConsumerRebalanceListener) and KafkaConsumer.subscribe(Collection) methods, respectively.

ConsumerStrategies Factory Object

LocationStrategy — Preferred Hosts per Topic Partitions

`LocationStrategy` allows a DirectKafkaInputDStream to request Spark executors to execute Kafka consumers as close topic leaders of topic partitions as possible.

`LocationStrategy` is used when `DirectKafkaInputDStream` computes a `KafkaRDD` for a given batch interval and is a means of distributing processing Kafka records across Spark executors.

Table 1. Location Strategies in Spark Streaming

Location Strategy	Description
PreferBrokers	Use when executors are on the same nodes as your Kafka brokers.
PreferConsistent	Use in most cases as it consistently distributes partitions across all executors.
PreferFixed	Use to place particular `TopicPartition` s on particular hosts if your load is uneven. Accepts a collection of topic partition and host pairs. Any topic partition not specified uses a consistent location.

Note	A topic partition is described using Kafka's TopicPartition.

You can create a `LocationStrategy` using LocationStrategies factory object.

```
import org.apache.spark.streaming.kafka010.LocationStrategies
val preferredHosts = LocationStrategies.PreferConsistent
```

LocationStrategies Factory Object

`LocationStrategies` holds the factory methods to access `LocationStrategy` objects.

```
PreferBrokers: LocationStrategy
PreferConsistent: LocationStrategy
PreferFixed(hostMap: collection.Map[TopicPartition, String]): LocationStrategy
```

KafkaRDD

`KafkaRDD` is a RDD of Kafka's `ConsumerRecords` from topics in Apache Kafka with support for HasOffsetRanges.

Note	Kafka's ConsumerRecord holds a topic name, a partition number, the offset of the record in the Kafka partition and the record itself (as a key-value pair).

`KafkaRDD` uses KafkaRDDPartition as the partitions that define their preferred locations (as the host of the topic).

Note	The feature of defining placement preference (aka *location preference*) very well maps a `KafkaRDD` partition to a Kafka topic partition on a Kafka-closest host.

`KafkaRDD` is created:

- On demand using KafkaUtils.createRDD
- In batches using KafkaUtils.createDirectStream

`KafkaRDD` is also created when a `DirectKafkaInputDStream` restores `KafkaRDDs` from checkpoint.

Note	`KafkaRDD` is a `private[spark]` class.

<table><tr><td>Tip</td><td>Enable <code>INFO</code> logging level for <code>org.apache.spark.streaming.kafka010.KafkaRDD</code> logger to see what happens inside.

Add the following line to <code>conf/log4j.properties</code> :

<pre>log4j.logger.org.apache.spark.streaming.kafka010.KafkaRDD=INFO</pre>
Refer to Logging.</td></tr></table>

`getPartitions` Method

Caution	FIXME

Creating KafkaRDD Instance

`KafkaRDD` takes the following when created:

- SparkContext
- Collection of Kafka parameters with their values
- Collection of OffsetRanges
- Kafka's `TopicPartitions` and their hosts
- Flag to control whether to use consumer cache

Caution	FIXME Are the hosts in `preferredHosts` Kafka brokers?

`KafkaRDD` initializes the internal registries and counters.

Computing KafkaRDDPartition (in TaskContext) — `compute` Method

```
compute(thePart: Partition, context: TaskContext): Iterator[ConsumerRecord[K, V]]
```

Note	`compute` is a part of the RDD Contract.

`compute` assumes that it works with `thePart` as KafkaRDDPartition only. It asserts that the offsets are correct, i.e. `fromOffset` is at most `untilOffset`.

If the beginning and ending offsets are the same, you should see the following INFO message in the logs and `compute` returns an empty collection.

```
INFO KafkaRDD: Beginning offset [fromOffset] is the same as ending offset skipping [to
pic] [partition]
```

Otherwise, when the beginning and ending offsets are different, a KafkaRDDIterator is created (for the partition and the input TaskContext) and returned.

Getting Placement Preferences of Partition — `getPreferredLocations` Method

```
getPreferredLocations(thePart: Partition): Seq[String]
```

Note	`getPreferredLocations` is a part of RDD contract to define the placement preferences (aka *preferred locations*) of a partition.

`getPreferredLocations` casts `thePart` to KafkaRDDPartition.

`getPreferredLocations` finds all executors.

Caution	FIXME Use proper name for executors.

`getPreferredLocations` requests `KafkaRDDPartition` for the Kafka `TopicPartition` and finds the preferred hosts for the partition.

Note	`getPreferredLocations` uses preferredHosts that was given when `KafkaRDD` was created.

If `getPreferredLocations` did not find the preferred host for the partition, all executors are used. Otherwise, `getPreferredLocations` includes only executors on the preferred host.

If `getPreferredLocations` found no executors, all the executors are considered.

`getPreferredLocations` returns one matching executor (for the `TopicPartition`) or an empty collection.

Creating ExecutorCacheTaskLocations for All Executors in Cluster — `executors` Internal Method

```
executors(): Array[ExecutorCacheTaskLocation]
```

`executors` requests `BlockManagerMaster` for all `BlockManager` nodes (peers) in a cluster (that represent all the executors available).

Note	`executors` uses `KafkaRDD` 's SparkContext to access the current `BlockManager` and in turn BlockManagerMaster.

`executors` creates `ExecutorCacheTaskLocations` using the peers' hosts and executor ids.

Note	`executors` are sorted by their host names and executor ids.

Caution	FIXME Image for sorted ExecutorCacheTaskLocations.

Note	`executors` is used exclusively when `KafkaRDD` is requested for its placement preferences (aka *preferred locations*).

`KafkaRDDPartition`

`KafkaRDDPartition` is…FIXME

`topicPartition`

Caution	FIXME

HasOffsetRanges and OffsetRange

HasOffsetRanges

`HasOffsetRanges` represents an object that has a collection of OffsetRanges (i.e. a range of offsets from a single Kafka topic partition).

`HasOffsetRanges` is part of `org.apache.spark.streaming.kafka010` package.

Note	KafkaRDD is a `HasOffsetRanges` object.

You can access `HasOffsetRanges` given a KafkaRDD as follows:

```
import org.apache.spark.streaming.kafka010.KafkaUtils
KafkaUtils.createDirectStream(...).foreachRDD { rdd =>
  import org.apache.spark.streaming.kafka010.OffsetRange
  val offsetRanges: Array[OffsetRange] = rdd.asInstanceOf[HasOffsetRanges].offsetRange
s
}
```

OffsetRange

`OffsetRange` represents a range of offsets from a single Kafka TopicPartition (i.e. a topic name and partition number).

`OffsetRange` holds a `topic`, `partition` number, `fromOffset` (inclusive) and `untilOffset` (exclusive) offsets.

You can create instances of `OffsetRange` using the factory methods from `OffsetRange` companion object. You can then count the number of records in a topic partition using count method.

```
// Start spark-shell with spark-streaming-kafka-0-10_2.11 dependency
// --packages org.apache.spark:spark-streaming-kafka-0-10_2.11:2.1.0-SNAPSHOT
import org.apache.spark.streaming.kafka010.OffsetRange

scala> val offsets = OffsetRange(topic = "spark-logs", partition = 0, fromOffset = 2,
untilOffset = 5)
offsets: org.apache.spark.streaming.kafka010.OffsetRange = OffsetRange(topic: 'spark-l
ogs', partition: 0, range: [2 -> 5])

scala> offsets.count
res0: Long = 3

scala> offsets.topicPartition
res1: org.apache.kafka.common.TopicPartition = spark-logs-0
```

`OffsetRange` is part of `org.apache.spark.streaming.kafka010` package.

Creating OffsetRange Instance

You can create instances of `OffsetRange` using the following factory methods (from `OffsetRange` companion object):

```
OffsetRange.create(
  topic: String,
  partition: Int,
  fromOffset: Long,
  untilOffset: Long): OffsetRange

OffsetRange.create(
  topicPartition: TopicPartition,
  fromOffset: Long,
  untilOffset: Long): OffsetRange

OffsetRange.apply(
  topic: String,
  partition: Int,
  fromOffset: Long,
  untilOffset: Long): OffsetRange

OffsetRange.apply(
  topicPartition: TopicPartition,
  fromOffset: Long,
  untilOffset: Long): OffsetRange
```

Counting Records in Topic Partition — `count` method

```
count(): Long
```

`count` counts the number of records in a `OffsetRange`.

RecurringTimer

```
class RecurringTimer(clock: Clock, period: Long, callback: (Long) => Unit, name: String
)
```

`RecurringTimer` (aka **timer**) is a `private[streaming]` class that uses a single daemon thread prefixed `RecurringTimer - [name]` that, once started, executes `callback` in a loop every `period` time (until it is stopped).

The wait time is achieved by `Clock.waitTillTime` (that makes testing easier).

Tip	Enable `INFO` or `DEBUG` logging level for `org.apache.spark.streaming.util.RecurringTimer` logger to see what happens inside. Add the following line to `conf/log4j.properties` : `log4j.logger.org.apache.spark.streaming.util.RecurringTimer=DEBUG` Refer to Logging.

When `RecurringTimer` triggers an action for a `period` , you should see the following DEBUG message in the logs:

```
DEBUG RecurringTimer: Callback for [name] called at time [prevTime]
```

Start and Restart Times

```
getStartTime(): Long
getRestartTime(originalStartTime: Long): Long
```

`getStartTime` and `getRestartTime` are helper methods that calculate time.

`getStartTime` calculates a time that is a multiple of the timer's `period` and is right after the current system time.

Note	`getStartTime` is used when JobGenerator is started.

`getRestartTime` is similar to `getStartTime` but includes `originalStartTime` input parameter, i.e. it calculates a time as `getStartTime` but shifts the result to accommodate the time gap since `originalStartTime` .

Note	`getRestartTime` is used when JobGenerator is restarted.

Starting Timer

```
start(startTime: Long): Long
start(): Long (1)
```

1. Uses the internal getStartTime method to calculate `startTime` and calls `start(startTime: Long)` .

You can start a `RecurringTimer` using `start` methods.

Note	`start()` method uses the internal getStartTime method to calculate `startTime` and calls `start(startTime: Long)` .

When `start` is called, it sets the internal `nextTime` to the given input parameter `startTime` and starts the internal daemon thread. This is the moment when the clock starts ticking…

You should see the following INFO message in the logs:

```
INFO RecurringTimer: Started timer for [name] at time [nextTime]
```

Stopping Timer

```
stop(interruptTimer: Boolean): Long
```

A timer is stopped using `stop` method.

Note	It is called when JobGenerator stops.

When called, you should see the following INFO message in the logs:

```
INFO RecurringTimer: Stopped timer for [name] after time [prevTime]
```

`stop` method uses the internal `stopped` flag to mark the stopped state and returns the last `period` for which it was successfully executed (tracked as `prevTime` internally).

Note	Before it fully terminates, it triggers `callback` one more/last time, i.e. `callback` is executed for a `period` after `RecurringTimer` has been (marked) stopped.

Fun Fact

You can execute `org.apache.spark.streaming.util.RecurringTimer` as a command-line standalone application.

```
$ ./bin/spark-class org.apache.spark.streaming.util.RecurringTimer
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
INFO RecurringTimer: Started timer for Test at time 1453787444000
INFO RecurringTimer: 1453787444000: 1453787444000
DEBUG RecurringTimer: Callback for Test called at time 1453787444000
INFO RecurringTimer: 1453787445005: 1005
DEBUG RecurringTimer: Callback for Test called at time 1453787445000
INFO RecurringTimer: 1453787446004: 999
DEBUG RecurringTimer: Callback for Test called at time 1453787446000
INFO RecurringTimer: 1453787447005: 1001
DEBUG RecurringTimer: Callback for Test called at time 1453787447000
INFO RecurringTimer: 1453787448000: 995
DEBUG RecurringTimer: Callback for Test called at time 1453787448000
^C
INFO ShutdownHookManager: Shutdown hook called
INFO ShutdownHookManager: Deleting directory /private/var/folders/0w/kb0d3rqn4zb9fcc91
pxhgn8w0000gn/T/spark-71dbd43d-2db3-4527-adb8-f1174d799b0d/repl-a6b9bf12-fec2-4004-923
6-3b0ab772cc94
INFO ShutdownHookManager: Deleting directory /private/var/folders/0w/kb0d3rqn4zb9fcc91
pxhgn8w0000gn/T/spark-71dbd43d-2db3-4527-adb8-f1174d799b0d
```

Backpressure (Back Pressure)

Quoting TD from his talk about Spark Streaming:

> Backpressure is to make applications robust against data surges.

With backpressure you can guarantee that your Spark Streaming application is **stable**, i.e. receives data only as fast as it can process it.

Note	Backpressure shifts the trouble of buffering input records to the sender so it keeps records until they could be processed by a streaming application. You could alternatively use dynamic allocation feature in Spark Streaming to increase the capacity of streaming infrastructure without slowing down the senders.

Backpressure is disabled by default and can be turned on using spark.streaming.backpressure.enabled setting.

You can monitor a streaming application using web UI. It is important to ensure that the batch processing time is shorter than the batch interval. Backpressure introduces a **feedback loop** so the streaming system can adapt to longer processing times and avoid instability.

Note	Backpressure is available since Spark 1.5.

RateController

Tip	Read up on back pressure in Wikipedia.

`RateController` is a contract for single-dstream StreamingListeners that listens to batch completed updates for a dstream and maintains a **rate limit**, i.e. an estimate of the speed at which this stream should ingest messages. With every batch completed update event it calculates the current processing rate and estimates the correct receving rate.

Note	`RateController` works for a single dstream and requires a RateEstimator.

The contract says that RateControllers offer the following method:

```
protected def publish(rate: Long): Unit
```

When created, it creates a daemon single-thread executor service called **stream-rate-update** and initializes the internal `rateLimit` counter which is the current message-ingestion speed.

When a batch completed update happens, a `RateController` grabs `processingEndTime`, `processingDelay`, `schedulingDelay`, and `numRecords` processed for the batch, computes a rate limit and publishes the current value. The computed value is set as the present rate limit, and published (using the sole abstract `publish` method).

Computing a rate limit happens using the RateEstimator's `compute` method.

Caution	FIXME Where is this used? What are the use cases?

InputDStreams can define a `RateController` that is registered to JobScheduler's `listenerBus` (using `ssc.addStreamingListener`) when JobScheduler starts.

RateEstimator

`RateEstimator` computes the rate given the input `time`, `elements`, `processingDelay`, and `schedulingDelay`.

It is an abstract class with the following abstract method:

```
def compute(
    time: Long,
    elements: Long,
    processingDelay: Long,
    schedulingDelay: Long): Option[Double]
```

You can control what `RateEstimator` to use through spark.streaming.backpressure.rateEstimator setting.

The only possible `RateEstimator` to use is the pid rate estimator.

PID Rate Estimator

PID Rate Estimator (represented as `PIDRateEstimator`) implements a proportional-integral-derivative (PID) controller which acts on the speed of ingestion of records into an input dstream.

Warning	The **PID rate estimator** is the only possible estimator. All other rate estimators lead to `IllegalArgumentException` being thrown.

It uses the following settings:

- `spark.streaming.backpressure.pid.proportional` (default: 1.0) can be 0 or greater.
- `spark.streaming.backpressure.pid.integral` (default: 0.2) can be 0 or greater.
- `spark.streaming.backpressure.pid.derived` (default: 0.0) can be 0 or greater.
- `spark.streaming.backpressure.pid.minRate` (default: 100) must be greater than 0.

Note	The PID rate estimator is used by DirectKafkaInputDStream and input dstreams with receivers (aka ReceiverInputDStreams).

Tip

Enable `INFO` or `TRACE` logging level for `org.apache.spark.streaming.scheduler.rate.PIDRateEstimator` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.streaming.scheduler.rate.PIDRateEstimator=TRACE
```

Refer to Logging.

When the PID rate estimator is created you should see the following INFO message in the logs:

```
INFO PIDRateEstimator: Created PIDRateEstimator with proportional = [proportional], in
tegral = [integral], derivative = [derivative], min rate = [minRate]
```

When the pid rate estimator computes the rate limit for the current time, you should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator:
time = [time], # records = [numElements], processing time = [processingDelay], schedul
ing delay = [schedulingDelay]
```

If the time to compute the current rate limit for is before the latest time or the number of records is 0 or less, or processing delay is 0 or less, the rate estimation is skipped. You should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator: Rate estimation skipped
```

And no rate limit is returned.

Otherwise, when this is to compute the rate estimation for next time and there are records processed as well as the processing delay is positive, it computes the rate estimate.

Once the new rate has already been computed, you should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator:
 latestRate = [latestRate], error = [error]
 latestError = [latestError], historicalError = [historicalError]
 delaySinceUpdate = [delaySinceUpdate], dError = [dError]
```

If it was the first computation of the limit rate, you should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator: First run, rate estimation skipped
```

No rate limit is returned.

Otherwise, when it is another limit rate, you should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator: New rate = [newRate]
```

And the current rate limit is returned.

Elastic Scaling (Dynamic Allocation)

Dynamic Allocation in Spark Streaming makes for **adaptive streaming applications** by scaling them up and down to adapt to load variations. It actively controls resources (as executors) and prevents resources from being wasted when the processing time is short (comparing to a batch interval) - **scale down** - or adds new executors to decrease the processing time - **scale up**.

Note	It is a work in progress in Spark Streaming and should be available in Spark 2.0.

The motivation is to control the number of executors required to process input records when their number increases to the point when the processing time could become longer than the batch interval.

Configuration

- `spark.streaming.dynamicAllocation.enabled` controls whether to enabled dynamic allocation (`true`) or not (`false`).

ExecutorAllocationManager

Caution	FIXME

requestExecutors

killExecutor

StreamingSource

Caution	FIXME

Settings

The following list are the settings used to configure Spark Streaming applications.

Caution	FIXME Describe how to set them in streaming applications.

- `spark.streaming.kafka.maxRetries` (default: `1`) sets up the number of connection attempts to Kafka brokers.
- `spark.streaming.receiver.writeAheadLog.enable` (default: `false`) controls what ReceivedBlockHandler to use: `WriteAheadLogBasedBlockHandler` or `BlockManagerBasedBlockHandler`.
- `spark.streaming.receiver.blockStoreTimeout` (default: `30`) time in seconds to wait until both writes to a write-ahead log and BlockManager complete successfully.
- `spark.streaming.clock` (default: `org.apache.spark.util.SystemClock`) specifies a fully-qualified class name that extends `org.apache.spark.util.Clock` to represent time. It is used in JobGenerator.
- `spark.streaming.ui.retainedBatches` (default: `1000`) controls the number of `BatchUIData` elements about completed batches in a first-in-first-out (FIFO) queue that are used to display statistics in Streaming page in web UI.
- `spark.streaming.receiverRestartDelay` (default: `2000`) - the time interval between a receiver is stopped and started again.
- `spark.streaming.concurrentJobs` (default: `1`) is the number of concurrent jobs, i.e. threads in streaming-job-executor thread pool.
- `spark.streaming.stopSparkContextByDefault` (default: `true`) controls whether (`true`) or not (`false`) to stop the underlying SparkContext (regardless of whether this `StreamingContext` has been started).
- `spark.streaming.kafka.maxRatePerPartition` (default: `0`) if non-`0` sets maximum number of messages per partition.
- `spark.streaming.manualClock.jump` (default: `0`) offsets (aka *jumps*) the system time, i.e. adds its value to checkpoint time, when used with the clock being a subclass of `org.apache.spark.util.ManualClock`. It is used when JobGenerator is restarted from checkpoint.
- `spark.streaming.unpersist` (default: `true`) is a flag to control whether output streams should unpersist old RDDs.

- `spark.streaming.gracefulStopTimeout` (default: 10 * batch interval)
- `spark.streaming.stopGracefullyOnShutdown` (default: `false`) controls whether to stop StreamingContext gracefully or not and is used by stopOnShutdown Shutdown Hook.

Checkpointing

- `spark.streaming.checkpoint.directory` - when set and StreamingContext is created, the value of the setting gets passed on to StreamingContext.checkpoint method.

Back Pressure

- `spark.streaming.backpressure.enabled` (default: `false`) - enables (`true`) or disables (`false`) back pressure in input streams with receivers or DirectKafkaInputDStream.
- `spark.streaming.backpressure.rateEstimator` (default: `pid`) is the RateEstimator to use.