pyspark.streaming module

Module contents

class `pyspark.streaming.`**`StreamingContext`**(*sparkContext*, *batchDuration=None*, *jssc=None*) [source]

Bases: **`object`**

Main entry point for Spark Streaming functionality. A StreamingContext represents the connection to a Spark cluster, and can be used to create **`DStream`** various input sources. It can be from an existing **`SparkContext`**. After creating and transforming DStreams, the streaming computation can be started and stopped using *context.start()* and *context.stop()*, respectively. *context.awaitTermination()* allows the current thread to wait for the termination of the context by *stop()* or by an exception.

`addStreamingListener`(*streamingListener*) [source]

Add a [[org.apache.spark.streaming.scheduler.StreamingListener]] object for receiving system events related to streaming.

`awaitTermination`(*timeout=None*) [source]

Wait for the execution to stop.

Parameters:
timeout – time to wait in seconds

`awaitTerminationOrTimeout`(*timeout*) [source]

Wait for the execution to stop. Return *true* if it's stopped; or throw the reported error during the execution; or *false* if the waiting time elapsed before returning from the method.

Parameters:
timeout – time to wait in seconds

`binaryRecordsStream`(*directory*, *recordLength*) [source]

Create an input stream that monitors a Hadoop-compatible file system for new files and reads them as flat binary files with records of fixed length. Files must be written to the monitored directory by "moving" them from another location within the same file system. File names starting with . are ignored.

Parameters:
- **directory** – Directory to load data from
- **recordLength** – Length of each record in bytes

`checkpoint`(*directory*) [source]

Sets the context to periodically checkpoint the DStream operations for master fault-tolerance. The graph will be checkpointed every batch interval.

Parameters:
directory – HDFS-compatible directory where the checkpoint data will be reliably stored

classmethod **`getActive`**() [source]

Return either the currently active StreamingContext (i.e., if there is a context started but not stopped) or None.

classmethod **`getActiveOrCreate`**(*checkpointPath*, *setupFunc*) [source]

Either return the active StreamingContext (i.e. currently started but not stopped), or recreate a StreamingContext from checkpoint data or create a new StreamingContext using the provided setupFunc function. If the checkpointPath is None or does not contain valid checkpoint data, then setupFunc will be called

to create a new context and setup DStreams.

Parameters:
- **checkpointPath** – Checkpoint directory used in an earlier streaming program. Can be None if the intention is to always create a new context when there is no active context.
- **setupFunc** – Function to create a new JavaStreamingContext and setup DStreams

classmethod **getOrCreate**(*checkpointPath*, *setupFunc*) [source]

Either recreate a StreamingContext from checkpoint data or create a new StreamingContext. If checkpoint data exists in the provided *checkpointPath*, then StreamingContext will be recreated from the checkpoint data. If the data does not exist, then the provided setupFunc will be used to create a new context.

Parameters:
- **checkpointPath** – Checkpoint directory used in an earlier streaming program
- **setupFunc** – Function to create a new context and setup DStreams

queueStream(*rdds*, *oneAtATime=True*, *default=None*) [source]

Create an input stream from a queue of RDDs or list. In each batch, it will process either one or all of the RDDs returned by the queue.

> **Note:** Changes to the queue after the stream is created will not be recognized.

Parameters:
- **rdds** – Queue of RDDs
- **oneAtATime** – pick one rdd each time or pick all of them once.
- **default** – The default rdd if no more in rdds

remember(*duration*) [source]

Set each DStreams in this context to remember RDDs it generated in the last given duration. DStreams remember RDDs only for a limited duration of time and releases them for garbage collection. This method allows the developer to specify how long to remember the RDDs (if the developer wishes to query old data outside the DStream computation).

Parameters:
duration – Minimum duration (in seconds) that each DStream should remember its RDDs

socketTextStream(*hostname*, *port*, *storageLevel=StorageLevel(True*, *True*, *False*, *False*, *2)*) [source]

Create an input from TCP source hostname:port. Data is received using a TCP socket and receive byte is interpreted as UTF8 encoded `\n` delimited lines.

Parameters:
- **hostname** – Hostname to connect to for receiving data
- **port** – Port to connect to for receiving data
- **storageLevel** – Storage level to use for storing the received objects

property **sparkContext**

Return SparkContext which is associated with this StreamingContext.

start() [source]

Start the execution of the streams.

stop(*stopSparkContext=True*, *stopGraceFully=False*) [source]

Stop the execution of the streams, with option of ensuring all received data has been processed.

Parameters:
- **stopSparkContext** – Stop the associated SparkContext or not

- **stopGracefully** – Stop gracefully by waiting for the processing of all received data to be completed

textFileStream(*directory*) [source]

Create an input stream that monitors a Hadoop-compatible file system for new files and reads them as text files. Files must be wrriten to the monitored directory by “moving” them from another location within the same file system. File names starting with . are ignored. The text files must be encoded as UTF-8.

transform(*dstreams*, *transformFunc*) [source]

Create a new DStream in which each RDD is generated by applying a function on RDDs of the DStreams. The order of the JavaRDDs in the transform function parameter will be the same as the order of corresponding DStreams in the list.

union(**dstreams*) [source]

Create a unified DStream from multiple DStreams of the same type and same slide duration.

class pyspark.streaming.**DStream**(*jdstream*, *ssc*, *jrdd_deserializer*) [source]

Bases: **object**

A Discretized Stream (DStream), the basic abstraction in Spark Streaming, is a continuous sequence of RDDs (of the same type) representing a continuous stream of data (see **RDD** in the Spark core documentation for more details on RDDs).

DStreams can either be created from live data (such as, data from TCP sockets, etc.) using a **StreamingContext** or it can be generated by transforming existing DStreams using operations such as *map*, *window* and *reduceByKeyAndWindow*. While a Spark Streaming program is running, each DStream periodically generates a RDD, either from live data or by transforming the RDD generated by a parent DStream.

DStreams internally is characterized by a few basic properties:

- A list of other DStreams that the DStream depends on
- A time interval at which the DStream generates an RDD
- A function that is used to generate an RDD after each time interval

cache() [source]

Persist the RDDs of this DStream with the default storage level (*MEMORY_ONLY*).

checkpoint(*interval*) [source]

Enable periodic checkpointing of RDDs of this DStream

Parameters:
interval – time in seconds, after each period of that, generated RDD will be checkpointed

cogroup(*other*, *numPartitions=None*) [source]

Return a new DStream by applying ‘cogroup’ between RDDs of this DStream and *other* DStream.

Hash partitioning is used to generate the RDDs with *numPartitions* partitions.

combineByKey(*createCombiner*, *mergeValue*, *mergeCombiners*, *numPartitions=None*) [source]

Return a new DStream by applying combineByKey to each RDD.

context() [source]

Return the StreamingContext associated with this DStream

count() [source]

Return a new DStream in which each RDD has a single element generated by counting each RDD of this DStream.

countByValue() [source]

Return a new DStream in which each RDD contains the counts of each distinct value in each RDD of this DStream.

countByValueAndWindow(*windowDuration*, *slideDuration*, *numPartitions=None*) [source]

Return a new DStream in which each RDD contains the count of distinct elements in RDDs in a sliding window over this DStream.

Parameters:

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

countByWindow(*windowDuration*, *slideDuration*) [source]

Return a new DStream in which each RDD has a single element generated by counting the number of elements in a window over this DStream. windowDuration and slideDuration are as defined in the window() operation.

This is equivalent to window(windowDuration, slideDuration).count(), but will be more efficient if window is large.

filter(*f*) [source]

Return a new DStream containing only the elements that satisfy predicate.

flatMap(*f*, *preservesPartitioning=False*) [source]

Return a new DStream by applying a function to all elements of this DStream, and then flattening the results

flatMapValues(*f*) [source]

Return a new DStream by applying a flatmap function to the value of each key-value pairs in this DStream without changing the key.

foreachRDD(*func*) [source]

Apply a function to each RDD in this DStream.

fullOuterJoin(*other*, *numPartitions=None*) [source]

Return a new DStream by applying 'full outer join' between RDDs of this DStream and *other* DStream.

Hash partitioning is used to generate the RDDs with *numPartitions* partitions.

glom() [source]

Return a new DStream in which RDD is generated by applying glom() to RDD of this DStream.

groupByKey(*numPartitions=None*) [source]

Return a new DStream by applying groupByKey on each RDD.

groupByKeyAndWindow(*windowDuration*, *slideDuration*, *numPartitions=None*) [source]

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

Parameters:

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

join(*other*, *numPartitions=None*) [source]

Return a new DStream by applying ‘join’ between RDDs of this DStream and *other* DStream.

Hash partitioning is used to generate the RDDs with *numPartitions* partitions.

leftOuterJoin(*other*, *numPartitions=None*) [source]

Return a new DStream by applying ‘left outer join’ between RDDs of this DStream and *other* DStream.

Hash partitioning is used to generate the RDDs with *numPartitions* partitions.

map(*f*, *preservesPartitioning=False*) [source]

Return a new DStream by applying a function to each element of DStream.

mapPartitions(*f*, *preservesPartitioning=False*) [source]

Return a new DStream in which each RDD is generated by applying mapPartitions() to each RDDs of this DStream.

mapPartitionsWithIndex(*f*, *preservesPartitioning=False*) [source]

Return a new DStream in which each RDD is generated by applying mapPartitionsWithIndex() to each RDDs of this DStream.

mapValues(*f*) [source]

Return a new DStream by applying a map function to the value of each key-value pairs in this DStream without changing the key.

partitionBy(*numPartitions*, *partitionFunc=<function portable_hash>*) [source]

Return a copy of the DStream in which each RDD are partitioned using the specified partitioner.

persist(*storageLevel*) [source]

Persist the RDDs of this DStream with the given storage level

pprint(*num=10*) [source]

Print the first num elements of each RDD generated in this DStream.

Parameters:
num – the number of elements from the first will be printed.

reduce(*func*) [source]

Return a new DStream in which each RDD has a single element generated by reducing each RDD of this DStream.

reduceByKey(*func*, *numPartitions=None*) [source]

Return a new DStream by applying reduceByKey to each RDD.

reduceByKeyAndWindow(*func*, *invFunc*, *windowDuration*, *slideDuration=None*, *numPartitions=None*, *filterFunc=None*) [source]

Return a new DStream by applying incremental *reduceByKey* over a sliding window.

The reduced value of over a new window is calculated using the old window’s reduce value :

1. reduce the new values that entered the window (e.g., adding new counts)
2. “inverse reduce” the old values that left the window (e.g., subtracting old counts)

invFunc can be None, then it will reduce all the RDDs in window, could be slower than having *invFunc*.

Parameters:

- **func** – associative and commutative reduce function
- **invFunc** – inverse function of *reduceFunc*

- **windowDuration** – width of the window; must be a multiple of this DStream's batching interval
- **slideDuration** – sliding interval of the window (i.e., the interval after which the new DStream will generate RDDs); must be a multiple of this DStream's batching interval
- **numPartitions** – number of partitions of each RDD in the new DStream.
- **filterFunc** – function to filter expired key-value pairs; only pairs that satisfy the function are retained set this to null if you do not want to filter

`reduceByWindow`(*reduceFunc*, *invReduceFunc*, *windowDuration*, *slideDuration*)

[source]

Return a new DStream in which each RDD has a single element generated by reducing all elements in a sliding window over this DStream.

if *invReduceFunc* is not None, the reduction is done incrementally using the old window's reduced value :

1. reduce the new values that entered the window (e.g., adding new counts)

2. "inverse reduce" the old values that left the window (e.g., subtracting old counts) This is more efficient than *invReduceFunc* is None.

Parameters:

- **reduceFunc** – associative and commutative reduce function
- **invReduceFunc** – inverse reduce function of *reduceFunc*; such that for all y, and invertible x: *invReduceFunc(reduceFunc(x, y), x) = y*
- **windowDuration** – width of the window; must be a multiple of this DStream's batching interval
- **slideDuration** – sliding interval of the window (i.e., the interval after which the new DStream will generate RDDs); must be a multiple of this DStream's batching interval

`repartition`(*numPartitions*) [source]

Return a new DStream with an increased or decreased level of parallelism.

`rightOuterJoin`(*other*, *numPartitions=None*) [source]

Return a new DStream by applying 'right outer join' between RDDs of this DStream and *other* DStream.

Hash partitioning is used to generate the RDDs with *numPartitions* partitions.

`saveAsTextFiles`(*prefix*, *suffix=None*) [source]

Save each RDD in this DStream as at text file, using string representation of elements.

`slice`(*begin*, *end*) [source]

Return all the RDDs between 'begin' to 'end' (both included)

begin, *end* could be datetime.datetime() or unix_timestamp

`transform`(*func*) [source]

Return a new DStream in which each RDD is generated by applying a function on each RDD of this DStream.

func can have one argument of *rdd*, or have two arguments of (*time*, *rdd*)

`transformWith`(*func*, *other*, *keepSerializer=False*) [source]

Return a new DStream in which each RDD is generated by applying a function on each RDD of this DStream and 'other' DStream.

func can have two arguments of (*rdd_a*, *rdd_b*) or have three arguments of (*time*, *rdd_a*, *rdd_b*)

`union`(*other*) [source]

Return a new DStream by unifying data of another DStream with this DStream.

Parameters:
other – Another DStream having the same interval (i.e., slideDuration) as this DStream.

updateStateByKey(*updateFunc*, *numPartitions=None*, *initialRDD=None*) [source]

Return a new “state” DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values of the key.

Parameters:
updateFunc – State update function. If this function returns None, then corresponding state key-value pair will be eliminated.

window(*windowDuration*, *slideDuration=None*) [source]

Return a new DStream in which each RDD contains all the elements in seen in a sliding window of time over this DStream.

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

class pyspark.streaming.**StreamingListener** [source]

Bases: **object**

class **Java** [source]

Bases: **object**

implements = *['org.apache.spark.streaming.api.java.PythonStreamingListener']*

onBatchCompleted(*batchCompleted*) [source]

Called when processing of a batch of jobs has completed.

onBatchStarted(*batchStarted*) [source]

Called when processing of a batch of jobs has started.

onBatchSubmitted(*batchSubmitted*) [source]

Called when a batch of jobs has been submitted for processing.

onOutputOperationCompleted(*outputOperationCompleted*) [source]

Called when processing of a job of a batch has completed

onOutputOperationStarted(*outputOperationStarted*) [source]

Called when processing of a job of a batch has started.

onReceiverError(*receiverError*) [source]

Called when a receiver has reported an error

onReceiverStarted(*receiverStarted*) [source]

Called when a receiver has been started

onReceiverStopped(*receiverStopped*) [source]

Called when a receiver has been stopped

onStreamingStarted(*streamingStarted*) [source]

Called when the streaming has been started.

pyspark.streaming.kinesis module

class pyspark.streaming.kinesis.**KinesisUtils** [source]

Bases: **object**

static **createStream**(*ssc*, *kinesisAppName*, *streamName*, *endpointUrl*, *regionName*, *initialPositionInStream*, *checkpointInterval*, *storageLevel=StorageLevel(True, True, False, False, 2)*, *awsAccessKeyId=None*, *awsSecretKey=None*, *decoder=<function utf8_decoder>*, *stsAssumeRoleArn=None*, *stsSessionName=None*, *stsExternalId=None*) [source]

Create an input stream that pulls messages from a Kinesis stream. This uses the Kinesis Client Library (KCL) to pull messages from Kinesis.

> **Note:** The given AWS credentials will get saved in DStream checkpoints if checkpointing is enabled. Make sure that your checkpoint directory is secure.

Parameters:

- **ssc** – StreamingContext object
- **kinesisAppName** – Kinesis application name used by the Kinesis Client Library (KCL) to update DynamoDB
- **streamName** – Kinesis stream name
- **endpointUrl** – Url of Kinesis service (e.g., ://kinesis.us-east-1.amazonaws.com)
- **regionName** – Name of region used by the Kinesis Client Library (KCL) to update DynamoDB (lease coordination and checkpointing) and CloudWatch (metrics)
- **initialPositionInStream** – In the absence of Kinesis checkpoint info, this is the worker's initial starting position in the stream. The values are either the beginning of the stream per Kinesis' limit of 24 hours (InitialPositionInStream.TRIM_HORIZON) or the tip of the stream (InitialPositionInStream.LATEST).
- **checkpointInterval** – Checkpoint interval for Kinesis checkpointing. See the Kinesis Spark Streaming documentation for more details on the different types of checkpoints.
- **storageLevel** – Storage level to use for storing the received objects (default is StorageLevel.MEMORY_AND_DISK_2)
- **awsAccessKeyId** – AWS AccessKeyId (default is None. If None, will use DefaultAWSCredentialsProviderChain)
- **awsSecretKey** – AWS SecretKey (default is None. If None, will use DefaultAWSCredentialsProviderChain)
- **decoder** – A function used to decode value (default is utf8_decoder)
- **stsAssumeRoleArn** – ARN of IAM role to assume when using STS sessions to read from the Kinesis stream (default is None).
- **stsSessionName** – Name to uniquely identify STS sessions used to read from Kinesis stream, if STS is being used (default is None).
- **stsExternalId** – External ID that can be used to validate against the assumed IAM role's trust policy, if STS is being used (default is None).

Returns:

A DStream object

class pyspark.streaming.kinesis.**InitialPositionInStream** [source]

Bases: **object**

LATEST = *0*

TRIM_HORIZON = *1*

pyspark.streaming.kinesis.**utf8_decoder**(*s*) [source]

Decode the unicode as UTF-8