API for StreamingContext	Meaning
new **StreamingContext**(sparkContext: SparkContext, batchDuration: Duration)	Create a StreamingContext using an existing SparkContext. *BatchDuration is* the time interval at which streaming data will be divided into batches. Duration can be initialized by *Milliseconds(nb), Seconds(nb)* or *Minutes(nb)*
checkpoint(directory: String)	Set the context to periodically checkpoint the DStream operations for driver fault-tolerance. The directory is HDFS-compatible directory where the checkpoint data will be reliably stored. Note that this must be a fault-tolerant file system like HDFS.
socketTextStream(hostname: String, port: Int, sl: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[String]	Creates an input stream from TCP source hostname:port. Data is received using a TCP socket and the receive bytes is interpreted as UTF8 encoded `\n` delimited lines. The *hostname* parameter is the host to connect to for receiving data, *port* is the port to connect to for receiving data, *sl* is the Storage level to use for storing the received objects.
socketStream[T: ClassTag](hostname: String, port: Int, converter: (InputStream) => Iterator[T], sl: StorageLevel): ReceiverInputDStream[T]	Create an input stream from network source hostname:port, where data is received as serialized blocks (serialized using the Spark's serializer) that can be directly pushed into the block manager without deserializing them. This is the most efficient way to receive data. The *hostname* parameter is the host to connect to for receiving data, *port* is the port to connect to for receiving data, *sl* is the Storage level to use for storing the received objects. T is the type of the objects in the received blocks.
textFileStream(directory: String): DStream[String]	Create an input stream that monitors a Hadoop-compatible filesystem for new files and reads them as text files (using key as LongWritable, value as Text and input format as TextInputFormat). 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.
start(): Unit	Start the execution of the streams. Throws IllegalStateException if the StreamingContext is already stopped.
awaitTermination()	Wait for the execution to stop. Any exceptions that occurs during the execution will be thrown in this thread.
awaitTerminationOrTimeout(timeout: Long): Boolean	Wait for the execution to stop. Any exceptions that occurs during the execution will be thrown in this thread. The *timeout* parameter is the time to wait in milliseconds. Returns 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.
stop(stopSparkContext: Boolean = conf.getBoolean("spark.streaming.stopSparkContextByDefault", true)): Unit	Stop the execution of the streams immediately (does not wait for all received data to be processed). By default, if *stopSparkContext* is not specified, the underlying SparkContext will also be stopped. This implicit behavior can be configured using the SparkConf configuration *spark.streaming.stopSparkContextByDefault.* If *stopSparkContext* is true, it stops the associated SparkContext. The underlying SparkContext will be stopped regardless of whether this StreamingContext has been started.

Transformation for DStream[T]	Meaning
filter(filterFunc: (T) => Boolean): DStream[T]	Return a new DStream containing only the elements that satisfy a predicate.
map[U](mapFunc: (T) => U): DStream[U]	Return a new DStream by applying a function to all elements of this DStream
flatMap[U](flatMapFunc: (T) => TraversableOnce[U]): DStream[U]	Return a new DStream by applying a function to all elements of this DStream, and then flattening the results
glom(): DStream[Array[T]]	Return a new DStream in which each RDD is generated by applying glom() to each RDD of this DStream. Applying glom() to an RDD coalesces all elements within each partition into an array.
union(that: Dstream[T]): DStream[T]	Return a new DStream by unifying data of another DStream with this DStream.
repartition(numPartitions: Int): DStream[T]	Return a new DStream with an increased or decreased level of parallelism. Each RDD in the returned DStream has exactly numPartitions partitions.
count(): DStream[Long]	Return a new DStream in which each RDD has a single element generated by counting each RDD of this DStream.
countByValue(numPartitions: Int = ssc.sc.defaultParallelism)(implicit ord: Ordering[T] = null): DStream[(T, Long)]	Return a new DStream in which each RDD contains the counts of each distinct value in each RDD of this DStream. Hash partitioning is used to generate the RDDs with `numPartitions` partitions (Spark's default number of partitions if `numPartitions` not specified).
countByValueAndWindow(windowDuration: Duration, slideDuration: Duration, numPartitions: Int = ssc.sc.defaultParallelism)(implicit ord: Ordering[T] = null): DStream[(T, Long)]	Return a new DStream in which each RDD contains the count of distinct elements in RDDs in a sliding window over this DStream. Hash partitioning is used to generate the RDDs with `numPartitions` partitions (Spark's default number of partitions if `numPartitions` not specified).
transform[U](transformFunc: (RDD[T], Time) => RDD[U])(implicit arg0: ClassTag[U]): DStream[U]	Return a new DStream in which each RDD is generated by applying a function on each RDD of 'this' DStream.
reduce(reduceFunc: (T, T) => T): DStream[T]	Return a new DStream in which each RDD has a single element generated by reducing each RDD of this DStream.
foreachRDD(foreachFunc: (RDD[T]) => Unit): Unit	Apply a function to each RDD in this DStream. This is an output operator, so 'this' DStream will be registered as an output stream and therefore materialized.
countByWindow(windowDuration: Duration, slideDuration: Duration): DStream[Long]	Return a new DStream in which each RDD has a single element generated by counting the number of elements in a sliding window over this DStream. Hash partitioning is used to generate the RDDs with Spark's default number of partitions
reduceByWindow(reduceFunc: (T, T) => T, windowDuration: Duration, slideDuration: Duration): DStream[T]	Return a new DStream in which each RDD has a single element generated by reducing all elements in a sliding window over this DStream.
window(windowDuration: Duration, slideDuration: Duration): DStream[T]	Return a new DStream in which each RDD contains all the elements in seen in a sliding window of time over this DStream.

Transformation for DStream[T], T= (K,V)	Meaning
combineByKey[C](createCombiner: (V) => C, mergeValue: (C, V) => C, mergeCombiner: (C, C) => C, partitioner: Partitioner, mapSideCombine: Boolean = true): DStream[(K, C)]	Combine elements of each key in DStream's RDDs using custom functions. This is similar to the combineByKey for RDDs. Please refer to combineByKey in org.apache.spark.rdd.PairRDDFunctions in the Spark core documentation for more information.
reduceByKey(reduceFunc: (V, V) => V): DStream[(K, V)]	Return a new DStream by applying `reduceByKey` to each RDD. The values for each key are merged using the associative and commutative reduce function. Hash partitioning is used to generate the RDDs with Spark's default number of partitions.
mapValues[U](mapValuesFunc: (V) => U): DStream[(K, U)]	Return a new DStream by applying a map function to the value of each key-value pairs in 'this' DStream without changing the key
flatMapValues[U](flatMapValuesFunc: (V) => TraversableOnce[U]): DStream[(K, U)]	Return a new DStream by applying a flatmap function to the value of each key-value pairs in 'this' DStream without changing the key.
join[W](other: DStream[(K, W)]): DStream[(K, (V, W))]	Return a new DStream by applying 'join' between RDDs of `this` DStream and `other` DStream. Hash partitioning is used to generate the RDDs with Spark's default number of partitions.
leftOuterJoin[W](other: DStream[(K, W)]): DStream[(K, (V, Option[W]))]	Return a new DStream by applying 'left outer join' between RDDs of `this` DStream and `other` DStream.
fullOuterJoin[W](other: DStream[(K, W)]): DStream[(K, (Option[V], Option[W]))]	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 Spark's default number of partitions.
rightOuterJoin[W](other: DStream[(K, W)]): DStream[(K, (Option[V], W))]	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 Spark's default number of partitions.
cogroup[W](other: DStream[(K, W)])(implicit arg0: ClassTag[W]): DStream[(K, (Iterable[V], Iterable[W]))]	Return a new DStream by applying 'cogroup' between RDDs of `this` DStream and `other` DStream. Hash partitioning is used to generate the RDDs with Spark's default number of partitions.
groupByKey(): DStream[(K, Iterable[V])]	Return a new DStream by applying `groupByKey` to each RDD. Hash partitioning is used to generate the RDDs with Spark's default number of partitions.
groupByKeyAndWindow(windowDuration: Duration): DStream[(K, Iterable[V])]	Return a new DStream by applying `groupByKey` over a sliding window. This is similar to `DStream.groupByKey()` but applies it over a sliding window. The new DStream generates RDDs with the same interval as this DStream. Hash partitioning is used to generate the RDDs with Spark's default number of partitions.
reduceByKeyAndWindow(reduceFunc: (V, V) => V, windowDuration: Duration): DStream[(K, V)]	Return a new DStream by applying `reduceByKey` over a sliding window on `this` DStream. Similar to `DStream.reduceByKey()`, but applies it over a sliding window. The new DStream generates RDDs with the same interval as this DStream. Hash partitioning is used to generate the RDDs with Spark's default number of partitions.

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)]	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 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 reduceByKeyAndWindow without "inverse reduce" function. However, it is applicable to only "invertible reduce functions". Hash partitioning is used to generate the RDDs with Spark's default number of partitions.
updateStateByKey[S](updateFunc: (Seq[V], Option[S]) => Option[S]):DStream[(K,S)]	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 each key. In every batch the updateFunc will be called for each state even if there are no new values. Hash partitioning is used to generate the RDDs with Spark's default number of partitions. If `updateFunc` function returns None, then corresponding state key-value pair will be eliminated.

Output Operation for DStream[T]	Meaning
foreachRDD(foreachFunc: (RDD[T]) => Unit): Unit	Apply a function to each RDD in this DStream. This is an output operator, so 'this' DStream will be registered as an output stream and therefore materialized.
print(num: Int): Unit	Print the first num elements of each RDD generated in this DStream. This is an output operator, so this DStream will be registered as an output stream and there materialized.
print(): Unit	Print the first ten elements of each RDD generated in this DStream. This is an output operator, so this DStream will be registered as an output stream and there materialized.
saveAsObjectFiles(prefix: String, suffix: String =""): Unit	Save each RDD in this DStream as a Sequence file of serialized objects. The file name at each batch interval is generated based on `prefix` and `suffix`: "prefix-TIME_IN_MS.suffix".
saveAsTextFiles(prefix: String, suffix: String = ""): Unit	Save each RDD in this DStream as at text file, using string representation of elements.
saveAsNewAPIHadoopFiles[F <: OutputFormat[K, V]](prefix: String, suffix: String): Unit	Save each RDD in `this` DStream as a Hadoop file. The file name at each batch interval is generated based on `prefix` and `suffix`: "prefix-TIME_IN_MS.suffix".