7

Stream Processing and Messaging Using Spark

Never confuse motion with action.

Benjamin Franklin, American founding father

In This Chapter:

- Introduction to Spark Streaming, the `StreamingContext`, and DStreams
- Operations on DStreams
- Sliding window and state operations on DStreams
- Introduction to Structured Streaming in Spark
- Spark with Apache Kafka
- Spark Streaming with Amazon Kinesis Streams

Real-time event processing has become a defining feature of Big Data systems. From sensors and network data processing to fraud detection to website monitoring and much more, the capability to consume, process, and derive insights from streaming data sources has never been more relevant. To this point in the book, the processing covered for the Spark core API and with Spark SQL has been batch oriented. This chapter focuses on stream processing and another key extension to Spark: Spark Streaming.

Introducing Spark Streaming

Event processing, also called *stream processing*, is a key component of Big Data platforms. The Spark project includes a subproject that enables low latency processing with fault tolerance and data guarantees: *Spark Streaming*.

Spark Streaming delivers an event-processing system integrated with its RDD-based batch framework, and it delivers a guarantee that each event will processed exactly once, even if a node failure or similar fault occurs.

The design goals for Spark Streaming include the following:

- Low (second-scale) latency
- One-time (and only one-time) event processing
- Linear scalability
- Integration with the Spark core and DataFrame APIs

Perhaps the biggest advantage of Spark Streaming (and its overriding design goal) is that it provides a unified programming model for both stream and batch operations.

Spark Streaming Architecture

Spark Streaming introduces the concept of discretized streams, or *DStreams*. DStreams are essentially batches of data stored in multiple RDDs, each batch representing a time window, typically in seconds. The resultant RDDs can then be processed using the core Spark RDD API and all the available transformations discussed so far in this book. (The section “Introduction to DStreams,” later in this chapter, discusses DStreams in more detail.) Figure 7.1 shows a high-level overview of Spark Streaming.

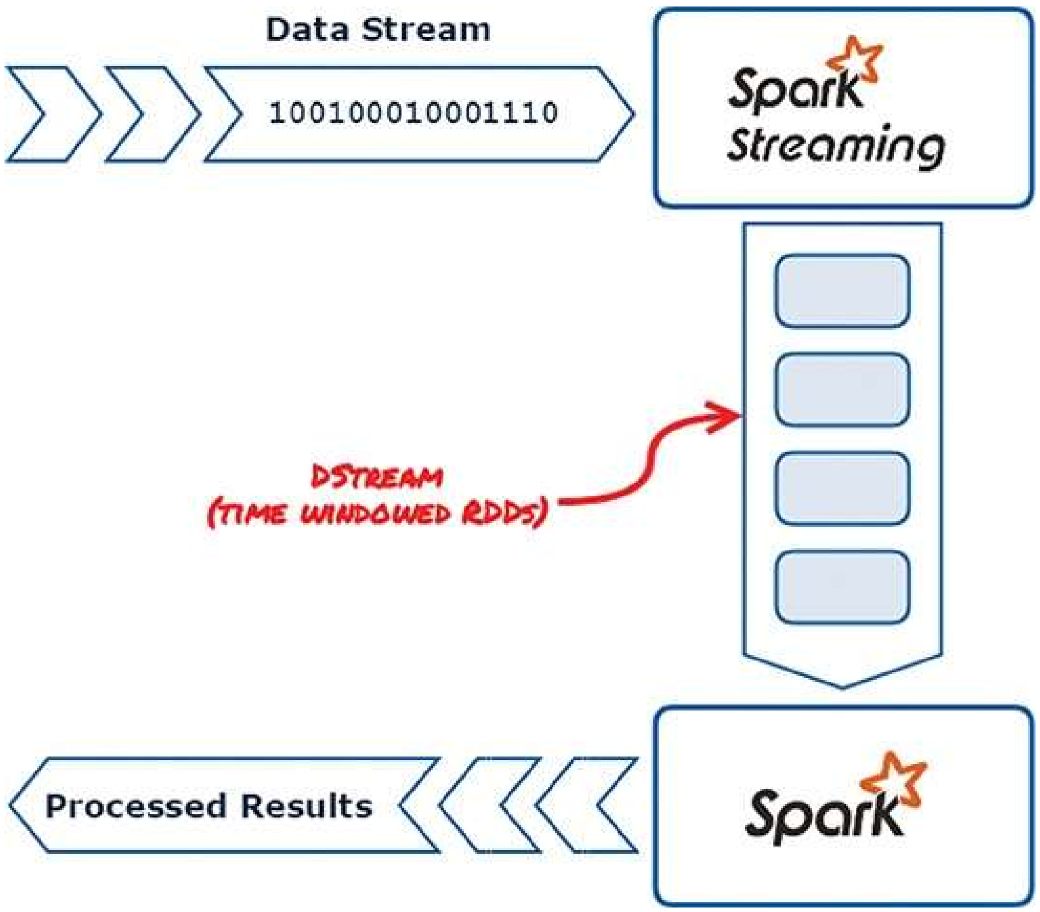


Figure 7.1 High-level overview of Spark Streaming.

As with the `SparkContext` and `SparkSession` program entry points discussed earlier in this book, Spark Streaming applications have an entry point called the `StreamingContext`. The `StreamingContext` represents a connection to a Spark platform or cluster using an existing `SparkContext`. You can use the `StreamingContext` to create DStreams from streaming input sources and govern streaming computation and DStream transformations.

The `StreamingContext` also specifies the `batchDuration` argument, which is a time interval, in seconds, by which streaming data is split into batches. After instantiating a `StreamingContext`, you create a connection to a data stream and define a series of transformations to be performed. You can use the `start()` method (or `ssc.start()`) to trigger evaluation of the incoming data after establishing a `StreamingContext`. You can stop the `StreamingContext` programmatically by using `ssc.stop()` or `ssc.awaitTermination()`, as shown in Listing 7.1.

Listing 7.1 **Creating a `StreamingContext`**

Click here to view code image

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 1)
...
# Initialize Data Stream
# DStream transformations
...
ssc.start()
...
# ssc.stop() or ssc.awaitTermination()
```

Note that just as `sc` and `sqlContext` are common conventions for object instantiations of the `SparkContext` and `SQLContext` or `HiveContext` classes, respectively, `ssc` is a common convention for an instance of the `StreamingContext`. Unlike the former entry points, however, the `StreamingContext` is not automatically instantiated in the interactive shells `pyspark` and `spark-shell`.

Introduction to DStreams

Discretized streams (DStreams) are the basic programming object in the Spark Streaming API. A DStream represents a continuous sequence of RDDs created from a continuous stream of data, with each underlying RDD representing a time window within the stream.

DStreams are created from streaming data sources such as TCP sockets, messaging systems, streaming APIs (such as the Twitter streaming API), and more. As an RDD abstraction, DStreams are also created from transformations performed on existing DStreams, such as `map()`, `flatMap()`, and other operations.

DStreams support two types of operations:

- Transformations
- Output operations

Output operations are analogous to RDD actions. DStreams execute lazily upon the request of an output operation, which is similar to lazy evaluation with Spark RDDs.

Figure 7.2 represents a DStream, with each *t* interval representing a window of time specified by the `batchDuration` argument in the `StreamingContext` instantiation.

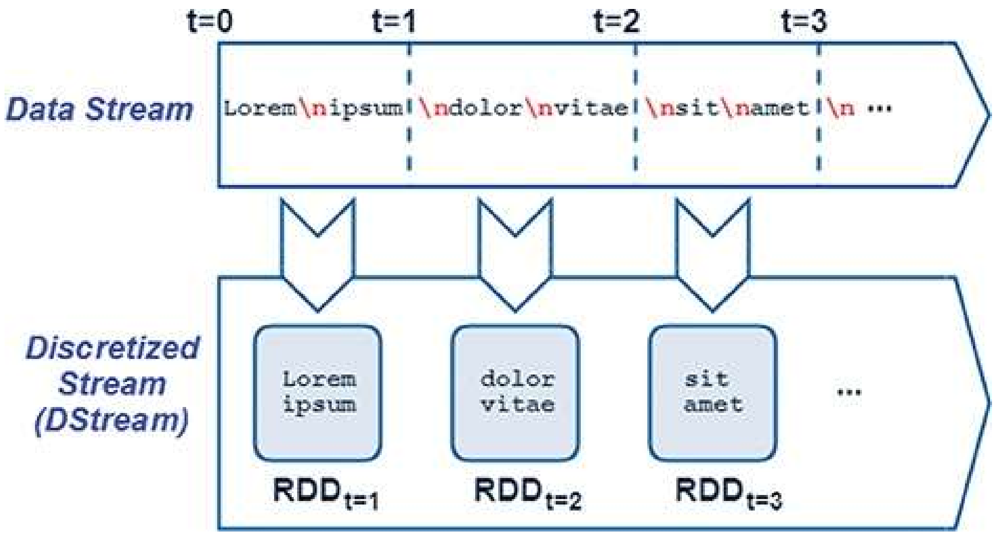


Figure 7.2 Spark discretized streams (DStreams).

DStream Sources

DStreams are defined within a `StreamingContext` for a specified input data stream, much the same way that RDDs are created for input data sources within a `SparkContext`. Many common streaming input sources are included in the Streaming API, such as sources to read data from a TCP socket or for reading data as it is written to HDFS.

The basic input data sources for creating DStreams are described in the following sections.

`socketTextStream()`

Syntax:

Click here to view code image

```
StreamingContext.socketTextStream(hostname,
        port,
        storageLevel=StorageLevel(True, True, False, False, 2))
```

Use the `socketTextStream()` method to create a DStream from an input TCP source defined by the `hostname` and `port` arguments. The data received is interpreted using UTF8 encoding, and new line termination is used to define new records. The `storageLevel` argument that defines the storage level for the DStream defaults to `MEMORY_AND_DISK_SER`. Listing 7.2 demonstrates the use of the `socketTextStream()` method.

Listing 7.2 **`socketTextStream()` Method**

Click here to view code image

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 1)
lines = ssc.socketTextStream('localhost', 9999)
counts = lines.flatMap(lambda line: line.split(" ")) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a+b)
counts.pprint() ssc.start()
ssc.awaitTermination()
```

`textFileStream()`

Syntax:

Click here to view code image

```
StreamingContext.textFileStream(directory)
```

Use the `textFileStream()` method to create a DStream by monitoring a directory from an instance of HDFS, as specified by the current system or application configuration settings. `textFileStream()` listens for the creation of new files in the directory specified by the `directory` argument and captures the data written as a streaming source. Listing 7.3 shows the use of the `textFileStream()` method.

Listing 7.3 **`textFileStream()` Method**

Click here to view code image

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 1)
```

```
lines = ssc.textFileStream('hdfs:///data/incoming/')
counts = lines.flatMap(lambda line: line.split(" ")) \
              .map(lambda x: (x, 1)) \
              .reduceByKey(lambda a, b: a+b)
counts.pprint()
ssc.start()
ssc.awaitTermination()
```

There are built-in sources for common messaging platforms such as Apache Kafka, Amazon Kinesis, Apache Flume, and more. We look at some of them shortly. You can also create custom streaming data sources by implementing a custom receiver for your desired source. At this stage, custom receivers must be written in Scala or Java.

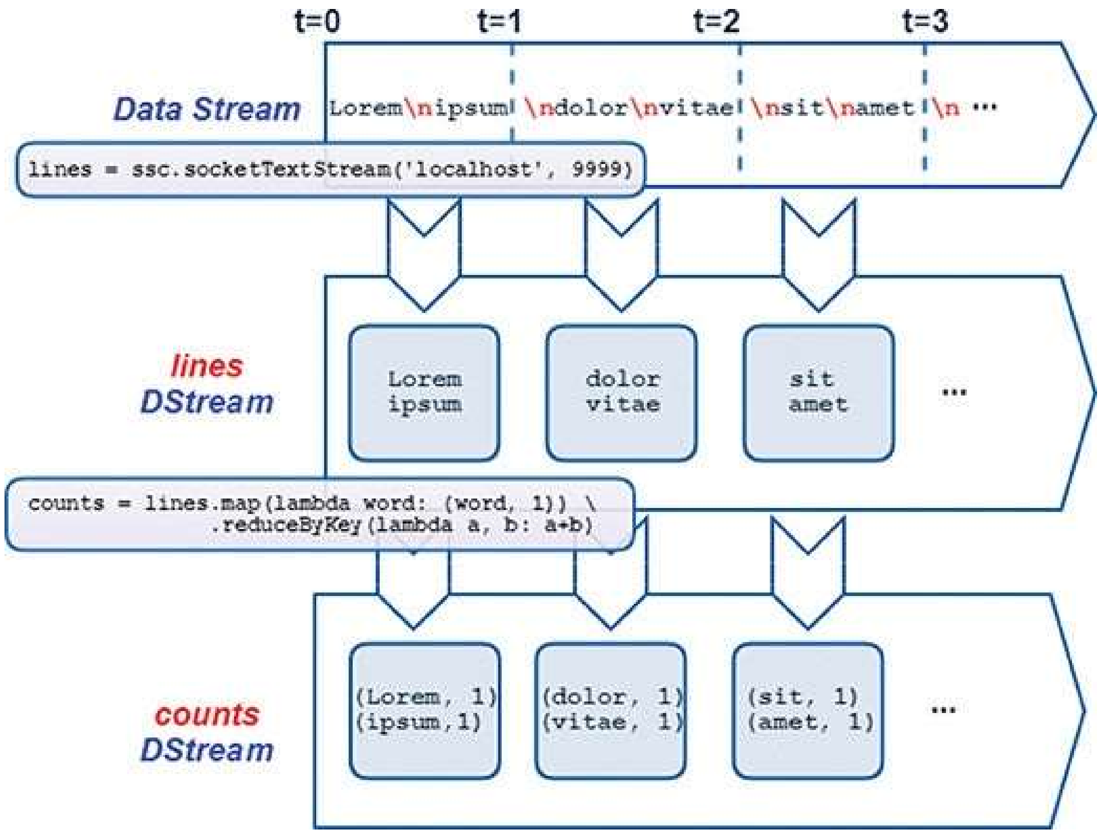
DStream Transformations

The DStream API contains many of the transformations available through the RDD API. DStream transformations, like RDD transformations, create new DStreams by applying functions to existing DStreams. Listing 7.4 and Figure 7.3 show a simplified example of DStream transformations.

Listing 7.4 **DStream Transformations**

Click here to view code image

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 30)
lines = ssc.socketTextStream('localhost', 9999)
counts = lines.map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a+b)
counts.pprint()
ssc.start()
ssc.awaitTermination()
 # output:
# ----------------------------
# Time: 2017-10-21 19:57:30
# ----------------------------
# (u'Lorem',1)
# (u'ipsum',1)
# ----------------------------
# Time: 2017-10-21 19:58:00
# ----------------------------
# (u'dolor',1)
# (u'vitae',1)
# ----------------------------
# Time: 2017-10-21 19:58:30
# ----------------------------
# (u'sit',1)
# (u'amet',1)
# ----------------------------
# Time: 2017-10-21 19:59:00
# ----------------------------
# ...
```



```
lines = ssc.socketTextStream('localhost', 9999)
```

```
counts = lines.map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a+b)
```

Figure 7.3 DStream transformations.

DStream Lineage and Checkpointing

The lineage of each DStream is maintained for fault tolerance much the same way that RDDs and DataFrames maintain their lineage. Because streaming applications are by definition long-lived applications, checkpointing is often necessary. Checkpointing with DStreams is similar to that in the RDD and DataFrame APIs. The methods are slightly different, however, and to make things confusing, the methods have the same names but are members of two separate classes. These are discussed in the following sections.

`StreamingContext.checkpoint()`

Syntax:

Click here to view code image

```
StreamingContext.checkpoint(directory)
```

The `StreamingContext.checkpoint()` method enables periodic checkpointing of DStream operations for durability and fault tolerance. The application DAG is checkpointed at each batch interval, as defined in the `StreamingContext`. The `directory` argument configures the directory, typically in HDFS, where the checkpoint data persists.

`DStream.checkpoint()`

Syntax:

```
DStream.checkpoint(interval)
```

The `DStream.checkpoint` method can enable periodic checkpointing of RDDs of a particular DStream. The `interval` argument is the time, in seconds, after which the underlying RDDs in a DStream are checkpointed.

Note that the `interval` argument must be a positive multiple of the `batchDuration` set in the `StreamingContext`.

Listing 7.5 demonstrates the use of the functions to control checkpointing behavior in Spark Streaming.

Listing 7.5 **Checkpointing in Spark Streaming**

Click here to view code image

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 30)
ssc.checkpoint('file:///opt/spark/data')
lines = ssc.socketTextStream('localhost', 9999)
```

```
counts = lines.map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a+b)
counts.checkpoint(30)
counts.pprint()
ssc.start()
ssc.awaitTermination()
```

Caching and Persistence with DStreams

DStreams support caching and persistence using interfaces with the same name and usage as their RDD counterparts, `cache()` and `persist()`. These options are especially handy for DStreams used more than once in downstream processing operations. Storage levels work the same with DStreams as they do with RDDs.

Broadcast Variables and Accumulators with Streaming Applications

Broadcast variables and accumulators are available for use in Spark Streaming applications in the same way they are implemented in native Spark applications. Broadcast variables are useful for distributing lookup or reference data associated with DStream RDD contents. You can use accumulators as counters.

There are some limitations with recovery when using broadcast variables or accumulators with checkpointing enabled. If you are developing production Spark Streaming applications and using broadcast variables or accumulators, consult the latest Spark Streaming programming guide.

DStream Output Operations

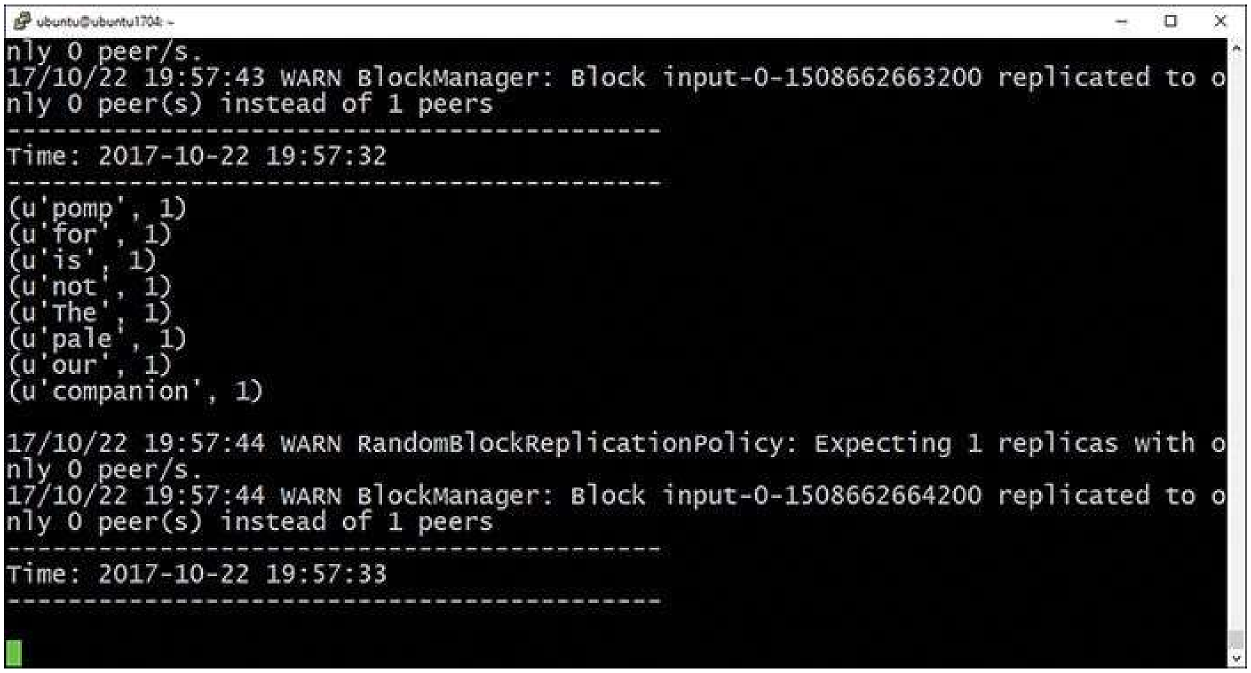
Output operations with DStreams are similar in concept to actions with RDDs. DStream output operations write data, results, events, or other data to a console, a filesystem, a database, or another destination, such as a messaging platform like Kafka. The basic DStream output operations are described in the following sections.

`pprint()`

Syntax:

```
DStream.pprint(num=10)
```

The `pprint()` method prints the first number of elements specified by the `num` argument for each RDD in the DStream (where `num` is `10` by default). Using `pprint()` is a common way to get interactive console feedback from a streaming application. Figure 7.4 shows the console output from a `pprint()` operation.



```
nly 0 peer/s.
17/10/22 19:57:43 WARN BlockManager: Block input-0-1508662663200 replicated to o
nly 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2017-10-22 19:57:32
-------------------------------------------
(u'pomp', 1)
(u'for', 1)
(u'is', 1)
(u'not', 1)
(u'The', 1)
(u'pale', 1)
(u'our', 1)
(u'companion', 1)

17/10/22 19:57:44 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with o
nly 0 peer/s.
17/10/22 19:57:44 WARN BlockManager: Block input-0-1508662664200 replicated to o
nly 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2017-10-22 19:57:33
-------------------------------------------
```

Figure 7.4 `pprint()` DStream console output.

`saveAsTextFiles()`

Syntax:

Click here to view code image

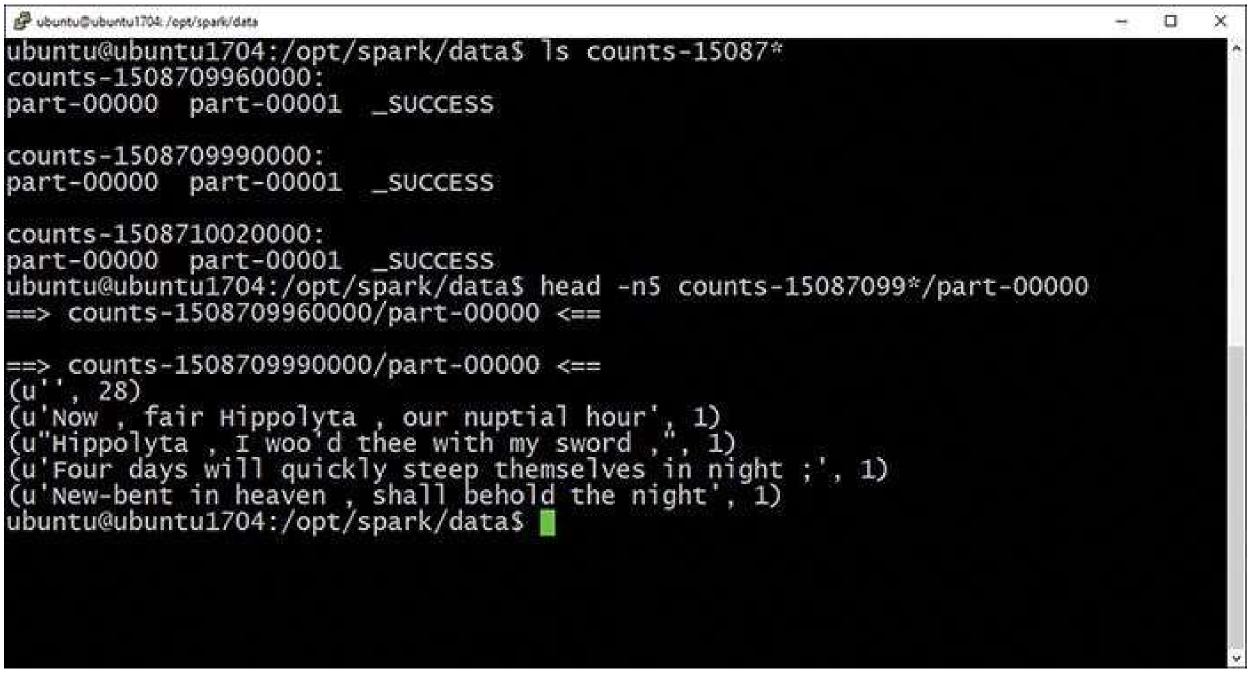
```
DStream.saveAsTextFiles(prefix, suffix-=None)
```

The `saveAsTextFiles()` method saves each RDD in a DStream as a text file in a target filesystem, local HDFS, or other filesystem. A directory of files is created with string representations of the elements contained in the DStream. Listing 7.6 shows the use of the `saveAsTextFiles()` method and the output directory created. Figure 7.5 provides a look at the file contents.

Listing 7.6 **Saving DStream Output to Files**

Click here to view code image

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 30)
lines = ssc.socketTextStream('localhost', 9999)
counts = lines.map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a+b)
counts.saveAsTextFiles("file:///opt/spark/data/counts")
ssc.start()
ssc.awaitTermination()
```



```
ubuntu@ubuntu1704:/opt/spark/data$ ls counts-15087*
counts-1508709960000:
part-00000  part-00001  _SUCCESS

counts-1508709990000:
part-00000  part-00001  _SUCCESS

counts-1508710020000:
part-00000  part-00001  _SUCCESS
ubuntu@ubuntu1704:/opt/spark/data$ head -n5 counts-15087099*/part-00000
==> counts-1508709960000/part-00000 <==

==> counts-1508709990000/part-00000 <==
(u'', 28)
(u'Now , fair Hippolyta , our nuptial hour', 1)
(u"Hippolyta , I woo'd thee with my sword ,", 1)
(u'Four days will quickly steep themselves in night ;', 1)
(u'New-bent in heaven , shall behold the night', 1)
ubuntu@ubuntu1704:/opt/spark/data$
```

Figure 7.5 Output from the `saveAsTextFiles()` DStream method.

`foreachRDD()`

Syntax:

```
DStream.foreachRDD(func)
```

The `foreachRDD()` output operation is similar to the `foreach()` action in the Spark RDD API. It applies the function specified by the `func` argument to each RDD in a DStream. The `foreachRDD()` method is executed by the Driver process running the streaming application and usually forces the computation of the DStream RDDs. The function used can be a named one or an anonymous `lambda` function, as with `foreach()`. Listing 7.7 shows a simple example of the `foreachRDD()` method.

Listing 7.7 **Performing Functions on Each RDD in a DStream**

Click here to view code image

```
from pyspark.streaming import StreamingContext
def printx(x): print("received : " + x)
ssc = StreamingContext(sc, 30)
lines = ssc.socketTextStream('localhost', 9999)
lines.foreachRDD(lambda x: x.foreach(lambda y: printx(y)))
ssc.start()
ssc.awaitTermination()
# output:
# received : Lorem
# received : ipsum
# received : dolor
# received : vitae
# received : sit
# received : amet
```

Exercise: Getting Started with Spark Streaming

This exercise shows how to stream lines from a Shakespeare text and consume the lines with the Spark Streaming application. It also shows how to perform a streaming word count against the incoming data, much like word count examples shown earlier in this book. Follow these steps:

1. Use `wget` or `curl` to download the `shakespeare.txt` file from https://s3.amazonaws.com/sparkusingpython/shakespeare/shakespeare.txt to a local directory such as `/opt/spark/data`.

2. Open a `pyspark` shell. Note that if you're using Local mode, you need to specify at least two worker threads, as shown here:

3. Enter the following commands, line by line, in the `pyspark` shell:

Click here to view code image

```
    import re
    from pyspark.streaming import StreamingContext
```

```
ssc = StreamingContext(sc, 30)
lines = ssc.socketTextStream('localhost', 9999)
wordcounts = lines.filter(lambda line: len(line) > 0) \
              .flatMap(lambda line: re.split('\W+', line)) \
              .filter(lambda word: len(word) > 0) \
              .map(lambda word: (word.lower(), 1)) \
              .reduceByKey(lambda x, y: x + y) wordcounts.pprint()
ssc.start()
ssc.awaitTermination()
```

Note that until you start a stream on the defined socket, you see exceptions appear in the console output. This is normal.

4. In another terminal, using the directory containing the local `shakespeare.txt` file from step 1 as the current directory, execute the following command:

Click here to view code image

```
$ while read line; do echo -e "$line\n"; sleep 1; done \
 < shakespeare.txt | nc -lk 9999
```

This command reads a line from the `shakespeare.txt` file every second and sends it to the `netcat` server.

You should see that every 30 seconds (the batchInterval set on the StreamingContext in step 3), the lines received from the latest batch are transformed into key/value pairs and counted, with output to the console similar to the output shown here:

Click here to view code image

```
-----------------------------------------
Time: 2017-10-21 20:10:00
-----------------------------------------
(u'and', 11)
(u'laugh', 1)
(u 'old', 1)
(u'have', 1)
(u'trifles', 1)
(u'imitate', 1)
(u'neptune', 1)
(u'is', 2)
(u'crown', 1)
(u'changeling', 1)
...
```

Find the complete source code for this exercise in the `streaming-wordcount` folder at https://github.com/sparktraining/spark_using_python.

State Operations

So far, the examples of Spark Streaming applications in this chapter have dealt with data statelessly processing each batch during a batch interval, independent of any other batches in a stream. Often you want or need to maintain state across batches of data, with the state updated as each new batch is processed. You can accomplish this by using a *state* DStream.

State DStreams are created and updated using the special `updateStateByKey()` transformation. This is preferred over using accumulators as shared variables in streaming applications because `updateStateByKey()` is automatically checkpointed for integrity, durability, and recoverability.

`updateStateByKey()`

Syntax:

Click here to view code image

```
DStream.updateStateByKey(updateFunc, numPartitions=None)
```

The `updateStateByKey()` method returns a new state DStream, where the state for each key updates by applying the function specified by the `updateFunc` argument against the previous state of the key and the new values of the key.

The `updateStateByKey()` method expects key/value pair input and returns a corresponding key/value pair output, with the values updated according to the `updateFunc` setting.

The `numPartitions` argument can repartition the output similarly to the RDD methods with this argument.

Note that checkpointing must be enabled using `ssc.checkpoint(directory)` in the `StreamingContext` before you can use the `updateStateByKey()` method and create and update state DStreams.

Consider this input stream:

Click here to view code image

```
Lorem ipsum dolor
<pause for more than 30 seconds>
Lorem ipsum dolor
```

Listing 7.8 shows how to use `updateStateByKey()` to create and update the counts for the words received on the stream.

Listing 7.8 **State DStreams**

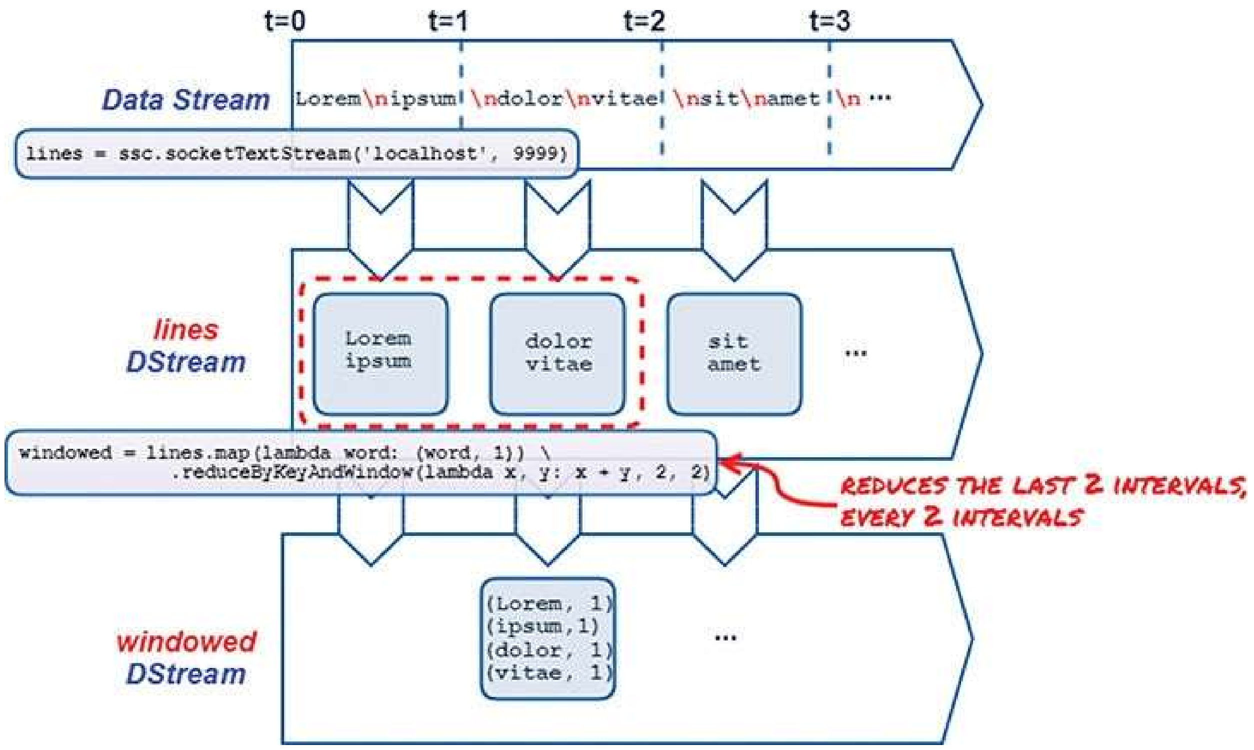
Click here to view code image

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 30)
ssc.checkpoint("checkpoint")
def updateFunc(new_values, last_sum):
return sum(new_values) + (last_sum or 0)
lines = ssc.socketTextStream('localhost', 9999)
wordcounts = lines.map(lambda word: (word, 1)) \
                  .updateStateByKey(updateFunc)
wordcounts.pprint()
ssc.start()
ssc.awaitTermination()
# output:
#...
# -------------------------------------------
# Time: 2016-03-31 00:02:30
# -------------------------------------------
# (u'Lorem', 1)
# (u'ipsum', 1)
# (u'dolor', 1)
#...
 # -------------------------------------------
# Time: 2016-03-31 00:03:00
# -------------------------------------------
# (u'Lorem', 2)
# (u'ipsum', 2)
# (u'dolor', 2)
```

Sliding Window Operations

The state operations you learned about in the previous section apply to all RDDs in the DStream. It is useful to look at aggregations over a specific window, such as the last hour or day. Because this window is relative to a point in time, it's called a *sliding* window.

Sliding window operations in Spark Streaming span RDDs within a DStream over a specified duration (the window length) and are evaluated at specific intervals (the slide interval). Consider Figure 7.6. If you want to count the words in the last two intervals (window length) every two intervals (slide interval), you can use the `reduceByKeyAndWindow()` function to create "windowed" RDDs.



```
lines = ssc.socketTextStream('localhost', 9999)
```

```
windowed = lines.map(lambda word: (word, 1)) \
            .reduceByKeyAndWindow(lambda x, y: x + y, 2, 2)
```

Figure 7.6 Sliding windows and windowed RDDs in Spark Streaming.

Sliding window functions available in the Spark Streaming API include `window()`, `countByWindow()`, `reduceByWindow()`, `reduceByKeyAndWindow()`, and `count ByValueAndWindow()`. The following sections cover a couple of these basic functions.

`window()`

Syntax:

Click here to view code image

```
DStream.window(windowLength, slideInterval)
```

The `window()` method returns a new DStream from specified batches of the input DStream. `window()` creates a new DStream object every interval, as specified by the `slideInterval` argument, consisting of elements from the input DStream for the specified `windowLength`.

Both `slideInterval` and `windowLength` must be multiples of the `batchDuration` set in the `StreamingContext`. Listing 7.9 demonstrates the use of the `window()` function.

Listing 7.9 **`window()` Function**

Click here to view code image

```
# send date to netcat every second:
# while sleep 1; do echo 'date'; done | nc -lk 9999
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 5)
dates = ssc.socketTextStream('localhost', 9999)
windowed = dates.window(10,10)
windowed.pprint()
ssc.start()
ssc.awaitTermination()
# output:
# ...
# -------------------------------------------
# Time: 2017-10-23 09:28:15
# -------------------------------------------
# Mon 23 Oct 09:28:05 AEDT 2017
# Mon 23 Oct 09:28:06 AEDT 2017
# Mon 23 Oct 09:28:07 AEDT 2017
# Mon 23 Oct 09:28:08 AEDT 2017
# Mon 23 Oct 09:28:09 AEDT 2017
# Mon 23 Oct 09:28:10 AEDT 2017
# Mon 23 Oct 09:28:11 AEDT 2017
# Mon 23 Oct 09:28:12 AEDT 2017
# Mon 23 Oct 09:28:13 AEDT 2017
#...
# -------------------------------------------
# Time: 2017-10-23 09:28:25
# -------------------------------------------
# Mon 23 Oct 09:28:14 AEDT 2017
# Mon 23 Oct 09:28:15 AEDT 2017
# Mon 23 Oct 09:28:16 AEDT 2017
# Mon 23 Oct 09:28:17 AEDT 2017
# Mon 23 Oct 09:28:18 AEDT 2017
# Mon 23 Oct 09:28:19 AEDT 2017
# Mon 23 Oct 09:28:20 AEDT 2017
# Mon 23 Oct 09:28:21 AEDT 2017 # Mon 23 Oct 09:28:22 AEDT 2017
# Mon 23 Oct 09:28:24 AEDT 2017
```

reduceByKeyAndWindow()

Syntax:

Click here to view code image

```
DStream.reduceByKeyAndWindow(func,
                             invFunc,
                             windowDuration,
                             slideDuration=None,
                             numPartitions=None,
                             filterFunc=None)
```

The `reduceByKeyAndWindow()` method creates a new DStream by performing an associative reduce function, as specified by the `func` argument, to a sliding window, as defined by the `windowDuration` and `slideDuration` arguments. The `invFunc` argument is an inverse function to the `func` argument. `invFunc` is included for efficiency to remove (or subtract) counts from the previous window; `numPartitions` is an optional argument supported for repartitioning the output DStream. The optional `filterFunc` argument can filter expired key/value pairs; in this case, only key/value pairs that satisfy the function are retained in the resultant DStream. Listing 7.10 demonstrates the use of the `reduceByKeyAndWindow()` function.

Note that checkpointing must be enabled when using the `reduceByKeyAndWindow()` function.

Listing 7.10 **The `reduceByKeyAndWindow()` Function**

Click here to view code image

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 5)
ssc.checkpoint("checkpoint")
lines = ssc.socketTextStream('localhost', 9999)
windowedWordCounts = lines.map(lambda word: (word, 1)) \
                          .reduceByKeyAndWindow(lambda x, y: x + y, \
                             lambda x, y: x - y, 30, 10)
windowedWordCounts.pprint()
ssc.start()
ssc.awaitTermination()
```

Structured Streaming

Stream processing in Spark is not limited to the RDD API; by using *Structured Streaming*, Spark Streaming is fully integrated with the Spark DataFrame API as well. Using Structured Streaming, streaming data sources are treated as an unbounded table that is continually appended to. SQL queries can run against these tables much as they are able to run from tables representing static DataFrames. Figure 7.7 shows a high-level overview of Structured Streaming with Spark.

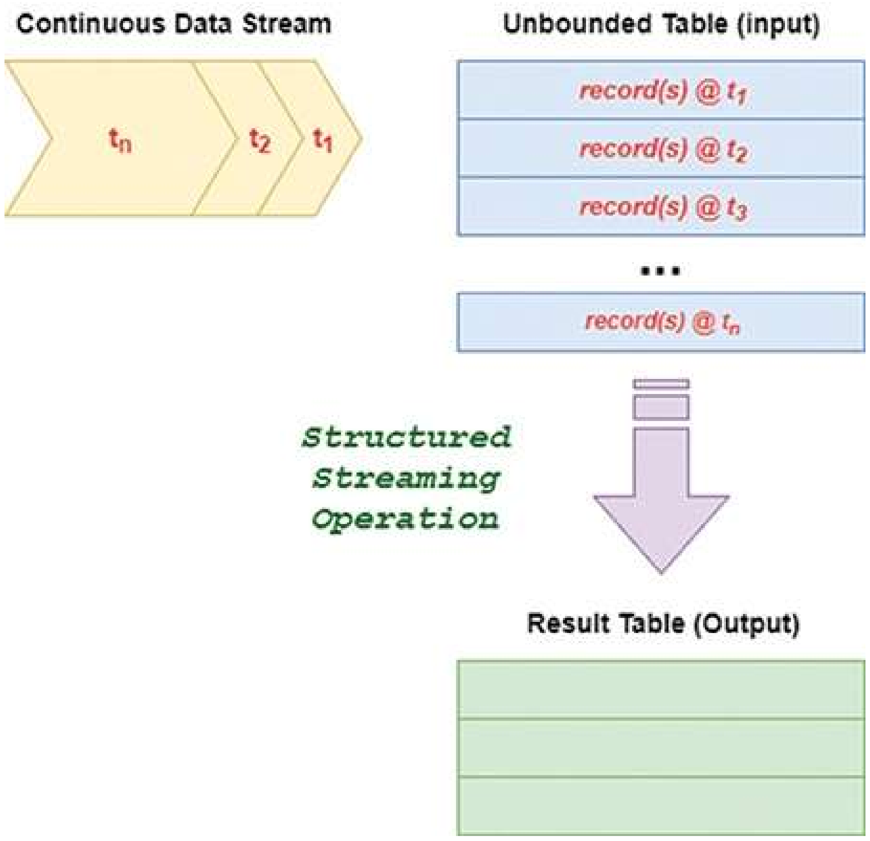


Figure 7.7 Structured Streaming.

Structured Streaming Data Sources

The `DataFrameReader` (detailed in Chapter 6) includes several built-in sources designed to ingest streaming data. These data sources include support for file, socket, and Kafka—which we discuss shortly—data streams. The `DataFrameReader.readStream()` method, available through the `SparkSession` object, includes a `format()` member used to define the streaming source.

File Sources

The file source reads new files written in a directory as a stream of data. Most of the file formats supported by the `DataFrameReader` are supported as Structured Streaming sources, including CSV, text, JSON, and Parquet ORC. Listing 7.11 demonstrates how to use a file source (CSV in this case) for a Structured Streaming application. Note that you must supply a schema unless there are existing files in the input directory from which the schema can be inferred.

Listing 7.11 **Structured Streaming Using a File Source**

Click here to view code image

```
from pyspark.sql.types import *
tripsSchema = StructType() \
        .add("TripID", "integer") \
        .add("Duration", "integer") \
        .add("StartDate", "string") \
        .add("StartStation", "string") \
        .add("StartTerminal", "integer") \
        .add("EndDate", "string") \
        .add("EndStation", "string") \ .add("EndTerminal", "integer") \
        .add("BikeNo", "integer") \
        .add("SubscriberType", "string") \
        .add("ZipCode", "string")
csv_input = spark \
    .readStream \
    .schema(tripsSchema) \
    .csv("/tmp/trips")
...
```

Socket Sources

The socket source reads text data in UTF8 format from a socket connection in much the same way as the `socketTextStream()` method in the Spark Streaming RDD API. Listing 7.12 demonstrates the use of a socket data source to perform a Structured Streaming word count.

Listing 7.12 **Structured Streaming Using a Socket Source**

Click here to view code image

```
socket_input = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()
...
```

Structured Streaming Data Sinks

As discussed earlier in this chapter, each data item arriving on a stream is treated like a new record appended to a table, referred to as the *Input Table*. The output from a Structured Streaming operation—that is, what is written out—is referred to as the *Result Table*. Output from Structured Streaming operations is written out by the `DataFrameWriter` object and, in particular, the `DataFrameWriter.writeStream()` method.

Output sinks in Spark's Structured Streaming define where the Result Table is written to. Output sinks themselves are defined using the `format()` member of the `DataFrameWriter.writeStream()` method.

There are built-in output sinks for writing data to files, memory, or the console.

File Sink

The file sink stores the Result Table in a directory in a supported filesystem (HDFS, local filesystem, S3, and so on). Listing 7.13 demonstrates the file sink.

Listing 7.13 **File Output Sink**

Click here to view code image

```
...
output.writeStream \
    .format("parquet") \
    .option("path", "/tmp/streaming_output") \
    .start()
# could also be "orc", "json", "csv"
```

To use a file output sink, you need to set a checkpoint location; this can be done as shown here:

Click here to view code image

```
spark.conf.set("spark.sql.streaming.checkpointLocation",
"/tmp/checkpoint_dir")
```

Console Sink

The console sink prints the Result Table to the console. This is useful for debugging but could be impractical for large output sets. Listing 7.14 demonstrates the console sink.

Listing 7.14 **Console Output Sink**

Click here to view code image

```
...
output.writeStream \
    .format("console") \
    .start()
```

Memory Sink

The memory stores the Result Table as a table in memory; this type of sink is also useful for debugging but should be used with caution for larger output data sets. Listing 7.15 demonstrates the memory sink.

Listing 7.15 **Memory Output Sink**

Click here to view code image

```
...
output.writeStream \
    .format("memory") \
    .queryName("trips") \
    .start()
spark.sql("select * from trips").show()
```

Output Modes

Just as output sinks define *where* to send the output of a Structured Streaming operation, output modes define *how* the output is treated. There are several different output modes:

- **append:** Outputs only new rows added to the Result Table since the last trigger. This mode, which is the default, is useful for operations that are simply projecting or filtering new data, including `where()`, `select()`, and `filter()`.

- **`complete`:** Outputs the entire Result Table, including all updates and transformations, after each trigger. This is useful for aggregate functions such as `count()`, `sum()`, and so on.
- **`update`:** Outputs only rows in the Result Table that have updated since the last trigger are output.

The output mode is specified using the `outputMode()` member of the `writeStream()` method.

Structured Streaming Operations

Because Structured Streaming builds on the DataFrame API, most DataFrame operations are available, including the following:

- Filtering records
- Projecting columns
- Performing column-level transformations using built-in or user-defined functions
- Grouping records and aggregating columns
- Joining streaming DataFrames with static DataFrames (with some limitations)

However, some operations in the DataFrame API are not available with streaming DataFrames, including the following:

- `limit` and `take(n)` operations
- `distinct` operations
- `sort` operations (supported only in `complete` output mode after an aggregation)
- Full outer join operations
- Any type of join between two streaming DataFrames

- Additional conditions on left and right outer join operations

Listing 7.16 puts together all the Structured Streaming concepts and showcases various operations that perform on streaming DataFrames.

Listing 7.16 **Structured Streaming Operations**

Click here to view code image

```
# declare a schema for a streaming source
from pyspark.sql.types import *
tripsSchema = StructType() \
        .add("TripID", "integer") \
        .add("Duration", "integer") \
        .add("StartDate", "string") \
        .add("StartStation", "string") \
        .add("StartTerminal", "integer") \
        .add("EndDate", "string") \
        .add("EndStation", "string") \
        .add("EndTerminal", "integer") \ .add("BikeNo", "integer") \
        .add("SubscriberType", "string") \
        .add("ZipCode", "string")
# read from an input stream
trips = spark \
    .readStream \
    .schema(tripsSchema) \
    .csv("/tmp/trips")
# perform a streaming DataFrame aggregation
result = trips.select(trips.StartTerminal, trips.StartStation) \
            .groupBy(trips.StartTerminal, trips.StartStation) \
            .agg({"*": "count"})
# write out the result table to the console
result.writeStream \
    .format("console") \
    .outputMode("complete") \
    .start()
# returns:
# <pyspark.sql.streaming.StreamingQuery object at 0x7fb1c5c2a0f0>
# -------------------------------------------
# Batch: 0
# -------------------------------------------
# +------------+------------------+--------+
# |StartTerminal|       StartStation|count(1)|
# +------------+------------------+--------+
# |           7|Paseo de San Antonio|     856|
# |          65|     Townsend at 7th|   13752|
# |          26|Redwood City Medi...|     150|
# |          38|       Park at Olive|     376|
# ...
```

Using Spark with Messaging Platforms

Messaging systems originally formed to provide middleware functionality—more specifically, message-oriented middleware (MOM). This area saw rapid expansion in the 1980s, in integrating legacy systems with newer systems, such as mainframe and early distributed systems. Today messaging systems and platforms provide much more functionality than just simple integration. They are a critical part of the mobile computing and Internet of Things (IoT) landscape. Projects such as JMS (Java Message Service), Kafka, ActiveMQ, ZeroMQ (ØMQ), RabbitMQ, Amazon SQS (Simple Queue Service), and Kinesis have added to the existing landscape of more established commercial solutions such as TIBCO EMS (Enterprise Message Service), IBM WebSphere MQ, and Microsoft Message Queuing (MSMQ).

The following sections look at some messaging systems commonly used with Big Data and Spark implementations.

Apache Kafka

Originally developed at LinkedIn, Apache Kafka is a popular open source project written in Scala and designed for message brokering and queuing between various Hadoop ecosystem projects.

Kafka Architecture

Kafka is a distributed, reliable, low-latency, pub-sub messaging platform. Conceptually, Kafka acts as a write-ahead log (WAL) for messages, much the same way a transaction log or journal functions in an ACID data store. This log-based design provides durability, consistency, and the capability for subscribers to replay messages.

Publishers, called *producers*, write data to topics. Subscribers, called *consumers*, read messages from specified *topics*. Figure 7.8 summarizes the relationships among producers, topics, and consumers. Messages themselves are uninterpreted byte arrays that can represent any object or primitive datatype. Common message content formats include JSON and *Avro*, an open source Hadoop ecosystem data-serialization project.

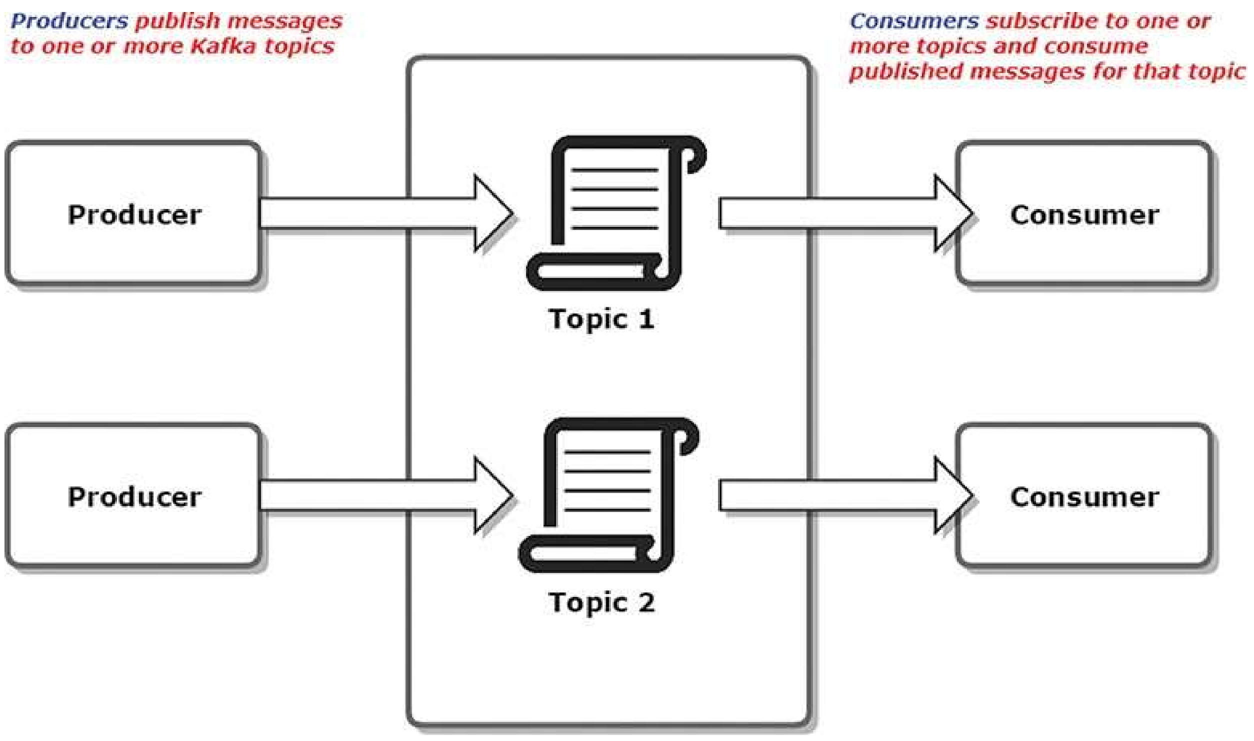

Producers publish messages to one or more Kafka topics
Consumers subscribe to one or more topics and consume published messages for that topic
Producer
Topic 1
Consumer
Producer
Topic 2
Consumer

Figure 7.8 Kafka producers, consumers, and topics.

Kafka is a distributed system that consists of one or more *brokers*, typically on separate nodes of a cluster. Brokers manage *partitions*, which are ordered, immutable sequences of messages for a particular topic. Partitions replicate across multiple nodes in a cluster to provide fault tolerance. A topic may have many partitions.

Each topic in Kafka is treated as a log—a collection of messages—with a unique offset assigned to each message. Topics are ordered within a partition. Consumers can access messages from a topic based on these offsets, which means a consumer can replay previous messages.

Kafka retains messages for only a specified period of time. After the specified retention period, messages are purged, and consumers no longer have access to these messages.

Kafka uses *Apache ZooKeeper* to maintain state between brokers. ZooKeeper is an open source distributed configuration and synchronization service used by many other Hadoop ecosystem projects, including HBase. ZooKeeper is typically implemented in a cluster configuration called an *ensemble*, and it's typically deployed in odd numbers, such as three or five.

A majority of nodes, or a *quorum* of nodes, successfully performing an action (such as updating a state) is required. A quorum must "elect" a leader, which in a Kafka cluster is the node responsible for all reads and writes for a specific partition; every partition has a leader.

Other nodes are considered followers. *Followers* consume messages and update their partitions. If the leader is unavailable, Kafka elects a new leader.

Figure 7.9 presents the Kafka cluster architecture.

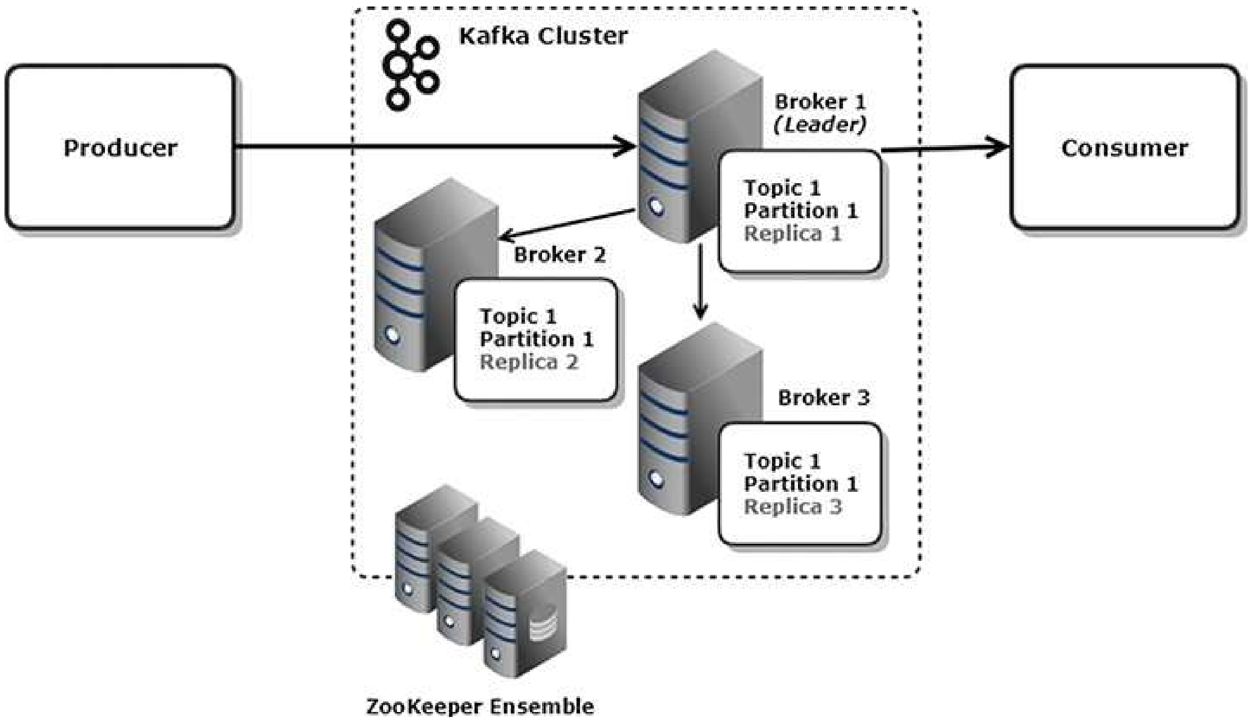


Figure 7.9 Kafka cluster architecture.

More detailed information about Kafka is available at http://kafka.apache.org/.

Using Spark with Kafka

Spark's support for Kafka closely aligns with the Spark Streaming project. Kafka's performance and durability make it a platform that's well suited to servicing Spark Streaming processes.

Common usage scenarios for Kafka and Spark include Spark Streaming processes reading data from a Kafka topic and performing event processing on the data stream or a Spark process serving as a producer and writing output to a Kafka topic.

There are two approaches to consuming messages from a Kafka topic using Spark:

- Using receivers
- Accessing messages directly from a broker (referred to as *Direct Stream Access*)

Receivers are processes that run within Spark Executors. Each receiver is responsible for an input DStream created from messages from a Kafka topic. Receivers query the ZooKeeper quorum for information about brokers, topics, partitions, and offsets. In addition, receivers implement a separate WAL—typically stored in HDFS—for durability and consistency. Messages and offsets are committed to the WAL, and then receipt of the message is acknowledged with an update of the consumed offset in ZooKeeper. This ensures that messages process once and only once across multiple receivers, if this guarantee is required.

The WAL implementation ensures durability and crash consistency in the event of receiver failure. Figure 7.10 summarizes the operation of Spark Streaming Kafka receivers.

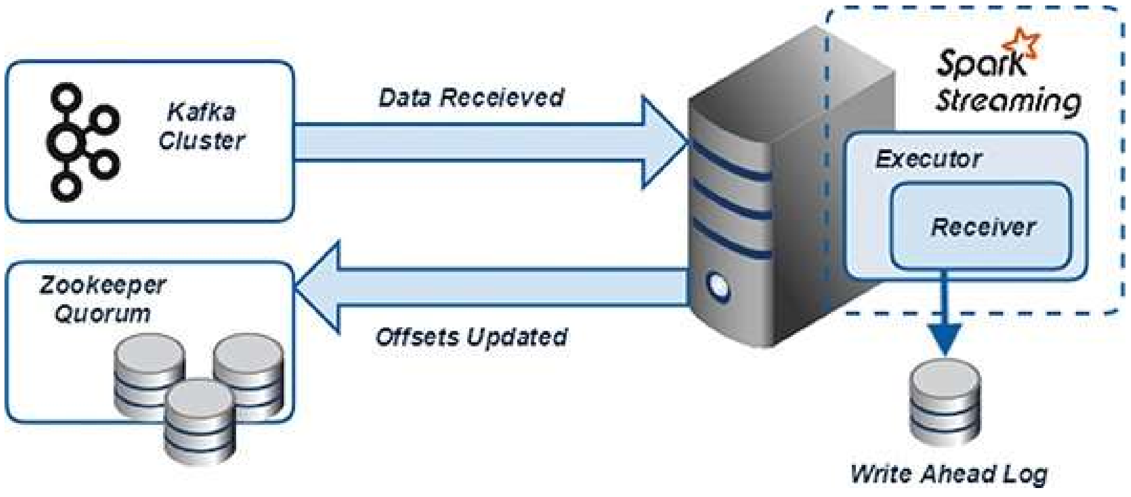


Figure 7.10 Spark Streaming Kafka receivers.

Although the receiver method for reading messages from Kafka provides durability and once- and-only-once processing, the blocking WAL write operations impact performance. A newer, alternative approach to stream consumption from Kafka is the direct approach. The direct approach does not use receivers or WAL. Instead, the Spark Driver queries Kafka for updates to offsets for each topic and directs application Executors to consume specified offsets in topic partitions directly from Kafka brokers.

The direct approach uses the `SimpleConsumer` Kafka API as opposed to the high-level `ConsumerConnector` API that is used with the receiver approach. The direct method provides durability and recoverability, and it enables "once-and-only-once" (transactional) processing semantics equivalent to the receiver approach without the WAL overhead. Figure 7.11 summarizes the operation of the Spark Streaming Kafka Direct API.

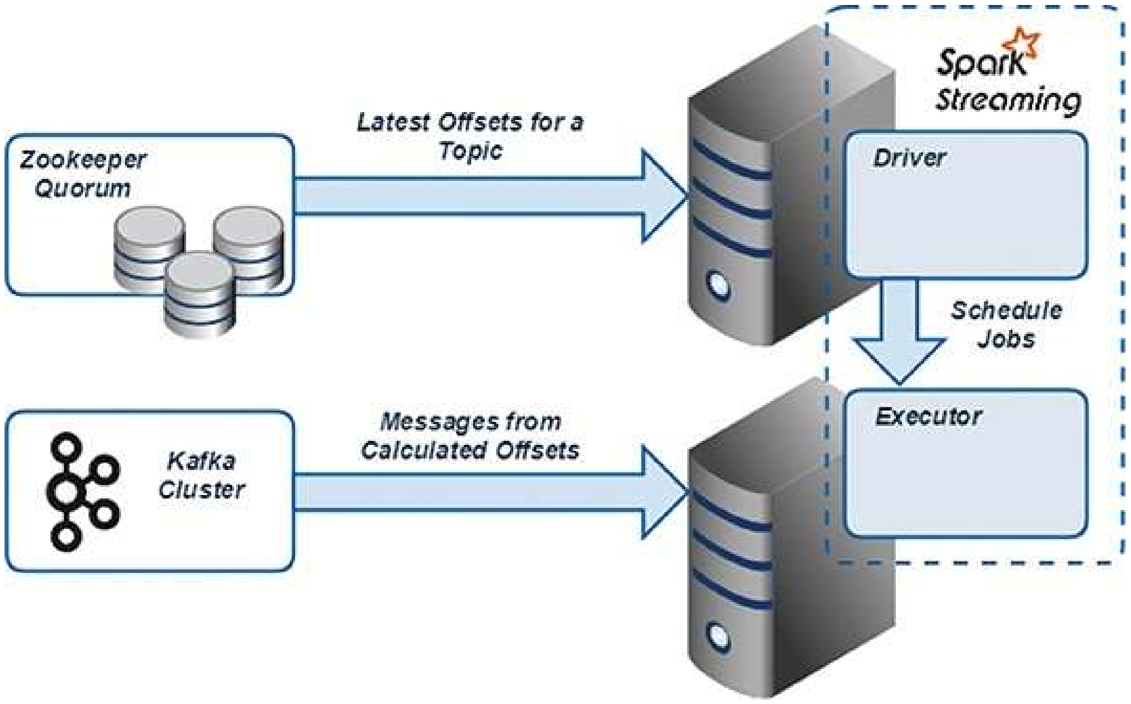


Figure 7.11 Spark Streaming Kafka Direct API.

`KafkaUtils`

In both the receiver and direct approaches to stream acquisition from a Kafka topic, you can use the `KafkaUtils` package with the Scala, Java, or Python

API. First, you need to download or compile the `spark-streaming-kafka-assembly.jar` file; alternatively, you can use a ready-made Spark package. Listing 7.17 shows an example of starting a `pyspark` session, including the `spark-streaming-kafka` package. The same process applies for `spark-shell` or `spark-submit`.

Listing 7.17 **Using Spark `KafkaUtils`**

Click here to view code image

```
$SPARK_HOME/bin/pyspark \
       --packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.2.0
```

With the `spark-streaming-kafka-assembly.jar` file or Spark package included in a Spark session and a `StreamingContext` available, you can access methods from the `KafkaUtils` class, including methods to create a stream using the receiver approach or direct approach. The following sections describe these methods.

`createDirectStream()`

Syntax:

Click here to view code image

```
KafkaUtils.createDirectStream(ssc,
                              topics,
                              kafkaParams,
                              fromOffsets=None,
                              keyDecoder=utf8_decoder,
                              valueDecoder=utf8_decoder,
                              messageHandler=None)
```

Use the `createDirectStream()` method to create a Spark Streaming DStream object from a Kafka topic or topics. The DStream consists of key/value pairs, where the key is the message key, and the value is the message itself. The `ssc` argument is a `StreamingContext` object. The `topics` argument is a list of one or more Kafka topics to consume. The `kafkaParams` argument passes additional parameters to Kafka, such as a list of Kafka brokers to communicate with. The `fromOffsets` argument specifies the reading start point for the stream. If it is not supplied, the stream is consumed from either the smallest or largest offset available in Kafka (controlled by the `auto.offset.reset` setting in the `kafkaParams` argument). The optional `keyDecoder` and `valueDecoder` arguments decode message key and value objects, defaulting to doing so using UTF8. The `messageHandler` argument is an optional argument for supplying a function to access message metadata. Listing 7.18 demonstrates the use of the `createDirectStream()` method.

Listing 7.18 `KafkaUtils.createDirectStream()` Method

Click here to view code image

```
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
ssc = StreamingContext(sc, 30)
stream = KafkaUtils.createDirectStream \
   (ssc, ["my_kafka_topic"], {"metadata.broker.list": "localhost:9092"})
```

There is a similar direct method in the `KafkaUtils` package, `createRDD()`, which is designed for batch access from a Kafka buffer; with it, you specify start and end offsets for a topic and partition.

`createStream()`

Syntax:

Click here to view code image

```
KafkaUtils.createStream(ssc,
                        zkQuorum,
                        groupId,
                        topics,
                        kafkaParams=None,
                        storageLevel=StorageLevel(True, True, False,
False, 2),
                        keyDecoder=utf8_decoder,
                        valueDecoder=utf8_decoder)
```

The `createStream()` method creates a Spark Streaming DStream object from a Kafka topic or topics using a high-level Kafka consumer API and receiver, including a WAL. The `ssc` argument is a `StreamingContext` object instantiation. The `zkQuorum` argument specifies a list of ZooKeeper nodes for the receiver to interact with. The `groupId` argument specifies the group ID for the consumer. The `topics` argument is a dictionary consisting of the topic name to consume and the number of partitions to create; each partition is consumed using a separate thread. The `kafkaParams` argument specifies additional parameters to pass to Kafka. The `storageLevel` argument is the storage level to use for the WAL; the default is `MEMORY_AND_DISK_SER_2`. The `keyDecoder` and `valueDecoder` arguments specify functions to decode message keys and values, respectively. Both default to the `utf8_decoder` function. Listing 7.19 demonstrates the use of the `createStream()` method.

Listing 7.19 **`KafkaUtils.createStream()`** **(Receiver) Method**

Click here to view code image

```
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
ssc = StreamingContext(sc, 1)
stream = KafkaUtils.createStream(ssc, \
          "localhost:2181", \
          "spark-streaming-consumer", \
          {"mykafkatopic": 1})
```

Exercise: Using Spark with Kafka

This exercise shows how to install a single-node Kafka system. You can use this platform to create messages through a producer and consume these messages as a DStream in a Spark Streaming application. For this exercise requires you need to have ZooKeeper installed, and it shows you how to do it. ZooKeeper is a requirement for installing HBase, so if you have an installation of HBase, you can use it. More information about ZooKeeper is available at https://zookeeper.apache.org/.

1. Download the latest release of Apache ZooKeeper (in this case, release 3.4.10) from https://zookeeper.apache.org/releases.html.
2. Unpack the ZooKeeper release:

   ```
   $ tar -xvf zookeeper-3.4.10.tar.gz
   ```

3. Change directories into your unpacked ZooKeeper release directory:

```
$ cd zookeeper-3.4.10
```

4. Create a simple ZooKeeper config file (`zoo.cfg`) in the ZooKeeper configuration directory, using a text editor such as `vi`:

```
$ vi conf/zoo.cfg
```

5. Add the following configuration to the `zoo.cfg` file:

Click here to view code image

```
tickTime=2000
dataDir=/tmp/zookeeper
clientPort=2181
```

   Save the file and then exit the text editor

6. Start the ZooKeeper Server service:

```
$ bin/zkServer.sh start
```

7. Download the latest Kafka release (in this case, Kafka release 1.0.0) from http://kafka.apache.org/downloads.html.

8. Unpack the `tar.gz` archive and create a Kafka home:

Click here to view code image

```
$ tar -xvf kafka_2.11-1.0.0.tgz
$ sudo mv kafka_2.11-1.0.0/ /opt/kafka/
$ export KAFKA_HOME=/opt/kafka
```

9. Start the Kafka server:

Click here to view code image

```
$KAFKA_HOME/bin/kafka-server-start.sh \
$KAFKA_HOME/config/server.properties
```

   You need to open several terminals concurrently for this exercise; refer to this terminal as terminal 1.

10. Open a second terminal (terminal 2) and create a test topic named `mykafkatopic`:

Click here to view code image

```
export KAFKA_HOME=/opt/kafka
$KAFKA_HOME/bin/kafka-topics.sh \
--create \
--zookeeper localhost:2181 \
--replication-factor 1 \
--partitions 1 \
--topic mykafkatopic
```

11. In terminal 2, list the available topics, and you should see the topic you just created:

Click here to view code image

```
$KAFKA_HOME/bin/kafka-topics.sh \
--list \
--zookeeper localhost:2181
```

12. In terminal 2, create a consumer process to read from your Kafka topic:

Click here to view code image

```
$KAFKA_HOME/bin/kafka-console-consumer.sh \
--bootstrap-server localhost:9092 \
--topic mykafkatopic \
--from-beginning
```

13. Open a new terminal (terminal 3) and use this terminal to start a new producer process to write to your Kafka topic:

Click here to view code image

```
export KAFKA_HOME=/opt/kafka
$KAFKA_HOME/bin/kafka-console-producer.sh \
--broker-list localhost:9092 \
--topic mykafkatopic
```

14. Enter messages, such as `this is a test message`, in your producer in terminal 3; you should see these messages appear in your consumer in terminal 2.
15. Use Ctrl+C to close the consumer process running in terminal 2 and the producer process running in terminal 3.
16. Using terminal 2, create a new topic named `shakespeare`:

Click here to view code image

```
$KAFKA_HOME/bin/kafka-topics.sh \
--create \
--zookeeper localhost:2181 \
--replication-factor 1 \
--partitions 1 \
--topic shakespeare
```

17. In terminal 2, open a `pyspark` session using the Spark Streaming assembly package (which is required for Kafka support):

Click here to view code image

```
$SPARK_HOME/bin/pyspark --master local[2] \
--jars spark-streaming-kafka-0-10-assembly_2.11-2.2.0.jar
```

This example uses Local mode; however, you can use a Standalone or YARN Cluster mode (if you have a YARN cluster available).

18. In the `pyspark` session, enter the following statements:

Click here to view code image

```
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
ssc = StreamingContext(sc, 30)
brokers = "localhost:9092"
topic = "shakespeare"
stream = KafkaUtils.createDirectStream \
(ssc, [topic], {"metadata.broker.list": brokers})
lines = stream.map(lambda x: x[1])
counts = lines.flatMap(lambda line: line.split(" ")) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a+b)
counts.pprint()
ssc.start()
ssc.awaitTermination()
```

19. In terminal 3, run the following command to send the contents of the `shakespeare.txt` file to your Kafka topic:

Click here to view code image

```
while read line; do echo -e "$line\n"; sleep 1; done \
< /opt/spark/data/shakespeare.txt \
| $KAFKA_HOME/bin/kafka-console-producer.sh \
--broker-list localhost:9092 \
--topic shakespeare
```

20. Check terminal 2; you should see results similar to the following:

Click here to view code image

```
-------------------------------------------
Time: 2017-11-03 05:24:30
-------------------------------------------
('', 37)
('step', 1)
('bring', 1)
('days', 2)
```

```
('quickly', 2)
('four', 1)
('but', 1)
('pomp', 2)
('thy', 1)
('dowager', 1)
...
```

The complete source code for this exercise is in the `kafka-streaming-wordcount` folder at https://github.com/sparktraining/spark_using_python.

Amazon Kinesis

Amazon Kinesis from Amazon Web Services is a fully managed distributed messaging platform inspired by, or at least very similar to, Apache Kafka. Kinesis is the AWS next-generation message queue service, and it introduces real-time stream processing at scale as an additional messaging alternative to the company’s original SQS offering.

The AWS Kinesis product family includes Amazon Kinesis Analytics, which enables SQL queries against streaming data, and Amazon Kinesis Firehose, which provides the capability to capture and load streaming data directly into Amazon S3, Amazon Redshift (an AWS cloud-based data warehouse platform), and other services. The Kinesis component we discuss here is Amazon Kinesis Streams.

Kinesis Streams

A Kinesis Streams application involves producers and consumers in the same way as the other messaging platforms already discussed in this chapter. Producers and consumers can be mobile applications or other systems within AWS (or note).

A Kinesis stream is an ordered sequence of data records; each record has a sequence number and is assigned to a shard (similar to a partition) based on a partition key. Shards are distributed across multiple instances in the AWS environment. Producers put data into shards, and consumers get data from shards. See Figure 7.12.

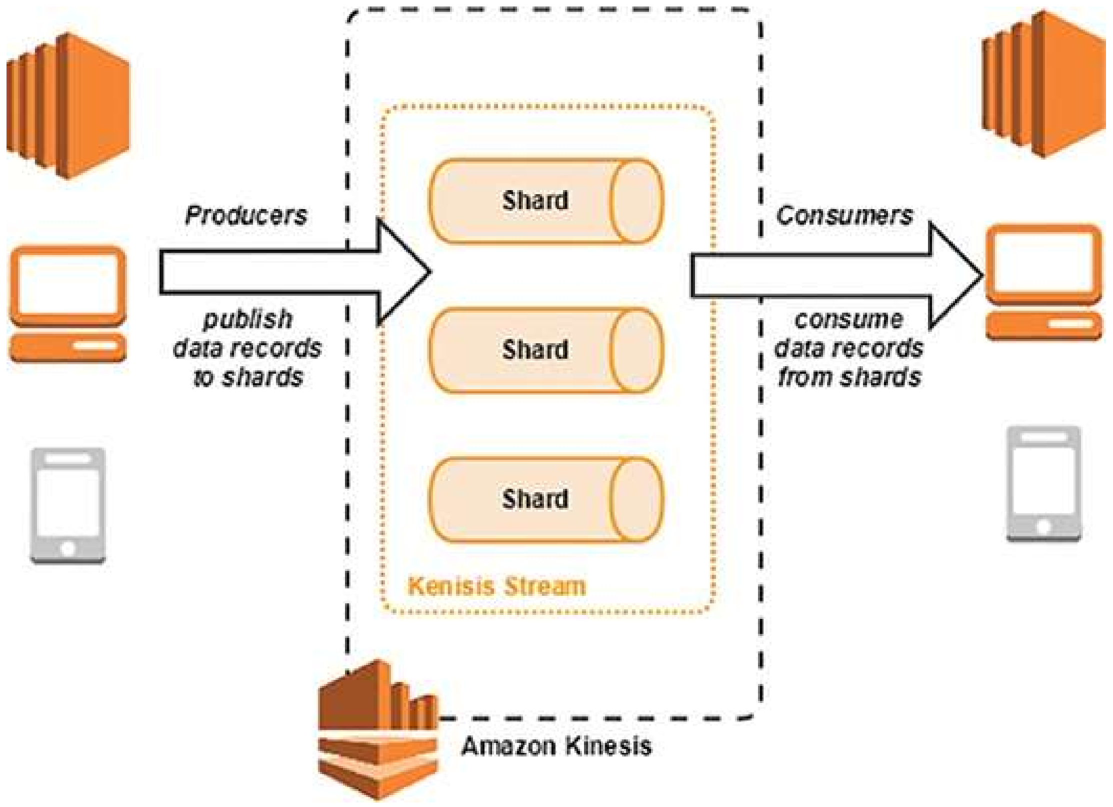
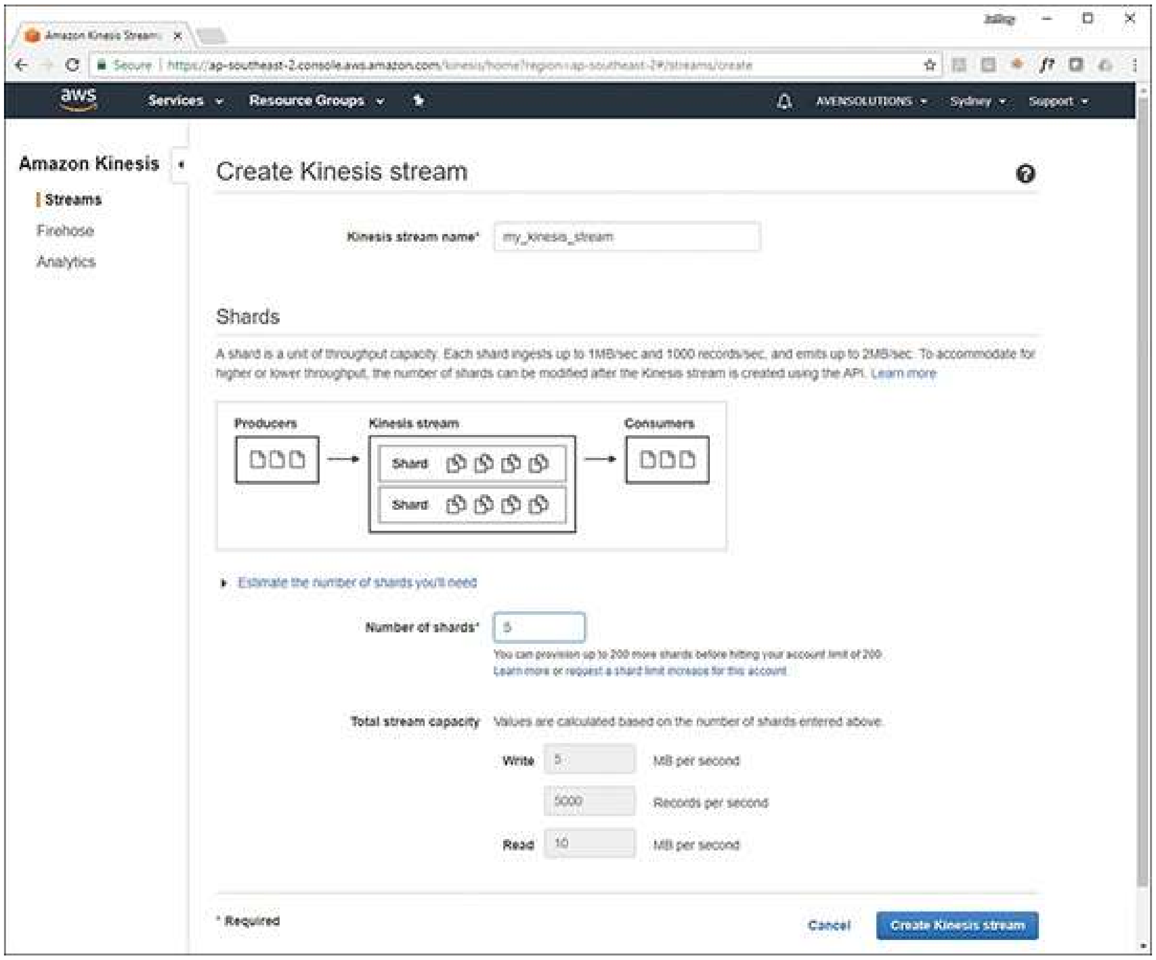

Producers
publish data records to shards
Shard
Shard
Shard
Kenisis Stream
Consumers
consume data records from shards
Amazon Kinesis

Figure 7.12 Amazon Kinesis streams.

You can create these streams by using the AWS console, the CLI, or the Streams API. Figure 7.13 demonstrates creating a stream using the AWS Management Console.


Amazon Kinesis Stream
Secure | https://ap-southeast-2.console.aws.amazon.com/kinesis/home?region=ap-southeast-2#/streams/create
aws
Services
Resource Groups
AVENSOLUTIONS
Sydney
Support
Amazon Kinesis
Streams
Firehose
Analytics
Producers
Kinesis stream
Shard
Shard
Consumers
Estimate the number of shards you'll need
Learn more
Learn more or request a shard limit increase for this account
Cancel
Create Kinesis stream

Create Kinesis stream

Kinesis stream name* my_kinesis_stream

Shards

A shard is a unit of throughput capacity. Each shard ingests up to 1MB/sec and 1000 records/sec, and emits up to 2MB/sec. To accommodate for higher or lower throughput, the number of shards can be modified after the Kinesis stream is created using the API.

Number of shards* 5

You can provision up to 200 more shards before hitting your account limit of 200.

Total stream capacity Values are calculated based on the number of shards entered above.

Write	5	MB per second
	5000	Records per second
Read	10	MB per second

* Required

Figure 7.13 Creating a Kinesis stream by using the AWS Management Console.

Amazon Kinesis Producer Library

The *Amazon Kinesis Producer Library* (*KPL*) is the set of API objects and methods used for producers to send records out to a Kinesis stream. The KPL enables producers to put records into Kinesis with the capability to buffer records, receive the result of a put as an asynchronous callback, write to multiple shards, and more.

Amazon Kinesis Client Library

The *Amazon Kinesis Client Library* (*KCL*) is the consumer API to connect to a stream and consume data records. The KCL is typically an entry point for record processing, such as event stream processing using Spark Streaming, from a Kinesis stream. The KCL also performs important functions, such as checkpointing processed records, using Amazon’s DynamoDB key/value store to maintain a durable copy of the Stream application's state. This table appears automatically in the same region as your Streams application, using your AWS credentials. The KCL is available in various common languages, including Java, Node.js, Ruby, and Python.

Using Spark with Amazon Kinesis

You can access Kinesis Streams from a Spark Streaming application by using the `createStream()` method and the `KinesisUtils` package (`pyspark.streaming.kinesis.KinesisUtils`). The `createStream()` method creates a receiver using the KCL and returns a DStream object.

Note that the KCL is licensed under the Amazon Software License (ASL). The terms and conditions of the ASL scheme vary somewhat from the Apache, GPL, and other open source licensing frameworks. Find more information at https://aws.amazon.com/asl/.

To use the `KinesisUtils.createStream()` function, you need an AWS account and API access credentials (Access Key ID and Secret Access Key); you also need to create a Kinesis Stream, though that process is beyond the scope of this book. You also need to supply the necessary Kinesis libraries, which you do by supplying the required `jar` using the `--jars` argument. Listing 7.20 shows an example of how to submit an application with Kinesis support.

Listing 7.20 **Submitting a Streaming Application with Kinesis Support**

Click here to view code image

```
spark-submit \
 --jars /usr/lib/spark/external/lib/spark-streaming-kinesis-asl-
assembly.jar
 ...
```

Given these prerequisites, the following section shows a description and an example of the `KinesisUtils.createStream()` method.

`createStream()`

Syntax:

Click here to view code image

```
KinesisUtils.createStream(ssc,
                          kinesisAppName,
                          streamName,
                          endpointUrl,
                          regionName,
                          initialPositionInStream,
                          checkpointInterval,
                          storageLevel=StorageLevel(True, True, False,
True, 2),
                          awsAccessKeyId=None,
                          awsSecretKey=None,
                          decoder=utf8_decoder)
```

The `createStream()` method creates an input stream that pulls messages from a Kinesis stream using the KCL and returns a DStream object. The `ssc` argument is an instantiated Spark `StreamingContext`. The `kinesisAppName` argument is a unique name used by the KCL to update state in the DynamoDB backing table. The `streamName` is the Kinesis stream name assigned when the stream was created. The `endpointUrl` and `regionName` arguments are references to the AWS Kinesis service and region—for example, `https://kinesis.us-east-1.amazonaws.com` and `us-east-1`. The `initialPositionInStream` is the initial starting position for messages in the stream; if checkpointing information is available, this argument is not used. The `checkpointInterval` is the interval for Kinesis checkpointing. The `storageLevel` argument is the RDD storage level to use for storing received objects; it defaults to `StorageLevel.MEMORY_AND_DISK_2`. The `awsAccessKeyId` and `awsSecretKey` arguments are your AWS API credentials. The `decoder` is the function used to decode message byte arrays, and it defaults to `utf8_decoder`. Listing 7.21 shows an example of using the `createStream()` method.

Listing 7.21 **Spark Streaming Using Amazon Kinesis**

Click here to view code image

```
from pyspark.streaming import StreamingContext
from pyspark import StorageLevel
from pyspark.streaming.kinesis import KinesisUtils
from pyspark.streaming.kinesis import InitialPositionInStream
ssc = StreamingContext(sc, 30)
appName = "KinesisCountApplication"
streamName = "my_kinesis_stream"
endpointUrl = "https://kinesis.ap-southeast-2.amazonaws.com"
regionName = "ap-southeast-2"
awsAccessKeyId = "YOURAWSACCESSKEYID"
awsSecretKey = "YOURAWSSECRETKEY"
# connect to Kinesis Stream
records = KinesisUtils.createStream(
           ssc, appName, streamName, endpointUrl, regionName,
           InitialPositionInStream.LATEST, 2,
           StorageLevel.MEMORY_AND_DISK_2,
           awsAccessKeyId, awsSecretKey)
# do some processing
output.pprint()
ssc.start()
ssc.awaitTermination()
```

Much more information about Kinesis is available at https://aws.amazon.com/kinesis/.

Summary

Spark Streaming is a key extension to the Spark core API, and it introduces objects and functions designed to process streams of data. One such object is the discretized stream (DStream), which is an RDD abstraction comprising streams of data batched into RDDs based on time intervals. Transformations applied to DStreams provide functions to each underlying RDD in the DStream. DStreams also have the capability to maintain state, which is accessible and updatable in real time—a key capability in stream processing use cases. Spark DStreams also support sliding window operations, which operate on data “windows” (such as the last hour, day, and so on).

This chapter covers some of the key open source messaging systems, such as Apache Kafka, which enable disparate systems to exchange messages, such as control messages or event messages, in an asynchronous yet reliable manner. The Spark Streaming project provides out-of-the-box support for Kafka, Kinesis, and other messaging platforms. When you use the messaging platform consumer libraries and utilities provided with the Spark Streaming subproject, Spark Streaming applications can connect to message brokers and consume messages into DStream objects.

Messaging systems are common data sources in complex event-processing pipelines powered by Spark applications. As the universe of connected devices continues to expand and machine- to-machine (M2M) data exchange proliferates, Spark Streaming and messaging will become even more important.