

The Purpose

The purpose of Spark Streaming: handle a continuous stream of live data in almost-real-time.

e.g. a long-running TCP connection; a directory where files regularly appear; message-passing systems (Kafka, Flume, RabbitMQ); Amazon Kinesis; Twitter streaming API.

Options

There are two streaming APIs in Spark:

- RDD-based: `pyspark.streaming.DStream`
- DataFrame-based: `pyspark.sql.streaming`


As you'd expect: DataFrame-based API is newer and assumes more structured data.

Let's take a look at both...

RDD-Based: The Idea

Input data comes in continuously, but you process it in batches (of maybe a few seconds). A *DStream* (discretized stream) objects listens to the input and creates the batches.

Each batch is an RDD. Use it like any other RDD. [\[*\]](#)



DStreams

With the RDD-based API, connecting to a source stream creates a `DStream` object.

Works something like an RDD. Has a [similar API](#) of things that can be done as the data arrives. It's really a collection of RDDs: one for each batch.

```
ssc = StreamingContext(sc, 2) # 2 second batches
lines = ssc.textFileStream(inputdir) # process files as they appear
data = lines.map(json.loads) # map DStream and return new DStream
ssc.start()
ssc.awaitTermination(timeout=3600) # listen for 1 hour
```

You need to actually *do* something with the RDD for each batch. That might be filtering/summarizing and saving to HDFS (as text/Parquet/etc) or Cassandra or....

```
def summarize(rdd):
    """ Process the RDD. """
    :
lines = ssc.textFileStream(inputdir)
data = lines.map(json.loads).filter(...)
data.foreachRDD(summarize)
data.saveAsTextFiles(outputprefix)
ssc.start()
ssc.awaitTermination()
```

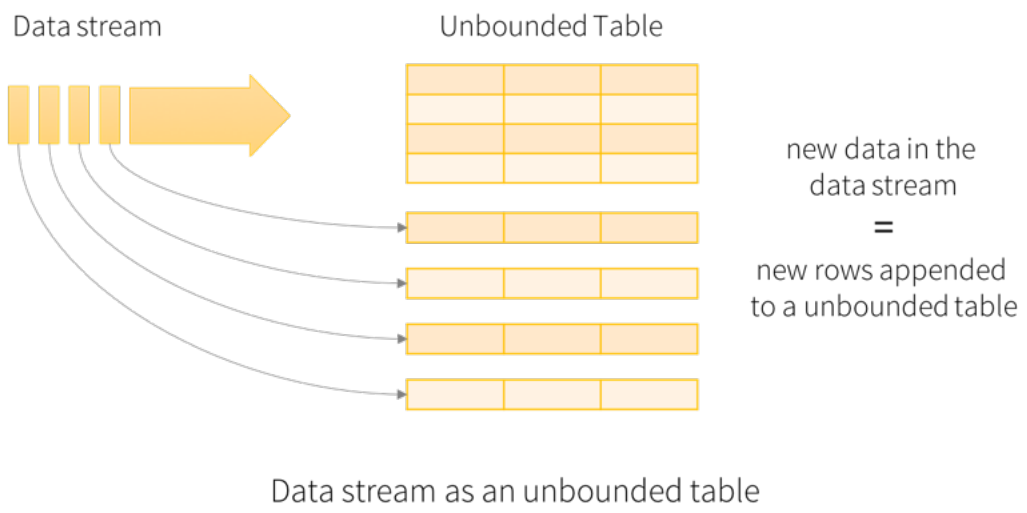
Like RDDs, DStreams are flexible and you end up doing most of the work in Python.

Structured Streaming

The new DataFrame-based API is [Spark Structured Streaming](#).

You work with a *Streaming DataFrame*, which is almost like any other DataFrame, but it grows (i.e. new rows are added) as data arrives on the stream. Still has a schema, and most of the operations you are used to.

Like a DataFrame, but grows in real time. [\[*\]](#)



You can build operations on the streaming DataFrame like any other DataFrame, but they get computed later, as the data arrives.

To create a streaming DataFrame, specify a source to read from. e.g. a directory of files, and process new files as they appear:

```
lines = spark.readStream.format('text') \
    .option('path', '/tmp/text').load()
```

Possible sources: files, TCP socket, Kafka message queue.

[Example code: [generate_lines.py](#) (a simple periodic file generator), [streaming_lines.py](#) (streaming job to consume them).]

Then operate on it like any other DataFrame. e.g.

```
lengths = lines.select(
    functions.length(lines['value']).alias('len')
)
counts = lengths.groupBy('len').count()
```

We're used to lazy evaluation: this isn't that different, except we specify the operations *before the data even exists*.

Then instead of the usual output options, we have a more limited set. e.g.

```
stream = counts.writeStream.format('console') \
    .outputMode('update').start()
stream.awaitTermination(600)
```

Output modes: complete (all data), append (new data), update (aggregated results that changed).

Output “sinks”: files, console, in-memory table, “foreach” (batch or row).

Combinations that are allowed depend on the operations you did on the source. e.g. “complete” only makes sense if you have a `.groupBy()`; “append” cannot be done if you did a `.groupBy()`.

The structured streaming API is still developing (e.g. the foreach sinks weren't accessible in Python until Spark 2.4). But, it seems compete enough to be usable, and it's likely where new streaming functionality will appear.

[Python foreach examples: [streaming_foreach.py](#).]