Spark Streaming

Big Data Analysis with Scala and Spark

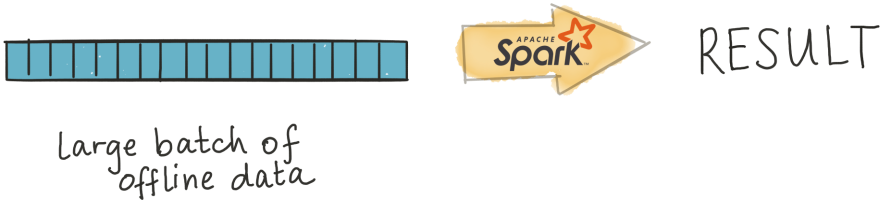
Heather Miller

Where Spark Streaming fits in (1)

Spark is focused on batching

Processing large, already-collected *batches* of data.

For example:



Where Spark Streaming fits in (1)

Spark is focused on batching

Processing large, already-collected *batches* of data.

Example batch jobs include:

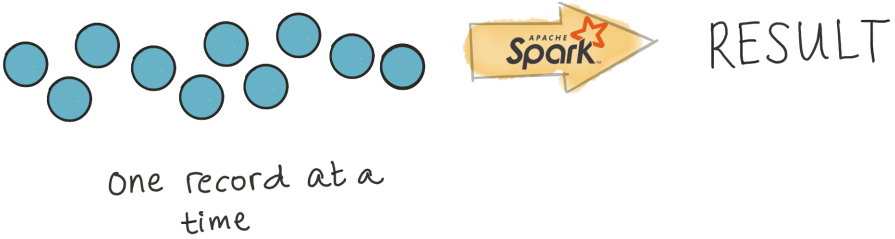
- analysis on terabytes of logs collected over a long period of time stored in S3 buckets
- analysis of code bases on GitHub, or on other large repositories of textual information such as Wikipedia
- nightly analysis on large data sets (images, text) collected over a 24 hour period

Where Spark Streaming fits in (2)

Spark Streaming is about streaming

Processing every value coming from a *stream* of data. That is, data values are constantly arriving.

For example:



Where Spark Streaming fits in (2)

Spark Streaming is about streaming

Processing every value coming from a *stream* of data. That is, data values are constantly arriving.

Example streaming jobs include:

- **real-time decision making**, *e.g., a bank that wants to automatically verify whether a new transaction on a customer's credit card represents fraud based on their recent history, and deny the transaction if the charge is determined fraudulent. (Stateful!)*
- **online machine learning**, *train a model on a combination of streaming and historical data from multiple users. E.g., fraud detection which continuously updates a model from all customers' behavior and tests each transaction against it.*

At odds with each other?

Looking at these two drawings, you may ask yourself...

Wait, how is it possible for me to put the streaming illustration into Spark?

At odds with each other?

Looking at these two drawings, you may ask yourself...

Wait, how is it possible for me to put the streaming illustration into Spark?

Everything that we've learned about optimizing operations on non-streaming Spark datasets have been based on the assumption that our data is fixed.

E.g., we assume we can stage up computation on a dataset that we might know a lot about, such as schema information in the case of DataFrames.

Microbatching

Spark supports streaming via ***microbatching***.

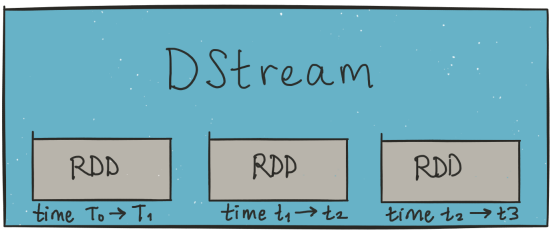
Micro-batch systems wait to accumulate small batches of input data (say, 500 ms' worth), then process each batch in parallel using a distributed collection of tasks, similar to the execution of a batch job in Spark.

For example:

DStreams

Spark Streaming provides a high-level abstraction called *discretized stream* or ***DStream***, which represents a continuous stream of data.

Internally, a DStream is represented as a sequence of RDDs.

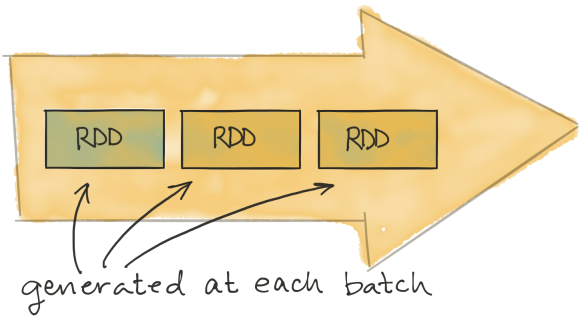


DStreams

Spark Streaming provides a high-level abstraction called *discretized stream* or ***DStream***, which represents a continuous stream of data.

Internally, a DStream is represented as a sequence of RDDs.

Another way to visualize it:



What do DStreams Look Like?

Let's say we want to count the number of words in text data received from a data server listening on a TCP socket.

What do DStreams Look Like?

Let's say we want to count the number of words in text data received from a data server listening on a TCP socket.

First we create a `StreamingContext`...

```
import org.apache.spark._
import org.apache.spark.streaming._

// Create a local StreamingContext with two working threads and batch
// interval of 1 second.
val conf = new SparkConf().setMaster("local[2]")
                          .setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

What do DStreams Look Like?

Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

What do DStreams Look Like?

The lines DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space characters into words.

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

// Split each line into words
val words = lines.flatMap(_.split(" "))
```

flatMap is a one-to-many DStream operation that creates a new DStream by generating multiple new records from each record in the source DStream.

—

What do DStreams Look Like?

Next, we want to count these words.

```
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in
// this DStream to the console
wordCounts.print()
```

The words DStream is further mapped (one-to-one transformation) to a DStream of (word, 1) pairs, which is then reduced to get the frequency of words in each batch of data. Finally, wordCounts.print() will print a few of the counts generated every second.

What do DStreams Look Like?

What happens now?

What do DStreams Look Like?

What happens now?

Nothing.

Spark Streaming only sets up the computation it will perform when it is started, and no real processing has started yet.

We still have to kick off computation:

```
ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

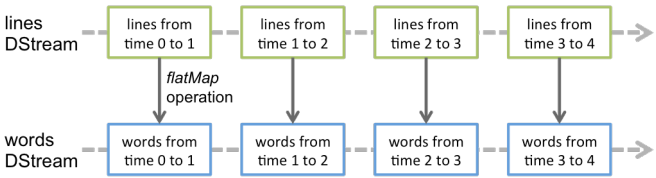
What do DStreams Look Like?

```
# TERMINAL 1:

# Running Netcat

$ nc -lk 9999

hello world
```

```
# TERMINAL 2: RUNNING NetworkWordCount

$ ./bin/run-example streaming.NetworkWordCount localhost 9999
...
-------------------------------------------
Time: 1357008430000 ms
-------------------------------------------
(hello,1)
(world,1)
...
```

What do DStreams Look Like?

Visualizing part of the previous computation:



Any operation applied on a DStream translates to operations on the underlying RDDs. For example, in the earlier example of converting a stream of lines to words, the flatMap operation is applied on each RDD in the lines DStream to generate the RDDs of the words DStream.

Creating the StreamingContext

In general, a StreamingContext object can be created from an existing SparkContext object.

```
import org.apache.spark.streaming._

val sc = ...                // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

The second parameter, Seconds(1) represents the time interval at which streaming data will be divided into batches.

Using the `StreamingContext`

After a `StreamingContext` is defined, the general workflow is the following:

1. Define the input sources by creating input DStreams.
2. Define the streaming computations by applying transformation and output operations to DStreams.
3. Start receiving data and processing it using `streamingContext.start()`.
4. Wait for the processing to be stopped (manually or due to any error) using `streamingContext.awaitTermination()`.
5. The processing can be manually stopped using `streamingContext.stop()`.

Using the StreamingContext (Important Points)

Important points to remember about StreamingContexts

1. Once a context has been started, no new streaming computations can be set up or added to it.
2. Once a context has been stopped, it cannot be restarted.
3. Only one StreamingContext can be active in a JVM at the same time.
4. stop() on StreamingContext also stops the SparkContext. To stop only the StreamingContext, set the optional parameter of stop() called stopSparkContext to false.
5. A SparkContext can be re-used to create multiple StreamingContext, as long as the previous StreamingContext is stopped (without stopping the SparkContext) before the next StreamingContext is created.

Kinds of Operations on DStreams

Rather than organizing operations on DStreams around transformations and actions, DStream operations are broken into the following categories:

1. **Ingestion**
2. **Transformation**
3. **Output**

Kinds of Operations on DStreams

Rather than organizing operations on DStreams around transformations and actions, DStream operations are broken into the following categories:

1. **Ingestion**
2. **Transformation**
3. **Output**

Note: transformations on DStreams are still lazy! Now instead, computation is kicked off explicitly by a call to the `start()` *method on the* `StreamingContext`

Ingestion: Getting Data Into DStreams

Every input DStream is associated with a ***Receiver*** object which receives the data from a source and stores it in Spark's memory for processing.

Spark Streaming provides three categories of streaming sources.

1. **Basic sources (built-in)**: Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
2. **Advanced sources (built-in)**: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes.
3. **Custom sources (user-provided)**: Input DStreams can also be created out of custom data sources. To do so, you must implement a your own `Receiver` (`apache.spark.streaming.receiver.Receiver`).

Basic Sources

The simplest sort of basic source available in Spark Streaming are **file streams**.

For reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.), a DStream can be created as via `StreamingContext.fileStream[KeyClass, ValueClass, InputFormatClass]`.

For simple text files, the easiest method is:

```
StreamingContext.textFileStream(dataDirectory)
```

Advanced Sources

Connectors have long been available for several popular message queues/pub-sub frameworks:

- Twitter
- Kafka
- Flume
- Kinesis

Usually: require external library.

Recently: Spark 2.3.0 shipped with methods in the Python API to read in data from Kafka, Kinesis and Flume.

Custom Sources

To create your own custom source, you must extend the abstract `Receiver` class, and implement two methods:

- `onStart()`: Things to do to start receiving data.
- `onStop()`: Things to do to stop receiving data.

Custom Sources

To create your own custom source, you must extend the abstract `Receiver` class, and implement two methods:

```
class CustomReceiver(host: String, port: Int)
  extends Receiver[String](StorageLevel.MEMORY_AND_DISK_2) with Logging {

  def onStart() {
    // Start the thread that receives data over a connection
    new Thread("Socket Receiver") {
      override def run() { receive() }
    }.start()
  }

  def onStop() {
    // There is nothing much to do as the thread calling receive()
    // is designed to stop by itself if isStopped() returns false
  }
}
```

Creating DStreams

Ways to create a DStream:

Creating DStreams

Ways to create a DStream:

1. **Ingest**: input data streams from sources such as Kafka, Flume, and Kinesis,
2. **Other DStreams**: or by applying high-level operations on other DStreams.

DStream Transformations (1)

DStreams support many of the transformations available on normal Spark RDDs.

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.

DStream Transformations (2)

DStreams support many of the transformations available on normal Spark RDDs.

countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property `spark.default.parallelism`) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.
join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
cogroup(*otherStream*, [*numTasks*])	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
transform(*func*)	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
updateStateByKey(*func*)	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 for the key. This can be used to maintain arbitrary state data for each key.

Outputting Results

Output operations allow DStream's data to be pushed out to external systems like a database or a file systems. Since the output operations actually allow the transformed data to be consumed by external systems, they trigger the actual execution of all the DStream transformations (similar to actions for RDDs).

Output Operation	Meaning
print()	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging. Python API This is called **pprint()** in the Python API.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as text files. The file name at each batch interval is generated based on *prefix* and *suffix*: "*prefix-TIME_IN_MS[.suffix]*".
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as `SequenceFiles` of serialized Java objects. The file name at each batch interval is generated based on *prefix* and *suffix*: "*prefix-TIME_IN_MS[.suffix]*". Python API This is not available in the Python API.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on *prefix* and *suffix*: "*prefix-TIME_IN_MS[.suffix]*". Python API This is not available in the Python API.
foreachRDD(*func*)	The most generic output operator that applies a function, *func*, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function *func* is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.