Integrating Kafka and Spark Streaming: Code Examples and State of the Game

Oct 1, 2014 · 22 min read

[Spark Streaming](https://spark.apache.org/streaming/) has been getting some attention lately as a real-time data processing tool, often mentioned alongside [Apache Storm](http://storm.apache.org/). If you ask me, no real-time data processing tool is complete without Kafka integration (smile), hence I added an example Spark Streaming application to [kafka-storm-starter](https://github.com/miguno/kafka-storm-starter)that demonstrates how to read from Kafka and write to Kafka, using [Avro](http://avro.apache.org/) as the data format and [Twitter Bijection](https://github.com/twitter/bijection) for handling the data serialization. In this post I will explain this Spark Streaming example in further detail and also shed some light on the current state of Kafka integration in Spark Streaming. All this with the disclaimer that this happens to be my first experiment with Spark Streaming.

1. [What is Spark Streaming?](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#what-is-spark-streaming)
   * [Spark Streaming vs. Apache Storm](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#spark-streaming-vs-apache-storm)
2. [Excursus: Machines, cores, executors, tasks, and receivers in Spark](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#excursus-machines-cores-executors-tasks-and-receivers-in-spark)
3. [Integrating Kafka with Spark Streaming](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#integrating-kafka-with-spark-streaming)
   * [Overview](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#overview)
   * [Primer on topics, partitions, and parallelism in Kafka](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#primer-on-topics-partitions-and-parallelism-in-kafka)
   * [Reading from Kafka](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#reading-from-kafka)
     + [Read parallelism in Spark Streaming](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#read-parallelism-in-spark-streaming)
     + [Downstream processing parallelism in Spark Streaming](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#downstream-processing-parallelism-in-spark-streaming)
   * [Writing to Kafka](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#writing-to-kafka)
   * [Complete example](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#complete-example)
4. [Known issues in Spark Streaming](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#known-issues-in-spark-streaming)
5. [Spark tips and tricks](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#spark-tips-and-tricks)
   * [General](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#general)
   * [Kafka integration](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#kafka-integration)
   * [Testing](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#testing)
   * [Performance tuning](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#performance-tuning)
6. [Wrapping up](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#wrapping-up)
7. [References](https://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/#references)

**The Spark Streaming example code is available at**[**kafka-storm-starter**](https://github.com/miguno/kafka-storm-starter)**on GitHub. And yes, the project's name might now be a bit misleading. :-)**

What is Spark Streaming?

[Spark Streaming](http://spark.apache.org/streaming/) is a sub-project of [Apache Spark](http://spark.apache.org/). Spark is a batch processing platform similar to Apache Hadoop, and Spark Streaming is a real-time processing tool that runs on top of the Spark engine.

Spark Streaming vs. Apache Storm

In terms of use cases Spark Streaming is closely related to [Apache Storm](http://storm.apache.org/), which is arguably today’s most popular real-time processing platform for Big Data. Bobby Evans and Tom Graves of Yahoo! Engineering recently gave a talk on [Spark and Storm at Yahoo!](http://yahoohadoop.tumblr.com/post/98213421641/storm-and-spark-at-yahoo-why-chose-one-over-the-other), in which they compare the two platforms and also cover the question of when and why choosing one over the other. Similarly, P. Taylor Goetz of HortonWorks shared a slide deck titled [Apache Storm and Spark Streaming Compared](http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming).

Here’s my personal, very brief comparison: Storm has higher industry adoption and better production stability compared to Spark Streaming. Spark on the other hand has a more expressive, higher level API than Storm, which is arguably more pleasant to use, at least if you write your Spark applications in Scala (I prefer the Spark API, too). But don’t trust my word, please do check out the talks/decks above yourself.

Both Spark and Storm are top-level Apache projects, and vendors have begun to integrate either or both tools into their commercial offerings, e.g. HortonWorks ([Storm](http://hortonworks.com/hadoop/storm/), [Spark](http://hortonworks.com/hadoop/spark/)) and Cloudera ([Spark](http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/spark.html)).

Excursus: Machines, cores, executors, tasks, and receivers in Spark

The subsequent sections of this article talk a lot about parallelism in Spark and in Kafka. You need at least a basic understanding of some Spark terminology to be able to follow the discussion in those sections.

* A Spark **cluster** contains 1+ worker nodes aka slave machines (simplified view; I exclude pieces like cluster managers here.)
* A **worker node** can run 1+ executors.
* An **executor** is a process launched for an application on a worker node, which runs tasks and keeps data in memory or disk storage across them. Each application has its own executors. An executor has a certain amount of cores aka “slots” available to run tasks assigned to it.
* A **task** is a unit of work that will be sent to one executor. That is, it runs (part of) the actual computation of your application. The SparkContext sends those tasks for the executors to run. Each task occupies one slot aka core in the parent executor.
* A **receiver** ([API](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.receiver.Receiver), [docs](http://spark.apache.org/docs/latest/streaming-custom-receivers.html)) is run within an executor as a long-running task. Each receiver is responsible for exactly one so-called *input DStream* (e.g. an input stream for reading from Kafka), and each receiver – and thus input DStream – occupies one core/slot.
* An **input DStream**: an *input* DStream is a special DStream that connects Spark Streaming to external data sources for reading input data. For each external data source (e.g. Kafka) you need one such input DStream implementation. Once Spark Streaming is “connected” to an external data source via such input DStreams, any subsequent DStream transformations will create “normal” DStreams.

In Spark’s execution model, each application gets its own executors, which stay up for the duration of the whole application and run 1+ tasks in multiple threads. This isolation approach is similar to Storm’s model of execution. This architecture becomes more complicated once you introduce cluster managers like YARN or Mesos, which I do not cover here. See [Cluster Overview](http://spark.apache.org/docs/latest/cluster-overview.html) in the Spark docs for further details.

Integrating Kafka with Spark Streaming

Overview

In short, Spark Streaming supports Kafka but there are still some rough edges.

A good starting point for me has been the [KafkaWordCount](https://github.com/apache/spark/blob/master/examples/scala-2.10/src/main/scala/org/apache/spark/examples/streaming/KafkaWordCount.scala) example in the Spark code base (**Update 2015-03-31:** see also [DirectKafkaWordCount](https://github.com/apache/spark/blob/master/examples/scala-2.10/src/main/scala/org/apache/spark/examples/streaming/DirectKafkaWordCount.scala)). When I read this code, however, there were still a couple of open questions left.

Notably I wanted to understand how to:

* Read from Kafka *in parallel*. In Kafka, a topic can have *N* partitions, and ideally we’d like to parallelize reading from those *N* partitions. This is what the [Kafka spout in Storm](https://github.com/apache/incubator-storm/tree/master/external/storm-kafka) does.
* Write to Kafka from a Spark Streaming application, also, *in parallel*.

On top of those questions I also ran into several known issues in Spark and/or Spark Streaming, most of which have been discussed in the Spark mailing list. I’ll summarize the current state and known issues of the Kafka integration further down below.

Primer on topics, partitions, and parallelism in Kafka

*For details see my articles* [*Apache Kafka 0.8 Training Deck and Tutorial*](https://www.michael-noll.com/blog/2014/08/18/apache-kafka-training-deck-and-tutorial/) *and* [*Running a Multi-Broker Apache Kafka 0.8 Cluster on a Single Node*](https://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/)*.*

Kafka stores data in *topics*, with each topic consisting of a configurable number of *partitions*. The number of partitions of a topic is very important for performance considerations as this number is an *upper bound on the* *consumer parallelism*: if a topic has *N* partitions, then your application can only consume this topic with a maximum of *N* threads in parallel. (At least this is the case when you use Kafka’s built-in Scala/Java consumer API.)

When I say “application” I should rather say *consumer group* in Kafka’s terminology. A consumer group, identified by a string of your choosing, is the cluster-wide identifier for a logical consumer application. All consumers that are part of the same consumer group share the burden of reading from a given Kafka topic, and only a maximum of *N* (= number of partitions) threads across all the consumers in the same group will be able to read from the topic. Any excess threads will sit idle.

**Multiple Kafka consumer groups can be run in parallel:** Of course you can run multiple, independent logical consumer applications against the same Kafka topic. Here, each logical application will run its consumer threads under a unique consumer group id. Each application can then also use different read parallelisms (see below). When I am talking about the various ways to configure read parallelism in the following sections, then I am referring to the settings of a *single*one of these logical consumer applications.

Here are some simplified examples.

* Your application uses the consumer group id “terran” to read from a Kafka topic “zerg.hydra” that has **10 partitions**. If you configure your application to consume the topic with only **1** thread, then this single thread will read data from all 10 partitions.
* Same as above, but this time you configure **5** consumer threads. Here, each thread will read from 2 partitions.
* Same as above, but this time you configure **10** consumer threads. Here, each thread will read from a single partition.
* Same as above, but this time you configure **14** consumer threads. Here, 10 of the 14 threads will read from a single partition each, and the remaining 4 threads will be idle.

Let’s introduce some real-world complexity in this simple picture – the *rebalancing* event in Kafka. Rebalancing is a lifecycle event in Kafka that occurs when consumers join or leave a consumer group (there are more conditions that trigger rebalancing but these are not important in this context; see my[Kafka training deck](https://www.michael-noll.com/blog/2014/08/18/apache-kafka-training-deck-and-tutorial/) for details on rebalancing).

* Your application uses the consumer group id “terran” and starts consuming with 1 thread. This thread will read from all 10 partitions. During runtime, you’ll increase the number of threads from 1 to 14. That is, there is suddenly a change of parallelism for the same consumer group. This triggers *rebalancing* in Kafka. Once rebalancing completes, you will have 10 of 14 threads consuming from a single partition each, and the 4 remaining threads will be idle. And as you might have guessed, the initial thread will now read from only one partition and will no longer see data from the other nine.

We have now a basic understanding of topics, partitions, and the number of partitions as an upper bound for the parallelism when reading from Kafka. But what are the resulting implications for an application – such as a Spark Streaming job or Storm topology – that reads its input data from Kafka?

1. **Read parallelism:** You typically want to read from all *N* partitions of a Kafka topic in parallel by consuming with *N* threads. And depending on the data volume you want to spread those threads across different NICs, which typically means across different machines. In Storm, this is achieved by setting the parallelism of the [Kafka spout](https://github.com/apache/storm/tree/master/external/storm-kafka) to *N* via TopologyBuilder#setSpout(). The Spark equivalent is a bit trickier, and I will describe how to do this in further detail below.
2. **Downstream processing parallelism:** Once retrieved from Kafka you want to process the data in parallel. Depending on your use case this level of parallelism must be different from the read parallelism. If your use case is CPU-bound, for instance, you want to have many more processing threads than read threads; this is achieved by shuffling or “fanning out” the data via the network from the few read threads to the many processing threads. Hence you pay for the access to more cores with increased network communication, serialization overhead, etc. In Storm, you perform such a shuffling via a [shuffle grouping](https://storm.apache.org/documentation/Concepts.html) from the Kafka spout to the next downstream bolt. The Spark equivalent is the [repartition](https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#transformations-on-dstreams) transformation on DStreams.

The important takeaway is that it is possible – and often desired – to decouple the level of parallelisms for *reading from Kafka* and for *processing the data once read*. In the next sections I will describe the various options you have at your disposal to configure read parallelism and downstream processing parallelism in Spark Streaming.

Reading from Kafka

Read parallelism in Spark Streaming

Like Kafka, Spark Streaming has the concept of *partitions*. It is important to understand that Kafka’s per-topic partitions are not correlated to the partitions of [RDDs in Spark](http://spark.apache.org/docs/1.1.0/programming-guide.html).

The [KafkaInputDStream](https://github.com/apache/spark/blob/master/external/kafka/src/main/scala/org/apache/spark/streaming/kafka/KafkaInputDStream.scala) of Spark Streaming – aka its Kafka “connector” – uses Kafka’s [high-level consumer API](http://kafka.apache.org/documentation.html#highlevelconsumerapi), which means you have two control knobs in Spark that determine read parallelism for Kafka:

1. **The number of input DStreams.** Because Spark will run one receiver (= task) per input DStream, this means using multiple input DStreams will parallelize the read operations across multiple cores and thus, hopefully, across multiple machines and thereby NICs.
2. **The number of consumer threads per input DStream.** Here, the same receiver (= task) will run multiple threads. That is, read operations will happen in parallel but on the same core/machine/NIC.

For practical purposes option 1 is the preferred.

Why is that? First and foremost because reading from Kafka is normally network/NIC limited, i.e. you typically do not increase read-throughput by running more threads *on the same* *machine*. In other words, it is rare though possible that reading from Kafka runs into CPU bottlenecks. Second, if you go with option 2 then multiple threads will be competing for the lock to push data into so-called *blocks* (the += method of BlockGenerator that is used behind the scenes is synchronized on the block generator instance).

**Number of partitions of the RDDs created by the input DStreams:** The KafkaInputDStream will store individual messages received from Kafka into so-called *blocks*. From what I understand, a new block is generated every [spark.streaming.blockInterval](http://spark.apache.org/docs/latest/configuration.html#spark-streaming)milliseconds, and each block is turned into a partition of the RDD that will eventually be created by the DStream. If this assumption of mine is true, then the number of partitions in the RDDs created by KafkaInputDStream is determined by batchInterval / spark.streaming.blockInterval, where batchInterval is the time interval at which streaming data will be divided into batches (set via a constructor parameter of StreamingContext). For example, if the batch interval is 2 seconds (default) and the block interval is 200ms (default), your RDD will contain 10 partitions. Please correct me if I'm mistaken.

Option 1: Controlling the number of input DStreams

The example below is taken from the [Spark Streaming Programming Guide](https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#reducing-the-processing-time-of-each-batch).

**val** ssc**:** **StreamingContext** **=** **???** *// ignore for now*

**val** kafkaParams**:** **Map[String**, **String]** **=** **Map(**"group.id" **->** "terran"**,** */\* ignore rest \*/***)**

**val** numInputDStreams **=** 5

**val** kafkaDStreams **=** **(**1 to numInputDStreams**).**map **{** **\_** **=>** **KafkaUtils.**createStream**(...)** **}**

In this example we create five input DStreams, thus spreading the burden of reading from Kafka across five cores and, hopefully, five machines/NICs. (I say “hopefully” because I am not certain whether Spark Streaming task placement policy will try to place receivers on different machines.) All input DStreams are part of the “terran” consumer group, and the Kafka API will ensure that these five input DStreams a) will see all available data for the topic because it assigns each partition of the topic to an input DStream and b) will not see overlapping data because each partition is assigned to only one input DStream at a time. In other words, this setup of “collaborating” input DStreams works because of the consumer group behavior provided by the Kafka API, which is used behind the scenes by KafkaInputDStream.

What I have not shown in the example is how many threads are created *per input DStream*, which is done via parameters to the KafkaUtils.createStream method (the actual input topic(s) are also specified as parameters of this method). We will do this in the next section.

But before we continue let me highlight several known issues with this setup and with Spark Streaming in particular, which are caused on the one hand by current limitations of Spark in general and on the other hand by the current implementation of the Kafka input DStream in particular:

*[When you use the multi-input-stream approach I described above, then] those consumers operate in one [Kafka] consumer group, and they try to decide which consumer consumes which partitions. And it may just fail to do syncpartitionrebalance, and then you have only a few consumers really consuming. To mitigate this problem, you can set rebalance retries very high, and pray it helps.*

*Then arises yet another “feature” — if your receiver dies (OOM, hardware failure), you just stop receiving from Kafka! (*[*source*](http://markmail.org/message/257a5l3oqyftsjxj)*)*

The “stop receiving from Kafka” issue requires [some explanation](http://apache-spark-user-list.1001560.n3.nabble.com/spark-streaming-and-the-spark-shell-tp3347.html). Currently, when you start your streaming application via ssc.start() the processing starts and continues indefinitely – even if the input data source (e.g. Kafka) becomes unavailable. That is, streams are not able to detect if they have lost connection to the upstream data source and thus cannot react to this event, e.g. by reconnecting or by stopping the execution. Similarly, if you lose a receiver that reads from the data source, then [your streaming application will generate empty RDDs](http://apache-spark-user-list.1001560.n3.nabble.com/spark-streaming-and-the-spark-shell-tp3347p3375.html).

This is a pretty unfortunate situation. One crude workaround is to restart your streaming application whenever it runs into an upstream data source failure or a receiver failure. This workaround may not help you though if your use case requires you to set the Kafka configuration option auto.offset.resetto “smallest” – because of a known bug in Spark Streaming the resulting behavior of your streaming application may not be what you want. See the section on *Known issues in Spark Streaming* below for further details.

Option 2: Controlling the number of consumer threads per input DStream

In this example we create a *single* input DStream that is configured to run three consumer threads – in the same receiver/task and thus on the same core/machine/NIC – to read from the Kafka topic “zerg.hydra”.

**val** ssc**:** **StreamingContext** **=** **???** *// ignore for now*

**val** kafkaParams**:** **Map[String**, **String]** **=** **Map(**"group.id" **->** "terran"**,** **...)**

**val** consumerThreadsPerInputDstream **=** 3

**val** topics **=** **Map(**"zerg.hydra" **->** consumerThreadsPerInputDstream**)**

**val** stream **=** **KafkaUtils.**createStream**(**ssc**,** kafkaParams**,** topics**,** **...)**

The KafkaUtils.createStream method is overloaded, so there are a few different method signatures. In this example we pick the Scala variant that gives us the most control.

Combining options 1 and 2

Here is a more complete example that combines the previous two techniques:

**val** ssc**:** **StreamingContext** **=** **???**

**val** kafkaParams**:** **Map[String**, **String]** **=** **Map(**"group.id" **->** "terran"**,** **...)**

**val** numDStreams **=** 5

**val** topics **=** **Map(**"zerg.hydra" **->** 1**)**

**val** kafkaDStreams **=** **(**1 to numDStreams**).**map **{** **\_** **=>**

**KafkaUtils.**createStream**(**ssc**,** kafkaParams**,** topics**,** **...)**

**}**

We are creating five input DStreams, each of which will run a single consumer thread. If the input topic “zerg.hydra” has five partitions (or less), then this is normally the best way to parallelize read operations if you care primarily about maximizing throughput.

Downstream processing parallelism in Spark Streaming

In the previous sections we covered parallelizing reads from Kafka. Now we can tackle parallelizing the downstream data processing in Spark. Here, you must keep in mind how Spark itself parallelizes its processing. Like Kafka, Spark ties the parallelism to the number of (RDD) partitions by running [*one task per RDD partition*](http://spark.apache.org/docs/1.1.0/programming-guide.html#resilient-distributed-datasets-rdds) (sometimes partitions are still called “slices” in the docs).

**Just like any Spark application:** Once a Spark Streaming application has received its input data, any further processing is identical to non-streaming Spark applications. That is, you use exactly the same tools and patterns to scale your application as you would for "normal" Spark data flows. See [Level of Parallelism in Data Processing](https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#level-of-parallelism-in-data-processing).

This gives us two control knobs:

1. **The number of input DStreams**, i.e. what we receive as a result of the previous sections on read parallelism. This is our starting point, which we can either take as-is or modify with the next option.
2. **The** [**repartition**](https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#transformations-on-dstreams) **DStream transformation.** It returns a new DStream with an increased or decreased level *N* of parallelism. Each RDD in the returned DStream has exactly *N* partitions. DStreams are a continuous series of RDDs, and behind the scenes DStream.repartition calls RDD.repartition. The latter “reshuffles the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.” In other words, DStream.repartition is very similar to Storm’s [shuffle grouping](https://storm.apache.org/documentation/Concepts.html).

Hence repartition is our primary means to decouple read parallelism from processing parallelism. It allows us to set the number of processing tasks and thus the number of cores that will be used for the processing. Indirectly, we also influence the number of machines/NICs that will be involved.

A related DStream transformation is [union](https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#transformations-on-dstreams). (This method also exists for StreamingContext, where it returns the unified DStream from multiple DStreams of the same type and same slide duration. Most likely you would use the StreamingContext variant.) A union will return a UnionDStream backed by a UnionRDD. A UnionRDD is comprised of all the partitions of the RDDs being unified, i.e. if you unite 3 RDDs with 10 partitions each, then your union RDD instance will contain 30 partitions. In other words,union will squash multiple DStreams into a single DStream/RDD, but it will not change the level of parallelism. Whether you need to use union or not depends on whether your use case requires information from all Kafka partitions “in one place”, so it’s primarily because of semantic requirements. One such example is when you need to perform a (global) count of distinct elements.

Note: [RDDs are not ordered.](http://apache-spark-user-list.1001560.n3.nabble.com/Union-of-2-RDD-s-only-returns-the-first-one-tp766p5089.html) So when you union RDDs, then the resulting RDD itself will not have a well-defined ordering either. If you need ordering sort the RDD.

Your use case will determine which knobs and which combination thereof you need to use. Let’s say your use case is CPU-bound. Here, you may want to consume the Kafka topic “zerg.hydra” (which has five Kafka partitions) with a read parallelism of 5 – i.e. 5 receivers with 1 consumer thread each – but bump up the processing parallelism to 20:

**val** ssc**:** **StreamingContext** **=** **???**

**val** kafkaParams**:** **Map[String**, **String]** **=** **Map(**"group.id" **->** "terran"**,** **...)**

**val** readParallelism **=** 5

**val** topics **=** **Map(**"zerg.hydra" **->** 1**)**

**val** kafkaDStreams **=** **(**1 to readParallelism**).**map **{** **\_** **=>**

**KafkaUtils.**createStream**(**ssc**,** kafkaParams**,** topics**,** **...)**

**}**

*//> collection of five \*input\* DStreams = handled by five receivers/tasks*

**val** unionDStream **=** ssc**.**union**(**kafkaDStreams**)** *// often unnecessary, just showcasing how to do it*

*//> single DStream*

**val** processingParallelism **=** 20

**val** processingDStream **=** unionDStream**(**processingParallelism**)**

*//> single DStream but now with 20 partitions*

In the next section we tie all the pieces together and also cover the actual data processing.

Writing to Kafka

Writing to Kafka should be done from the foreachRDD output operation:

*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 a external system, like saving the RDD to files, or writing it over the network to a database. Note that the function func is executed at the driver, and will usually have RDD actions in it that will force the computation of the streaming RDDs.*

Note: The remark "the function func is executed at the driver" does not mean that, say, a Kafka producer itself would be run from the driver. Rather, read this remark more as "the function funcis *evaluated* at the driver". The actual behavior will become more clear once you read *Design Patterns for using foreachRDD*.

You should read the section [Design Patterns for using foreachRDD](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#output-operations-on-dstreams) in the Spark docs, which explains the recommended patterns as well as common pitfalls when using foreachRDD to talk to external systems.

In my case, I decided to follow the recommendation to re-use Kafka producer instances across multiple RDDs/batches via a pool of producers. I implemented such a pool with [Apache Commons Pool](http://commons.apache.org/proper/commons-pool/), see [PooledKafkaProducerAppFactory](https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/kafka/PooledKafkaProducerAppFactory.scala). Factories are helpful in this context because of Spark’s execution and serialization model. The pool itself is provided to the tasks via a [broadcast variable](http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables).

The end result looks as follows:

**val** producerPool **=** **{**

*// See the full code on GitHub for details on how the pool is created*

**val** pool **=** createKafkaProducerPool**(**kafkaZkCluster**.**kafka**.**brokerList**,** outputTopic**.**name**)**

ssc**.**sparkContext**.**broadcast**(**pool**)**

**}**

stream**.**map **{** **...** **}.**foreachRDD**(**rdd **=>** **{**

rdd**.**foreachPartition**(**partitionOfRecords **=>** **{**

*// Get a producer from the shared pool*

**val** p **=** producerPool**.**value**.**borrowObject**()**

partitionOfRecords**.**foreach **{** **case** tweet**:** **Tweet** **=>**

*// Convert pojo back into Avro binary format*

**val** bytes **=** converter**.**value**.**apply**(**tweet**)**

*// Send the bytes to Kafka*

p**.**send**(**bytes**)**

**}**

*// Returning the producer to the pool also shuts it down*

producerPool**.**value**.**returnObject**(**p**)**

**})**

**})**

Keep in mind that Spark Streaming creates many RRDs per minute, each of which contains multiple partitions, so preferably you shouldn’t create new Kafka producers for each partition, let alone for each Kafka message. The setup above minimizes the creation of Kafka producer instances, and also minimizes the number of TCP connections that are being established with the Kafka cluster. You can use this pool setup to precisely control the number of Kafka producer instances that are being made available to your streaming application (if in doubt, use fewer).

Complete example

The code example below is the gist of my example Spark Streaming application ([see the full code for details and explanations](https://github.com/miguno/kafka-storm-starter/blob/develop/src/test/scala/com/miguno/kafkastorm/spark/KafkaSparkStreamingSpec.scala)). Here, I demonstrate how to:

* Read Avro-encoded data (the Tweet class) from a Kafka topic in parallel. We use a the optimal read parallelism of one single-threaded input DStream per Kafka partition.
* Deserialize the Avro-encoded data back into pojos, then serializing them back into binary. The serialization is performed via [Twitter Bijection](https://github.com/twitter/bijection).
* Write the results back into a different Kafka topic via a Kafka producer pool.

*// Set up the input DStream to read from Kafka (in parallel)*

**val** kafkaStream **=** **{**

**val** sparkStreamingConsumerGroup **=** "spark-streaming-consumer-group"

**val** kafkaParams **=** **Map(**

"zookeeper.connect" **->** "zookeeper1:2181"**,**

"group.id" **->** "spark-streaming-test"**,**

"zookeeper.connection.timeout.ms" **->** "1000"**)**

**val** inputTopic **=** "input-topic"

**val** numPartitionsOfInputTopic **=** 5

**val** streams **=** **(**1 to numPartitionsOfInputTopic**)** map **{** **\_** **=>**

**KafkaUtils.**createStream**(**ssc**,** kafkaParams**,** **Map(**inputTopic **->** 1**),** **StorageLevel.MEMORY\_ONLY\_SER).**map**(\_.**\_2**)**

**}**

**val** unifiedStream **=** ssc**.**union**(**streams**)**

**val** sparkProcessingParallelism **=** 1 *// You'd probably pick a higher value than 1 in production.*

unifiedStream**.**repartition**(**sparkProcessingParallelism**)**

**}**

*// We use accumulators to track global "counters" across the tasks of our streaming app*

**val** numInputMessages **=** ssc**.**sparkContext**.**accumulator**(**0L**,** "Kafka messages consumed"**)**

**val** numOutputMessages **=** ssc**.**sparkContext**.**accumulator**(**0L**,** "Kafka messages produced"**)**

*// We use a broadcast variable to share a pool of Kafka producers, which we use to write data from Spark to Kafka.*

**val** producerPool **=** **{**

**val** pool **=** createKafkaProducerPool**(**kafkaZkCluster**.**kafka**.**brokerList**,** outputTopic**.**name**)**

ssc**.**sparkContext**.**broadcast**(**pool**)**

**}**

*// We also use a broadcast variable for our Avro Injection (Twitter Bijection)*

**val** converter **=** ssc**.**sparkContext**.**broadcast**(SpecificAvroCodecs.**toBinary**[Tweet])**

*// Define the actual data flow of the streaming job*

kafkaStream**.**map **{** **case** bytes **=>**

numInputMessages **+=** 1

*// Convert Avro binary data to pojo*

converter**.**value**.**invert**(**bytes**)** **match** **{**

**case** **Success(**tweet**)** **=>** tweet

**case** **Failure(**e**)** **=>** *// ignore if the conversion failed*

**}**

**}.**foreachRDD**(**rdd **=>** **{**

rdd**.**foreachPartition**(**partitionOfRecords **=>** **{**

**val** p **=** producerPool**.**value**.**borrowObject**()**

partitionOfRecords**.**foreach **{** **case** tweet**:** **Tweet** **=>**

*// Convert pojo back into Avro binary format*

**val** bytes **=** converter**.**value**.**apply**(**tweet**)**

*// Send the bytes to Kafka*

p**.**send**(**bytes**)**

numOutputMessages **+=** 1

**}**

producerPool**.**value**.**returnObject**(**p**)**

**})**

**})**

*// Run the streaming job*

ssc**.**start**()**

ssc**.**awaitTermination**()**

[*See the full source code for further details and explanations.*](https://github.com/miguno/kafka-storm-starter/blob/develop/src/test/scala/com/miguno/kafkastorm/spark/KafkaSparkStreamingSpec.scala)

Personally, I really like the conciseness and expressiveness of the Spark Streaming code. As Bobby Evans and Tom Graves are eluding to in their talk, the Storm equivalent of this code is more verbose and comparatively lower level: The [KafkaStormSpec](https://github.com/miguno/kafka-storm-starter/blob/develop/src/test/scala/com/miguno/kafkastorm/integration/KafkaStormSpec.scala) in [kafka-storm-starter](https://github.com/miguno/kafka-storm-starter) wires and runs a Storm topology that performs the same computations. Well, the spec file itself is only a few lines of code once you exclude the code comments, which I only keep for didactic reasons; however, keep in mind that in Storm’s Java API you cannot use Scala-like anonymous functions as I show in the Spark Streaming example above (e.g. the map and foreach steps). Instead you must write “full” classes – bolts in plain Storm, functions/filters in Storm Trident – to achieve the same functionality, see e.g. [AvroDecoderBolt](https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/storm/bolts/AvroDecoderBolt.scala). This feels a bit similar to, say, having to code against Spark’s own API using Java, where juggling with anonymous functions is IMHO just as painful.

Lastly, I also liked the [Spark documentation](http://spark.apache.org/documentation.html). It was very easy to get started, and even some more advanced use is covered (e.g. [Tuning Spark](http://spark.apache.org/docs/1.1.0/tuning.html)). I still had to browse the mailing list and also dive into the source code, but the general starting experience was ok – only the Kafka integration part was lacking (hence this blog post). Good job to everyone involved maintaining the docs!

Known issues in Spark Streaming

Update Jan 20, 2015: Spark 1.2+ includes features such as write ahead logs (WAL) that help to minimize some of the data loss scenarios for Spark Streaming that are described below. See[Improved Fault-tolerance and Zero Data Loss in Spark Streaming](http://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html).

You might have guessed by now that there are indeed a number of unresolved issues in Spark Streaming. I try to summarize my findings below.

On the one hand there are issues due to some confusion about how to correctly read from and write to Kafka, which you can follow in mailing list discussions such as [Multiple Kafka Receivers and Union](http://apache-spark-user-list.1001560.n3.nabble.com/Multiple-Kafka-Receivers-and-Union-td14901.html) and [How to scale more consumer to Kafka stream](http://apache-spark-user-list.1001560.n3.nabble.com/How-to-scale-more-consumer-to-Kafka-stream-td13883.html).

On the other hand there are apparently still some inherent issues in Spark Streaming as well as Spark itself, notably with regard to data loss in failure scenarios. In other words, issues that you do not want to run into in production!

* The current (v1.1) driver in Spark does not recover such raw data that has been received but not processed ([source](https://www.mail-archive.com/user@spark.apache.org/msg10572.html)). Here, your Spark application may lose data under certain conditions. Tathagata Das points out that driver recovery should be fixed in Spark v1.2, which will be released around the end of 2014.
* The current Kafka “connector” of Spark is based on Kafka’s high-level consumer API. One effect of this is that Spark Streaming cannot rely on its KafkaInputDStream to properly replay data from Kafka in case of a downstream data loss (e.g. Spark machines died).
  + Some people even advocate that the current [Kafka connector of Spark should not be used in production](http://markmail.org/message/2lb776ta5sq6lgtw) because it is based on the high-level consumer API of Kafka. Instead Spark should use the simple consumer API (like Storm’s Kafka spout does), which allows you to control offsets and partition assignment deterministically.
* The Spark community has been working on filling the previously mentioned gap with e.g. Dibyendu Bhattacharya’s [kafka-spark-consumer](https://github.com/dibbhatt/kafka-spark-consumer). The latter is a port of Apache Storm’s [Kafka spout](https://github.com/apache/storm/tree/master/external/storm-kafka), which is based on Kafka’s so-called simple consumer API, which provides better replaying control in case of downstream failures.
* Even given those volunteer efforts, the Spark team would prefer to not special-case data recovery for Kafka, as their goal is “to provide strong guarantee, exactly-once semantics in all transformations” ([source](https://www.mail-archive.com/user@spark.apache.org/msg10572.html)), which is understandable. On the flip side it still feels a bit like a wasted opportunity to not leverage Kafka’s built-in replaying capabilities. Tough call!
* [SPARK-1340](https://spark-project.atlassian.net/browse/SPARK-1340): In the case of Kafka input DStreams, receivers are not getting restarted if the worker running the receiver fails. So if a worker dies in production, you will simply miss the data the receiver(s) was/were responsible to retrieve from Kafka.
* See also [Failure of a Worker Node](http://spark.apache.org/docs/latest/streaming-programming-guide.html#failure-of-a-worker-node) for further discussions on data loss scenarios (“lost input data!”) as well as data duplication scenarios (“wrote output data twice!”). Applies to Kafka, too.
* Spark’s usage of the Kafka consumer parameter [auto.offset.reset](http://kafka.apache.org/documentation.html#consumerconfigs) is different from Kafka’s semantics. In Kafka, the behavior of setting auto.offset.reset to “smallest” is that the consumer will automatically reset the offset to the smallest offset when a) there is no existing offset stored in ZooKeeper or b) there is an existing offset but it is out of range. Spark however will *always* remove existing offsets and then start all the way from zero again. This means whenever you restart your application with auto.offset.reset = "smallest", your application will completely re-process all available Kafka data. Doh! See [this discussion](http://apache-spark-user-list.1001560.n3.nabble.com/spark-streaming-and-the-spark-shell-tp3347p3387.html) and [that discussion](http://markmail.org/message/257a5l3oqyftsjxj).
* [SPARK-1341](https://spark-project.atlassian.net/browse/SPARK-1341): Ability to control the data rate in Spark Streaming. This is relevant in so far that if you are already in trouble because of the other Kafka-relatd issues above (e.g. the auto.offset.resetmisbehavior), then what may happen is that your streaming application must or thinks it must re-process a lot of older data. But since there is no built-in rate limitation this may cause your workers to become overwhelmed and run out of memory.

Apart from those failure handling and Kafka-focused issues there are also scaling and stability concerns. Again, please refer to the [Spark and Storm](http://yahoohadoop.tumblr.com/post/98213421641/storm-and-spark-at-yahoo-why-chose-one-over-the-other) talk of Bobby and Tom for further details. Both of them have more experience with Spark than I do.

I also came across [one comment](https://www.mail-archive.com/user@spark.apache.org/msg11505.html) that there may be issues with the (awesome!) G1 garbage collector that is available in Java 1.7.0u4+, but I didn’t run into any such issue so far.

Spark tips and tricks

I compiled a list of notes while I was implementing the example code. This list is by no means a comprehensive guide, but it may serve you as a starting point when implementing your own Spark Streaming jobs. It contains references to the [Spark Streaming programming guide](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html) as well as information compiled from the spark-user mailing list.

General

* When creating your Spark context pay special attention to the configuration that sets the number of cores used by Spark. You must configure enough cores for running both all the required for *receivers*(see below) and for the actual data processing part. In Spark, each receiver is responsible for exactly one input DStream, and each receiver (and thus each input DStream) occies one core – the only exception is when reading from a file stream ([see docs](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#input-dstreams)). So if, for instance, your job reads from 2 input streams but only has access to 2 cores than the data will be read but no processing will happen.
  + Note that in a streaming application, you can create multiple input DStreams to receive multiple streams of data in parallel. I demonstrate such a setup in the example job where we parallelize reading from Kafka.
* You can use [broadcast variables](http://spark.apache.org/docs/1.1.0/programming-guide.html#broadcast-variables) to share common, read-only variables across machines (see also the relevant section in the [Tuning Guide](http://spark.apache.org/docs/1.1.0/tuning.html#broadcasting-large-variables)). In the example job I use broadcast variables to share a) a Kafka producer pool (through which the job writes its output to Kafka) and b) an injection for encoding/decoding Avro data (from Twitter Bijection). [Passing functions to Spark](http://spark.apache.org/docs/1.1.0/programming-guide.html#passing-functions-to-spark).
* You can use [accumulator](http://spark.apache.org/docs/1.1.0/programming-guide.html#accumulators) variables to track global “counters” across the tasks of your streaming job (think: Hadoop job counters). In the example job I use accumulators to track how many total messages the job has been consumed from and produced to Kafka, respectively. If you give your accumulators a name (see link), then they will also be displayed in the Spark UI.
* Do not forget to import the relevant implicits of Spark in general and Spark Streaming in particular:
* // Required to gain access to RDD transformations via implicits.
* import org.apache.spark.SparkContext.\_
* // Required when working on `PairDStreams` to gain access to e.g. `DStream.reduceByKey`
* // (versus `DStream.transform(rddBatch => rddBatch.reduceByKey()`) via implicits.
* //
* // See also http://spark.apache.org/docs/1.1.0/programming-guide.html#working-with-key-value-pairs
* import org.apache.spark.streaming.StreamingContext.toPairDStreamFunctions
* If you’re a fan of [Twitter Algebird](https://github.com/twitter/algebird), then you will like how you can leverage Count-Min Sketch and friends in Spark. Typically you’ll use operations such as reduce or reduceByWindow (cf.[transformations on DStreams](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#transformations-on-dstreams)). The Spark project includes examples for [Count-Min Sketch](https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/TwitterAlgebirdCMS.scala) and[HyperLogLog](https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/TwitterAlgebirdHLL.scala).
* If you need to determine the memory consumption of, say, your fancy Algebird data structure – e.g. Count-Min Sketch, HyperLogLog, or Bloom Filters – as it is being used in your Spark application, then the SparkContext logs might be an option for you. See [Determining Memory Consumption](http://spark.apache.org/docs/1.1.0/tuning.html#determining-memory-consumption).

Kafka integration

Beyond what I already said in the article above:

* You may need to tweak the Kafka consumer configuration of Spark Streaming. For example, if you need to read large messages from Kafka you must increase the fetch.message.max.bytesconsumer setting. You can pass such custom Kafka parameters to Spark Streaming when calling KafkaUtils.createStream(...).

Testing

* Make sure you stop the StreamingContext and/or SparkContext (via stop()) within a finally block or your test framework’s tearDown method, as Spark does not support two contexts running concurrently in the same program (or JVM?). ([source](http://spark.apache.org/docs/1.1.0/programming-guide.html#accumulators))
* In my experience, when using sbt, you want to configure your build to fork JVMs during testing. At least in the case of [kafka-storm-starter](https://github.com/miguno/kafka-storm-starter) the tests must run several threads in parallel, e.g. in-memory instances of ZooKeeper, Kafka, Spark. See [build.sbt](https://github.com/miguno/kafka-storm-starter/blob/develop/build.sbt) for a starting point.
* Also, if you are on Mac OS X, you may want to disable IPv6 in your JVMs to prevent DNS-related timeouts. This issue is unrelated to Spark. See [.sbtopts](https://github.com/miguno/kafka-storm-starter/blob/develop/.sbtopts) for how to do disable IPv6.

Performance tuning

* Make sure you understand the runtime implications of your job if it needs to talk to external systems such as Kafka. You should read the section *Design Patterns for using foreachRDD* in the [Spark Streaming programming guide](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#output-operations-on-dstreams). For instance, my example application uses a pool of Kafka producers to optimize writing from Spark Streaming to Kafka. Here, “optimizing” means sharing the same (few) producers across tasks, notably to reduce the number of new TCP connections being established with the Kafka cluster.
* Use Kryo for serialization instead of the (slow) default Java serialization (see [Tuning Spark](http://spark.apache.org/docs/1.1.0/tuning.html#serialized-rdd-storage)). My example enables Kryo and registers e.g. the Avro-generated Java classes with Kryo to speed up serialization. See [KafkaSparkStreamingRegistrator](https://github.com/miguno/kafka-storm-starter/blob/develop/src/main/scala/com/miguno/kafkastorm/spark/serialization/KafkaSparkStreamingRegistrator.scala). By the way, the use of Kryo is recommended in Spark for the very same reason it is recommended in Storm.
* Configure Spark Streaming jobs to clear persistent RDDs by setting spark.streaming.unpersistto true. This is likely to reduce the RDD memory usage of Spark, potentially improving GC behavior as well. ([source](http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html#memory-tuning))
* Start your P&S tests with storage level MEMORY\_ONLY\_SER (here, RDD are stored as serialized Java objects, one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer like Kryo, but more CPU-intensive to read. This option is often the best for Spark Streaming jobs. For local testing you may want to not use the \*\_2variants (2 = replication factor).

Wrapping up

The full Spark Streaming code is available in [kafka-storm-starter](https://github.com/miguno/kafka-storm-starter/). I’d recommend to begin reading with the [KafkaSparkStreamingSpec](https://github.com/miguno/kafka-storm-starter/blob/develop/src/test/scala/com/miguno/kafkastorm/spark/KafkaSparkStreamingSpec.scala). This spec launches in-memory instances of Kafka, ZooKeeper, and Spark, and then runs the example streaming application I covered in this post.

In summary I enjoyed my initial Spark Streaming experiment. While there are still several problems with Spark/Spark Streaming that need to be sorted out, I am sure the Spark community will eventually be able to address those. I have found the Spark community to be positive and willing to help, and I am looking forward to what will be happening over the next few months.

Given that Spark Streaming still needs some [TLC](http://en.wiktionary.org/wiki/tender_loving_care) to reach Storm’s capabilities in large-scale production settings, would I use it in 24x7 production? Most likely not, with the addendum “not yet”. So where would I use Spark Streaming in its current state right now? Here are two ideas, and I am sure there are even more:

1. It seems a good fit to prototype data flows very rapidly. If you run into scalability issues because your data flows are too large, you can e.g. opt to run Spark Streaming against only a sample or subset of the data.
2. What about combining Storm and Spark Streaming? For example, you could use Storm to crunch the raw, large-scale input data down to manageable levels, and then perform follow-up analysis with Spark Streaming, benefitting from the latter’s out-of-the-box support for many interesting algorithms and computations. use cases.

Thanks to the Spark community for all their great work!

References

* [Spark Streaming + Kafka Integration Guide](https://spark.apache.org/docs/latest/streaming-kafka-integration.html)
* [Deep Dive with Spark Streaming](http://www.slideshare.net/spark-project/deep-divewithsparkstreaming-tathagatadassparkmeetup20130617), by Tathagata Das, Jun 2013
* Mailing list discussions:
  + [Spark Streaming threading model](https://www.mail-archive.com/dev@spark.incubator.apache.org/msg00531.html) – also contains some information on how Spark Streaming pushes input data into blocks
  + [Low Level Kafka Consumer for Spark](http://apache-spark-user-list.1001560.n3.nabble.com/Low-Level-Kafka-Consumer-for-Spark-td11258.html) – lots of information about the current state of Kafka integration in Spark Streaming, known issues, possible remedies, etc.
  + [How are the executors used in Spark Streaming in terms of receiver and driver program?](http://apache-spark-user-list.1001560.n3.nabble.com/How-are-the-executors-used-in-Spark-Streaming-in-terms-of-receiver-and-driver-program-td9336.html)– machines vs. cores vs. executors vs. receivers vs. DStreams in Spark