

Aug 6 2015

Spark and Kafka integration patterns

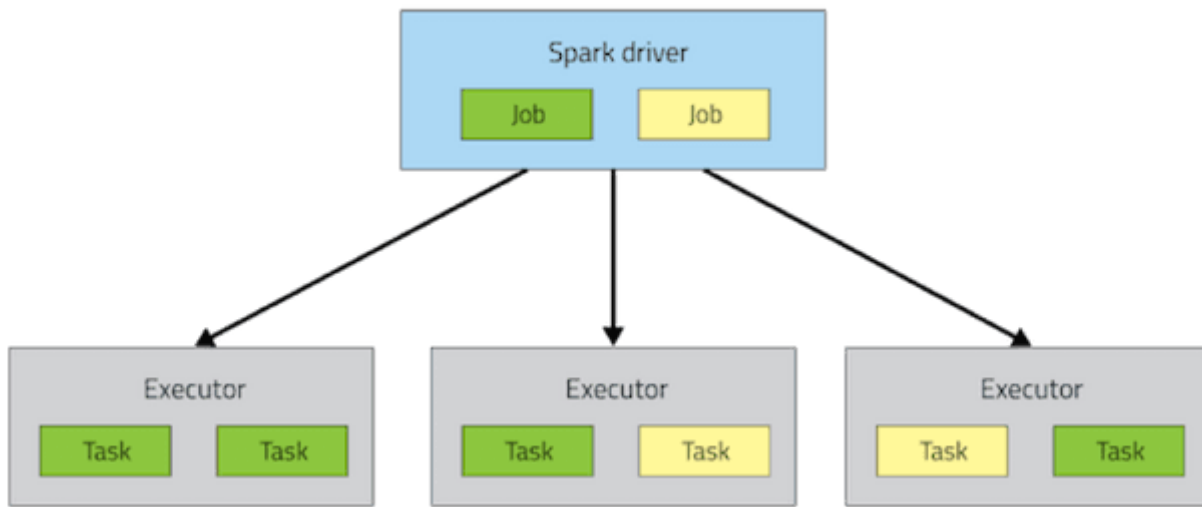
[allegro.tech blog](#) · [Marcin Kuthan](#) · Spark and Kafka integration patterns

Today we would like to share our experience with [Apache Spark](#), and how to deal with one of the most annoying aspects of the framework. This article assumes basic knowledge of Apache Spark. If you feel uncomfortable with the basics of Spark, we recommend you to participate in an excellent [online course](#) prepared by the creators of Spark.

The devil is in the detail

Big Data developers familiar with [MapReduce programming model](#) are usually impressed with elegant, expressive and concise Spark API which complies with functional programming style. Unfortunately, as soon as they start to develop more complex applications, they run into subtle issues related to the distributed nature of Spark's internals. Spark API is only a thin abstraction and, as we all know, abstractions in computer science tend to leak.

The problem we would like to discuss here is the problem of object serialization and lifecycle management in Spark applications. Sooner or later you may observe strange `java.io.NotSerializableException` exceptions in your application stack trace. This happens because some part of your application code is evaluated on the Spark driver, other part on the Spark executors. In the example below, Spark creates two jobs on the driver and delegates work to executors on a cluster of remote nodes. The number of jobs depends on your application business logic and the number of tasks depends on data partitioning.



When you gain some experience with Spark, it should be easy to look at the code, and tell where it will be eventually executed. Look at the code snippet below.

```
dstream.foreachRDD { rdd =>
  val where1 = "on the driver"
  rdd.foreach { record =>
    val where2 = "on different executors"
  }
}
```

The outer loop against `rdd` is executed locally on the driver. [RDD](#) (Resilient Distributed Dataset) is a structure where data is transparently distributed to cluster nodes. The only place you can access `rdd` is the driver.

But the inner loop will be evaluated in a distributed manner. RDD will be partitioned and inner loop iterates over subset of `rdd` elements on every Spark executor.

Spark uses Java (or [Kryo](#)) serialization to send application objects from the driver to the executors. At first you will try to add `scala.Serializable` or `java.io.Serializable` marker interface to all of your application classes to avoid weird exceptions. But this blind approach has at least two disadvantages:

- There might be a performance penalty when complex object graph is serialized and sent to a dozen of remote cluster nodes. It might be mitigated by using [Spark broadcast variables](#), though.
- Not everything is serializable, e.g: TCP socket cannot be serialized and sent between nodes.

Naive attempt to integrate Spark Streaming and [Kafka](#) producer

After this introduction we are ready to discuss the problem we had to solve in our application. The application is a long running Spark Streaming job deployed on [YARN](#) cluster. The job receives unstructured data from Apache Kafka, validates data, converts it into [Apache Avro](#) binary format and publishes it back to another Apache Kafka topic.

Our very first attempt was very similar to the code presented below. Can you see the problem?

```
dstream.foreachRDD { rdd =>
  val producer = createKafkaProducer()
  rdd.foreach { message =>
    producer.send(message)
  }
  producer.close()
}
```

The producer is created (and disposed of) once on the driver but the message is sent to an executor. The producer keeps open sockets to the Kafka brokers so it cannot be serialized and sent over the network.

The producer initialization and disposal code can be moved to the inner loop as presented below.

```
dstream.foreachRDD { rdd =>
  rdd.foreach { message =>
    val producer = createKafkaProducer()
    producer.send(message)
    producer.close()
  }
}
```

Kafka producer is created and closed on an executor and does not need to be serialized. But it does not scale at all, the producer is created and closed for every single message. Establishing a connection to the cluster takes time. It is a much more time consuming operation than opening plain socket connection, as Kafka producer needs to discover leaders for all partitions. Kafka Producer itself is a “heavy” object, so you can also expect high CPU utilization by the JVM garbage collector.

The previous example could be improved by using `foreachPartition` loop. The partition of records is always processed by a Spark task on a single executor using single JVM. You can safely share a thread-safe Kafka producer instance. But in our scale (20k messages / second, 64 partitions, 2 seconds batch) it did not scale as well. Kafka producer was created and closed 64 times every 2 seconds and sent only 625 messages on average.

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
```

```

val producer = createKafkaProducer()
partitionOfRecords.foreach { message =>
    connection.send(message)
}
producer.close()
}
}

```

If you are interested in more details about above optimizations, look at the design patterns published in the official [Spark Streaming documentation](#). Perhaps the patterns were prepared for regular database connection pools, when single database connection cannot be shared between clients concurrently. But a Kafka producer is a thread-safe object, so it can be easily shared by multiple Spark tasks within the same JVM. Moreover, Kafka producer is asynchronous and buffers data heavily before sending. How to return a Kafka producer to the pool, if it is still processing data?

Problem solved

Finally we ended up with a solution based on Scala lazy evaluation (you can do it in Java as well). The application code is very similar to our first naive attempt when Kafka producer is managed fully on the driver. To ensure that `kafkaSink` object is sent only once we use the Spark broadcast mechanism.

```

val kafkaSink = sparkContext.broadcast(KafkaSink(conf))

dstream.foreachRDD { rdd =>
    rdd.foreach { message =>
        kafkaSink.value.send(message)
    }
}

```

`KafkaSink` class is a smart wrapper for a Kafka producer. Instead of sending the producer itself, we send only a “recipe” how to create it in an executor. The class is serializable because Kafka producer is initialized just before first use on an executor. Constructor of `KafkaSink` class takes a function which returns Kafka producer lazily when invoked. Once the Kafka producer is created, it is assigned to `producer` variable to avoid initialization on every `send()` call.

```

class KafkaSink(createProducer: () => KafkaProducer[String, String]) extends Serializable {

    lazy val producer = createProducer()

    def send(topic: String, value: String): Unit = producer.send(new ProducerRecord(
    })

```

```
object KafkaSink {  
  def apply(config: Map[String, Object]): KafkaSink = {  
    val f = () => {  
      new KafkaProducer[String, String](config)  
    }  
    new KafkaSink(f)  
  }  
}
```

Before production deployment, `KafkaSink` needs to be improved a little. We have to close the Kafka producer before an executor JVM is closed. Without it, all messages buffered internally by Kafka producer would be lost.

```
object KafkaSink {  
  def apply(config: Map[String, Object]): KafkaSink = {  
    val f = () => {  
      val producer = new KafkaProducer[String, String](config)  
  
      sys.addShutdownHook {  
        producer.close()  
      }  
  
      producer  
    }  
    new KafkaSink(f)  
  }  
}
```

The function `f` is evaluated on an executor so we can register shutdown hook there to close the Kafka producer. The shutdown hook will be executed before the executor JVM is closed, and Kafka producer will flush all buffered messages.

Summary

We hope that this post will be helpful for others looking for a better way to integrate Spark Streaming and Apache Kafka. While this article refers to Kafka, the approach could be easily adapted to other cases where a limited number of instances of “heavy”, non-serializable objects should be created.

**Marcin Kuthan**

Marcin is a Big Data engineer working at Allegro Group since 2014. He genuinely likes programming, problem solving and learning new trends in software methodologies, tools and languages.


[see post by Marcin Kuthan](#) ▶

Share this post



21 Comments

allegro.tech

 Login ▼ Recommend 9 Share

Sort by Newest ▼



Join the discussion...

**Prakash tirumalaseti** • 4 days ago

Thank you for very informative post.

^ | ▼ • Reply • Share ›

**Kumar** • 6 days ago

Can someone share the same code in Java?

^ | ▼ • Reply • Share ›

**Tim** • 2 months ago

Excellent post, thank you! FWIW, just a minor edit, but I'm sure everyone gets the point:

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    val producer = createKafkaProducer()
    partitionOfRecords.foreach { message =>
      connection.send(record))
    }
    producer.close()
  }
}
```

Should be:

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    val producer = createKafkaProducer()
    partitionOfRecords.foreach { message =>
      producer.send(message)
    }
  }
}
```

```

producer.send(message))
}
producer.close()
}
}

```

^ | v • Reply • Share ›



mkuthan → Tim • 2 months ago

Thank you, Tim. Code snippet has been fixed :-)

^ | v • Reply • Share ›



Tim → mkuthan • 2 months ago

Most welcome :)

By the way, I was curious, have you tried putting the lazy val in the KafkaSink "object" itself (or a case class), and just dropping the KafkaSink "class" altogether?

Something like:

```

case class KafkaWrapper(config: Properties) {

  lazy val producer = {
    val kp = new KafkaProducer[String, String](config)

    sys.addShutdownHook {
      kp.close()
    }

    kp
  }

  def send(topic: String, value: String): Unit = {
    producer.send(new ProducerRecord(topic, value))
  }
}

```

^ | v • Reply • Share ›



mkuthan → Tim • 2 months ago

It will work for sure, but KafkaWrapper is not testable anymore. You can not mock Kafka producer if the producer is created in KafkaWrapper constructor.

^ | v • Reply • Share ›



Artur Bartnik • 5 months ago

thanks, great tip, in my case more important is knowledge that data have been saved to kafka before I go to next batch, so I made a little change

```

def send(topic: String, value: String): Unit = {
  producer.send(new ProducerRecord(topic, value))
}

```

```
der send(topic: String, value: String): Unit = producer.send(new ProducerRecord(topic,
value)).get(SEND_TIMEOUT_MS, TimeUnit.MILLISECONDS)
```

^ | v • Reply • Share ›



sunny • 5 months ago

Hi, I rewrote my streaming project with the method in the post. The job could receive from and write to kafka just fine when I first wrote data to kafka. But the second time I wrote large records, like 400 million, this exception occurred:

```
WARN scheduler.ReceiverTracker: Error reported by receiver for stream 0: Error in block
pushing thread - java.io.NotSerializableException:
org.apache.kafka.clients.producer.KafkaProducer
```

Serialization stack:

```
- object not serializable (class: org.apache.kafka.clients.producer.KafkaProducer, value:
org.apache.kafka.clients.producer.KafkaProducer@66aa52fa)
```

```
- field (class: com.user.util.KafkaSink, name: producer, type: class
org.apache.kafka.clients.producer.KafkaProducer)
```

```
- object (class com.user.util.KafkaSink, com.user.util.KafkaSink@19a09800)
```

The thing is this exception only occurred when the data is large. If I wrote 10 thousands each time, everything just works fine.

Hope someone can help me with this. Thanks!

^ | v • Reply • Share ›



mkuthan → sunny • 5 months ago

Hi Sunny

It looks that producer field in KafkaSink has been initialized on the driver not on the executors. You should debug why producer is initialized so early or you could mark producer field with `@transient` annotation. Even if the producer is initialized on the driver, it will not be serialized. And finally the producer will be initialized on the executor again because transient fields are not initialized after deserialization.

1 ^ | v • Reply • Share ›



William Braik → mkuthan • 14 days ago

Thanks. I had the same issue (although it was not related to the size of the batch) -- annotating the field as transient did the trick.

```
@transient lazy val producer = {
val conf = ...
val producer = new KafkaProducer[String, String](conf)
```

```
...
```

```
producer
}
```

^ | v • Reply • Share ›



This comment was deleted.



mkuthan → Guest • 5 months ago

Hi MD

Be aware that Cloudera's spark-kafka-writer uses deprecated Scala Kafka client and does not publish messages to Kafka in reliable way. Please refer to second part of this blog post <http://mkuthan.github.io/blog/...> How to publish processing results from Spark Streaming to Kafka in reliable way.

^ | v • Reply • Share ›



Mark Kerzner • 8 months ago

Thank you, it is useful

^ | v • Reply • Share ›



vijayrc • 9 months ago

thanks for the post, very helpful.

^ | v • Reply • Share ›



Radek • a year ago

Dzieki, super jasno wyjasnione.

^ | v • Reply • Share ›



Marius • a year ago

Nice post, I just have two questions:

- Does the Kafka topic also have 64 partitions?
- Is a single producer as fast as multiple producers? As far as I understand it only uses one thread per broker.

^ | v • Reply • Share ›



mkuthan → Marius • a year ago

Hi Marius

Thank you for your comment :-)

- I did not mention in the post, we use direct stream integration approach. So number of Kafka partitions equals number of Spark partitions.

- Indeed, there is one Kafka producer per broker and Kafka producer shares single thread for all I/O operations. It could be a bottleneck if your streaming logic is much faster then sending messages to Kafka brokers, or for executors with many cores. We also found that Kafka compression is the one of the most important Kafka producer performance factor, especially when gzip compression is enabled.

To mitigate this issue, you could create pool of Kafka producers in the f() recipe function. The pool might be implemented as simple array of Kafka producers with round robin selector on top. Much, much simpler than regular DB connection pool like dbcp or c3p0.

^ | v • Reply • Share ›



Marius → mkuthan • a year ago

Hi Marcin,

thanks for your answer. Good idea with the pooling, I will try that. I will also switch to the direct approach.

Currently I'm experiencing an increase in processing time with 16 partitions on two brokers already at rather low data input rates.

^ | v • Reply • Share ›



Atul Kulkarni • a year ago

Hi - I am having tough time making this work on Spark on a yarn cluster. The producer creation code runs on the driver and not on the executor - I have a log statement in there which shows out put in the stderr of driver and not on every executor's stderr. What in the above code guarantees creation of KafaSink() singleton on the executor JVM?

^ | v • Reply • Share ›



mkuthan → Atul Kulkarni • a year ago

Hi Atul

We deploy our spark jobs using YARN cluster mode. The producer is lazily created on the executors, perhaps there is some tiny difference in your code. Please check that you don't touch function responsible for Kafka producer initialisation "f()" on Spark driver. You can throw an exception just after new KafkaProducer() line, and check the stacktrace for place where the f() function was called.

Look at the KafkaSink constructor, it takes the function how to create producer, let's call it recipe. And this recipe is evaluated lazily before first use, there is no point to send message using Kafka producer on the driver, so the producer is initialised on the executors.

I hope that my explanations will be helpful for you.

^ | v • Reply • Share ›



Olivier NOUGUIER • a year ago

Proudly built by **allegro** Tech **engineers**

Follow us:

