Kafka, Spark and Avro - Part 2, Consuming Kafka messages with Spark

2016-03-03

This post is the second post in a series in which we will learn how to send messages in the Avro format into Kafka so that they can be consumed by Spark Streaming. As a reminder there will be 3 posts:

1. Kafka 101: producing and consuming plain-text messages with standard Java code
2. Kafka + Spark: consuming plain-text messages from Kafka with Spark Streaming
3. Kafka + Spark + Avro: same as 2. with Avro-encoded messages

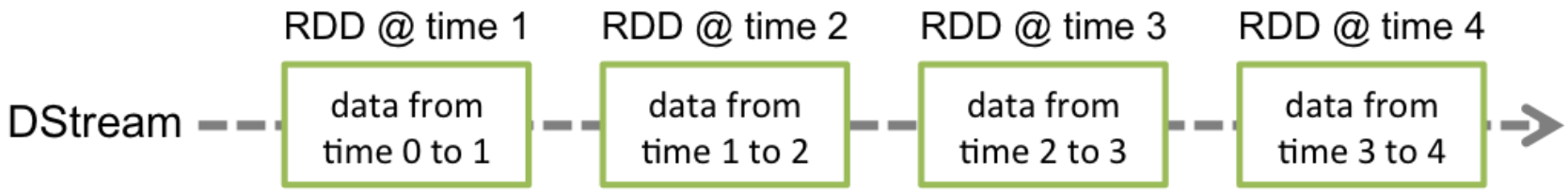
In this post, we will reuse the Java producer we created in the first post to send messages into Kafka. This time, however, we will consume the messages with Spark.

Spark Streaming

Kafka is a messaging system, meaning messages are supposed to be consumed as they arrive, i.e. with a streaming technology. That's where Spark Streaming comes into play.

If you don't know Spark Streaming, this is basically an extension of Spark to implement streaming applications on top of a batch engine. Events - messages - are not processed one by one but, instead, they are accumulated over a short period of time - say a few seconds - and then processed as **micro-batches**. In Spark Streaming's terminology, the flow of messages is a **Discretized Streams**, which materializes into a `DStream` in the API.

Here is how it looks. The events will be grouped into small batches, one RDD per period of time.



The structure of a Spark Streaming application always looks the same:

- Initialize the SparkContext.
- Initialize the StreamingContext with a duration.
- Describe your processing pipeline.
- Start the StreamingContext and keep the application alive.

```
package com.ipponusa;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.streaming.Duration;
import org.apache.spark.streaming.api.java.JavaStreamingContext;

public class SparkStringConsumer {


    public static void main(String[] args) {

        SparkConf conf = new SparkConf()
                .setAppName("kafka-sandbox")
                .setMaster("local[*]");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaStreamingContext ssc = new JavaStreamingContext(sc, new Duration(2000));

        // TODO: processing pipeline
```

```
        // TODO: processing pipeline

        ssc.start();
        ssc.awaitTermination();
    }
}
```

I usually prefer writing Scala code when it comes to using Spark, but this time I will write Java code to be consistent with the previous post.

We have configured the period to 2 seconds (2000 ms). Notice that Spark Streaming is not designed for periods shorter than about half a second. If you need a shorter delay in your processing, try Flink or Storm instead.

Spark Streaming with Kafka

Spark Streaming has a connector for Kafka. First thing you need to do is add the dependency:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka_2.10</artifactId>
    <version>1.6.0</version>
</dependency>
```

Spark Streaming's Kafka integration guide provides two ways to link Spark Streaming to Kafka: the **receiver-based approach** and the **direct approach**. All you need to know is that the direct approach is newer, more efficient, and easier to understand. That's the one we will use.

Here is the code we need to write to setup the Kafka connector:

```
Map<String, String> kafkaParams = new HashMap<>();
kafkaParams.put("metadata.broker.list", "localhost:9092");
Set<String> topics = Collections.singleton("mytopic");
```

```
JavaPairInputDStream<String, String> directKafkaStream = KafkaUtils.createDirectStream(ssc,
        String.class, String.class, StringDecoder.class, StringDecoder.class, kafkaParams, topics);
```

You have to define the connection to Kafka, the topic, the types and deserializers for the key and the value, and finally the processing to apply.

Here is our processing code:

```
directKafkaStream.foreachRDD(rdd -> {
    System.out.println("--- New RDD with " + rdd.partitions().size()
            + " partitions and " + rdd.count() + " records");
    rdd.foreach(record -> System.out.println(record._2));
});
```

If you launch this application, you should see a message every 2 seconds:

```
--- New RDD with 2 partitions and 0 records
--- New RDD with 2 partitions and 0 records
...
```

Now, go back to the `SimpleStringProducer` (see the previous post) and modify the main loop so that it posts a message every 100 ms:

```
for (int i = 0; i < 1000; i++) {
    ProducerRecord<String, String> record = new ProducerRecord<>("mytopic", "value-" + i);
    producer.send(record);
    Thread.sleep(250);
}
```

If you launch the producer, the consumer should now display the incoming messages:

```
--- New RDD with 2 partitions and 3 records
value-1
value-0
value-2
--- New RDD with 2 partitions and 7 records
value-3
value-5
value-7
value-9
value-4
value-6
value-8
--- New RDD with 2 partitions and 8 records
value-11
value-10
value-13
...
```

You should get about 8 messages per micro-batch (2 seconds divided by 250 ms).

Now, you may be wondering why there are 2 partitions in the RDD and why the messages are not in the same order as they were published.

Spark Streaming's direct approach makes a **one-to-one mapping between partitions in a Kafka topic and partitions in a Spark RDD**. Because we left off the previous post with a Kafka topic with 2 partitions, we now have a Spark RDD with 2 partitions.

The answer to the second question is related to the partitioning. When publishing to Kafka, our messages got dispatched to the 2 partitions in a round-robin fashion (partition 0 would hold messages 0, 1, 3… while partition 1 would hold messages 2, 4, 6…). Then, when they are being consumed by Spark, each RDD partition is processed in parallel by separate threads. This explains why the messages seem to be in a random order. In fact, the ordering within each Kafka partition is preserved.

Conclusion

We've seen in this post that we can use Spark to process a stream of messages coming from Kafka. The 2 technologies fit well together as you can increase or decrease the size of either your Kafka cluster or Spark cluster to adapt to the load on your system.

In the next post, we will see how to send and receive messages in a more robust format than plain strings of texts, namely by using Avro.