Kafka, Spark and Avro - Part 3, Producing and consuming Avro messages

2016-03-04

This post is the third and last post in a series in which we learn how to send messages in the Avro format into Kafka so that they can be consumed by Spark Streaming. As a reminder there are 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 and the Spark consumer we created in the previous posts. Instead of dealing with plain-text messages, though, we will serialize our messages with Avro. That will allow us to send much more complex data structures over the wire.

Avro

Avro is a data serialization system and, as Spark and Kafka, it is an open source Apache project.

Avro relies on schemas so as to provide efficient serialization of the data. The schema is written in JSON format and describes the fields and their types. There are 2 cases:

- when serializing to a file, the schema is written to the file
- in RPC - such as between Kafka and Spark - both systems should know the schema prior to exchanging data, or they could exchange the schema during the connection handshake.

What makes Avro handy is that you do not need to generate data classes. In this post, we will actually not even define a data class at all but, instead, use generic records (probably not the most efficient, I would agree).

Let's start by adding a dependency to Avro:

```
<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.8.0</version>
</dependency>
```

We will define the schema as follows:

```
{
    "fields": [
        { "name": "str1", "type": "string" },
        { "name": "str2", "type": "string" },
        { "name": "int1", "type": "int" }
    ],
    "name": "myrecord",
    "type": "record"
}
```

We are basically defining a record as being an object with two text fields named "str1" and "str2" and one integer field named "int1".

This schema can be instantiated as follows:

```
Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(USER_SCHEMA);
```

Here, `USER_SCHEMA` is the JSON listed above as a Java String.

Bijection, by Twitter

Now, we could use Avro's API to serialize and deserialize objects but this is not the most friendly API. Instead, we will use Bijection which makes it easy to convert objects back and forth.

```
<dependency>
    <groupId>com.twitter</groupId>
    <artifactId>bijection-avro_2.10</artifactId>
    <version>0.9.2</version>
</dependency>
```

We can now create an `Injection` which is an object that can make the conversion in one way or the other:

```
Injection<GenericRecord, byte[]> recordInjection = GenericAvroCodecs.toBinary(schema);
```

This allows us to create a record and serialize it:

```
GenericData.Record record = new GenericData.Record(schema);
avroRecord.put("str1", "My first string");
avroRecord.put("str2", "My second string");
avroRecord.put("int1", 42);

byte[] bytes = recordInjection.apply(record);
```

Or the other way around:

```
GenericRecord record = recordInjection.invert(bytes).get();

String str1 = (String) record.get("str1");
String str2 = (String) record.get("str2");
int int1 = (int) record.get("int1");
```

The producer

We can easily update the code of the producer to send messages which payload is an object serialized with Avro.

Anywhere where the value was defined as being a `String`, we will now use `byte[]`. Also, we need to switch from the `StringSerializer` to the `ByteArraySerializer` (this change only applies to the value, we can keep the `StringSerializer` for the key).

The code becomes:

```
package com.ipponusa.avro;

import com.twitter.bijection.Injection;
import com.twitter.bijection.avro.GenericAvroCodecs;
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class SimpleAvroProducer {

    public static final String USER_SCHEMA = "{"
            + "\"type\":\"record\","
            + "\"name\":\"myrecord\","
            + "\"fields\":["
            + "  { \"name\":\"str1\", \"type\":\"string\" },"

            + "  { \"name\":\"str2\", \"type\":\"string\" },"
            + "  { \"name\":\"int1\", \"type\":\"int\" }"
            + "]}";
```

```java
    public static void main(String[] args) throws InterruptedException {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.ByteArraySerializer");

        Schema.Parser parser = new Schema.Parser();
        Schema schema = parser.parse(USER_SCHEMA);
        Injection<GenericRecord, byte[]> recordInjection = GenericAvroCodecs.toBinary(schema);

        KafkaProducer<String, byte[]> producer = new KafkaProducer<>(props);

        for (int i = 0; i < 1000; i++) {
            GenericData.Record avroRecord = new GenericData.Record(schema);
            avroRecord.put("str1", "Str 1-" + i);
            avroRecord.put("str2", "Str 2-" + i);
            avroRecord.put("int1", i);

            byte[] bytes = recordInjection.apply(avroRecord);

            ProducerRecord<String, byte[]> record = new ProducerRecord<>("mytopic", bytes);
            producer.send(record);

            Thread.sleep(250);

        }

        producer.close();
    }
}
```

(On a side node, I wish Java had multi-lines String literals...)

The consumer

In the same way we updated the producer to send binary messages, we can update the Spark consumer to receive these messages.

We don't want the Kafka connecter to decode the values as strings. Therefore, we need to replace the `StringDecoder` with the `DefaultDecoder` which will basically give us the raw array of bytes.

Here is a *basic* implementation:

```
package com.ipponusa.avro;

import com.twitter.bijection.Injection;
import com.twitter.bijection.avro.GenericAvroCodecs;
import kafka.serializer.DefaultDecoder;
import kafka.serializer.StringDecoder;
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericRecord;
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.JavaPairInputDStream;
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import org.apache.spark.streaming.kafka.KafkaUtils;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class SparkAvroConsumer {


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


        JavaPairInputDStream<String, byte[]> directKafkaStream = KafkaUtils.createDirectStream(ssc,
                String.class, byte[].class, StringDecoder.class, DefaultDecoder.class, kafkaParams, topics);

        directKafkaStream.foreachRDD(rdd -> {
            rdd.foreach(avroRecord -> {
                Schema.Parser parser = new Schema.Parser();
                Schema schema = parser.parse(SimpleAvroProducer.USER_SCHEMA);
                Injection<GenericRecord, byte[]> recordInjection = GenericAvroCodecs.toBinary(schema);
                GenericRecord record = recordInjection.invert(avroRecord._2).get();

                System.out.println("str1= " + record.get("str1")
                        + ", str2= " + record.get("str2")
                        + ", int1=" + record.get("int1"));
            });
        });

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

You have probably spotted the issue in this implementation: the parsing of the schema and the creation of the injection object are done for each and every record. This is obviously adding unnecessary computations on your Spark jobs, thus limiting the ability to handle large volumes of messages.

We could be tempted to define the injection within the main method of our job:

```
Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(SimpleAvroProducer.USER_SCHEMA);
Injection<GenericRecord, byte[]> recordInjection = GenericAvroCodecs.toBinary(schema);

directKafkaStream.foreachRDD(rdd -> {
    rdd.foreach(avroRecord -> {
        GenericRecord record = recordInjection.invert(avroRecord._2).get();
        System.out.println("str1= " + record.get("str1")
                + ", str2= " + record.get("str2")
                + ", int1=" + record.get("int1"));
    });
});
```

This wouldn't work, though. We would receive the following error:

```
Exception in thread "main" org.apache.spark.SparkException: Task not serializable
...
Caused by: java.io.NotSerializableException: org.apache.avro.generic.GenericDatumReader
...
```

This is a typical problem with Spark: the closure (the lamba expression) references the `recordInjection` object and it would therefore be serialized to be sent to all the workers. In this case, Avro (de)serializer objects are not serializable.

We can work around this issue by defining the `recordInjection` object as a `static` member of our class:

```
private static Injection<GenericRecord, byte[]> recordInjection;

static {
    Schema.Parser parser = new Schema.Parser();
    Schema schema = parser.parse(SimpleAvroProducer.USER_SCHEMA);
    recordInjection = GenericAvroCodecs.toBinary(schema);
}

public static void main(String[] args) {
    ...
```

In case you are not familiar with this syntax, we are using a static initialization block. An instance of the `recordInjection` object will be created per JVM, i.e. we will have one instance per Spark worker.

We can now refine our code by introducing a `map` transformation to deserialize our records:

```
directKafkaStream
        .map(message -> recordInjection.invert(message._2).get())
        .foreachRDD(rdd -> {
            rdd.foreach(record -> {
                System.out.println("str1= " + record.get("str1")
                        + ", str2= " + record.get("str2")
                        + ", int1=" + record.get("int1"));
            });
        });
```

This code is now more efficient as well as being more readable.

Conclusion

In these 3 posts, we have seen how to produce messages encoded with Avro, how to send them into Kafka, how to consume them with Spark, and finally how to decode them. This allows us to build a powerful streaming platform, one that can scale by adding nodes either to the Kafka or the Spark cluster.

One thing we have not covered is how to share Avro schemas and how to handle changes of schemas. That's where the Confluent Platform comes into play with its Schema Registry.

The complete code can be found on GitHub.