Stateless Processing

The simplest form of stream processing requires no memory of previously seen events. Each event is consumed, processed, and subsequently forgotten. This paradigm is called *stateless processing*, and Kafka Streams includes a rich set of operators for working with data in a stateless way.

In this chapter, we will explore the *stateless operators* that are included in Kafka Streams, and in doing so, we'll see how some of the most common stream processing tasks can be tackled with ease. The topics we will explore include:

- Filtering records
- Adding and removing fields
- Rekeying records
- Branching streams
- Merging streams
- Transforming records into one or more outputs
- Enriching records, one at a time

We'll take a tutorial-based approach for introducing these concepts. Specifically, we'll be streaming data about cryptocurrencies from Twitter and applying some stateless operators to convert the raw data into something more meaningful: investment signals. By the end of this chapter, you will understand how to use stateless operators in Kafka Streams to enrich and transform raw data.

Before we jump into the tutorial, let's get a better frame of reference for what stateless processing is by comparing it to the other form of stream processing: *stateful processing*.

Stateless Versus Stateful Processing

One of the most important things you should consider when building a Kafka Streams application is whether or not your application requires stateful processing. The following describes the distinction between stateless and stateful stream processing:

- In *stateless applications*, each event handled by your Kafka Streams application is processed independently of other events, and only *stream* views are needed by your application. In other words, your application treats each event as a self-contained insert and requires no memoryof previously seen events.
- *Stateful applications*, on the other hand, need to remember information about previously seen events *in one or more steps* of your processor topology, usually for the purpose of aggregating, windowing, or joining event streams. These applications are more complex under the hood since they need to track additional data, or *state*.

In the high-level DSL, the type of stream processing application you ultimately build boils down to the individual *operators* that are used in your topology. Operators are stream processing functions (e.g., `filter`, `map`, `flatMap`, `join`, etc.) that are applied to events as they flow through your topology. Some operators, like `filter`, are considered *stateless* because they only need to look at the current record to perform an action (in this case, `filter` looks at each record individually to determine whether ornot the record should be forwarded to downstream processors). Other operators, like `count`, are *stateful* since they require knowledge of previous events (`count` needs toknow how many events it has seen so far in order to track the number of messages).

If your Kafka Streams application requires *only* stateless operators (and therefore does not need to maintain any memory of previously seen events), then your application is considered *stateless*. However, if you introduce one or more stateful operators, regardless of whether or not your application also uses stateless operators, then your application is considered *stateful*. The added complexity of stateful applications warrants additional considerations with regards to maintenance, scalability, and fault tolerance.

Introducing Our Tutorial: Processing a Twitter Stream

In this tutorial, we will explore the use case of algorithmic trading. Sometimes called *high-frequency trading* (HFT), this lucrative practice involves building software to evaluate and purchase securities automatically, by processing and responding tomany types of market signals with minimal latency.

To assist our fictional trading software, we will build a stream processing application that will help us gauge market sentiment around different types of cryptocurrencies (Bitcoin, Ethereum, Ripple, etc.), and use these sentiment scores as investment/divestment signals in a custom trading algorithm. Since millions of people use Twitter to share their thoughts on cryptocurrencies and other topics, we will use Twitter as the data source for our application.

Before we get started, let's look at the steps required to build our stream processing application. We will then use these requirements to design a processor topology, which will be a helpful guide as we build our stateless Kafka Streams application. The key concepts in each step are italicized:

1. Tweets that mention certain digital currencies (#bitcoin, #ethereum) should be consumed from a source topic called `tweets`:
    - Since each record is JSON-encoded, we need to figure out how to properly *deserialize* these records into higher-level data classes.
    - Unneeded fields should be removed during the deserialization process to simplify our code. Selecting only a subset of fields to work with is referred to as *projection*, and is one of the most common tasks in stream processing.
2. Retweets should be excluded from processing. This will involve some form of data *filtering*.
3. Tweets that aren't written in English should be *branched* into a separate stream for translating.
4. Non-English tweets need to be translated to English. This involves *mapping* one input value (the non-English tweet) to a new output value (an English-translated tweet).
5. The newly translated tweets should be *merged* with the English tweets stream to create one unified stream.
6. Each tweet should be enriched with a sentiment score, which indicates whether Twitter users are conveying positive or negative emotion when discussing certain digital currencies. Since a single tweet could mention multiple crypto currencies, we will demonstrate how to convert each input (tweet) into a variable number of outputs using a `flatMap` operator.
7. The enriched tweets should be serialized using Avro, and written to an output topic called `crypto-sentiment`. Our fictional trading algorithm will read from this topic and make investment decisions based on the signals it sees.

Now that the requirements have been captured, we can design our processor topology. Figure 3-1 shows what we'll be building in this chapter and how data will flow through our Kafka Streams application.

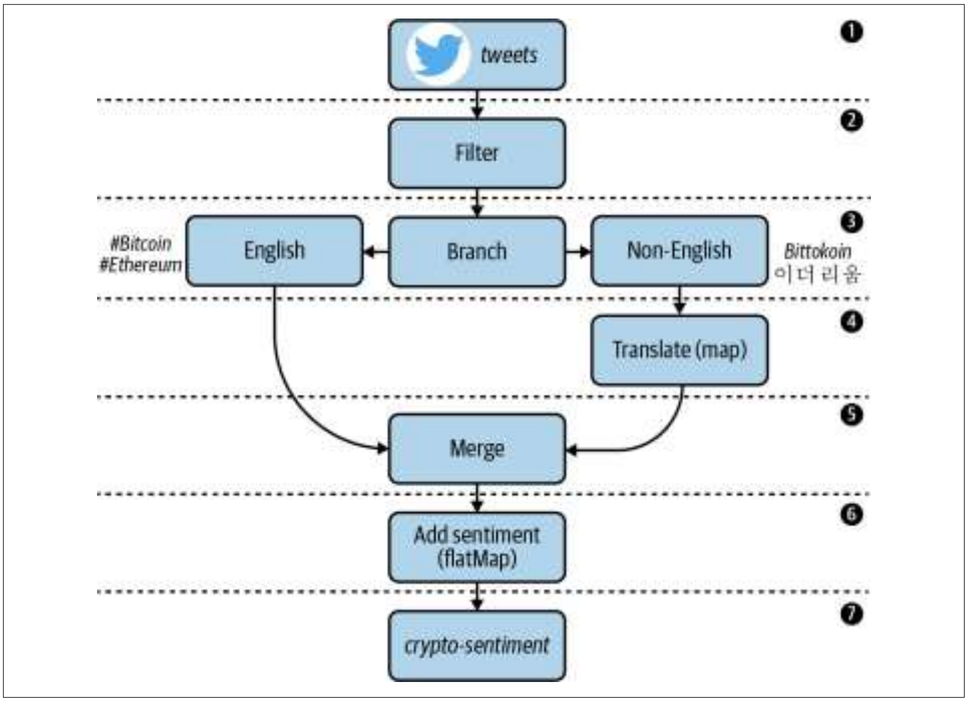

Figure 3-1. The topology that we will be implementing for our tweet enrichment application

With our topology design in hand, we can now start implementing our Kafka Streams application by working our way through each of the processing steps (labeled1–7) in Figure 3-1. We will start by setting up our project, and then move on to the first step in our topology: streaming tweets from the source topic.

Project Setup

The code for this chapter is located at *https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git*.

If you would like to reference the code as we work our way through each topology step, clone the repository and change to the directory containing this chapter's tutorial. The following command will do the trick:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-03/crypto-sentiment
```

You can build the project anytime by running the following command:

```
$ ./gradlew build --info
```

Now that our project is set up, let's start creating our Kafka Streams application.

Adding a KStream Source Processor

All Kafka Streams applications have one thing in common: they consume data from one or more source topics. In this tutorial, we only have one source topic: `tweets`. This topic is populated with tweets from the Twitter source connector, which streams tweets from Twitter's streaming API and writes JSON-encoded tweet records to Kafka. An example tweet value is shown in Example 3-1.

Example 3-1. Example record value in the tweets source topic

```
{
    "CreatedAt": 1602545767000,
    "Id": 1206079394583924736,
    "Text": "Anyone else buying the Bitcoin dip?",
    "Source": "",
    "User": {
        "Id": "123",
        "Name": "Mitch",
        "Description": "",
        "ScreenName": "timeflown",
        "URL": "https://twitter.com/timeflown",
        "FollowersCount": "1128",
        "FriendsCount": "1128"
    }
}
```

Now that we know what the data looks like, the first step we need to tackle is getting the data from our source topic into our Kafka Streams application. As you can see in the following code block, adding a `KStream` source processor in Kafka Streams is simple and requires just a couple of lines of code:

```
StreamsBuilder builder = new StreamsBuilder(); ❶

KStream<byte[], byte[]> stream = builder.stream("tweets"); ❷
```

❶ When using the high-level DSL, processor topologies are built using a `StreamsBuilder` instance.

❷ `KStream` instances are created by passing the topic name to the `StreamsBuilder.stream` method. The `stream` method can optionally accept additional parameters.

One thing you may notice is that the `KStream` we just created is parameterized with `byte[]` types:

```
KStream<byte[], byte[]>
```

The `KStream` interface leverages two generics: one for specifying the type of keys (`K`) in our Kafka topic and the other for specifying the type of values (`V`). If we were to peel back the floorboards in the Kafka Streams library, we would see an interface that looks like this:

```
public interface KStream<K, V> {
  // omitted for brevity
}
```

Therefore, our `KStream` instance, which is parameterized as `KStream<byte[], byte[]>`, indicates that the record keys and values coming out of the `tweets` topic arebeing encoded as byte arrays. However, we just mentioned that the tweet records are actually encoded as JSON objects by the source connector (see Example 3-1), so whatgives?

Kafka Streams, *by default*, represents data flowing through our application as byte arrays. This is due to the fact that Kafka itself stores and transmits data as raw byte sequences, so representing the data as a byte array will always work (and is therefore a sensible default). Storing and transmitting raw bytes makes Kafka flexible because it doesn't impose any particular data format on its clients, and also fast, since it requires less memory and CPU cycles on the brokers to transfer a raw byte stream over the network. However, this means that Kafka clients, including Kafka Streams applications, are responsible for serializing and deserializing these byte streams in order to work with higher-level objects and formats, including strings (delimited or non-delimited), JSON, Avro, Protobuf, etc.

Before we address the issue of deserializing our tweet records into higher-level objects, let's add the additional boilerplate code needed to run our Kafka Streams application. For testability purposes, it's often beneficial to separate the logic for building a Kafka Streams topology from the code that actually runs the application. So the boilerplate code will include two classes. First, we'll define a class for building our Kafka Streams topology, as shown in Example 3-2.

Example 3-2. A Java class that defines our Kafka Streams topology

```
class CryptoTopology {

  public static Topology build() {
    StreamsBuilder builder = new StreamsBuilder();

    KStream<byte[], byte[]> stream = builder.stream("tweets");
    stream.print(Printed.<byte[], byte[]>toSysOut().withLabel("tweets-stream")); ❶

    return builder.build();
  }
}
```

❶ The `print` operator allows us to easily view data as it flows through our application. It is generally recommended for development use only.

The second class, which we'll call `App`, will simply instantiate and run the topology, as shown in Example 3-3.

Example 3-3. A separate Java class used to run our Kafka Streams application

```
class App {
  public static void main(String[] args) {
    Topology topology = CryptoTopology.build();

    Properties config = new Properties(); ❶
    config.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev");
    config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:29092");

    KafkaStreams streams = new KafkaStreams(topology, config); ❷

    Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ❸

    System.out.println("Starting Twitter streams");
    streams.start(); ❹
  }
}
```

❶ Kafka Streams requires us to set some basic configuration, including an application ID (which corresponds to a consumer group) and the Kafka bootstrap servers. We set these configs using a `Properties` object.

❷ Instantiate a `KafkaStreams` object with the processor topology and streams config.

❸ Add a shutdown hook to gracefully stop the Kafka Streams application when a global shutdown signal is received.

❹ Start the Kafka Streams application. Note that `streams.start()` does not block, and the topology is executed via background processing threads. This is the reason why a shutdown hook is required.

Our application is now ready to run. If we were to start our Kafka Streams application and then produce some data to our `tweets` topic, we would see the raw byte arrays (the cryptic values that appear after the comma in each output row) being printed to the screen:

```
[tweets-stream]: null, [B@c52d992
[tweets-stream]: null, [B@a4ec036
[tweets-stream]: null, [B@3812c614
```

As you might expect, the low-level nature of byte arrays makes them a little difficult to work with. In fact, additional stream processing steps will be much easier to implement if we find a different method of representing the data in our source topic. This is where the concepts of data serialization and deserialization come into play.

Serialization/Deserialization

Kafka is a bytes-in, bytes-out stream processing platform. This means that clients, like Kafka Streams, are responsible for converting the byte streams they consume into higher-level objects. This process is called *deserialization*. Similarly, clients must also convert any data they want to write back to Kafka back into byte arrays. This process is called *serialization*. These processes are depicted in Figure 3-2.

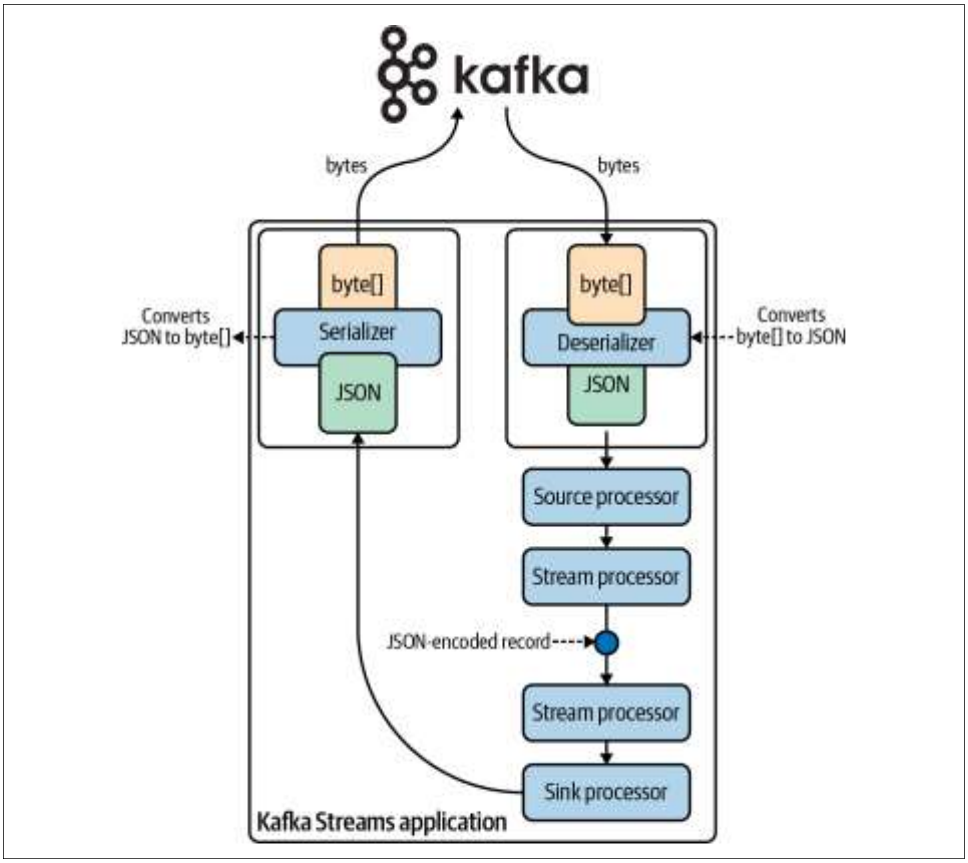

Figure 3-2. An architectural view of where the deserialization and serialization processes occur in a Kafka Streams application

In Kafka Streams, serializer and deserializer classes are often combined into a single class called a *Serdes*, and the library ships with several implementations, shown in Table 3-1. For example, the String Serdes (accessible via the `Serdes.String()` method) includes both the String serializer *and* deserializer class.

Table 3-1. Default Serdes implementations that are available in Kafka Streams

Data type	Serdes class
byte[]	`Serdes.ByteArray()`, `Serdes.Bytes()`
ByteBuffer	`Serdes.ByteBuffer()`
Double	`Serdes.Double()`
Integer	`Serdes.Integer()`
Long	`Serdes.Long()`
String	`Serdes.String()`
UUID	`Serdes.UUID()`
Void	`Serdes.Void()`

Whenever you need to deserialize/serialize data in Kafka Streams, you should first check to see whether or not one of the built-in Serdes classes fits your needs. However, as you may have noticed, Kafka Streams doesn't ship with Serdes classes for some common formats, including JSON, Avro, and Protobuf. However, we can implement our own Serdes when the need arises, and since tweets are represented as JSON objects, we will learn how to build our own custom Serdes to handle this format next.

Building a Custom Serdes

As mentioned earlier, the tweets in our source topic are encoded as JSON objects, but only the raw bytes are stored in Kafka. So, the first thing we will do is write some code for deserializing tweets as higher-level JSON objects, which will ensure the data is easy to work with in our stream processing application. We could just use the built-in String Serdes, `Serdes.String()`, instead of implementing our own, but that would make working with the Twitter data difficult, since we couldn't easily access each field in the tweet object.

If your data is in a common format, like JSON or Avro, then you won't need to reinvent the wheel by implementing your own low-level JSON serialization/deserialization logic. There are many Java libraries for serializing and deserializing JSON, but the one we will be using in this tutorial is Gson. Gson was developed by Google and has an intuitive API for converting JSON bytes to Java objects. The following code block demonstrates the basic method for deserializing byte arrays with Gson, which will come in handy whenever we need to read JSON records from Kafka:

```
Gson gson = new Gson();
byte[] bytes = ...; ❶
```

```
Type type = ...; ❷
gson.fromJson(new String(bytes), type); ❸
```

❶ The raw bytes that we need to deserialize.

❷ `type` is a Java class that will be used to represent the deserialized record.

❸ The `fromJson` method actually converts the raw bytes into a Java class.

Furthermore, Gson also supports the inverse of this process, which is to say it allows us to convert (i.e., serialize) Java objects into raw byte arrays. The following code block shows how serialization works in Gson:

```
Gson gson = new Gson();
gson.toJson(instance).getBytes(StandardCharsets.UTF_8);
```

Since the Gson library takes care of the more complex task of serializing/deserializing JSON at a low level, we just need to leverage Gson's capabilities when implementing a custom Serdes. The first step is to define a data class, which is what the raw byte arrays will be deserialized into.

Defining Data Classes

One feature of Gson (and some other JSON serialization libraries, like Jackson) is that it allows us to convert JSON byte arrays into Java objects. In order to do this, we simply need to define a data class, or POJO (Plain Old Java Object), that contains the fields that we want to deserialize from the source object.

As you can see in Example 3-4, our data class for representing raw tweet records in our Kafka Streams application is pretty simple. We simply define a class property for each field we want to capture from the raw tweet (e.g., `createdAt`), and a corresponding getter/setter method for accessing each property (e.g., `getCreatedAt`).

Example 3-4. A data class to use for deserializing tweet records

```
public class Tweet {
  private Long createdAt;
  private Long id;
  private String lang;
  private Boolean retweet;
  private String text;

  // getters and setters omitted for brevity
}
```

If we don't want to capture certain fields in the source record, then we can just omit the field from the data class, and Gson will drop it automatically. Whenever you create your own deserializer, you should consider dropping any fields in the source record that aren't needed by your application or downstream applications during the deserialization process. This process of reducing the available fields to a smaller subset is called *projection* and is similar to using a `SELECT` statement in the SQL world to only select the columns of interest.

Now that we have created our data class, we can implement a Kafka Streams deserializer for converting the raw byte arrays from the `tweets` topic to higher-level `Tweet` Java objects (which will be much easier to work with).

Implementing a Custom Deserializer

The code required for implementing a custom deserializer is pretty minimal, especially when you leverage a library that hides most of the complexity of deserializ-ing byte arrays (as we're doing with Gson). We simply need to implement the `Deserializer` interface in the Kafka client library, and invoke the deserialization logic whenever `deserialize` is invoked. The following code block shows how to implement the `Tweet` deserializer:

```
public class TweetDeserializer implements Deserializer<Tweet> {
  private Gson gson =
      new GsonBuilder()
          .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE) ❶
          .create();

  @Override
  public Tweet deserialize(String topic, byte[] bytes) { ❷
    if (bytes == null) return null; ❸
    return gson.fromJson(
      new String(bytes, StandardCharsets.UTF_8), Tweet.class); ❹
  }
}
```

❶ Gson supports several different formats for JSON field names. Since the Twitter Kafka connector uses upper camel case for field names, we set the appropriate field naming policy to ensure the JSON objects are deserialized correctly. This is pretty implementation-specific, but when you write your own deserializer, feel free to leverage third-party libraries and custom configurations like we're doing here to help with the heavy lifting of byte array deserialization.

❷ We override the `deserialize` method with our own logic for deserializing records in the `tweets` topic. This method returns an instance of our data class (`Tweet`).

❸ Don't try to deserialize the byte array if it's null.

❹ Use the Gson library to deserialize the byte array into a `Tweet` object.

Now, we are ready to implement our serializer.

Implementing a Custom Serializer

The code for implementing a custom serializer is also very straightforward. This time, we need to implement the `serialize` method of the `Serializer` interface that is included in the Kafka client library. Again, we will leverage Gson's serialization capabilities to do most of the heavy lifting. The following code block shows the `Tweet` serializer we will be using in this chapter:

```
class TweetSerializer implements Serializer<Tweet> {
  private Gson gson = new Gson();

  @Override
  public byte[] serialize(String topic, Tweet tweet) {
    if (tweet == null) return null; ❶
    return gson.toJson(tweet).getBytes(StandardCharsets.UTF_8); ❷
  }
}
```

❶ Don't try to deserialize a null `Tweet` object.

❷ If the `Tweet` object is not null, use Gson to convert the object into a byte array.

With both a `Tweet` deserializer *and* serializer in place, we can now combine these two classes into a Serdes.

Building the Tweet Serdes

So far, the deserializer and serializer implementations have been very light on code. Similarly, our custom Serdes class, whose sole purpose is to combine the deserializer and serializer in a convenient wrapper class for Kafka Streams to use, is also pretty minimal. The following code block shows how easy it is to implement a custom Serdes class:

```
public class TweetSerdes implements Serde<Tweet> {

  @Override
  public Serializer<Tweet> serializer() {
    return new TweetSerializer();
  }

  @Override
  public Deserializer<Tweet> deserializer() {
    return new TweetDeserializer();
  }
}
```

Now that our custom Serdes is in place, we can make a slight modification to the code in Example 3-2. Let's change:

```
KStream<byte[], byte[]> stream = builder.stream("tweets");
stream.print(Printed.<byte[], byte[]>toSysOut().withLabel("tweets-stream"));
```

to:

```
KStream<byte[], Tweet> stream = ❶
  builder.stream(
    "tweets",
    Consumed.with(Serdes.ByteArray(), new TweetSerdes())); ❷

stream.print(Printed.<byte[], Tweet>toSysOut().withLabel("tweets-stream"));
```

❶ Notice that our value type has changed from `byte[]` to `Tweet`. Working with `Tweet` instances will make our life much easier, as you will see in upcoming sections.

❷ Explicitly set the key and value Serdes to use for this `KStream` using Kafka Stream's `Consumed` helper. By setting the value Serdes to `new TweetSerdes()`, our stream will now be populated with `Tweet` objects instead of byte arrays. The key Serdes remains unchanged.

Note, the keys are still being serialized as byte arrays. This is because our topology doesn't require us to do anything with the record keys, so there is no point in deserializing these. Now, if we were to run our Kafka Streams application and produce some data to the source topic, you would see that we are now working with `Tweet` objects, which include the helpful getter methods for accessing the original JSON fields that we will leverage in later stream processing steps:

```
[tweets-stream]: null, Tweet@182040a6
[tweets-stream]: null, Tweet@46efe0cb
[tweets-stream]: null, Tweet@176ef3db
```

Now that we have created a `KStream` that leverages our `Tweet` data class, we can start implementing the rest of the topology. The next step is to filter out any retweets from our Twitter stream.

Handling Deserialization Errors in Kafka Streams

You should always be explicit with how your application handles deserialization errors, otherwise you may get an unwelcome surprise if a malformed record ends up in your source topic. Kafka Streams has a special config, `DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG`, that can be used to specify a deserialization exception handler. You can implement your own exception handler, or use one of the built-in defaults, including `LogAndFailExceptionHandler` (which logs an error and then sends a shutdown signal to Kafka Streams) or `LogAndContinueExceptionHandler` (which logs an error and continues processing).

Filtering Data

One of the most common stateless tasks in a stream processing application is filtering data. Filtering involves selecting only a subset of records to be processed, and ignoring the rest. Figure 3-3 shows the basic idea of filtering.

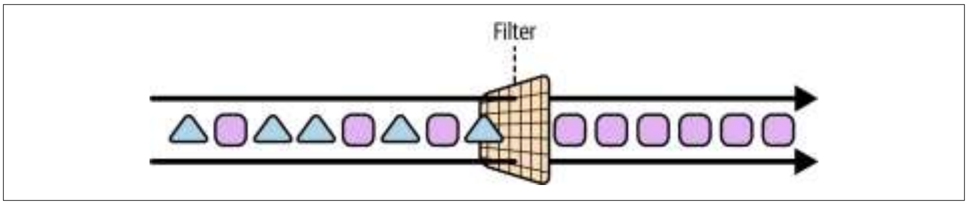

Figure 3-3. Filtering data allows us to process only a subset of data in our event stream

In Kafka Streams, the two primary operators[11] that are used for filtering are:

- `filter`
- `filterNot`

We'll start by exploring the `filter` operator. `filter` simply requires us to pass in a Boolean expression, called a `Predicate`, in order to determine if a message should be kept. If the predicate returns `true`, the event will be forwarded to downstream processors. If it returns `false`, then the record will be excluded from further processing. Our use case requires us to filter out retweets, so our predicate will leverage the `Tweet.isRetweet()` method.

Since `Predicate` is a functional interface, we can use a lambda with the `filter` operator. Example 3-5 shows how to filter out retweets using this method.

Example 3-5. An example of how to use the DSL's **filter** *operator*

```
KStream<byte[], Tweet> filtered =
  stream.filter(
      (key, tweet) -> {
        return !tweet.isRetweet();
      });
```

The `filterNot` operator is very similar to `filter`, except the Boolean logic is inverted (i.e., returning `true` will result in the record being dropped, which is the opposite of `filter`). As you can see in Example 3-5, our filtering condition is negated: `!tweet.isRetweet()`. We could have just as easily negated at the operator levelinstead, using the code in Example 3-6.

Example 3-6. An example of how to use the DSL's filterNot operator

```
KStream<byte[], Tweet> filtered =
  stream.filterNot(
      (key, tweet) -> {
        return tweet.isRetweet();
      });
```

These two approaches are functionally equivalent. However, when the filtering logic contains a negation, I prefer to use `filterNot` to improve readability. For this tutorial, we will use the `filterNot` implementation shown in Example 3-6 to exclude retweets from further processing.

> If your application requires a filtering condition, you should filter as early as possible. There's no point in transforming or enriching data that will simply be thrown away in a subsequent step, especially if the logic for processing the unneeded event is computationally expensive.

We have now implemented steps 1 and 2 in our processor topology (Figure 3-1). We are ready to move on to the third step: separating our filtered stream into sub streams based on the source language of the tweet.

Branching Data

In the previous section, we learned how to use Boolean conditions called predicates to filter streams. Kafka Streams also allows us to use predicates to separate (or *branch*) streams. Branching is typically required when events need to be routed to different stream processing steps or output topics based on some attribute of the event itself.

Revisiting our requirements list, we see that tweets can appear in multiple languages in our source topic. This is a great use case for branching, since a subset of records in our stream (non-English tweets) require an extra processing step: they need to be translated to English. The diagram in Figure 3-4 depicts the branching behavior we need to implement in our application.

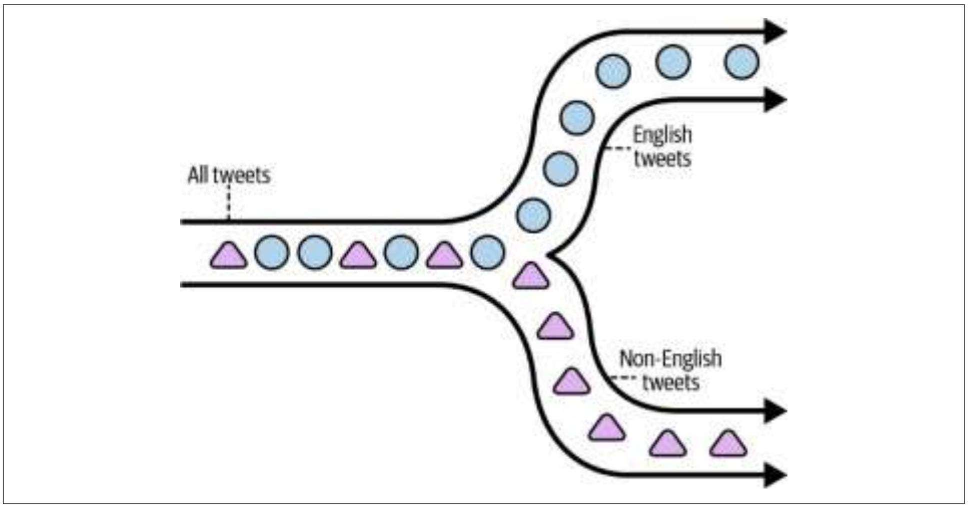

Figure 3-4. Branching operations split a single stream into multiple output streams

Let's create two lambda functions: one for capturing English tweets, and another for capturing everything else. Thanks to our earlier deserialization efforts, we can leverage another getter method in our data class to implement our branching logic: `Tweet.getLang()`. The following code block shows the predicates we will use for branching our stream:

```
Predicate<byte[], Tweet> englishTweets =
  (key, tweet) -> tweet.getLang().equals("en"); ❶

Predicate<byte[], Tweet> nonEnglishTweets =
  (key, tweet) -> !tweet.getLang().equals("en"); ❷
```

❶ This predicate matches all English tweets.

❷ This predicate is the inverse of the first, and captures all non-English tweets.

Now that we have defined our branching conditions, we can leverage Kafka Streams' `branch` operator, which accepts one or more predicates and returns a list of output streams that correspond to each predicate. Note that each predicate is evaluated in order, and a record can only be added to a single branch. If a record doesn't match any predicate, then it will be dropped:

```
KStream<byte[], Tweet>[] branches =
  filtered.branch(englishTweets, nonEnglishTweets); ❶

KStream<byte[], Tweet> englishStream = branches[0]; ❷

KStream<byte[], Tweet> nonEnglishStream = branches[1]; ❸
```

❶ We create two branches of our stream by evaluating the source language of a tweet.

❷ Since we passed the `englishTweets` predicate first, the first `KStream` in our list (at the 0 index) contains the English output stream.

❸ Since the `nonEnglishTweets` predicate was used as the final branching condition, the tweets that need translating will be the last `KStream` in our list (at index position 1).

The method of branching streams may be changing in an upcoming Kafka Streams release. If you are using a Kafka Streams version greater than 2.7.0, you may want to check "KIP-418: A method-chaining way to branch KStream" for potential API changes. Some backward compatibility is expected, and the current implementation does work on the latest version of Kafka Streams at the time of writing (version 2.7.0).

Now that we have created two substreams (`englishStream` and `nonEnglishStream`), we can apply different processing logic to each. This takes care of the third step of our processor topology (see Figure 3-1), so we're now ready to move on to the next step: translating non-English tweets into English.

Translating Tweets

At this point, we have two record streams: tweets that are written in English (`englishStream`), and tweets that are written in another language (`nonEnglishStream`). We would like to perform sentiment analysis on each tweet, but the API we will be using to perform this sentiment analysis only supports a few languages (English being one of them). Therefore, we need to translate each tweet in the `nonEnglishStream` to English.

We're talking about this as a business problem, though. Let's look at the requirement from a stream processing perspective. What we really need is a way of transforming one input record into exactly one new output record (whose key or value may or may not be the same type as the input record). Luckily for us, Kafka Streams has two operators that fit the bill:

- `map`
- `mapValues`

A visualization of a mapping operation is shown in Figure 3-5.

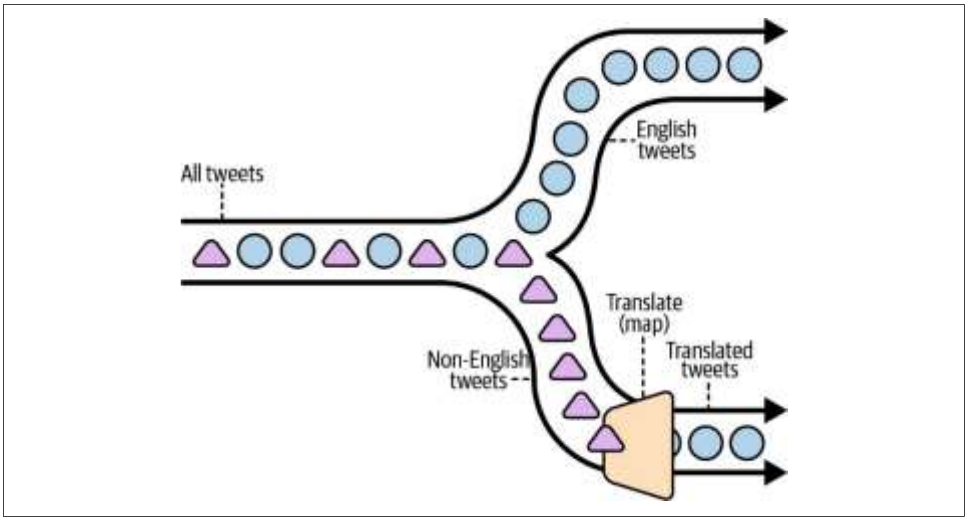

Figure 3-5. Mapping operations allow us to perform a 1:1 transformation of records

The map and mapValues operators are very similar (they both have a 1:1 mapping between input and output records), and both could work for this use case. The only difference is that map requires us to specify a new record value *and* record key, while mapValues requires us to just set a new value.

Let's first take a look at how we might implement this with map. Assume that we not only want to translate the tweet text, but also rekey each record by the Twitter username associated with a given tweet. We could implement the tweet translation step as shown in the following code block:

```
KStream<byte[], Tweet> translatedStream =
  nonEnglishStream.map(
      (key, tweet) -> { ❶
        byte[] newKey = tweet.getUsername().getBytes();
        Tweet translatedTweet = languageClient.translate(tweet, "en");
        return KeyValue.pair(newKey, translatedTweet); ❷
      });
```

❶ Our mapping function is invoked with the current record key and record value (called tweet).

❷ Our mapping function is required to return a new record key *and* value, represented using Kafka Streams' KeyValue class. Here, the new key is set to the Twitter username, and the new value is set to the translated tweet. The actual logic for translating text is out of scope for this tutorial, but you can check the source code for implementation details.

However, the requirements we outlined don't require us to rekey any records. As we'll see in the next chapter, rekeying records is often used when stateful operations will be performed in a latter stream processing step. So while the preceding code works, we can simplify our implementation by using the mapValues operator instead, which is only concerned with transforming record values.

An example of how to use the mapValues operator is shown in the following code block:

```
KStream<byte[], Tweet> translatedStream =
  nonEnglishStream.mapValues(
      (tweet) -> { ❶
        return languageClient.translate(tweet, "en"); ❷
      });
```

❶ Our mapValues function is invoked with *only* the record value.

❷ We are only required to return the new record value when using the mapValues operator. In this case, the value is a translated Tweet.

We will stick with the mapValues implementation since we don't need to rekey any records.

It is recommended to use `mapValues` instead of `map` whenever possible, as it allows Kafka Streams to potentially execute the program more efficiently.

After translating all non-English tweets, we now have two `KStreams` that contain English tweets:

- `englishStream`, the original stream of English tweets
- `translatedStream`, the newly translated tweets, which are now in English

This wraps up step 4 in our processor topology (see Figure 3-1). However, the goal of our application is to perform sentiment analysis on *all* English tweets. We *could* duplicate this logic across both of these streams: `englishStream` and `translatedStream`. However, instead of introducing unnecessary code duplication, it would be better to *merge* these two streams. We'll explore how to merge streams in the next section.

Merging Streams

Kafka Streams provides an easy method for combining multiple streams into a single stream. Merging streams can be thought of as the opposite of a branching operation, and is typically used when the same processing logic can be applied to more than one stream in your application.

The equivalent of a `merge` operator in the SQL world is a union query. For example:

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

Now that we have translated all tweets in the non-English stream, we have two separate streams that we need to perform sentiment analysis on: `englishStream` and `translatedStream`. Furthermore, all tweets that we enrich with sentiment scores will need to be written to a single output topic: `crypto-sentiment`. This is an ideal use case for merging since we can reuse the same stream/sink processors for both of these separate streams of data.

The diagram in Figure 3-6 depicts what we are trying to accomplish.

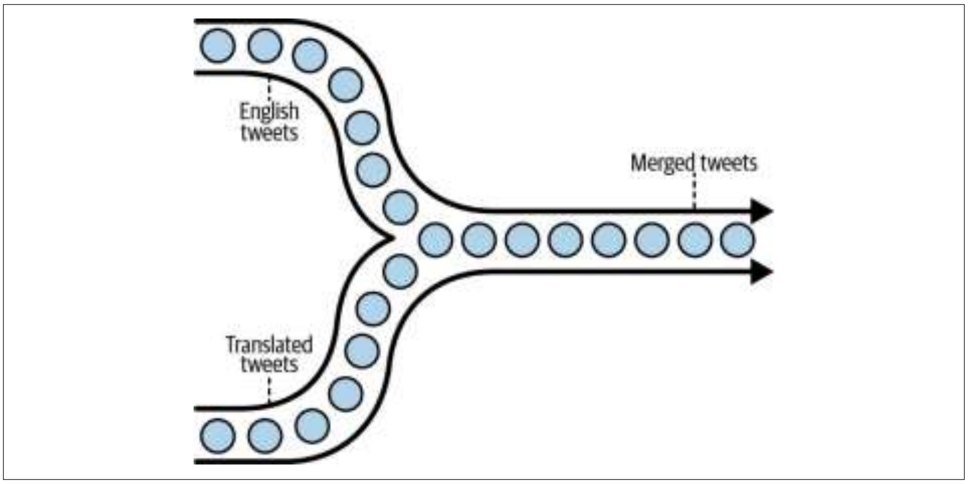

Figure 3-6. Merging operations combine multiple streams into a single stream

The code for merging streams is very simple. We only need to pass the streams we want to combine to the `merge` operator, as shown here:

```
KStream<byte[], Tweet> merged = englishStream.merge(translatedStream);
```

Now that our streams are combined, we are ready to move to the next step.

Enriching Tweets

We are nearing the goal of being able to enrich each tweet with a sentiment score. However, as you may have noticed, our current data class, `Tweet`, represents the structure of the raw tweets in our *source topic*. We need a new data class for representing the enriched records that we will be writing to our *output topic* (`crypto-sentiment`). This time, instead of serializing our data using JSON, we will be using a data serialization format called Avro. Let's take a deeper look at Avro and learn how to create an Avro data class for representing the enriched records in our Kafka Streams application.

Avro Data Class

Avro is a popular format in the Kafka community, largely due to its compact byte representation (which is advantageous for high throughput applications), native support for record schemas, and a schema management tool called Schema Registry, which works well with Kafka Streams and has had strong support for Avro since its inception. There are other advantages as well. For example, some Kafka Connectors can use Avro schemas to automatically infer the table structure of downstream datastores, so encoding our output records in this format can help with data integration downstream.

When working with Avro, you can use either *generic records* or *specific records*.

Generic records are suitable when the record schema isn't known at runtime. They allow you to access field names using generic getters and setters. For example: `GenericRecord.get(String key)` and `GenericRecord.put(String key, Object value)`.

Specific records, on the other hand, are Java classes that are generated from Avro schema files. They provide a much nicer interface for accessing record data. For example, if you generate a specific record class named `EntitySentiment`, then you can access fields using dedicated getters/setters for each field name. For example: `entitySentiment.getSentimentScore()`.

Since our application defines the format of its output records (and therefore, the schema is known at build time), we'll use Avro to generate a specific record (which we'll refer to as a data class from here on out). A good place to add a schema definition for Avro data is in the *src/main/avro* directory of your Kafka Streams project. Example 3-7 shows the Avro schema definition we will be using for our sentiment-enriched output records. Save this schema in a file named *entity_sentiment.avsc* in the *src/main/avro/* directory.

Example 3-7. An Avro schema for our enriched tweets

```
{
  "namespace": "com.magicalpipelines.model", ❶
  "name": "EntitySentiment", ❷
  "type": "record",
  "fields": [
    {
      "name": "created_at",
      "type": "long"
    },
    {
      "name": "id",
      "type:" "long"
    },
    {
      "name": "entity",
      "type": "string"
    },
    {
      "name": "text",
      "type": "string"
    },
    {
      "name": "sentiment_score",
      "type": "double"
    },
    {
      "name": "sentiment_magnitude",
      "type": "double"
    },
    {
      "name": "salience",
      "type": "double"
    }
  ]
}
```

❶ The desired package name for your data class.

❷ The name of the Java class that will contain the Avro-based data model. This class will be used in subsequent stream processing steps.

Now that we have defined our schema, we need to generate a data class from this definition. In order to do that, we need to add some dependencies to our project. This can be accomplished by adding the following lines in the project's *build.gradle* file:

```
plugins {
  id 'com.commercehub.gradle.plugin.avro' version '0.9.1' ❶
}

dependencies {
  implementation 'org.apache.avro:avro:1.8.2' ❷
}
```

❶ This Gradle plug-in is used to autogenerate Java classes from Avro schema definitions, like the one we created in Example 3-7.

❷ This dependency contains the core classes for working with Avro.

Now, when we build our project, a new data class named `EntitySentiment` will be automatically generated for us. This generated dataclass contains a new set of fields for storing tweet sentiment (`sentiment_score`, `sentiment_magnitude`, and `salience`) and corresponding getter/setter methods. With our data class in hand, we can now proceed with adding sentiment scores to ourtweets. This will introduce us to a new set of DSL operators that are extremely usefulfor transforming data.

Sentiment Analysis

We have already seen how to transform records during the tweet translation step by using `map` and `mapValues`. However, both `map` and `mapValues` produce exactly one output record for each input record they receive. In some cases, we may want to produce zero, one, or even multiple output records for a single input record.

Consider the example of a tweet that mentions multiple cryptocurrencies:

> #bitcoin is looking super strong. #ethereum has me worried though

This fictional tweet explicitly mentions two cryptocurrencies, or *entities* (Bitcoin and Ethereum) and includes two separate sentiments (positive sentiment for Bitcoin, and negative for Ethereum). Modern Natural Language Processing (NLP) libraries and services are often smart enough to calculate the sentiment for each entity within the text, so a single input string, like the one just shown, could lead to multiple output records. For example:

```
{"entity": "bitcoin", "sentiment_score": 0.80}
{"entity": "ethereum", "sentiment_score": -0.20}
```

Once again, let’s consider this requirement outside of the business problem itself. Here, we have a very common stream processing task: we need to convert a single input record into a variable number of output records. Luckily for us, Kafka Streams includes two operators that can help with this use case:

- `flatMap`
- `flatMapValues`

Both `flatMap` and `flatMapValues` are capable of producing zero, one, or multiple output records every time they are invoked. Figure 3-7 visualizes this 1:N mapping of input to output records.

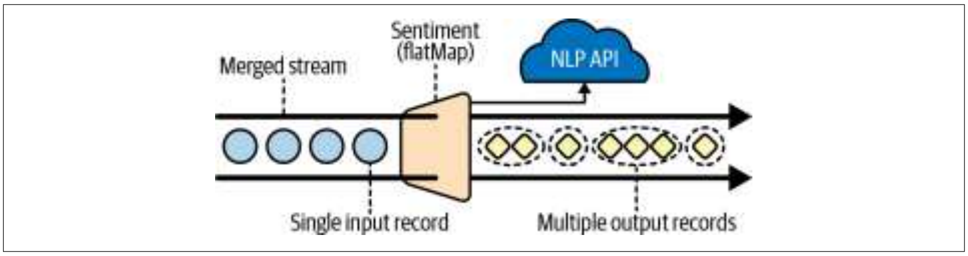

Figure 3-7. flatMap operations allow us to transform one input record into zero or more output records

Similar to the `map` operation we saw earlier, `flatMap` requires us to set a new record key and value. On the other hand, `flatMapValues` only requires us to specify a new value. Since we don’t need to process the record key in this example, we will use `flatMapValues` to perform entity-level sentiment analysis for our tweets, as shown in the following code block (notice that we are using our new Avro-based data class, `EntitySentiment`):

```
KStream<byte[], EntitySentiment> enriched =
  merged.flatMapValues(
      (tweet) -> {
        List<EntitySentiment> results =
          languageClient.getEntitySentiment(tweet); ❶

        results.removeIf(
            entitySentiment -> !currencies.contains(
              entitySentiment.getEntity())); ❷

        return results; ❸
      });
```

❶ Get a list of sentiment scores for each entity in the tweet.

❷ Remove all entities that don’t match one of the crypto currencies we are tracking.

The final list size after all of these removals is variable (a key characteristic of `flatMap` and `flatMapValues` return values), and could have zero or more items.

❸ Since the `flatMap` and `flatMapValues` operators can return any number of records, we will return a list that contains every record that should be added to the output stream. Kafka Streams will handle the “flattening” of the collection we return (i.e., it will break out each element in the list into a distinct record in the stream).

Similar to our recommendation about using `mapValues`, it is also recommended to use `flatMapValues` instead of `flatMap` whenever possible, as it allows Kafka Streams to potentially execute the program more efficiently.

We are now ready to tackle the final step: writing the enriched data (the sentiment scores) to a new output topic. In order to do this, we need to build an Avro Serdes that can be used for serializing the Avro-encoded `EntitySentiment` records that we created.

Serializing Avro Data

As mentioned earlier, Kafka is a bytes-in, bytes-out stream processing platform. Therefore, in order to write the `EntitySentiment` records to our output topic, we need to serialize these Avro records into byte arrays.

When we serialize data using Avro, we have two choices:

- Include the Avro schema in each record.
- Use an even more compact format, by saving the Avro schema in Confluent Schema Registry, and only including a much smaller schema ID in each record instead of the entire schema.

As shown in Figure 3-8, the benefit of the first approach is you don’t have to set up and run a separate service alongside your Kafka Streams application. Since Confluent Schema Registry is a REST service for creating and retrieving Avro, Protobuf, and JSON schemas, it requires a separate deployment and therefore introduces a maintenance cost and an additional point of failure. However, with the first approach, you do end up with larger message sizes since the schema is included.

However, if you are trying to eke every ounce of performance out of your Kafka Streams application, the smaller payload sizes that Confluent Schema Registry enables may be necessary. Furthermore, if you anticipate ongoing evolution of your record schemas and data model, the schema compatibility checks that are included in Schema Registry help ensure that future schema changes are made in safe, non- breaking ways.

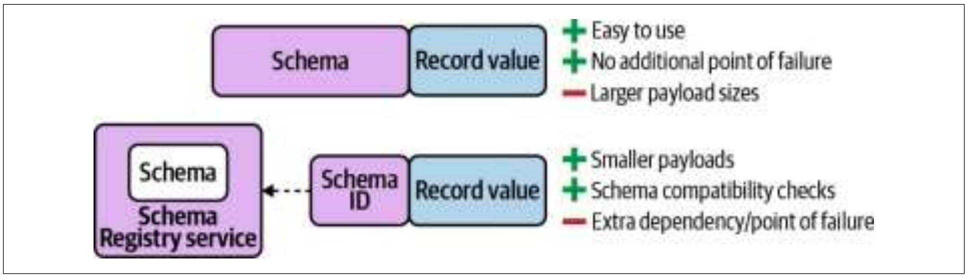

Figure 3-8. Advantages and disadvantages of including an Avro schema in each record

Since Avro Serdes classes are available for both approaches, the amount of code you need to introduce to your application for serializing Avro data is minimal. The following sections show how to configure both a registryless and Schema Registry-aware Avro Serdes.

Registryless Avro Serdes

While a registryless Avro Serdes can be implemented very simply by yourself, I have open sourced a version under the `com.mitchseymour:kafka-registryless-avro-serdes` package. You can use this package by updating your *build.gradle* file with the following code:

```
dependencies {
  implementation 'com.mitchseymour:kafka-registryless-avro-serdes:1.0.0'
}
```

Whenever you need to use this Serdes in your Kafka Streams application, just provide an Avro-generated class to the `AvroSerdes.get` method, as shown here:

```
AvroSerdes.get(EntitySentiment.class)
```

The resulting Serdes can be used anywhere you would normally use one of Kafka's built-in Serdes.

Schema Registry-Aware Avro Serdes

Confluent has also published a package for distributing its Schema Registry–aware Avro Serdes. If you want to leverage the Confluent Schema Registry, update your *build.gradle* file as follows:

```
repositories {
  mavenCentral()

  maven {
      url "https://packages.confluent.io/maven/" ❶
  }
}

dependencies {
  implementation ('io.confluent:kafka-streams-avro-serde:6.0.1') { ❷
```

```
      exclude group: 'org.apache.kafka', module: 'kafka-clients' ❸
    }
  }
```

❶ Add the Confluent Maven repository since this is where the artifact for the Schema Registry–aware Avro Serdes lives.

❷ Add the required dependency for using a Schema Registry–aware Avro Serdes.

❸ Exclude an incompatible transitive dependency included in the `kafka-streams-avro-serde` at the time of writing.[20]

The Schema Registry–aware Avro Serdes requires some additional configuration, so you can improve the readability of your code by creating a factory class for instantiating Serdes instances for each of the data classes in your project. For example, the following code block shows how to create a registry-aware Avro Serdes for the `TweetSentiment` class:

```
public class AvroSerdes {
  public static Serde<EntitySentiment> EntitySentiment(
    String url, boolean isKey) {

      Map<String, String> serdeConfig =
        Collections.singletonMap("schema.registry.url", url); ❶
      Serde<EntitySentiment> serde = new SpecificAvroSerde<>();
      serde.configure(serdeConfig, isKey);
      return serde;
  }
}
```

❶ The register-aware Avro Serdes requires us to configure the Schema Registry endpoint.

Now, you can instantiate the Serdes wherever you need it in your Kafka Streams application with the following code:

```
AvroSerdes.EntitySentiment("http://localhost:8081", false) ❶
```

❶ Update the Schema Registry endpoint to the appropriate value.

You can interact with Schema Registry directly for registering, deleting, modifying, or listing schemas. However, when using a registry-aware Avro Serdes from Kafka Streams, your schema will automatically be registered for you. Furthermore, to improve performance, the registry-aware Avro Serdes minimizes the number of schema lookups by caching schema IDs and schemas locally.

Now that we have our Avro Serdes in place, we can create our sink processor.

Adding a Sink Processor

The last step is to write the enriched data to our output topic: `crypto-sentiment`. There are a few operators for doing this:

- `to`
- `through`
- `repartition`

If you want to return a new `KStream` instance for appending additional operators/ stream processing logic, then you should use the `repartition` or `through` operator (the latter was deprecated right before this book was published, but is still widely used and backward compatibility is expected). Internally, these operators call `builder.stream` again, so using them will result in additional sub-topologies being created by Kafka Streams. However, if you havereached a terminal step in your stream, as we have, then you should use the `to` opera-tor, which returns `void` since no other stream processors need to be added to the underlying `KStream`.

In this example, we will use the `to` operator since we have reached the last step of our processor topology, and we will also use the Schema Registry–aware Avro Serdes since we want better support for schema evolution and also smaller message sizes. The following code shows how to add the sink processor:

```
enriched.to(
  "crypto-sentiment",
  Produced.with(
    Serdes.ByteArray(),
    AvroSerdes.EntitySentiment("http://localhost:8081", false)));
```

We have now implemented each of the steps in our processor topology (see Figure 3-1). The final step is to run our code and verify that it works as expected.

Running the Code

In order to run the application, you'll need to stand up a Kafka cluster and Schema Registry instance. The source code for this tutorial includes a Docker Compose envi-ronment to help you accomplish this. Once the Kafka cluster and Schema Registry service are running, you can start your Kafka Streams application with the following command:

```
./gradlew run --info
```

Now we're ready to test. The next section shows how to verify that our application works as expected.

Empirical Verification

One of the easiest ways to verify that your application is working as expected is through empirical verification. This involves generating data in a local Kafka cluster and subsequently observing the data that gets written to the output topic. The easiest way to do this is to save some example `tweet` records in a text file, and use the `kafka-console-producer` to write these records to our source topic: `tweets`.

The source code includes a file called *test.json*, with two records that we will use for testing. Note: the actual example records are flattened in *test.json*, but we have shown the pretty version of each record in Example 3-8 to improve readability.

Example 3-8. Two example tweets used for testing our Kafka Streams application

```
{
  "CreatedAt": 1577933872630,
  "Id": 10005,
  "Text": "Bitcoin has a lot of promise. I'm not too sure about #ethereum",
  "Lang": "en",
  "Retweet": false, ❶
  "Source": "",
  "User": {
    "Id": "14377871",
    "Name": "MagicalPipelines",
    "Description": "Learn something magical today.",
    "ScreenName": "MagicalPipelines",
    "URL": "http://www.magicalpipelines.com",
    "FollowersCount": "248247",
    "FriendsCount": "16417"
  }
}
{
  "CreatedAt": 1577933871912,
  "Id": 10006,
  "Text": "RT Bitcoin has a lot of promise. I'm not too sure about #ethereum",
  "Lang": "en",
  "Retweet": true, ❷
  "Source": "",
  "User": {
    "Id": "14377870",
    "Name": "Mitch",
    "Description": "",
    "ScreenName": "Mitch",
    "URL": "http://mitchseymour.com",
    "FollowersCount": "120",
    "FriendsCount": "120"
  }
}
```

❶ This first tweet (ID 10005) is *not* a retweet. We expect this tweet to be enriched with sentiment scores.

❷ The second tweet (ID 10006) *is* a retweet, and we expect this record to be ignored.

Now, let’s produce these example records to our local Kafka cluster using the following command:

```
kafka-console-producer \
  --bootstrap-server kafka:9092 \
  --topic tweets < test.json
```

In another tab, let’s use `kafka-console-consumer` to consume the enriched records. Run the following command:

```
kafka-console-consumer \
  --bootstrap-server kafka:9092 \
  --topic crypto-sentiment \
  --from-beginning
```

You should see some cryptic-looking output with lots of strange symbols:

```
8888[88|Bitcoin has a lot of promise.
I'm not too sure about #ethereumbitcoin`ff8?`ff8? -88?
8888[88|Bitcoin has a lot of promise.
I'm not too sure about #ethereumethereum888ℸ888ℸ888?
```

This is because Avro is a binary format. If you’re using Schema Registry, as we are, then you can use a special console script developed by Confluent that improves the readability of Avro data. Simply change `kafka-console-consumer` to `kafka-avro-console-consumer` as shown here:

```
kafka-avro-console-consumer \
 --bootstrap-server kafka:9092 \
 --topic crypto-sentiment \
 --from-beginning
```

Finally, you should see output similar to the following:

```
{
  "created_at": 1577933872630,
  "id": 10005,
  "text": "Bitcoin has a lot of promise. I'm not too sure about #ethereum",
  "entity": "bitcoin",
  "sentiment_score": 0.699999988079071,
  "sentiment_magnitude": 0.699999988079071,
  "salience": 0.47968605160713196
}
{
  "created_at": 1577933872630,
  "id": 10005,
  "text": "Bitcoin has a lot of promise. I'm not too sure about #ethereum",
  "entity": "ethereum",
  "sentiment_score": -0.20000000298023224,
  "sentiment_magnitude": -0.20000000298023224,
  "salience": 0.030233483761548996
}
```

Notice that tweet ID `10006` does not appear in the output. This was a retweet, and therefore was filtered in the second topology step in Figure 3-1. Also notice that tweet ID `10005` resulted in two output records. This is expected since this tweet mentioned two separate cryptocurrencies (each with a separate sentiment), and verifies that our `flatMapValues` operator worked as expected (the sixth topology step in Figure 3-1).

If you would like to verify the language translation step, feel free to update the example records in *test.json* with a foreign language tweet. We will leave this as an exercise for you.