5 Consumers: Unlocking data

This chapters covers

- Exploring the consumer and how it works
- Using consumer groups to coordinate reading data from topics
- Learning about offsets and how to use them
- Examining various configuration options that change consumer behavior

In our previous chapter, we started writing data into our Kafka system. However, as you know, that is only one part of the story. Consumers get the data from Kafka and provide those values to other systems or applications. Because consumers are clients that exist outside of brokers, they can be written in various programming languages just like producer clients. Take note that when we look at how things work in this chapter, we will try to lean towards the defaults of the Java consumer client. After reading this chapter, we will be on our way to solving our previous business requirements by consuming data in a couple of different ways.

5.1 An example

The consumer client is the program that subscribes to the topic or topics that interest them [1]. As with producer clients, the actual consumer processes can run on separate machines and are not required to run on a specific server. In fact, most consumer clients in production settings are on separate hosts. As long as the clients can connect to the Kafka brokers, they can read messages. Figure 5.1 reintroduces the broad scope of Kafka and shows consumers running outside the brokers to get data from Kafka.

Why is it important to know that the consumer is subscribing to topics (pulling messages) and not being pushed to instead? The power of processing control shifts to the consumer in this situation. Figure 5.1 shows where consumer clients fit into the overall Kafka ecosystem. Clients are responsible for reading data from topics and

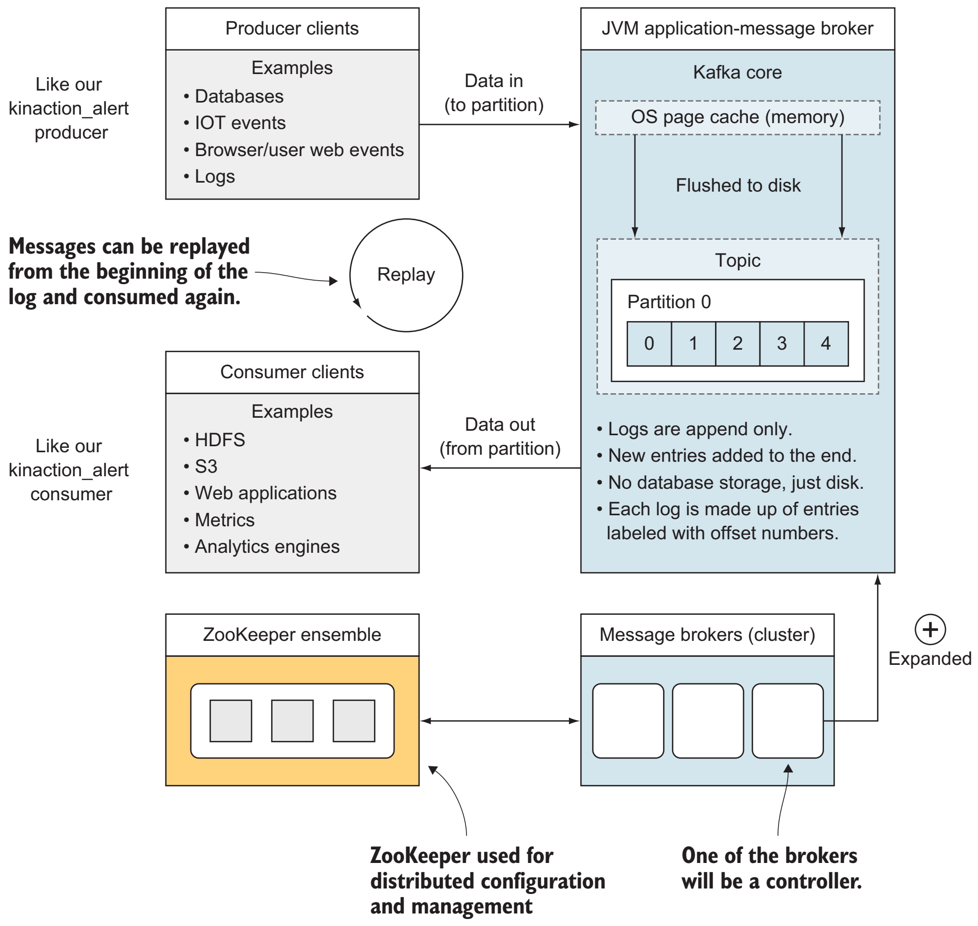
Like our kinaction_alert producer
Producer clients
Examples
• Databases
• IOT events
• Browser/user web events
• Logs
Data in (to partition)
JVM application-message broker
Kafka core
OS page cache (memory)
Flushed to disk
Messages can be replayed from the beginning of the log and consumed again.
Replay
Topic
Partition 0
0
1
2
3
4
Like our kinaction_alert consumer
Consumer clients
Examples
• HDFS
• S3
• Web applications
• Metrics
• Analytics engines
Data out (from partition)
• Logs are append only.
• New entries added to the end.
• No database storage, just disk.
• Each log is made up of entries labeled with offset numbers.
ZooKeeper ensemble
Message brokers (cluster)
Expanded
ZooKeeper used for distributed configuration and management
One of the brokers will be a controller.

Figure 5.1 Overview of Kafka consumer clients

making it available to application (like metrics dashboards or analytics engines) or storing it in other systems. Consumers themselves control the rate of consumption.

With consumers in the driver's seat, if a failure occurs and the consumer applications come back online, they can start pulling again. There's no need to always have the consumers up and running to handle (or miss) notifications. Although you can develop applications that are capable of handling this constant data flow or even a buildup of back pressure due to volume, you need to know that you are not a listener for the brokers; consumers are the ones pulling the data. For those readers that have used Kafka before, you might know that there are reasons why you probably will not want to have your consumers down for extended periods. When we discuss more details about topics, we will look at how data might be removed from Kafka due to size or time limits that users can define.

5.1.1 Consumer options

In our discussion, you will notice a couple of properties that are related to the ones that were needed for the producer clients as well. We always need to know the brokers we can attempt to connect to on client startup. One minor "gotcha" is to make sure you use the deserializers for the keys and values that match the serializers you produced the message with. For example, if you produce using a `StringSerializer` but try to consume using the `LongDeSerializer`, you will get an exception that you will need to fix.

Table 5.1 lists some of the configuration values that we should know as we start writing our own consumers [2].

Table 5.1 Consumer configuration

Key	Purpose
`bootstrap.servers`	One or more Kafka brokers to connect on startup
`value.deserializer`	Needed for deserialization of the value
`key.deserializer`	Needed for deserialization of the key
`group.id`	A name that's used to join a consumer group
`client.id`	An ID to identify a user (we will use this in chapter 10)
`heartbeat.interval.ms`	Interval for consumer's pings to the group coordinator

One way to deal with all of the consumer configuration key names is to use the constants provided in the Java class `ConsumerConfig` (see http://mng.bz/oGgy) and by looking for the Importance label of "high" in the Confluent website (http://mng.bz/drdv). However, in our examples, we will use the property names themselves for clarity. Listing 5.1 shows four of these keys in action. The values for the configurations in table 5.1 determine how our consumer interacts with the brokers as well as other consumers.

We will now switch to reading from a topic with one consumer as we did in chapter 2. For this example, we have an application similar to how Kafka could have started in LinkedIn, dealing with user activity events (mentioned in chapter 1) [3]. Let's say that we have a specific formula that uses the time a user spends on the page as well as the number of interactions they have, which is sent as a value to a topic to project future click rates with a new promotion. Imagine that we run the consumer and process all of the messages on the topic and that we are happy with our application of the formula (in this case, multiplying by a magic number).

Listing 5.1 shows an example of looking at the records from the topic `kinaction_promos` and printing a value based on the data from each event. This listing has many similarities to the producer code that we wrote in chapter 4, where properties are used to determine the behavior of the consumer. This use of deserializers for the keys and values is different than having serializers for producers, which varies depending on the topic we consume.

> **NOTE** Listing 5.1 is not a complete code listing but is meant to highlight specific consumer lines. Remember, a consumer can subscribe to multiple topics, but in this instance, we are only interested in the `kinaction_promos` topic.

In the listing, a loop is also used to poll the topic partitions that our consumer is assigned in order to process messages. This loop is toggled with a Boolean value. This sort of loop can cause errors, especially for beginner programmers! Why this loop then? Part of the streaming mindset encompasses events as a continuous stream, and this is reflected in the logic. Notice that this example uses `250` for the value of the poll duration, which is in milliseconds. This timeout indicates how long the call blocks a main application thread by waiting, but it can return immediately when records are ready for delivery [4]. This value is something that you can fine-tune and adjust, based on the needs of your applications. The reference (and more details) for the Java 8 style of using `addShutdownHook` we use in the following listing can be seen at https://docs.confluent.io/platform/current/streams/developer-guide/write-streams.html.

Listing 5.1 Promotion consumer

```
...
  private volatile boolean keepConsuming = true;

  public static void main(String[] args) {
    Properties kaProperties = new Properties();
    kaProperties.put("bootstrap.servers",
              "localhost:9092,localhost:9093,,localhost:9094");
    kaProperties.put("group.id",
              "kinaction_webconsumer");                          <- Defines group.id. (We'll discuss this with consumer groups.)
    kaProperties.put("enable.auto.commit", "true");
    kaProperties.put("auto.commit.interval.ms", "1000");
    kaProperties.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");    <- Defines deserializers for the key and values
    kaProperties.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
```

```
    WebClickConsumer webClickConsumer = new WebClickConsumer();
    webClickConsumer.consume(kaProperties);

    Runtime.getRuntime()
      .addShutdownHook(
        new Thread(webClickConsumer::shutdown)
      );
  }

  private void consume(Properties kaProperties) {
    try (KafkaConsumer<String, String> consumer =
      new KafkaConsumer<>(kaProperties)) {              <-- Passes the properties into the KafkaConsumer constructor
      consumer.subscribe(
        List.of("kinaction_promos")                     <-- Subscribes to one topic, kinaction_promos
      );

      while (keepConsuming) {                           <-- Uses a loop to poll for topic records
         ConsumerRecords<String, String> records =
          consumer.poll(Duration.ofMillis(250));
        for (ConsumerRecord<String, String> record : records) {
          log.info("kinaction_info offset = {}, key = {}",
                   record.offset(),
                   record.key());
          log.info("kinaction_info value = {}",
            Double.parseDouble(record.value()) * 1.543);
        }
      }
    }
  }

  private void shutdown() {
    keepConsuming = false;
  }
}
```

After generating a value for every message in the topic in listing 5.1, we find out that our modeling formula isn't correct! So what should we do now? Attempt to recalculate the data we have from our end results (assuming the correction would be harder than in the example) and then apply a new formula?

This is where we can use our knowledge of consumer behavior in Kafka to replay the messages we already processed. By having the raw data retained, we do not have to worry about trying to recreate the original data. Developer mistakes, application logic mistakes, and even dependent application failures can be corrected because the data is not removed from our topics once it is consumed. This also explains how time travel, in a way, is possible with Kafka.

Let's switch to looking at how to stop our consumer. You already saw where you used Ctrl-C to end your processing or stopped the process on the terminal. However, the proper way includes calling a `close` method on the consumer [23].

Listing 5.2 shows a consumer that runs on a thread and a different class controls shutdown. When the code in listing 5.2 is started, the thread runs with a consumer instance. By calling the public method `shutdown`, a different class can flip the Boolean and stop our consumer from polling for new records. The stopping variable is our guard, which decides whether to continue processing or not. Calling the `wakeup` method also causes a `WakeupException` to be thrown that leads to the final block closing the consumer resource correctly [5]. Listing 5.2 used https://kafka.apache.org/26/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html as a reference documentation.

Listing 5.2 Closing a consumer

```
public class KinactionStopConsumer implements Runnable {
     private final KafkaConsumer<String, String> consumer;
     private final AtomicBoolean stopping =
                              new AtomicBoolean(false);
     ...

    public KinactionStopConsumer(KafkaConsumer<String, String> consumer) {
      this.consumer = consumer;
    }

     public void run() {
         try {
             consumer.subscribe(List.of("kinaction_promos"));
             while (!stopping.get()) {
                 ConsumerRecords<String, String> records =
                   consumer.poll(Duration.ofMillis(250));
                 ...
             }
         } catch (WakeupException e) {
             if (!stopping.get()) throw e;
         } finally {
             consumer.close();
         }
     }

     public void shutdown() {
         stopping.set(true);
         consumer.wakeup();
     }
 }
```

The variable stopping determines whether to continue processing.

The client shutdown hook triggers WakeupException.

Stops the client and informs the broker of the shutdown

Calls shutdown from a different thread to stop the client properly

As we move on to the next topic, to go further, we need to understand offsets and how they can be used to control how consumers will read data.

5.1.2 Understanding our coordinates

One of the items that we have only talked about in passing so far is the concept of *offsets*. We use offsets as index positions in the log that the consumer sends to the broker.

This lets the log know which messages it wants to consume and from where. If you think back to our console consumer example, we used the flag `--from-beginning`. This sets the consumer's configuration parameter `auto.offset.reset` to `earliest` behind the scenes. With that configuration, you should see all the records for that topic for the partitions you are connected to, even if they were sent before you started the console consumer. The top part of figure 5.2 shows reading from the start of the log every time you run in this mode.

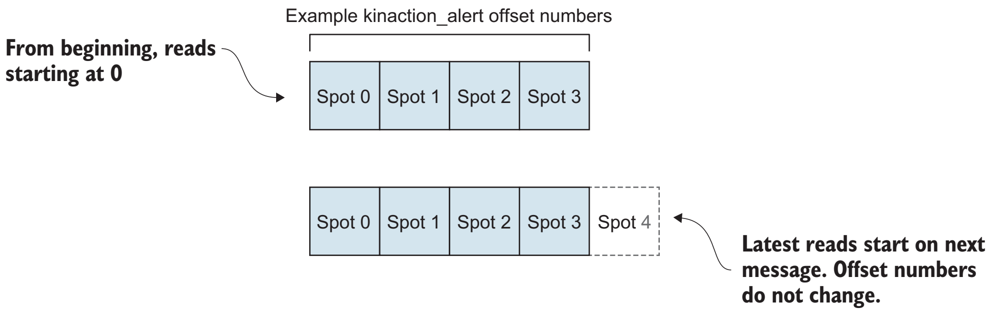
Example kinaction_alert offset numbers
From beginning, reads starting at 0
Spot 0
Spot 1
Spot 2
Spot 3
Spot 0
Spot 1
Spot 2
Spot 3
Spot 4
Latest reads start on next message. Offset numbers do not change.

Figure 5.2 Kafka offsets [6]

If you don't add the option `auto.offset.reset`, the default is `latest`. Figure 5.2 shows this mode as well. In this case, you will not see any messages from the producer unless you send them after you start the consumer. This option says to disregard processing the messages that already are in the topic partition your consumer is reading from; we only want to process what comes in after the consumer client starts polling the topic. You can think of this as an infinite array that has an index starting at `0`. However, there are no updates allowed for an index. Any changes need to be appended to the end of the log.

Note that offsets always increase for each partition. Once a topic partition has seen offset 0, even if that message is removed at a later point, the offset number is not used again. Some of you might have run into the issue of numbers that keep increasing until they hit the upper bound of a data type. Each partition has its own offset sequence, so the hope is that the risk will be low.

For a message written to a topic, what are the coordinates to find the message? First, we would find the partition within the topic that it was written to, and then we would find the index-based offset. As figure 5.3 shows, consumers usually read from the consumer's partition leader replica. This consumer leader replica could be different from any producer's leader replica due to changes in leadership over time; however, they are generally similar in concept.

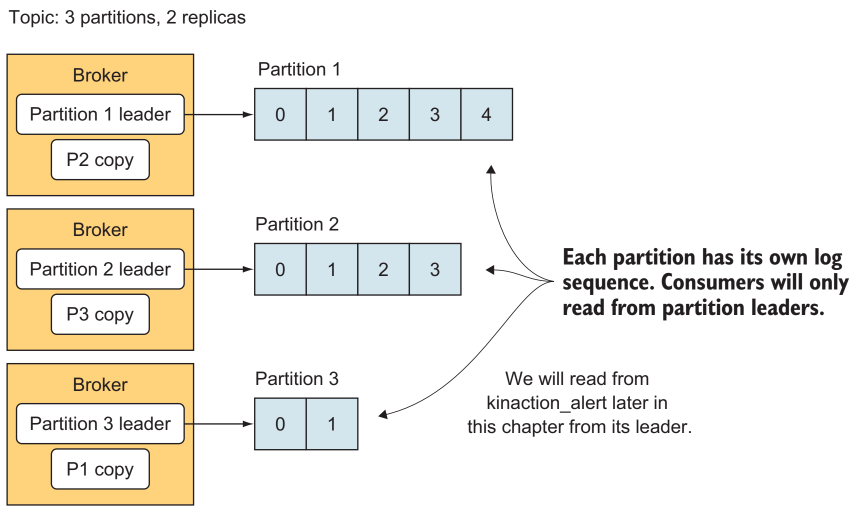
Topic: 3 partitions, 2 replicas
Broker
Partition 1 leader
P2 copy
Broker
Partition 2 leader
P3 copy
Broker
Partition 3 leader
P1 copy
Partition 1
0 1 2 3 4
Partition 2
0 1 2 3
Partition 3
0 1
Each partition has its own log sequence. Consumers will only read from partition leaders.
We will read from kinaction_alert later in this chapter from its leader.

Figure 5.3 Partition leaders

Also, when we talk about partitions, it is okay to have the same offset number across partitions. The ability to tell messages apart needs to include the details of which partition we are talking about within a topic, as well as the offset.

As a side note, if you do need to fetch from a follower replica due to an issue like network latency concerns (for example, having a cluster that stretches across data centers), KIP-392 introduced this ability in version 2.4.0 [7]. As you are starting out with your first clusters, we recommend starting with the default behavior and only reaching for this feature as it becomes necessary to impart a real impact. If you do not have your cluster across different physical sites, you likely will not need this feature at the current time.

Partitions play an important role in how we can process messages. Although the topic is a logical name for what your consumers are interested in, they will read from the leader replicas of their assigned partitions. But how do consumers figure out which partition to connect to? And not just which partition, but where does the leader exist for that partition? For each group of consumers, a specific broker takes on the role of being a group coordinator [8]. The consumer client talks to this coordinator in order to get a partition assignment along with other details it needs in order to consume messages.

The number of partitions also comes into play when talking about consumption. Some consumers will not get any work with more consumers than partitions. An example would be four consumers and only three partitions. Why might you be okay with that? In some instances, you might want to make sure that a similar rate of consumption occurs if a consumer dies unexpectedly. The *group coordinator* is not only in charge of assigning which consumers read which partitions at the beginning of group startup but also when consumers are added or fail and exit the group [8]. And, in an instance where there are more partitions than consumers, consumers handle more than one partition if needed.

Figure 5.4 shows a generic view of how four consumers read all of the data on the brokers where the subscribed topic has partition leader replicas spread evenly, with one on each of the three brokers. In this figure, the data is roughly the same size, which might not always be the case. One consumer sits ready without work because each partition leader replica is handled by one consumer only.

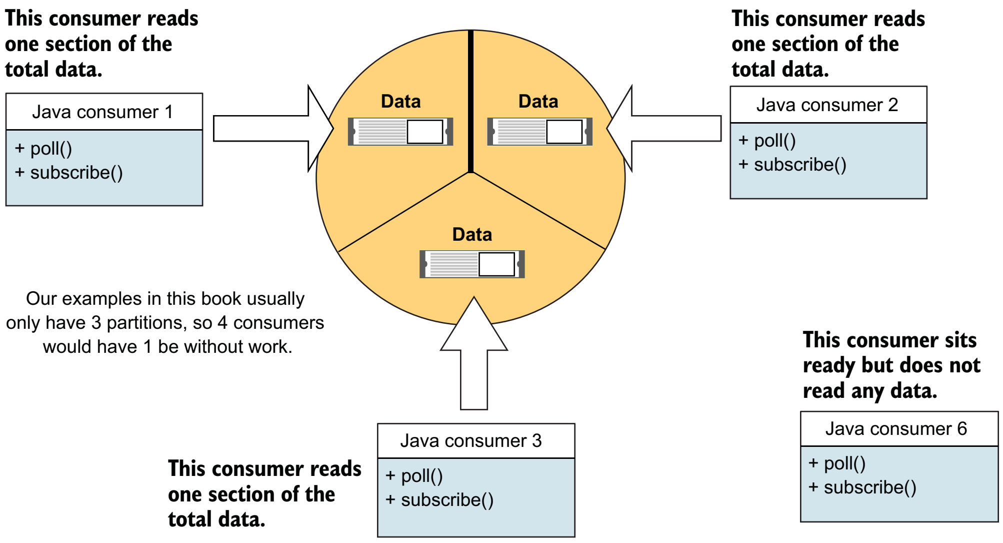
This consumer reads one section of the total data.
Java consumer 1
+ poll()
+ subscribe()
Data
Data
This consumer reads one section of the total data.
Java consumer 2
+ poll()
+ subscribe()
Data
Our examples in this book usually only have 3 partitions, so 4 consumers would have 1 be without work.
This consumer sits ready but does not read any data.
Java consumer 6
+ poll()
+ subscribe()
This consumer reads one section of the total data.
Java consumer 3
+ poll()
+ subscribe()

Figure 5.4 An extra Kafka consumer

Because the number of partitions determines the amount of parallel consumers you can have, some might ask why you don't always choose a large number such as 500 partitions. This quest for higher throughput is not free [9]. This is why you need to choose what best matches the shape of your data flow.

One key consideration is that many partitions might increase end-to-end latency. If milliseconds count in your application, you might not be able to wait until a partition is replicated between brokers [9]. This is key to having in-sync replicas, and it is done before a message is available for delivery to a consumer. You would also need to make sure that you watch the memory usage of your consumers. If you do not have a 1-to-1 mapping of partitions to consumers, each consumer's memory requirements can increase as it is assigned more partitions [9].

If you run across older documentation for Kafka, you might notice consumer client configurations for Apache ZooKeeper. Unless one is using an old consumer client, Kafka does not have consumers rely directly on ZooKeeper. Although consumers used ZooKeeper to store the offsets that they consume to a certain point, now the offsets are often stored inside a Kafka internal topic [10]. As a side note, consumer clients do not have to store their offsets in either of these locations, but this will likely be the case. If you want to manage your own offset storage you can! You can either store it in a local file, in cloud storage with a provider like AWS™, or a database. One of the advantages of moving away from ZooKeeper storage was to reduce the clients' dependency on ZooKeeper.

5.2 How consumers interact

Why is the concept of consumer groups paramount? Probably the most important reason is that scaling is impacted by either adding customers to or removing consumers from a group. Consumers that are not part of the same group do not share the same coordination of offset knowledge.

Listing 5.3 shows an example of a group named `kinaction_team0group`. If you instead make up a new `group.id` (like a random GUID), you will start a new consumer with no stored offsets and with no other consumers in your group [11]. If you join an existing group (or one that had offsets stored already), your consumer can share work with others or can even resume where it left off reading from any previous runs [1].

Listing 5.3 Consumer configuration for consumer group

```
Properties kaProperties = new Properties();
kaProperties.put("group.id", "kinaction_team0group");     <-- group.id determines
                                                              consumer behavior
                                                              with other consumers.
```

It is often the case that you will have many consumers reading from the same topic. An important detail to decide on if you need a new group ID is whether your consumers are working as part of one application or as separate logic flows. Why is this important?

Let's think of two use cases for data that come from a human resource system. One team wonders about the number of hires from specific states, and the other team is more interested in the data for the impact on travel budgets for interviews. Would anyone on the first team care about what the other team is doing or would either of the teams want to consume only a portion of the messages? Likely not! How can we keep this separation? The answer is to assign a separate `group.id` to each application. Each consumer that uses the same `group.id` as another consumer will be considered to be working together to consume the partitions and offsets of the topic as one logical application.

5.3 Tracking

Going through our usage patterns so far, we have not talked too much about how we keep a record of what each client has read. Let's briefly talk about how some message brokers handle messages in other systems. In some systems, consumers do not record what they have read. They pull the message and then it does not exist on a queue after acknowledgment. This works well for a single message that needs to have exactly one application process it. Some systems use topics in order to publish the message to all those that are subscribers. And often, future subscribers will have missed this message entirely because they were not actively part of the receiver list when the event happened.

Figure 5.5 shows non-Kafka message broker scenarios, including how messages are often removed after consumption. It also shows a second pattern where a message might come from the original source and then be replicated to other queues. In systems where the message would be consumed and not available for more than one consumer, this approach is needed so that separate applications each get a copy.

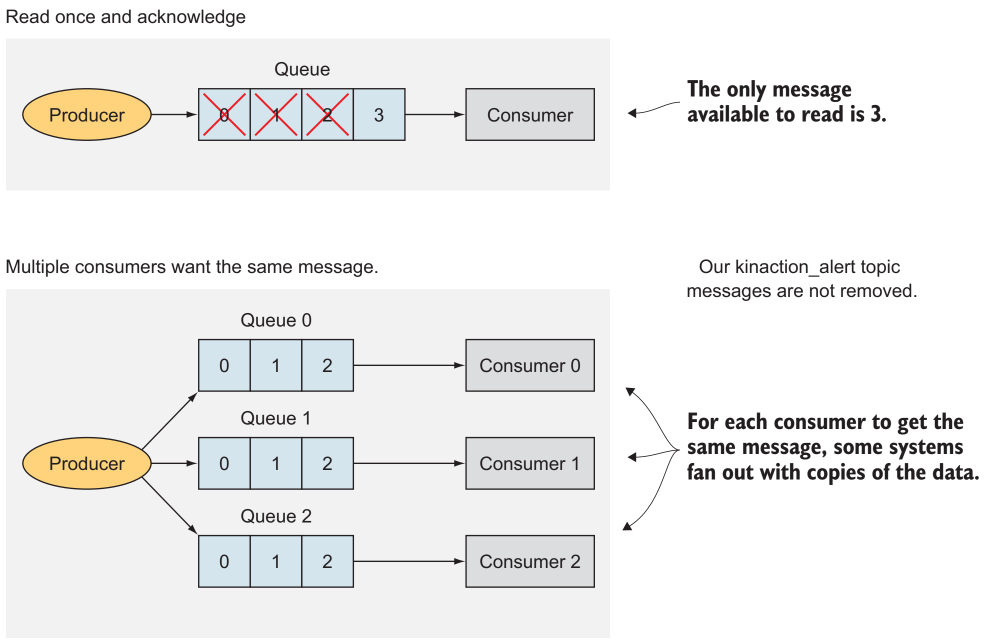
Read once and acknowledge
Queue
Producer
0
1
2
3
Consumer
The only message available to read is 3.
Multiple consumers want the same message.
Our kinaction_alert topic messages are not removed.
Queue 0
Queue 1
Queue 2
Producer
0 1 2
Consumer 0
Consumer 1
Consumer 2
For each consumer to get the same message, some systems fan out with copies of the data.

Figure 5.5 Other broker scenarios

You can imagine that the copies grow in number as an event becomes a popular source of information. Rather than have entire copies of the queue (besides those for replication or failover), Kafka can serve multiple applications from the same partition leader replica.

Kafka, as we mentioned in the first chapter, is not limited to having only one consumer. Even if a consuming application does not exist when a message is first created on a topic, as long as Kafka retains the message in its log, then it can still process the data. Because messages are not removed from other consumers or delivered once, consumer clients need a way to keep a record of where they have read in the topic. In addition, because many applications can read the same topic, it is important that the offsets and partitions are specific to a certain consumer group. The key coordinates to let your consumer clients work together is a unique blend of the following: group, topic, and partition number.

5.3.1 Group coordinator

As mentioned earlier, the group coordinator works with the consumer clients to keep a record of where inside the topic that specific group has read [8]. The partition's coordinates of a topic and group ID make it specific to an offset value.

Looking at figure 5.6, notice that we can use the offset commits as coordinates to find out where to read from next. For example, in the figure, a consumer that is part of a group called `kinaction_teamoffka0` and is assigned partition 0 would be ready to read offset 3 next.

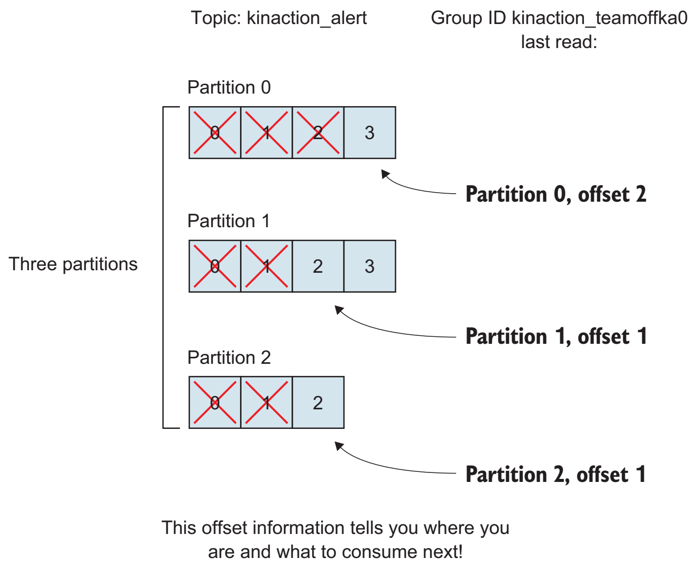

Figure 5.6 Coordinates

Figure 5.7 shows a scenario where the same partitions of interest exist on three separate brokers for two different consumer groups, `kinaction_teamoffka0` and `kinaction_teamsetka1`. The consumers in each group will get their own copy of the data from the partitions on each broker. They do not work together unless they are part of the same group. Correct group membership is important for each group to have their metadata managed accurately.

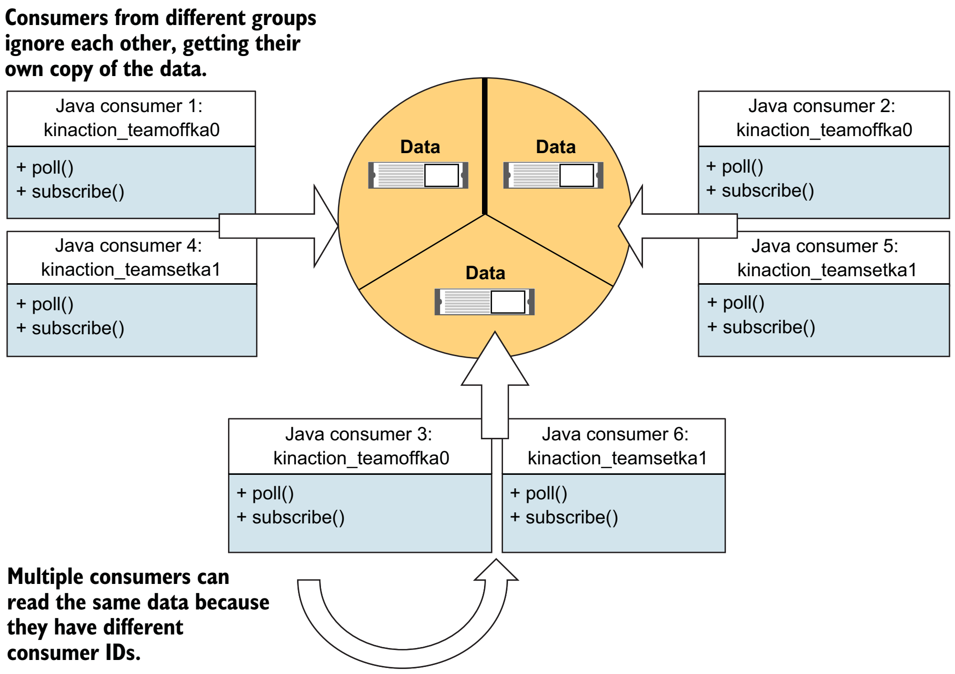
Consumers from different groups ignore each other, getting their own copy of the data.
Java consumer 1: kinaction_teamoffka0
+ poll()
+ subscribe()
Java consumer 4: kinaction_teamsetka1
+ poll()
+ subscribe()
Data
Data
Data
Java consumer 2: kinaction_teamoffka0
+ poll()
+ subscribe()
Java consumer 5: kinaction_teamsetka1
+ poll()
+ subscribe()
Java consumer 3: kinaction_teamoffka0
+ poll()
+ subscribe()
Java consumer 6: kinaction_teamsetka1
+ poll()
+ subscribe()
Multiple consumers can read the same data because they have different consumer IDs.

Figure 5.7 Consumers in separate groups [12]

As a general rule, only one consumer per consumer group can read one partition. In other words, whereas a partition might be read by many consumers, it can only be read by one consumer from each group at a time. Figure 5.8 highlights how one consumer can read two partitions leader replicas, where the second consumer can only read the data from a third partition leader [8]. A single partition replica is not to be divided or shared between more than one consumer with the same ID.

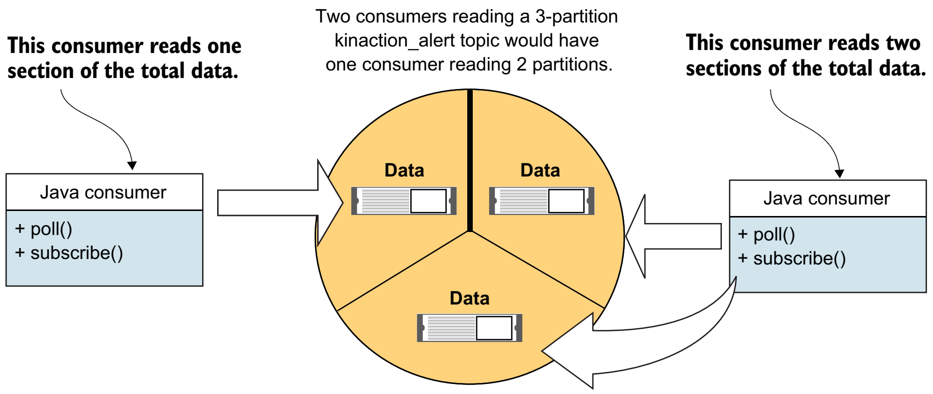
This consumer reads one section of the total data.
Two consumers reading a 3-partition kinaction_alert topic would have one consumer reading 2 partitions.
This consumer reads two sections of the total data.
Java consumer
+ poll()
+ subscribe()
Data
Data
Data
Java consumer
+ poll()
+ subscribe()

Figure 5.8 Kafka consumers in a group

One of the neat things about being part of a consumer group is that when a consumer fails, the partitions that it was reading are reassigned [8]. An existing consumer takes the place of reading a partition that was once read by the consumer that dropped out of the group.

Table 5.1 listed `heartbeat.interval.ms`, which determines the amount of pings to the group coordinator [13]. This heartbeat is the way that the consumer communicates with the coordinator to let it know it is still replying in a timely fashion and working away diligently [8].

Failure by a consumer client to send a heartbeat over a period of time can happen in a couple of ways, like stopping the consumer client by either termination of the process or failure due to a fatal exception. If the client isn't running, it cannot send messages back to the group coordinator [8].

5.3.2 Partition assignment strategy

One item that we will want to be aware of is how consumers get assigned to partitions. This matters since it will help you figure out how many partitions each of your consumers might be taxed with processing. The property `partition.assignment.strategy` is what determines which partitions are assigned to each consumer [14]. `Range` and `RoundRobin` are provided, as are `Sticky` and `CooperativeSticky` [15].

The *range assigner* uses a single topic to find the number of partitions (ordered by number) and then is broken down by the number of consumers. If the split is not even, then the first consumers (using alphabetical order) get the remaining partitions [16]. Make sure that you employ a spread of partitions that your consumers can handle and consider switching the assignment strategy if some consumer clients use all their resources, though others are fine. Figure 5.9 shows how three clients will grab three out of seven total partitions and end up with more partitions than the last client.

The *round-robin* strategy is where the partitions are uniformly distributed down the row of consumers [1]. Figure 5.9 is a modified figure from the article "What I have learned from Kafka partition assignment strategy," which shows an example of three clients that are part of the same consumer group and assigned in a round-robin fash-

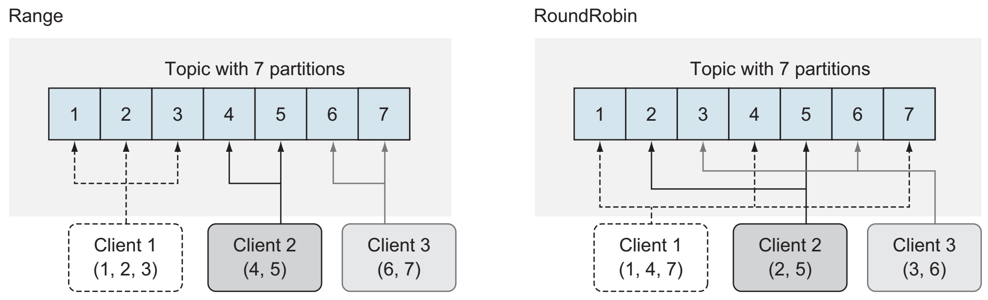

Figure 5.9 Partition assignments

ion for one topic made of seven partitions [17]. The first consumer gets the first partition, the second consumer the second, and so on until the partitions run out.

The *sticky* strategy was added in version 0.11.0 [18]. However, since we will use range assigner in most of our examples internally and already looked at round-robin as well, we will not dig into `Sticky` and `CooperativeSticky`.

5.4 Marking our place

One of the important things to think about is your need for assuring that your applications read all messages from your topic. Is it okay to miss a few, or do you need each message confirmed as it's read? The real decision comes down to your requirements and any trade-offs you are willing to make. Are you okay with sacrificing some speed in order to have a safer method of seeing each message? These choices are discussed in this section.

One option is to use `enable.auto.commit` set to `true`, the default for consumer clients [19]. Offsets are committed on our behalf. One of the nicest parts of this option is that we do not have to make any other calls to commit the offsets that are consumed.

Kafka brokers resend messages if they are not automatically acknowledged due to a consumer client failure. But what sort of trouble can we get into? If we process messages that we get from our latest poll, say, in a separate thread, the automatic commit offset can be marked as being read even if everything is not actually done with those specific offsets. What if we had a message fail in our processing that we would need to retry? With our next poll, we could get the next set of offsets after what was already committed as being consumed [8]. It is possible and easy to lose messages that look like they have been consumed despite not being processed by your consumer logic.

When looking at what you commit, notice that timing might not be perfect. If you do not call a commit method on a consumer with metadata noting your specific offset to commit, you might have some undefined behavior based on the timing of polls, expired timers, or even your own threading logic. If you need to be sure to commit a record at a specific time as you process it or a specific offset in particular, you should make sure that you send the offset metadata into the commit method.

Let's explore this topic more by talking about using code-specific commits enabled by `enable.auto.commit` set to `false`. This method can be used to exercise the most management over when your application actually consumes a message and commits it. At-least-once delivery guarantees can be achieved with this pattern.

Let's talk about an example in which a message causes a file to be created in Hadoop in a specific location. As you get a message, you poll a message at offset 999. During processing, the consumer stops because of an error. Because the code never actually committed offset 999, the next time a consumer of that same group starts reading from that partition, it gets the message at offset 999 again. By receiving the message twice, the client was able to complete the task without missing the message. On the flip side, you did get the message twice! If for some reason your processing actually works and you achieve a successful write, your code has to handle the fact that you might have duplicates.

Now let's look at some of the code that we would use to control our offsets. As we did with a producer when we sent a message earlier, we can also commit offsets in a synchronous or asynchronous manner. Listing 5.4 shows a synchronous commit. Looking at that listing for `commitSync`, it is important to note that the commit takes place in a manner that blocks any other progress in the code until a success or failure occurs [20].

Listing 5.4 Waiting on a commit

```
        consumer.commitSync();
#// Any code here will wait on line before
```

commitSync waits for a success or fail.

As with producers, we can also use a callback. Listing 5.5 shows how to create an asynchronous commit with a callback by implementing the `OffsetCommitCallback` interface (the `onComplete` method) with a lambda expression [21]. This instance allows for log messages to determine our success or failure even though our code does not wait before moving on to the next instruction.

Listing 5.5 Commit with a callback

```
public static void commitOffset(long offset,
                                 int partition,
                                 String topic,
                                 KafkaConsumer<String, String> consumer) {
    OffsetAndMetadata offsetMeta = new OffsetAndMetadata(++offset, "");

    Map<TopicPartition, OffsetAndMetadata> kaOffsetMap = new HashMap<>();
    kaOffsetMap.put(new TopicPartition(topic, partition), offsetMeta);

    consumer.commitAsync(kaOffsetMap, (map, e) -> {
      if (e != null) {
        for (TopicPartition key : map.keySet()) {
          log.info("kinaction_error: offset {}", map.get(key).offset());
        }
      } else {
        for (TopicPartition key : map.keySet()) {
          log.info("kinaction_info: offset {}", map.get(key).offset());
        }
      }
    });
  }
```

A lambda that creates an OffsetCommitCallback instance

If you think back to chapter 4, this is similar to how we used asynchronous sends with a callback for acknowledgments. To implement your own callback, you need to use the interface `OffsetCommitCallback`. You can define an `onComplete` method definition to handle exceptions or successes as needed.

Why would you want to choose synchronous or asynchronous commit patterns? Keep in mind that your latency is higher if you wait for a blocking call. This time factor might be worth the delay if your requirements include needs for data consistency [21]. These decisions help determine the amount of control you need to exercise when informing Kafka which messages your logic considers as processed.

5.5 Reading from a compacted topic

Consumers should be made aware of reading from a compacted topic. Kafka compacts the partition log in a background process, and records with the same key might be removed except for the last one. Chapter 7 will go further into how these topics work, but in short, we need to update records that have the same key value. If you do not need a history of messages, but rather just the last value, you might wonder how this concept works with an immutable log that only adds records to the end. The biggest "gotcha" for consumers that might cause an error is that when reading records from a compacted topic, consumers can still get multiple entries for a single key [22]! How is this possible? Because compaction runs on the log files that are on disk, compaction may not see every message that exists in memory during cleanup.

Clients need to handle this case, where there is more than one value per key. We should have the logic in place to handle duplicate keys and, if needed, ignore all but the last value. To pique your interest about compacted topics, note that Kafka uses its own compacted internal topic, called `__consumer_offsets`, which relates directly to your consumer offsets themselves [23]. Compaction makes sense here because for a specific combination of a consumer group, partition, and topic, only the latest value is needed as it will have the latest offset consumed.

5.6 Retrieving code for our factory requirements

Let's try to use the information we gathered about how consumers work to see if we can start working on our own solutions designed in chapter 3 for use with Kafka in our e-bike factory but from the consumer client perspective. As noted in chapter 3, we want to ensure that we do not lose any audit messages when operators complete commands against the sensors. First, let's look at the options we have in reading our offsets.

5.6.1 Reading options

Although there is no lookup of a message by a key option in Kafka, it is possible to seek to a specific offset. Thinking about our log of messages being an ever increasing array with each message having an index, we have a couple of options for this, including starting from the beginning, going to the end, or finding offsets based on specific times. Let's take a look at these options.

One issue that we might run into is that we want to read from the beginning of a topic even if we have already done so. Reasons could include logic errors and a desire to replay the entire log or a failure in our data pipeline after starting with Kafka. The important configuration to set for this behavior is `auto.offset.reset` to `earliest`[24]. Another technique that we can use is to run the same logic but use a different group ID. In effect, this means that the commit offset topics that Kafka uses internally will not be able to find an offset value but will be able to start at the first index found because the commit offset topic does not have any data on the new consumer group.

Listing 5.6 is an example of setting the property `auto.offset.reset` to `"earliest"` to seek to a specific offset [24]. Setting a group ID to a random UUID also helps to achieve starting with no offset history for a consumer group. This is the type of reset we could use to look at `kinaction_alerttrend` with different code logic to determine trends against all of the data in that topic.

Listing 5.6 Earliest offset

```
Properties kaProperties = new Properties();
kaProperties.put("group.id",
                 UUID.randomUUID().toString());      <- Creates a group ID for which Kafka does not have a stored offset
kaProperties.put("auto.offset.reset", "earliest");   <- Uses the earliest offset retained in our logs
```

Sometimes you just want to start your logic from when the consumers start up and forget about past messages [24]. Maybe the data is already too old to have business value in your topic. Listing 5.7 shows the properties you would set to get this behavior of starting with the latest offset. If you want to make sure that you don't find a previous consumer offset and want to instead default to the latest offset Kafka has for your subscriptions, using a UUID isn't necessary except for testing. If we are only interested about new alerts coming into our `kinaction_alert topic`, this might be a way for a consumer to see only those alerts.

Listing 5.7 Latest offset

```
Properties kaProperties = new Properties();
kaProperties.put("group.id",
                 UUID.randomUUID().toString());    <- Creates a group ID for which Kafka does not have a stored offset
kaProperties.put("auto.offset.reset", "latest");   <- Uses the latest record offset
```

One of the trickier offset search methods is `offsetsForTimes`. This method allows you to send a map of topics and partitions as well as a timestamp for each in order to get a map back of the offset and timestamp for the given topics and partitions [25]. This can be useful in situations where a logical offset is not known, but a timestamp is known. For example, if you have an exception related to an event that was logged, you might be able to use a consumer to determine the data that was processed around your specific timestamp. Trying to locate an audit event by time might be used for our topic `kinaction_audit` to locate commands happening as well.

As listing 5.8 shows, we have the ability to retrieve the offset and timestamps per a topic or partition when we map each to a timestamp. After we get our map of metadata returned from the `offsetsForTimes` call, we then can seek directly to the offset we are interested in by seeking to the offset returned for each respective key.

Listing 5.8 Seeking to an offset by timestamps

```
...
Map<TopicPartition, OffsetAndTimestamp> kaOffsetMap =
consumer.offsetsForTimes(timeStampMapper);        ◁ Finds the first offset greater or
...                                                  equal to that timeStampMapper
// We need to use the map we get
consumer.seek(partitionOne,
  kaOffsetMap.get(partitionOne).offset());     ◁ Seeks to the first offset
                                                  provided in kaOffsetMap
```

One thing to be aware of is that the offset returned is the first message with a timestamp that meets your criteria. However, due to the producer resending messages on failures or variations in when timestamps are added (by consumers, perhaps), times might appear out of order.

Kafka also gives you the ability to find other offsets as can be referenced in the consumer Javadoc [26]. With all of these options, let's see how they apply to our use case.

5.6.2 Requirements

One requirement for our audit example was that there is no need to correlate (or group together) any events across the individual events. This means that there are no concerns on the order or need to read from specific partitions; any consumer reading any partition should be good. Another requirement was to not lose any messages. A safe way to make sure that our logic is executed for each audit event is to specifically commit the offset per record after it is consumed. To control the commit as part of the code, we can set `enable.auto.commit` to `false`.

Listing 5.9 shows an example of leveraging a synchronous commit after each record is processed for the audit feature. Details of the next offset to consume in relation to the topic and partition of the offset that was just consumed are sent as a part of each loop through the records. One "gotcha" to note is that it might seem odd to add 1 to the current offset, but the offset sent to your broker is supposed to be your future index. The method `commitSync` is called and passed the offset map containing the offset of the record that was just processed [20].

Listing 5.9 Audit consumer logic

```
...
kaProperties.put("enable.auto.commit", "false");    ◁ Sets autocommit
                                                       to false
try (KafkaConsumer<String, String> consumer =
     new KafkaConsumer<>(kaProperties)) {

      consumer.subscribe(List.of("kinaction_audit"));

      while (keepConsuming) {
        var records = consumer.poll(Duration.ofMillis(250));
        for (ConsumerRecord<String, String> record : records) {
          // audit record process ...
          OffsetAndMetadata offsetMeta =
            new OffsetAndMetadata(++record.offset(), "");

          Map<TopicPartition, OffsetAndMetadata> kaOffsetMap =
            new HashMap<>();
          kaOffsetMap.put(
            new TopicPartition("kinaction_audit",
                               record.partition()), offsetMeta);

          consumer.commitSync(kaOffsetMap);
        }
      }
    }
...
```

Adding a record to the current offset determines the next offset to read.

Allows for a topic and partition key to be related to a specific offset

Commits the offsets

Another goal of the design for our e-bike factory was to capture our alert status and monitor the alert trend over time. Even though we know our records have a key that is the stage ID, there is no need to consume one group at a time or worry about the order. Listing 5.10 shows how to set the `key.deserializer` property so the consumer knows how to deal with the binary data that was stored in Kafka when we produced the message. In this example, `AlertKeySerde` is used for the key to deserialize. Because message loss isn't a huge concern in our scenario, allowing autocommit of messages is good enough in this situation.

Listing 5.10 Alert trending consumer

```
...
kaProperties.put("enable.auto.commit", "true");
kaProperties.put("key.deserializer",
  AlertKeySerde.class.getName());
kaProperties.put("value.deserializer",
  "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<Alert, String> consumer =
  new KafkaConsumer<Alert, String>(kaProperties);
consumer.subscribe(List.of("kinaction_alerttrend"));

while (true) {
    ConsumerRecords<Alert, String> records =
    consumer.poll(Duration.ofMillis(250));
    for (ConsumerRecord<Alert, String> record : records) {
        // ...
    }
}
...
```

Uses autocommit as lost messages are not an issue

AlertKeySerde key deserializer

Another large requirement is to have any alerts quickly processed to let operators know about critical issues. Because the producer in chapter 4 used a custom `Partitioner`, we will assign a consumer directly to that same partition to alert us to critical issues. Because a delay in case of other alerts is not desirable, the commit will be for each offset in an asynchronous manner.

Listing 5.12 shows the consumer client logic focused on critical alerts assigning themselves to the specific topic and partition that is used for producing alerts when the custom partitioner class `AlertLevelPartitioner` is used. In this case, it is partition 0 and topic `kinaction_alert`.

We use `TopicPartition` objects to tell Kafka which specific partitions we are interested in for a topic. Passing the `TopicPartition` objects to the `assign` method takes the place of allowing a consumer to be at the discretion of a group coordinator assignment [27].

For listing 5.11, each record that comes back from the consumer poll, an asynchronous commit is used with a callback. A commit of the next offset to consume is sent to the broker and should not block the consumer from processing the next record, per our requirements. The options in the following listing seem to satisfy our core design requirements from chapter 3.

Listing 5.11 Alert consumer

```
kaProperties.put("enable.auto.commit", "false");

KafkaConsumer<Alert, String> consumer =
  new KafkaConsumer<Alert, String>(kaProperties);
TopicPartition partitionZero =
  new TopicPartition("kinaction_alert", 0);          <- Uses TopicPartition for critical messages
consumer.assign(List.of(partitionZero));             <- Consumer assigns itself the partition rather than subscribing to the topic

while (true) {
    ConsumerRecords<Alert, String> records =
      consumer.poll(Duration.ofMillis(250));
    for (ConsumerRecord<Alert, String> record : records) {
        // ...
        commitOffset(record.offset(),
          record.partition(), topicName, consumer);   <- Commits each record asynchronously
    }
}

...
public static void commitOffset(long offset,int part, String topic,
  KafkaConsumer<Alert, String> consumer) {
    OffsetAndMetadata offsetMeta = new OffsetAndMetadata(++offset, "");

    Map<TopicPartition, OffsetAndMetadata> kaOffsetMap =
      new HashMap<TopicPartition, OffsetAndMetadata>();
    kaOffsetMap.put(new TopicPartition(topic, part), offsetMeta);

    OffsetCommitCallback callback = new OffsetCommitCallback() {
    ...
    };
    consumer.commitAsync(kaOffsetMap, callback);      <- The asynchronous commit uses the kaOffsetMap and callback arguments.
}
```

Overall, the consumer can be a complex piece of our interactions with Kafka. Some options can be done with property configurations alone, but if not, you can use your knowledge of topics, partitions, and offsets to navigate your way to the data you need.

Summary

- Consumer clients provide developers with a way to get data out of Kafka. As with producer clients, consumer clients have a large number of available configuration options we can set rather than using custom coding.
- Consumer groups allow more than one client to work as a group to process records. With grouping, clients can process data in parallel.
- Offsets represent the position of a record in the commit log that exists on a broker. By using offsets, consumers can control where they want to start reading data.
- An offset can be a previous offset that consumers have already seen, which gives us the ability to replay records.
- Consumers can read data in a synchronous or an asynchronous manner.
- If asynchronous methods are used, the consumer can use code in callbacks to run logic once data is received.

References

1 S. Kozlovski. “Apache Kafka Data Access Semantics: Consumers and Membership.” Confluent blog (n.d.). https://www.confluent.io/blog/apache-kafka-data-access-semantics-consumers-and-membership (accessed August 20, 2021).

2 “Consumer Configurations.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html (accessed June 19, 2019).

3 N. Narkhede. “Apache Kafka Hits 1.1 Trillion Messages Per Day – Joins the 4 Comma Club.” Confluent blog (September 1, 2015). https://www.confluent.io/blog/apache-kafka-hits-1-1-trillion-messages-per-day-joins-the-4-comma-club/ (accessed October 20, 2019).

4 “Class KafkaConsumer<K,V>.” Kafka 2.7.0 API. Apache Software Foundation (n.d.). https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html#poll-java.time.Duration- (accessed August 24, 2021).

5 “Class WakeupException.” Kafka 2.7.0 API. Apache Software Foundation (n.d.). https://kafka.apache.org/27/javadoc/org/apache/kafka/common/errors/WakeupException.html (accessed June 22, 2020).

6 “Documentation: Topics and Logs.” Confluent documentation (n.d.). https://docs.confluent.io/5.5.1/kafka/introduction.html#topics-and-logs (accessed October 20, 2021).

7 “KIP-392: Allow consumers to fetch from closest replica.” Wiki for Apache Kafka. Apache Software Foundation (November 05, 2019). https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica (accessed December 10, 2019).

8 J. Gustafson. "Introducing the Kafka Consumer: Getting Started with the New Apache Kafka 0.9 Consumer Client." Confluent blog (January 21, 2016). https://www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/ (accessed June 01, 2020).

9 J. Rao. "How to choose the number of topics/partitions in a Kafka cluster?" Confluent blog (March 12, 2015). https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/ (accessed May 19, 2019).

10 "Committing and fetching consumer offsets in Kafka." Wiki for Apache Kafka. Apache Software Foundation (March 24, 2015). https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=48202031 (accessed December 15, 2019).

11 "Consumer Configurations: group.id." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_group.id (accessed May 11, 2018).

12 "Documentation: Consumers." Apache Software Foundation (n.d.). https://kafka.apache.org/23/documentation.html#intro_consumers (accessed December 11, 2019).

13 "Consumer Configurations: heartbeat.interval.ms." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_heartbeat.interval.ms (accessed May 11, 2018).

14 "Consumer Configurations: partition.assignment.strategy." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_partition.assignment.strategy (accessed December 22, 2020).

15 S. Blee-Goldman. "From Eager to Smarter in Apache Kafka Consumer Rebalances." Confluent blog (n.d.). https://www.confluent.io/blog/cooperative-rebalancing-in-kafka-streams-consumer-ksqldb/ (accessed August 20, 2021).

16 "RangeAssignor.java." Apache Kafka GitHub (n.d.). https://github.com/apache/kafka/blob/c9708387bb1dd1fd068d6d8cec2394098d5d6b9f/clients/src/main/java/org/apache/kafka/clients/consumer/RangeAssignor.java (accessed August 25, 2021).

17 A. Li. "What I have learned from Kafka partition assignment strategy." Medium (December 1, 2017). https://medium.com/@anyili0928/what-i-have-learned-from-kafka-partition-assignment-strategy-799fdf15d3ab (accessed October 20, 2021).

18 "Release Plan 0.11.0.0." Wiki for Apache Kafka. Apache Software Foundation (June 26, 2017). https://cwiki.apache.org/confluence/display/KAFKA/Release+Plan+0.11.0.0 (accessed December 14, 2019).

19 "Consumer Configurations: enable.auto.commit." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_enable.auto.commit (accessed May 11, 2018).

20 Synchronous Commits. Confluent documentation (n.d.). https://docs.confluent.io/3.0.0/clients/consumer.html#synchronous-commits (accessed August 24, 2021).

21 Asynchronous Commits. Confluent documentation (n.d.). https://docs.confluent.io/3.0.0/clients/consumer.html#asynchronous-commits (accessed August 24, 2021).

22 Kafka Design. Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/design.html (accessed August 24, 2021).

23 Kafka Consumers. Confluent documentation (n.d.). https://docs.confluent.io/3.0.0/clients/consumer.html (accessed August 24, 2021).

24 "Consumer Configurations: auto.offset.reset." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#consumerconfigs_auto.offset.reset (accessed May 11, 2018).

25 `offsetsForTimes`. Kafka 2.7.0 API. Apache Software Foundation (n.d.). https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/Consumer.html#offsetsForTimes-java.util.Map- (accessed June 22, 2020).

26 `seek`. Kafka 2.7.0 API. Apache Software Foundation (n.d.). https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/Consumer.html#seek-org.apache.kafka.common.TopicPartition-long- (accessed June 22, 2020).

27 `assign`. Kafka 2.7.0 API. Apache Software Foundation (n.d.). https://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html#assign-java.util.Collection- (accessed August 24, 2021).