7 Topics and partitions

This chapters covers

- Creation parameters and configuration options
- How partitions exist as log files
- How segments impact data inside partitions
- Testing with `EmbeddedKafkaCluster`
- Topic compaction and how data can be retained

In this chapter, we will look further into how we might store our data across topics as well as how to create and maintain topics. This includes how partitions fit into our design considerations and how we can view our data on the brokers. All of this information will help us as we also look at how to make a topic update data rather than appending it to a log.

7.1 Topics

To quickly refresh our memory, it is important to know that a topic is a non-concrete concept rather than a physical structure. It does not usually exist on only one broker. Most applications consuming Kafka data view that data as being in a

single topic; no other details are needed for them to subscribe. However, behind the topic name are one or more partitions that actually hold the data [1]. Kafka writes the data that makes up a topic in the cluster to logs, which are written to the broker filesystems.

Figure 7.1 shows partitions that make up one topic named `kinaction_helloworld`. A single partition's copy is not split between brokers and has a physical footprint on each disk. Figure 7.1 also shows how those partitions are made up of messages that are sent to the topic.

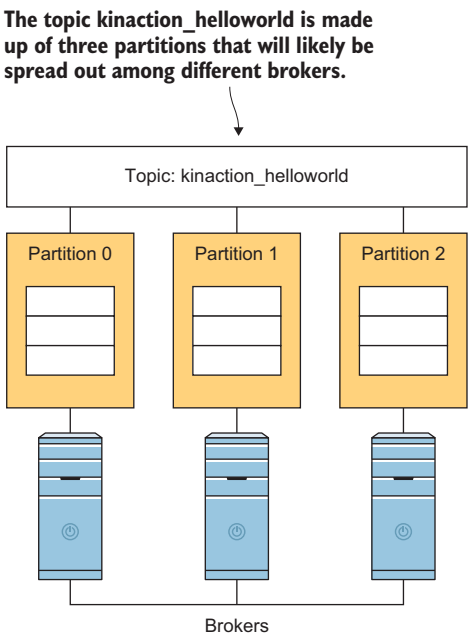

Figure 7.1 Example topic with partitions

If writing to a topic is so simple in getting-started examples, why do we need to understand the role and pieces that make up a topic? At the highest level, this impacts how our consumers get to the data. Let's say that our company is selling spots for a training class using a web-based application that sends the events of user actions into our Kafka cluster. Our overall application process could generate droves of events. For example, there would be an event for the initial search on the location, one for the specific training being selected by the customer, and a third for classes that are confirmed. Should the producing applications send all of this data to a single topic or several topics? Is each message a specific type of event, and should each remain separated in different topics? There are adjustments with each approach and some things to consider that will help us determine the best method to take in every situation.

We see topic design as a two-step process. The first looks at the events we have. Do they belong in one topic or more than one? The second considers each topic. What is the number of partitions we should use? The biggest takeaway is that partitions are a per-topic design question and not a cluster-wide limitation or mandate. Although we can set a default number of partitions for topic creation, in most cases, we should consider how the topic will be used and what data it will hold.

We should have a solid reason to pick a specific number of partitions. Jun Rao wrote a fantastic article titled "How to choose the number of topics/partitions in a Kafka cluster?" on the Confluent blog about this very subject [2]! Let's say that we want to have a partition for each server as a generic rule. However, because we have one partition on each server does not mean producers will write evenly among them. To do so, we would have to ensure that each partition leader is spread out in that manner and stays that way.

We also need to get familiar with our data. Let's take a look at a list of items to think about, both in general and in this training class scenario:

- Data correctness
- The volume of messages of interest per consumer
- How much data you will have or need to process

Data correctness is at the top of most data concerns in real-world designs. This term could be considered vague, so our defintion is explained here as our opinion. With regard to topics, this involves making sure that events that must be ordered end up in the same partition and, thus, the same topic. Although we can place events by our consumers in an order based on a timestamp, it is more trouble (and error prone) to handle cross-topic event coordination than it is worth, in our opinion. If we use keyed messages and need those in order, we should care about partitions and any future changes to those partitions [1].

For data correctness with our three previous example events, it might be helpful to place the events with a message key (including the student ID) in two separate topics for the actual booked and confirmed/billed events. These events are student-specific, and this approach would be helpful to ensure that confirmation of a class occurs for that specific student. The search events themselves, however, may not be of interest or need to be ordered for a specific student if, for example, our analytics team is looking for the most popular searched cities rather than student information.

Next, we should consider the *volume of messages* of interest per consumer. For our theoretical training system, let's look at the number of events as we consider the topic placement. The search events themselves would far outnumber the other events. Let's say that a training location near a large city gets 50,000 searches a day but only has room for 100 students. Traffic on most days produces 50,000 search events and fewer than 100 actual booked training events. Will our confirmation team have an application that would want to subscribe to a generic event topic in which it uses or cares about less than 1% of the total messages? Most of the consumer's time would be, in effect, filtering out the mass of events to process only a select few.

Another point to account for is *the quantity of data* we will be processing. Will the number of messages require multiple consumers to be running in order to process within the time constraints required by our applications? If so, we have to be aware of how the number of consumers in a group is limited by the partitions in our topic [2]. It is easier at this point to create more partitions than we think we might require. Having more capacity for consumers to grow allows us to increase in volume without having to deal with repartitioning data. However, it is important to know that partitions are not an unlimited free resource, as talked about in Rao's article that we mentioned earlier. It also means having more brokers to migrate in case of a broker failure, which could be a potential headache in the making.

It's best to find a happy medium and to go with that as we design our systems. Figure 7.2 shows how our design might be best suited to two topics for the three event

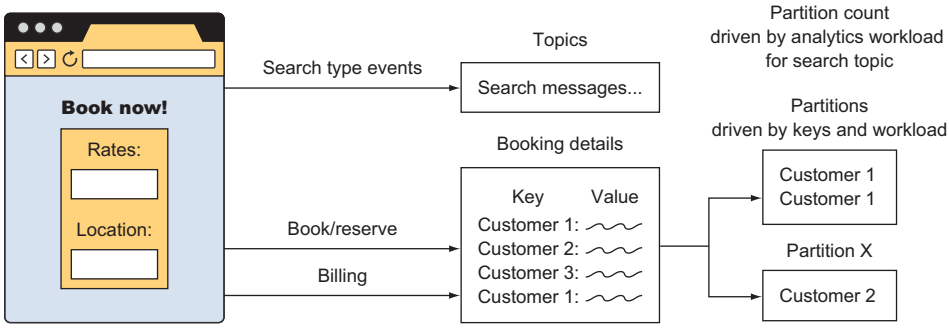

Figure 7.2 Example training event topic design

types we used in our scenario. As always, more requirements or details can change our future implementations.

A last thing to consider when deciding on the number of partitions for a topic is that reducing that number is not currently supported [3]. There may be ways to do this, but it is definitely not advised! Let's take a moment to think about why this would not be desirable.

When consumers subscribe to a topic, they really are attached to a partition. The removal of a partition could lose its current position when or if a consumer starts reading from a reassigned partition. This is where we need to make sure our keyed messages and consuming clients can follow any changes we make at the broker level. We impact consumers with our actions. Now that we've discussed topic design, let's dig a little deeper into the options that we can set when creating topics. We touched on these briefly when we created topics to produce messages in chapter 3, so we'll dive a bit deeper here.

7.1.1 Topic-creation options

Kafka topics have a couple of core options that must be set in order to create a topic. Although we have created topics since chapter 2 (with our `kinaction_helloworld` topic), we need to make sure we dig into the basic parameters that were glossed over. For these parameters, it's best to treat these decisions with thought and care and be intentional [4].

Another important decision to make at creation time is if you will ever need to delete a topic. Because this operation is significant, we want to make sure it cannot happen without a logical confirmation. For this, Kafka requires us to enable the `delete.topic.enable` option. If this is switched to `true`, we will be able to successfully delete the topic and it will then be removed [5].

It is nice to know that Kafka scripts have good usage documentation in general. We recommend running the command `kafka-topics.sh` first to see what various actions you can attempt. The following listing shows an incomplete command to get help.

Listing 7.1 Listing our topic options

```
bin/kafka-topics.sh
```

Runs the generic Kafka topic-related command

In the output that we'll see, one obvious command stands out: `--create`. Adding that parameter helps us get further information related to the `create` action itself (for example, "Missing required argument `"[topic]"`"). The following listing shows our still incomplete generic command built a little further.

Listing 7.2 Listing our topic options with `--create`

```
bin/kafka-topics.sh --create
```

Lists command-specific errors and the help documentation

Why spend time even talking about these steps, as some users are familiar with manual (man) pages as part of their Linux® work? Even though Kafka does not present data about how to use the tooling in that manner, this command is available before you have to search on Google.

Once we have a name that does not have over 249 characters (it's been attempted before), we can create our topic [6]. For our examples, we'll create `kinaction_topicandpart` with a replication factor of 2 and with two partitions. The next listing shows the syntax to use in the command prompt [3].

Listing 7.3 Creating another topic

```
bin/kafka-topics.sh
  --create --bootstrap-server localhost:9094 \
  --topic kinaction_topicandpart \
  --partitions 2 \
  --replication-factor 2
```

Adds the create option to our command

Names our topic

Ensures that we have two copies of our data

Creates our topic with two partitions

After we create our topic, we can `describe` that topic to make sure our settings look correct. Notice in figure 7.3 how our partition and replication factor match the command we just ran.

```
❯ bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic kinaction_topicandpart
Topic: kinaction_topicandpart   PartitionCount: 2       ReplicationFactor: 2    Configs:
        Topic: kinaction_topicandpart   Partition: 0    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: kinaction_topicandpart   Partition: 1    Leader: 0       Replicas: 0,2   Isr: 0,2
```

Figure 7.3 Describing a topic with two partitions

In our opinion, another option that is good to take care of at the broker level is to set `auto.create.topics.enable` to `false` [7]. Doing this ensures that we create our

topics on purpose and not from a producer sending a message to a topic name that was mistyped and never actually existed before a message was attempted. Although not tightly coupled, usually producers and consumers do need to know the correct topic name of where their data should live. This automatic topic creation can cause confusion. But while testing and learning Kafka, autocreated topics can be helpful. For a concrete example, if we run the command

```
kafka-console-producer.sh --bootstrap-server localhost:9094 --topic notexisting
```

without that topic existing, Kafka creates that topic for us. And if we run

```
kafka-topics.sh --bootstrap-server localhost:9094 --list
```

we would now have that topic in our cluster.

Although we usually focus on not removing data from production environments, as we continue in our own exploration of topics, we might run across some mistakes. It's good to know that we can indeed remove a topic if needed [3]. When we do that, all the data in the topic is removed. This is not something we would do unless we're ready to get rid of that data for good! Listing 7.4 shows how to use the `kafka-topic` command we used before, but this time to delete a topic named `kinaction_topicandpart` [3].

Listing 7.4 Deleting a topic

```
bin/kafka-topics.sh --delete --bootstrap-server localhost:9094
  --topic kinaction_topicandpart
```

Removes topic kinaction_topicandpart

Note that the `--delete` option is passed to our Kafka topics command. After running this command, you will not be able to work with this topic for your data as before.

7.1.2 Replication factors

For practical purposes, we should plan on having the total number of replicas less than or equal to the number of brokers. In fact, attempting to create a topic with the number of replicas being greater than the total number of brokers results in an error: `InvalidReplicationFactorException` [8]. We may imagine why this is an error. Imagine, we only have two brokers, and we want three replicas of a partition. One of those replicas would exist on one broker and two on the other broker. In this case, if we lost the broker that was hosting two of the replicas, we would be down to only one copy of the data. Losing multiple replicas of your data at once is not the ideal way to provide recovery in the face of failure.

7.2 Partitions

Moving on from dealing with Kafka commands at a (mostly) topic level, let's start to look deeper at partitions. From a consumer standpoint, each partition is an immutable log of messages. It should only grow and append messages to our data store. Although this data does not grow forever in practice, thinking of the data as

being added to rather than modified in place is a good mental model to maintain. Also, consumer clients cannot directly delete messages. This is what makes it possible to replay messages from a topic, which is a feature that can help us in many scenarios.

7.2.1 Partition location

One thing that might be helpful is to look at how the data is stored on our brokers. To start, let's find the location of the log.dirs (or log.dir) directory. Its location can be found by looking for log.dirs in your server.properties file if you followed along from appendix A. Under that directory, we should be able to see subfolders with a topic name and a partition number. If we pick one of those folders and look inside, we will see a couple of different files with these extensions: .index, .log, and .timeindex. Figure 7.4 shows how a single partition (in this case, 1) in our test topic looks by issuing a directory listing (`ls`).

```
❯ ls /tmp/kafkainaction/kafka-logs-0/kinaction_topicandpart-1
00000000000000000000.index      00000000000000000000.log        00000000000000000000.timeindex  leader-epoch-checkpoint
```

Figure 7.4 Partition directory listing

Sharp-eyed readers might see the file named leader-epoch-checkpoint and maybe even files with a .snapshot extension (not shown above) in their own directory. The leader-epoch-checkpoint file and snapshot files are those that we will not spend time looking at.

The files with the .log extension are where our data payload is stored. Other important information in the log file includes the offset of the message as well as the `CreateTime` field. Why the need for any other files then? Because Kafka is built for speed, it uses the .index and .timeindex files to store a mapping between the logical message offset and a physical position inside the index file [9].

As shown so far, partitions are made up of many files. In essence, this means that on a physical disk, a partition is not one single file but is rather split into several segments [10]. Figure 7.5 shows how multiple segments might make up a partition.

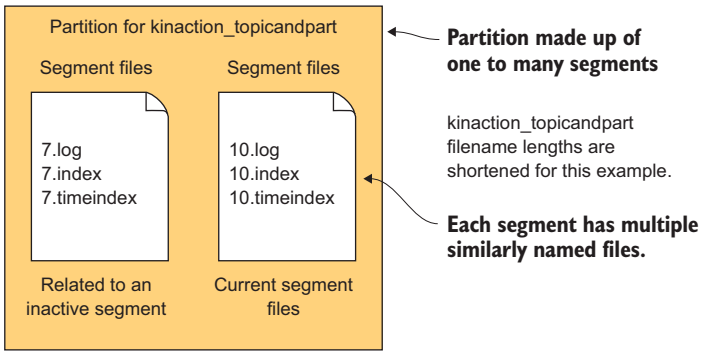

Figure 7.5 Segments make up a partition.

An active segment is the file to which new messages are currently written [11]. In our illustration, 10.log is where messages are being written in the partition directory. Older segments are managed by Kafka in various ways in which the active segment will not be; this includes being governed for retention based on the size of the messages or time configuration. These older segments (like 7.log in figure 7.5) can be eligible for topic compaction, which we will touch on later in this chapter.

To recap what we now know about segments, we know why we might have multiple files with the same name in a partition directory but with an .index, .timeindex, or .log extension. For example, if we have 4 segments, we would have a set of 4 files, each with one of the previous 3 extensions, for a total of 12 files. If we only see 1 of each file extension, we only have 1 segment.

7.2.2 Viewing our logs

Let's try to take a peek at a log file to see the messages we have produced for our topic so far. If we open it in a text editor, we will not see those messages in a human-readable format. Confluent has a script that we can use to look at those log segments [12]. Listing 7.5 shows us passing the command to `awk` and `grep` to look at a segment log file for partition 1 of the topic `kinaction_topicandpart`.

Listing 7.5 Looking at a dump of a log segment

```
bin/kafka-dump-log.sh --print-data-log \
  --files /tmp/kafkainaction/kafka-logs-0/
➥ kinaction_topicandpart-1/*.log \
| awk -F: '{print $NF}' | grep kinaction
```

Prints data that cannot be viewed easily with a text editor

Passes a file to read

By using the `--files` option, which is required, we chose to look at a segment file. Assuming the command is successful, we should see a list of messages printed to the screen. Without using `awk` and `grep`, you would also see offsets as well as other related metadata like compression codecs. This is definitely an interesting way to see how Kafka places messages on the broker and the data it retains around those messages. The ability to see the actual messages is empowering as it really helps you see the log in action that drives Kafka.

Looking at figure 7.6, we can see a payload in text that is a little easier to read than when we tried to `cat` the log file directly. For example, we can see a message in the segment file with the payload `kinaction_helloworld`. Hopefully, you will have more valuable data!

```
❯ bin/kafka-dump-log.sh --print-data-log --files /tmp/kafkainaction/kafka-logs-0/kinaction_topicandpart-1/*.
log | awk -F: '{print $NF}'| grep kinaction
Dumping /tmp/kafkainaction/kafka-logs-0/kinaction_topicandpart-1/00000000000000000000.log
 kinaction_helloworld
```

Figure 7.6 Viewing a log segment

As for the large number in the log filename, it is not random. The segment name should be the same as the first offset in that file.

One of the impacts of being able to see this data is that we now have to be concerned with who else can see it. Because data security and access controls are common concerns with most data that holds values, we will look at ways you can secure Kafka and topics in chapter 10. Facts about the segment log and index files are details that we would not normally rely on in our applications. However, knowing how to look at these logs might be helpful when understanding how our logs really exist.

It helps to imagine Kafka as a living and complex system (it is distributed, after all) that might need some care and feeding from time to time. In this next section, we will tackle testing our topic.

7.3 Testing with EmbeddedKafkaCluster

With all of the configuration options we have, it might be nice to test them as well. What if we could spin up a Kafka cluster without having a real production-ready cluster handy? Kafka Streams provides an integration utility class called `EmbeddedKafkaCluster` that serves as a middle ground between mock objects and a full-blown cluster. This class provides an in-memory Kafka cluster [13]. Although built with Kafka Streams in mind, we can use it to test our Kafka clients.

Listing 7.6 is set up like the tests found in the book *Kafka Streams in Action* by William P. Bejeck Jr., for example, his `KafkaStreamsYellingIntegrationTest` class [14]. That book and his following book, *Event Streaming with Kafka Streams and ksqlDB*, show more in-depth testing examples. We recommend checking those out, including his suggestion of using Testcontainers (https://www.testcontainers.org/). The following listing shows testing with `EmbeddedKafkaCluster` and JUnit 4.

Listing 7.6 Testing with `EmbeddedKafkaCluster`

```
@ClassRule
public static final EmbeddedKafkaCluster embeddedKafkaCluster
    = new EmbeddedKafkaCluster(BROKER_NUMBER);

private Properties kaProducerProperties;
private Properties kaConsumerProperties;

@Before
public void setUpBeforeClass() throws Exception {
    embeddedKafkaCluster.createTopic(TOPIC,
      PARTITION_NUMBER, REPLICATION_NUMBER);
    kaProducerProperties = TestUtils.producerConfig(
      embeddedKafkaCluster.bootstrapServers(),
      AlertKeySerde.class,
      StringSerializer.class);

    kaConsumerProperties = TestUtils.consumerConfig(
      embeddedKafkaCluster.bootstrapServers(),
      AlertKeySerde.class,
      StringDeserializer.class);
}
```

Uses JUnit-specific annotation to create the cluster with a specific number of brokers

Sets the consumer configuration to point to the embedded cluster brokers

```
@Test
public void testAlertPartitioner() throws InterruptedException {
    AlertProducer alertProducer =  new AlertProducer();
    try {
        alertProducer.sendMessage(kaProducerProperties);
    } catch (Exception ex) {
        fail("kinaction_error EmbeddedKafkaCluster exception"
        ➥ + ex.getMessage());
    }

    AlertConsumer alertConsumer = new AlertConsumer();
    ConsumerRecords<Alert, String> records =
      alertConsumer.getAlertMessages(kaConsumerProperties);
    TopicPartition partition = new TopicPartition(TOPIC, 0);
    List<ConsumerRecord<Alert, String>> results = records.records(partition);
    assertEquals(0, results.get(0).partition());
}
```

Calls the client without any changes, which is clueless of the underlying cluster being embedded

Asserts that the embedded cluster handled the message from production to consumption

When testing with `EmbeddedKafkaCluster`, one of the most important parts of the setup is to make sure that the embedded cluster is started before the actual testing begins. Because this cluster is temporary, another key point is to make sure that the producer and consumer clients know how to point to this in-memory cluster. To discover those endpoints, we can use the method `bootstrapServers()` to provide the needed configuration to the clients. Injecting that configuration into the client instances is again up to your configuration strategy, but it can be as simple as setting the values with a method call. Besides these configurations, the clients should be able to test away without the need to provide mock Kafka features!

The test in listing 7.6 verifies that the `AlertLevelPartitioner` logic was correct. Using that custom partitioner logic with a critical message should have landed the alert on partition 0 with our example code in chapter 4. By retrieving the messages for `TopicPartition(TOPIC, 0)` and looking at the included messages, the message partition location was confirmed. Overall, this level of testing is usually considered integration testing and moves you beyond just a single component under test. At this point, we have tested our client logic together with a Kafka cluster, integrating more than one module.

> **NOTE** Make sure that you reference the pom.xml changes in the source code for chapter 7. There are various JARs that were not needed in previous chapters. Also, some JARs are only included with specific classifiers, noting that they are only needed for test scenarios.

7.3.1 Using Kafka Testcontainers

If you find that you are having to create and then tear down your infrastructure, one option that you can use (especially for integration testing) is Testcontainers (https://www.testcontainers.org/modules/kafka/). This Java library uses Docker and one of a variety of JVM testing frameworks like JUnit. Testcontainers depends on Docker images to provide you with a running cluster. If your workflow is Docker-based or a development technique your team uses well, Testcontainers is worth looking into to get a Kafka cluster set up for testing.

NOTE One of the coauthors of this book, Viktor Gamov, maintains a repository (https://github.com/gAmUssA/testcontainers-java-module-confluent-platform) of integration testing Confluent Platform components (including Kafka, Schema Registry, ksqlDB).

7.4 Topic compaction

Now that we have a solid foundation on topics being made up of partitions and partitions being made up of segments, it is time to talk about the details of log compaction. With compaction, the goal is not to expire messages but rather to make sure that the latest value for a key exists and not to maintain any previous state. As just referenced, compaction depends on a key being part of the messages and that key not being `null` [10].

The configuration option that we used to create a compacted topic is `cleanup.policy=compact` [15]. This differs from the default configuration value that was set to `delete` before our override. In other words, we have to choose to create a compacted topic or the topic won't exist in that way. The following listing adds the configuration option needed for this new compacted topic.

Listing 7.7 Creating a compacted topic

```
bin/kafka-topics.sh --create --bootstrap-server localhost:9094 \
  --topic kinaction_compact --partitions 3 --replication-factor 3 \
  --config cleanup.policy=compact
```

Creates the topic like any other topic

Creates our topic type to be compacted

One of the easiest comparisons for how a compacted topic presents data can be seen in how code would update an array's existing field rather than appending more data. Let's say that we want to keep a current membership status for an online membership. A user can only be in one state at a time, either a Basic or a Gold membership. At first, a user enrolls in the Basic plan, but over time, upgrades to the Gold plan for more features. Although this is still an event that Kafka stores, in our case, we only want the most recent membership level for a specific customer (our key). Figure 7.7 shows an example using three customers.

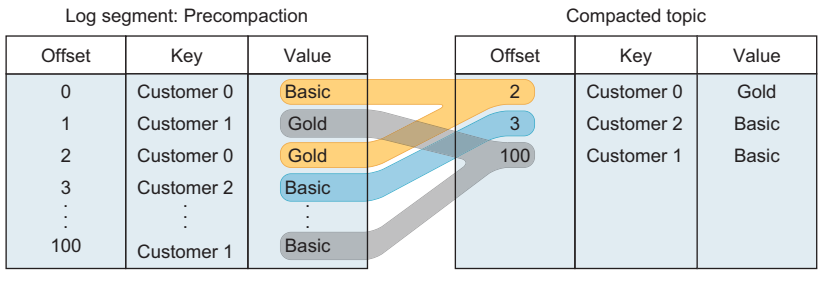

Figure 7.7 Compaction in general

After compaction is done, the latest customer 0 update (in our example) is all that exists in the topic. A message with offset 2 replaces the old value of `Basic` (message offset 0) for customer 0 with `Gold`. Customer 1 has a current value of `Basic` because the latest key-specific offset of 100 updates the previous offset 1 `Gold` state. As customer 2 only has one event, that event carries over to the compacted topic without any changes.

Another real-world example of why one would want to use a compacted topic is Kafka's internal topic, `__consumer_offsets`. Kafka does not need a history of offsets that a consumer group consumes; it just needs the latest offset. By storing the offsets in a compacted topic, the log, in effect, gets an updated view of the current state of its world.

When a topic is marked for compaction, we can view a single log in a couple of different states: compacted or not. For older segments, duplicate values for each key should have been reduced to just one value once compaction is completed. The active segment messages are those that have not yet been through compaction [11]. Multiple values can exist for a message for a specific key until all the messages are cleaned. Figure 7.8 illustrates how a pointer is used to show which messages have been processed with compaction and which messages have yet to be visited [16].

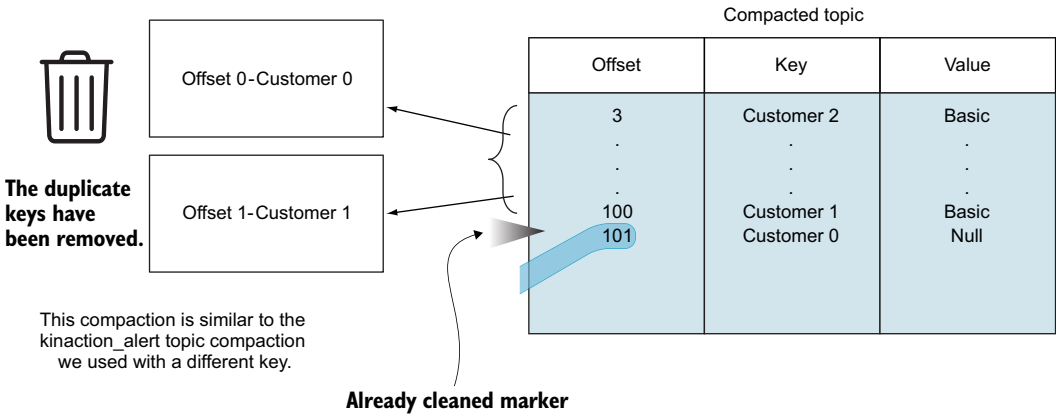

Figure 7.8 Compaction cleaning

Looking closely at the offsets in figure 7.8, we can see that there are gaps in the cleaned segment offset numbers. Because duplicate key messages are left with the latest value only, we might have some offset numbers removed from the segment file, for example, offset 2 was removed. In the active sections, we will likely see the ever-increasing offset numbers that we are used to, without random jumping numbers.

Let's now switch to a subscriber who wanted to delete their account. By sending an event with the subscriber key, like Customer 0, with a message value of `null`, this

message will be treated as a delete. This message is considered a `tombstone` [10]. If you have used other systems like Apache HBase™, the notion is similar. Figure 7.9 shows that the null value does not remove a message but is served like any other message [10].

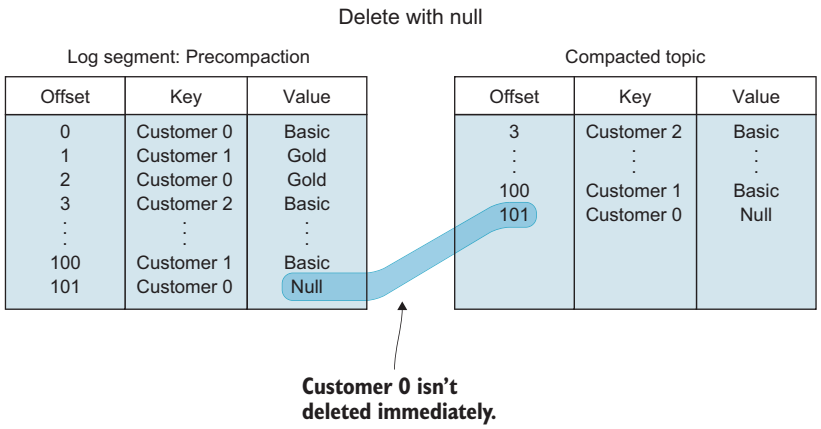

Figure 7.9 Compaction for a deleted value

With delete rules that an application may or may not have to deal with, Kafka can help us fulfill those data requirements with its core feature set.

Throughout this chapter, we have looked at the various details of topics, partitions, and segments. Although broker-specific, they can indeed impact our clients. Because we have experience now with how Kafka stores some of its own data, we are going to spend some time in our next chapter discussing how we can store our data. This includes longer-term storage options for data.

Summary

- Topics are non-concrete rather than physical structures. To understand the topic's behavior, a consumer of that topic needs to know about the number of partitions and the replication factors in play.
- Partitions make up topics and are the basic unit for parallel processing of data inside a topic.
- Log file segments are written in partition directories and are managed by the broker.
- Testing can be used to help validate partition logic and may use an in-memory cluster.
- Topic compaction is a way to provide a view of the latest value of a specific record.

References

1. "Main Concepts and Terminology." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/introduction.html#main-concepts-and-terminology (accessed August 28, 2021).
2. J. Rao. "How to choose the number of topics/partitions in a Kafka cluster?" (March 12, 2015). Confluent blog. https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/ (accessed May 19, 2019).
3. "Documentation: Modifying topics." Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#basic_ops_modify_topic (accessed May 19, 2018).
4. "Documentation: Adding and removing topics." Apache Software Foundation (n.d.). https://kafka.apache.org/documentation/#basic_ops_add_topic (accessed December 11, 2019).
5. "delete.topic.enable." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html#brokerconfigs_delete.topic.enable (accessed January 15, 2021).
6. `Topics.java`. Apache Kafka GitHub. https://github.com/apache/kafka/blob/99b9b3e84f4e98c3f07714e1de6a139a004cbc5b/clients/src/main/java/org/apache/kafka/common/internals/Topic.java (accessed August 27, 2021).
7. "auto.create.topics.enable." Apache Software Foundation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html#brokerconfigs_auto.create.topics.enable (accessed December 19, 2019).
8. `AdminUtils.scala`. Apache Kafka GitHub. https://github.com/apache/kafka/blob/d9b898b678158626bd2872bbfef883ca60a41c43/core/src/main/scala/kafka/admin/AdminUtils.scala (accessed August 27, 2021).
9. "Documentation: index.interval.bytes." Apache Kafka documentation. https://kafka.apache.org/documentation/#topicconfigs_index.interval.bytes (accessed August 27, 2021).
10. "Log Compaction." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/design.html#log-compaction (accessed August 20, 2021).
11. "Configuring The Log Cleaner." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/design.html#configuring-the-log-cleaner (accessed August 27, 2021).
12. "CLI Tools for Confluent Platform." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/cli-reference.html (accessed August 25, 2021).
13. `EmbeddedKafkaCluster.java`. Apache Kafka GitHub. https://github.com/apache/kafka/blob/9af81955c497b31b211b1e21d8323c875518df39/streams/src/test/java/org/apache/kafka/streams/integration/utils/EmbeddedKafkaCluster.java (accessed August 27, 2021).

14 W. P. Bejeck Jr. *Kafka Streams in Action*. Shelter Island, NY, USA: Manning, 2018.

15 "cleanup.policy." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/topic-configs.html#topicconfigs_cleanup.policy (accessed November 22, 2020).

16 "Log Compaction Basics." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/design.html#log-compaction-basics (accessed August 20, 2021).