2 Kafka brokers

This chapter covers

- Explaining how the Kafka Broker is the storage layer in the Kafka event streaming platform
- Describing how Kafka brokers handle requests from clients for writing and reading records
- Understanding topics and partitions
- Using JMX metrics to check for a healthy broker

In chapter one, I provided an overall view of the Kafka event streaming platform and the different components that make up the platform. In this chapter, we will focus on the heart of the system, the Kafka broker. The Kafka broker is the server in the Kafka architecture and serves as the storage layer.

In the course of describing the broker behavior in this chapter, we'll get into some lower-level details. I feel it's essential to cover them to give you an understanding of how the broker operates. Additionally, some of the things we'll cover, such as topics and partitions, are essential concepts you'll need to understand when we get into the chapter on clients. But in practice, as a developer, you won't have to handle these topics daily.

As the storage layer, the broker is responsible for data management, including retention and replication. Retention is how long the brokers store records. Replication is how brokers make copies of the data for durable storage, meaning if you lose a machine, you won't lose data.

But the broker also handles requests from clients. Here's an illustration showing the client applications and the brokers:

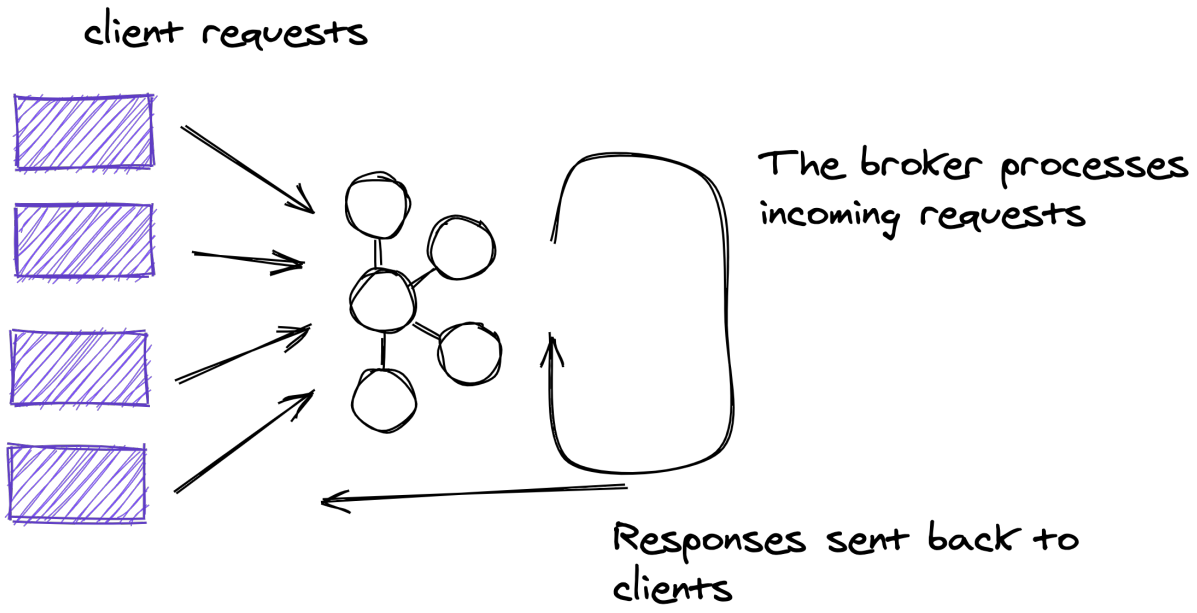

Figure 2.1 Clients communicating with brokers

To give you a quick mental model of the broker's role, we can summarize the illustration above: Clients send requests to the broker. The broker then processes those requests and sends a response. While I'm glossing over several details of the interaction, that is the gist of the operation.

> **NOTE** **Kafka is a deep subject, so I won't cover every aspect. I'll go over enough information to get you started working with the Kafka event streaming platform. For in-depth Kafka coverage, look at Kafka in Action by Dylan Scott (Manning, 2018).**

You can deploy Kafka brokers on commodity hardware, containers, virtual machines, or in cloud environments. In this book, you'll use Kafka in a docker container, so you won't need to install it directly. I'll cover the necessary Kafka installation in an appendix.

While you're learning about the Kafka broker, I'll need to talk about the producer and consumer clients. But since this is chapter is about the broker, I'll focus more on the broker's responsibilities. So at times, I'll leave out some of the client details. But not to worry, we'll get to those details in a later chapter.

So, let's get started with some walkthroughs of how a broker handles client requests, starting with producing.

2.1 Produce record requests

When a client wants to send records to the broker, it does so with a produce request. Clients send records to the broker for storage so that consuming clients can later read those records.

Here's an illustration of a producer sending records to a broker. It's important to note these illustrations aren't drawn to scale. What I mean is that typically you'll have many clients communicating with several brokers in a cluster. A single client will work with more than one broker. But it's easier to get a mental picture of what's going on if I keep the illustrations simple. Also, note that I'm simplifying the interaction, but we'll cover more details when discussing clients in chapter 4.

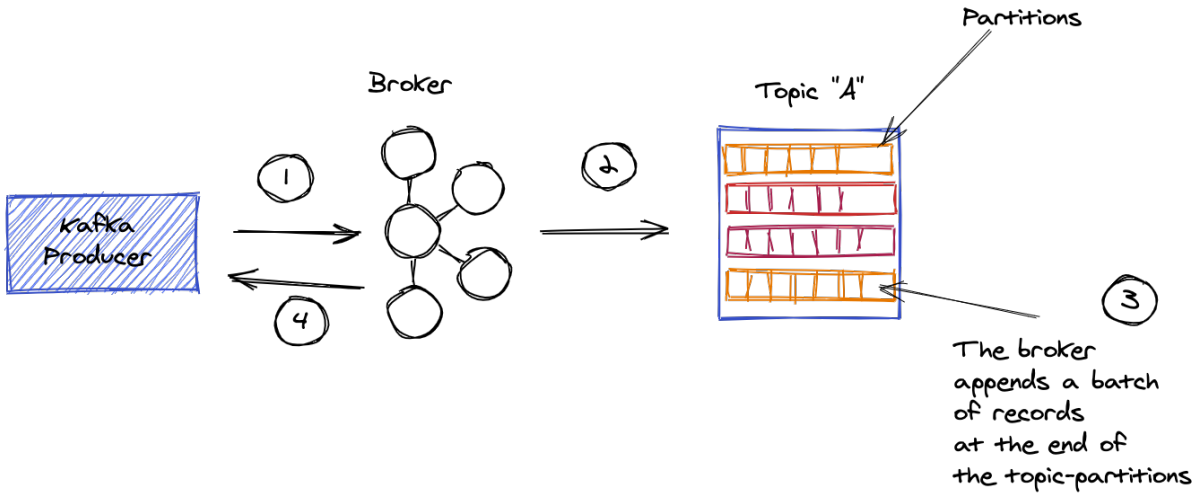

Figure 2.2 Brokers handling produce records request

Let's walk through the steps in the "Producing records" illustration.

1. The producer sends a batch of records to the broker. Whether it's a producer or consumer, the client APIs always work with a collection of records to encourage batching.
2. The broker takes the produce request out of the request queue.
3. The broker stores the records in a topic. Inside the topic, there are partitions; you can consider a partition way of bucketing the different records for now. A single batch of records always belongs to a specific partition within a topic, and the records are ***always*** appended at the end.
4. Once the broker completes the storing of the records, it sends a response back to the producer. We'll talk more about what makes up a successful write later in this chapter and again in chapter 4.

Now that we've walked through an example produce request, let's walk through another request type, fetch, which is the logical opposite of producing records; consuming records.

2.2 Consume record requests

Now let's take a look at the other side of the coin from a produce request to a consume request. Consumer clients issue requests to a broker to read (or consume) records from a topic. A critical point to understand is that consuming records does not affect data retention or records availability to other consuming clients. Kafka brokers can handle hundreds of consume requests for records from the same topic, and each request has no impact on the other. We'll get into data retention a bit later, but the broker handles it utterly separate from consumers.

It's also important to note that producers and consumers are unaware of each other. The broker handles produce and consume requests separately; one has nothing to do with the other. The example here is simplified to emphasize the overall action from the broker's point of view.

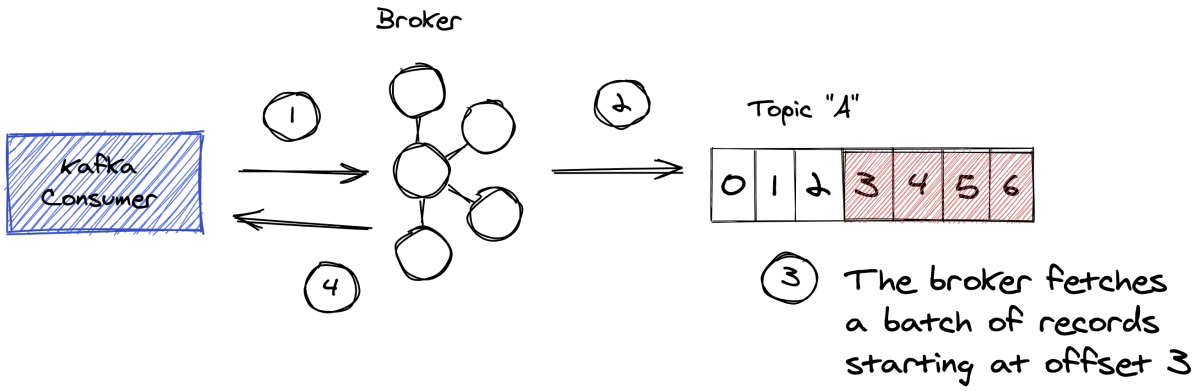

Figure 2.3 Brokers handling requests from a consumer

So let's go through the steps of the illustrated consume request.

1. The consumer sends a fetch request specifying the offset it wants to start reading records from. We'll discuss offsets in more detail later in the chapter.
2. The broker takes the fetch request out of the request queue
3. Based on the offset and the topic partition in the request, the broker fetches a batch of records
4. The broker sends the fetched batch of records in the response to the consumer

Now that we've completed a walk through two common request types, produce and fetch, I'm sure you noticed a few terms I haven't mentioned yet in the text, topics, partitions, and offsets. Topics, partitions, and offsets are fundamental, essential concepts in Kafka, so let's take some time now to explore what they mean.

2.3 Topics and partitions

In chapter one, we discussed that Kafka provides storage for data. Kafka durably stores your data as an unbounded series of key-value pair messages for as long as you want (there are other fields included in the messages, such as a timestamp, but we'll get to those details later on). Kafka replicates data across multiple brokers, so losing a disk or an entire broker means no data is lost.

Specifically, Kafka brokers use the file system for storage by appending the incoming records to the end of a file in a topic. A topic represents the name of the directory containing the file Kafka appends the records to.

> **NOTE** **Kafka receives the key-value pair messages as raw bytes, stores them that way, and serves the read requests in the same format. The Kafka broker is unaware of the type of record that it handles. By merely working with raw bytes, the brokers don't spend any time deserializing or serializing the data, allowing for higher performance. We'll see in chapter 3 how you can ensure that topics contain the expected byte format when we cover Schema Registry in chapter 3.**

Topics are partitioned, which is a way of further organizing the topic data into slots or buckets. A partition is an integer starting at 0. So if a topic has three partitions, the partitions numbers are 0, 1, and 2. Kafka appends the partition number to the end of the topic name, creating the same number of directories as partitions with the form `topic-N` where the `N` represents the partition number.

Kafka brokers have a configuration, `log.dirs`, where you place the top-level directory's name, which will contain all topic-partition directories. Let's take a look at an example. We're going to assume you've configured `log.dirs` with the value `/var/kafka/topic-data` and you have a topic named `purchases` with three partitions

Listing 2.1 Topic directory structure example

```
root@broker:/#  tree /var/kafka/topic-data/purchases*

/var/kafka/topic-data/purchases-0
 00000000000000000000.index
 00000000000000000000.log
 00000000000000000000.timeindex
 leader-epoch-checkpoint
/var/kafka/topic-data/purchases-1
 00000000000000000000.index
 00000000000000000000.log
 00000000000000000000.timeindex
 leader-epoch-checkpoint
/var/kafka/topic-data/purchases-2
 00000000000000000000.index
 00000000000000000000.log
 00000000000000000000.timeindex
 leader-epoch-checkpoint
```

So you can see here, the topic `purchases` with three partitions ends up as three directories `purchases-0`, `purchases-1`, and `purchases-2` on the file system. So it's fair to say that the topic name is more of a logical grouping while the partition is the storage unit.

TIP **The directory structure shown here was generated by using the `tree` command which a small command line tool used to display all contents of a directory.**

While we'll want to spend some time talking about those directories' contents, we still have some details to fill in about topic partitions.

Topic partitions are the unit of parallelism in Kafka. For the most part, the higher the number of partitions, the higher your throughput. As the primary storage mechanism, topic partitions allow messages to be spread across several machines. The given topic's capacity isn't limited to the available disk space on a single broker. Also, as mentioned before, replicating data across several brokers ensures you won't lose data should a broker lose disks or die.

We'll talk about load distribution more when discussing replication, leaders, and followers later in this chapter. We'll also cover a new feature, tiered storage, where data is seamlessly moved to external storage, providing virtually limitless capacity later in the chapter.

So how does Kafka map records to partitions? The producer client determines the topic and partition for the record before sending it to the broker. Once the broker processes the record, it appends it to a file in the corresponding topic-partition directory.

There are three possible ways of setting the partition for a record:

1. Kafka works with records in key-value pairs. Suppose the key is non-null (keys are optional). In that case, the producer maps the record to a partition using the deterministic formula of taking the hash of key modulo the number of partitions. Using this approach means that records with the same keys always land on the same partition.
2. When building the `ProducerRecord` in your application, you can explicitly set the partition for that record, which the producer then uses before sending it.
3. If the message has no key and no partition specified then, then partitions are alternated per batch. I'll cover how Kafka handles records without keys and partition assignment in detail in chapter four.

Now that we've covered how topic partitions work let's revisit that records are always appended at the end of the file. I'm sure you noticed the files in the directory example with an extension of `.log` (we'll talk about how Kafka names this file in an upcoming section). But these `log` files aren't the type developers think of, where an application prints its status or execution steps. The term log here is meant as a transaction log, storing a sequence of events in the order of

occurrence. So each topic partition directory contains its own transaction log. At this point, it would be fair to ask a question about log file growth. We'll talk about log file size and management when we cover segments a bit later in this chapter.

2.3.1 Offsets

As the broker appends each record, it assigns it an id called an offset. An offset is a number (starting at 0) the broker increments by 1 for each record. In addition to being a unique id, it represents the logical position in the file. The term logical position means it's the nth record in the file, but its physical location is determined by the size in bytes of the preceding records. We'll talk about how brokers use an offset to find the physical position of a record in a later section. The following illustration demonstrates the concept of offsets for incoming records:

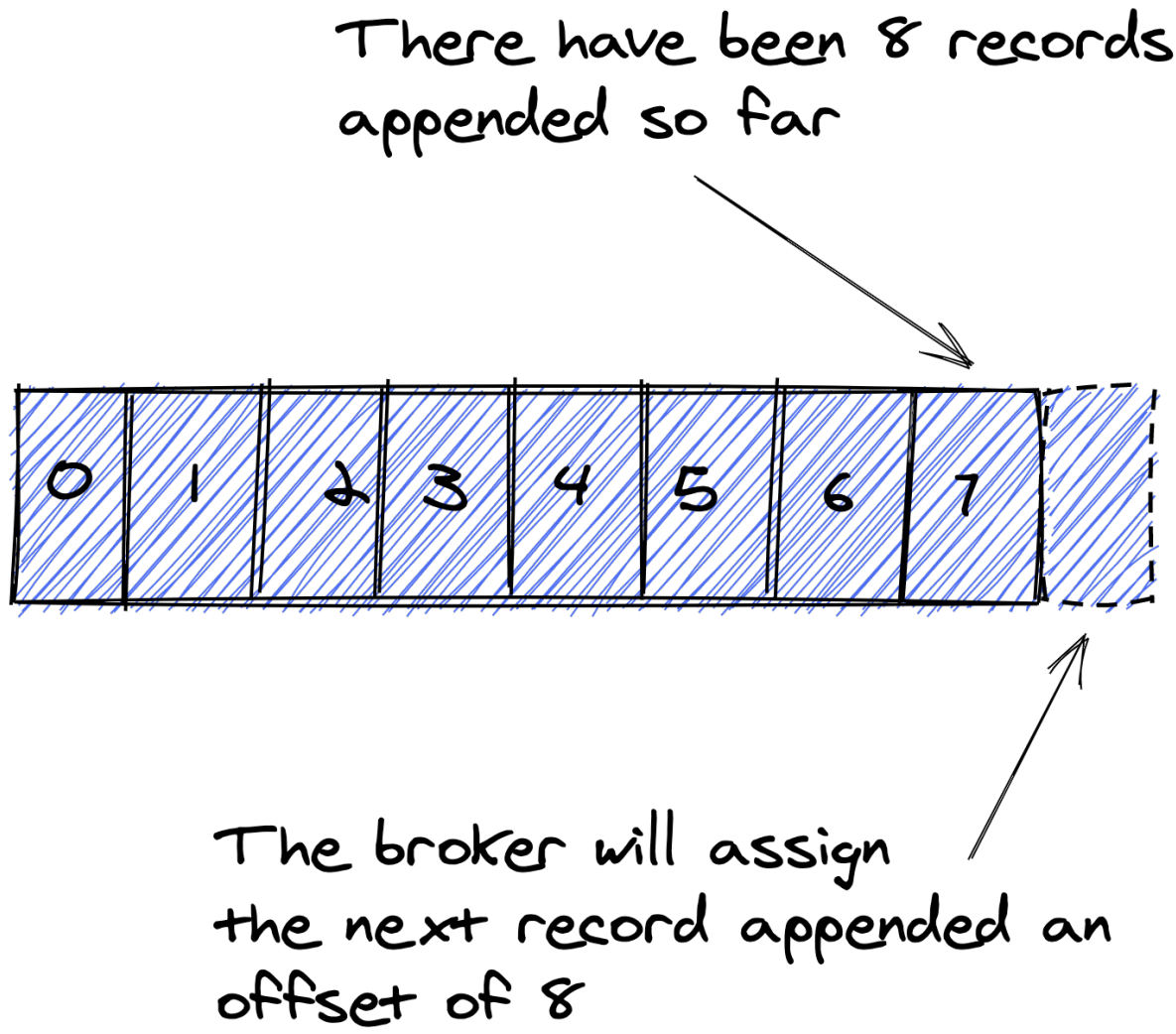

Figure 2.4 Assigning the offset to incoming records

Since new records always go at the end of the file, they are in order by offset. Kafka guarantees that records are in order within a partition, but not *across* partitions. Since records are in order by offset, we could be tempted to think they are in order by time as well, but that's not necessarily the case. The records are in order by their ***arrival*** time at the broker, but not necessarily by ***event time***. We'll get more into time semantics in the chapter on clients when we discuss timestamps. We'll also cover event-time processing in depth when we get to the chapters on Kafka Streams.

Consumers use offsets to track the position of records they've already consumed. That way, the broker fetches records starting with an offset one higher than the last one read by a consumer. Let's look at an illustration to explain how offsets work:

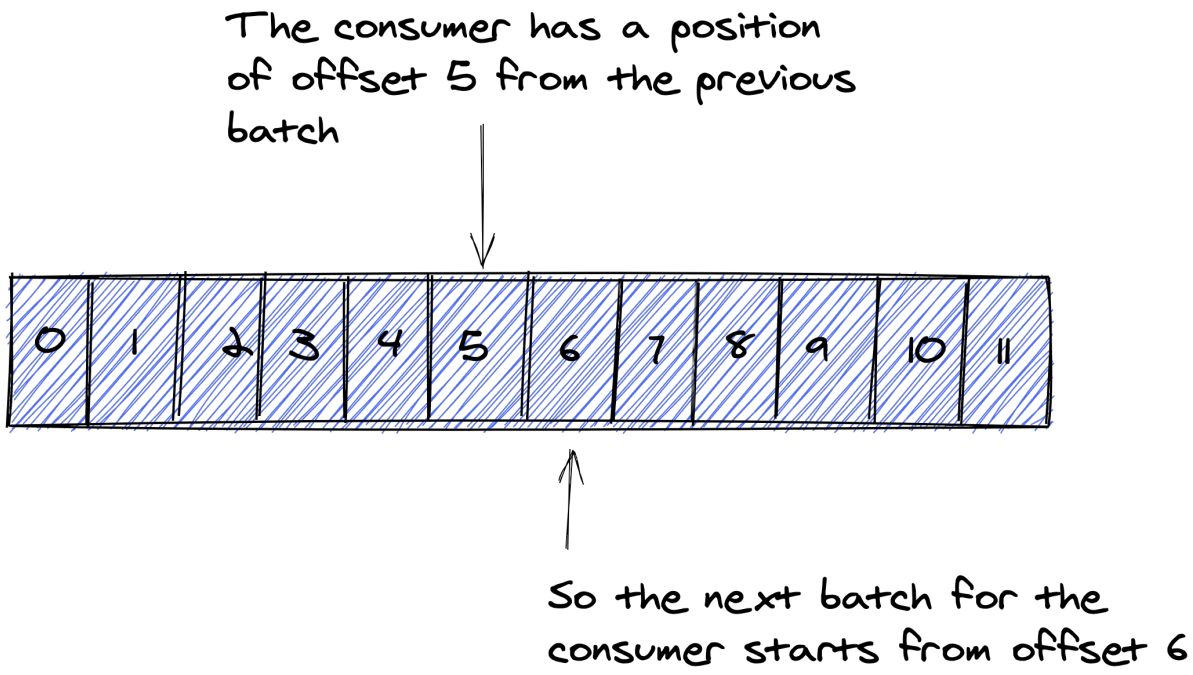

Figure 2.5 Offsets indicate where a consumer has left off reading records

In the illustration here, if a consumer reads records with offsets 0-5, in the next consumer request, the broker only fetches records starting at offset 6. The offsets used are unique for each consumer and are stored in an internal topic named `{underscore}consumer{underscore}offsets`. We'll go into more details about consumers and offsets in chapter four.

Now that we've covered topics, partitions and offsets, let's quickly discuss some trade-offs regarding the number of partitions to use.

2.3.2 Determining the correct number of partitions

Choosing the number of partitions to use when creating a topic is part art and part science. One of the critical considerations is the amount of data flowing into a given topic. More data implies more partitions for higher throughput. But as with anything in life, there are trade-offs.

Increasing the number of partitions increases the number of TCP connections and open file handles. Additionally, how long it takes to process an incoming record in a consumer will also determine throughput. If you have heavyweight processing in your consumer, adding more partitions may help, but the slower processing will ultimately hinder performance.[2]

Here are some considerations to keep in mind for setting the number of partitions. You want to choose a high enough number to cover high-throughput situations, but not so high so that you hit limits for the number of partitions a broker can handle as you create more and more topics. A good starting point could be the number of 30, which is evenly divisible by several numbers, which results in a more even distribution of keys in the processing layer.[3] We'll talk more about the importance of key-distribution in later chapters on clients and Kafka Streams.

At this point, you've learned that the broker handles requests from clients and is the storage layer for the Kafka event streaming platform. You've also learned about topics and partitions and the role they play in the storage layer.

Your next step is to get your hands dirty, producing and consuming records to see these concepts in action.

NOTE	**We'll cover the producer and consumer clients in chapter 4. Console clients are useful for learning, quick prototypes, and debugging. But in practice, you'll use the clients in your code.**

2.4 Sending your first messages

To run the following examples, you'll need to run a Kafka broker. In the previous edition of this book, the instructions were to download a binary version of Kafka tar file and extract it locally. In this edition, I've opted to run Kafka via docker instead. Specifically, we'll use docker compose, which makes running a multi-container docker application very easy. If you are running Mac OS or Windows, you can install docker desktop, which includes docker compose. For more information on installing docker, see the installation instructions on the docker site docs.docker.com/get-docker/.

Now, let's get started working with a Kafka broker by producing and consuming some records.

2.4.1 Creating a topic

Your first step for producing or consuming records is to create a topic. But to do that, you'll need running Kafka broker so let's take care of that now. I'm going to assume you've already installed docker at this point. To start Kafka, download the `docker-compose.yml` file from the source code repo here TOOD-create GitHub repo. After you've downloaded the file, open a new terminal window and CD to the directory with the `docker-compose.yml` file, and run this command `docker-compose up -d'.

> **TIP** **Starting docker-compose with the `-d` flag runs the docker services in the background. While it's OK to start docker-compose without the `-d` flag, the containers print their output to the terminal, so you need to open a new terminal window to do any further operations.**

Wait a few seconds, then run this command to open a shell on the docker broker container: `docker-compose exec broker bash`.

Using the docker broker container shell you just opened up run this command to create a topic:

```
kafka-topics --create --topic first-topic\
 --bootstrap-server localhost:9092\   ❶
 --replication-factor 1\   ❷
 --partitions 1   ❸
```

❶ The host:port to connect to the broker

❷ Specifying the replication factor

❸ The number of partitions

> **IMPORTANT** **Although you're using kafka in a docker container, the commands to create topics and run the console producer and consumer are the same.**

Since you're running a local broker for testing, you don't need a replication factor greater than 1. The same thing goes for the number of partitions; at this point, you only need one partition for this local development.

Now you have a topic, let's write some records to it.

2.4.2 Producing records on the command line

Now from the same window you ran the create topic command start a console producer:

```
kafka-console-producer --topic first-topic\   ❶
  --broker-list localhost:9092\   ❷
  --property parse.key=true\   ❸
  --property key.separator=":"   ❹
```

❶ The topic you created in the previous step

❷ host:port for the producer client to connect to the broker

❸ Specifying that you'll provide a key

❹ Specifying the separator of the key and value

When using the console producer, you need to specify if you are going to provide keys. Although Kafka works with key-value pairs, the key is optional and can be null. Since the key and value go on the same line, you also need to specify how Kafka can parse the key and value by providing a delimiter.

After you enter the above command and hit enter, you should see a prompt waiting for your input. Enter some text like the following:

```
key:my first message
key:is something
key:very simple
```

You type in each line, then hit enter to produce the records. Congratulations, you have sent your first messages to a Kafka topic! Now let's consume the records you just wrote to the topic. Keep the console producer running, as you'll use it again in a few minutes.

2.4.3 Consuming records from the command line

Now it's time to consume the records you just produced. Open a new terminal window and run the `docker-compose exec broker bash` command to get a shell on the broker container. Then run the following command to start the console consumer:

```
kafka-console-consumer --topic first-topic\   ❶
 --bootstrap-server localhost:9092\   ❷
 --from-beginning\   ❸
 --property print.key=true\   ❹
 --property key.separator="-"   ❺
```

❶ Specifying the topic to consume from

❷ The host:port for the consumer to connect to the broker

❸ Start consuming from the head of the log

❹ Print the keys

❺ Use the "-" character to separate keys and values

You should see the following output on your console:

```
key-my first message
key-is something
key-very simple
```

I should briefly talk about why you used the `--from-beginning` flag. You produced values before starting the consumer. As a result, you wouldn't have seen those messages as the console consumer reads from the end of the topic. So the `--from-beginning` parameter sets the consumer to read from the beginning of the topic. Now go back to the producer window and enter a new key-value pair. The console window with your consumer will update by adding the latest record at the end of the current output.

This completes your first example, but let's go through one more example where you can see how partitions come into play.

2.4.4 Partitions in action

In the previous exercise, you just produced and consumed some key-value records, but the topic only has one partition, so you didn't see the effect of partitioning. Let's do one more example, but this time we'll create a new topic with two partitions, produce records with different keys, and see the differences.

You should still have a console producer and console consumer running at this point. Go ahead and shut both of them down by entering a `CTRL+C` command on the keyboard.

Now let's create a new topic with partitions. Execute the following command from one of the terminal windows you used to either produce or consume records:

```
kafka-topics --create --topic second-topic\
 --bootstrap-server localhost:9092\
 --replication-factor 1\
 --partitions 2
```

For your next step, let's start a console consumer.

```
kafka-console-consumer --topic second-topic\
 --bootstrap-server broker:9092 \
 --property print.key=true \
 --property key.separator="-" \
 --partition 0    ❶
```

❶ Specifying the partition we'll consume from

This command is not too different from the one you executed before, but you're specifying the partition you'll consume the records from. After running this command, you won't see anything on the console until you start producing records in your next step. Now let's start up another console producer.

```
kafka-console-producer --topic second-topic\
  --broker-list localhost:9092\
  --property parse.key=true\
  --property key.separator=":"
```

After you've started the console producer, enter these key-value pairs:

```
key1:The lazy
key2:brown fox
key1:jumped over
key2:the lazy dog
```

You should only see the following records from the console consumer you have running:

```
key1:The lazy
key1:jumped over
```

The reason you don't see the other records here is the producer assigned them to partition 1. You can test this for yourself by running executing a `CTRL+C` in the terminal window of the current console consumer, then run the following:

```
kafka-console-consumer --topic second-topic\
 --bootstrap-server broker:9092\
  --property print.key=true\
  --property key.separator="-"\
  --partition 1\
  --from-beginning
```

You should see the following results:

```
key2:brown fox
key2:the lazy dog
```

If you were to re-run the previous consumer without specifying a partition, you would see all the records produced to the topic. We'll go into more details about consumers and topic partitions in chapter 4.

At this point, we're done with the examples, so you can shut down the producer and the consumer by entering a `CTRL+C` command. Then you can stop all the docker containers now by running `docker-compose down`.

To quickly recap this exercise, you've just worked with the core Kafka functionality. You produced some records to a topic; then, in another process, you consumed them. While in practice, you'll use topics with higher partition counts, a much higher volume of messages, and something more sophisticated than the console tools, the concepts are the same.

We've also covered the basic unit of storage the broker uses, partitions. We discussed how Kafka assigns each incoming record a unique, per partition id-the offset, and always appends records at the end of the topic partition log. But as more data flows into Kafka, do these files continue to grow indefinitely? The answer to this question is no, and we'll cover how the brokers manage data in the next section.

2.5 Segments

So far, you've learned that brokers append incoming records to a topic partition file. But they don't just continue to append to the same one creating huge monolithic files. Instead, brokers break up the files into discrete parts called segments. Using segments enforcing the data retention settings and retrieving records by offset for consumers is much easier.

Earlier in the chapter, I stated the broker writes to a partition; it appends the record to a file. But a more accurate statement is the broker appends the record to the ***active segment***. The broker creates a new segment when a log file reaches a specific size (1 MB by default). The broker still uses previous segments for serving read (consume) requests from consumers. Let's look at an illustration of this process:

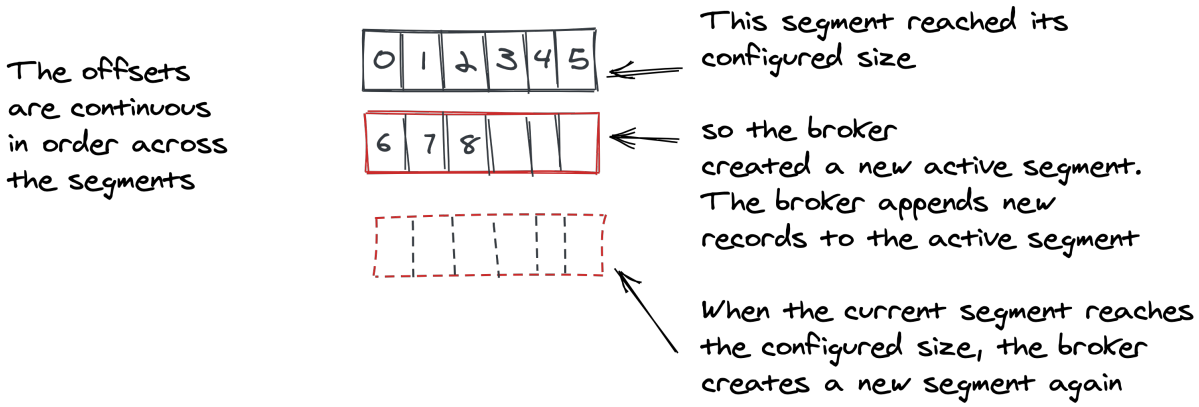

Figure 2.6 Creating new segments

Following along in the illustration here, the broker appends incoming records to the currently active segment. Once it reaches the configured size, the broker creates a segment that is considered the active segment. This process is repeated indefinitely.

The configuration controlling the size of a segment is `log.segment.bytes` which again has a default value of 1MB. Additionally, the broker will create new segments by time as well. The `log.roll.ms` or `log.roll.hours` governs the maximum time before the broker creates a new segment. The `log.roll.ms` is the primary configuration, but it has no default value, but the `log.roll.hours` has a default value of 168 hours (7 days). It's important to note when a broker creates a new segment based on time, and it means a new record has a timestamp greater than the earliest timestamp in the currently active segment plus the `log.roll.ms` or `log.roll.hours` configuration. It's not based on wall-clock time or when the file was last modified.

NOTE	The number of records in a segment won't necessarily be uniform, as the illustration might suggest here. In practice, they could vary in the total number of records. Remember, it's the total size or the age of the segment that triggers the broker to create a new one.

Now that we covered how the brokers create segments, we can talk about their data retention role.

2.5.1 Data retention

As records continue to come into the brokers, the brokers will need to remove older records to free up space on the file system over time. Brokers use a two-tiered approach to deleting data, time, and size. For time-based deletion, Kafka deletes records that are older than a configured retention time based on the timestamp of the record. If the broker placed all records in one big file, it would have to scan the file to find all those records eligible for deletion. But with the records stored in segments, the broker can remove segments where the latest timestamp in the segment exceeds the configured retention time. There are three time-based configurations for data deletion presented here in order of priority:

- `log.retention.ms` — How long to keep a log file in milliseconds
- `log.retention.minutes` — How long to keep a log file in minutes
- `log.retention.hours` — How long to keep a log file in hours

By default, only the `log.retention.hours` configuration has a default value, 168 (7 days). For size-based retention Kafka has the `log.retention.bytes` configuration. By default, it's set to `-1`. If you configure both size and time-based retention, then brokers will delete segments whenever either condition is met.

So far, we've focused our discussion on data retention based on the elimination of entire segments. If you remember, Kafka records are in key-value pairs. What if you wanted to retain the latest record per key? That would mean not removing entire segments but only removing the oldest records for each key. Kafka provides just such a mechanism called compacted topics.

2.5.2 Compacted topics

Consider the case where you have keyed data, and you're receiving updates for that data over time, meaning a new record with the same key will update the previous value. For example, a stock ticker symbol could be the key, and the price per share would be the regularly updated value. Imagine you're using that information to display stock values, and you have a crash or restart—you need to be able to start back up with the latest data for each key.[4]

If you use the deletion policy, a broker could remove a segment between the last update and the

application's crash or restart. You wouldn't have all the records on startup. It would be better to retain the final known value for a given key, treating the next record with the same key as an update to a database table.

Updating records by key is the behavior that compacted topics (logs) deliver. Instead of taking a coarse-grained approach and deleting entire segments based on time or size, compaction is more fine-grained and deletes old records *per key* in a log. At a high level, the log cleaner (a pool of threads) runs in the background, recopying log-segment files and removing records if there's an occurrence later in the log with the same key. Figure 2.13 illustrates how log compaction retains the most recent message for each key.

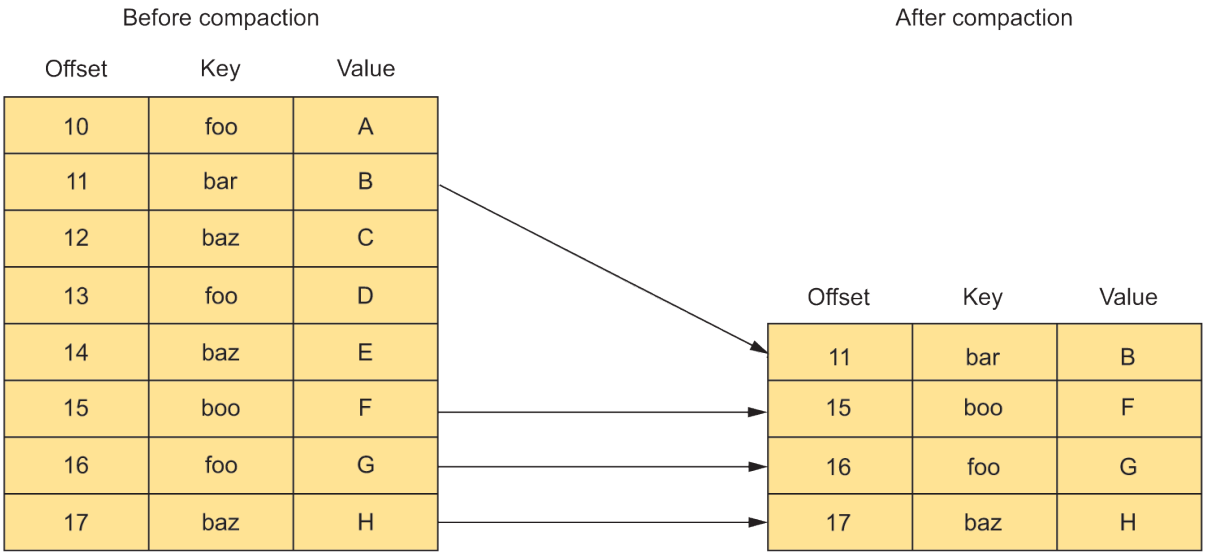

Figure 2.7 On the left is a log before compaction—you'll notice duplicate keys with different values. These duplicates are updates. On the right is after compaction—retaining the latest value for each key, but it's smaller in size.

This approach guarantees that the last record for a given key is in the log. You can specify log retention per topic, so it's entirely possible to use time-based retention and other ones using compaction.

By default, the log cleaner is enabled. To use compaction for a topic, you'll need to set the `log.cleanup.policy=compact` property when creating it.

Compaction is used in Kafka Streams when using state stores, but you won't be creating those logs/topics yourself—the framework handles that task. Nevertheless, it's essential to understand how compaction works. Log compaction is a broad subject, and we've only touched on it here. For more information, see the Kafka documentation: kafka.apache.org/documentation/{hash}compaction.

NOTE **With a `cleanup.policy` of `compact`, you might wonder how you can remove a record from the log. You delete with compaction by using a `null` value for the given key, creating a tombstone marker. Tombstones ensure that compaction removes prior records with the same key. The tombstone marker itself is removed later to free up space.**

The key takeaway from this section is that if you have independent, standalone events or messages, use log deletion. If you have updates to events or messages, you'll want to use log compaction.

Now that we've covered how Kafka brokers manage data using segments, it would be an excellent time to reconsider and discuss the topic-partition directories' contents.

2.5.3 Topic partition directory contents

Earlier in this chapter, we discussed that a topic is a logical grouping for records, and the partition is the actual physical unit of storage. Kafka brokers append each incoming record to a file in a directory corresponding to the topic and partition specified in the record. For review, here are the contents of a topic-partition

Listing 2.2 Contents of topic-partition directory

```
/var/kafka/topic-data/purchases-0
 00000000000000000000.index
 00000000000000000000.log
 00000000000000000000.timeindex
```

NOTE **In practice, you'll most likely not interact with a Kafka broker on this level. We're going into this level of detail to provide a deeper understanding of how broker storage works.**

We already know the `log` file contains the Kafka records, but what are the `index` and `timeindex` files? When a broker appends a record, it stores other fields along with the key and value. Three of those fields are the offset (which we've already covered), the size, and the record's physical position in the segment. The `index` is a memory-mapped file that contains a mapping of offset to position. The `timeindex` is also a memory-mapped file containing a mapping of timestamp to offset.

Let's look at the `index` files first.

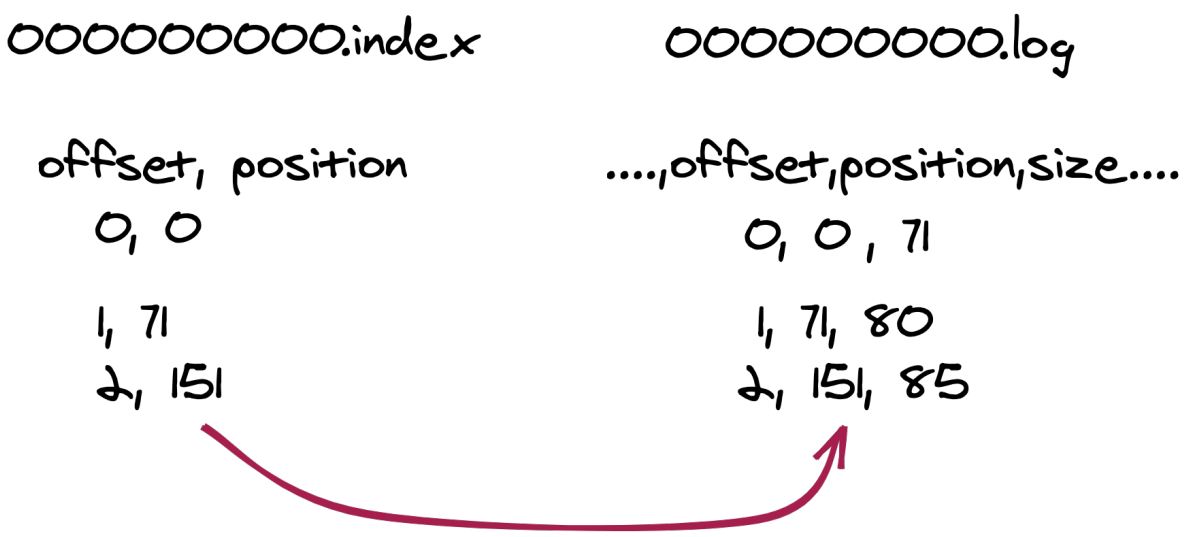

Figure 2.8 Searching for start point based on offset 2

Brokers use the index files to find the starting point for retrieving records based on the given offset. The brokers do a binary search in the `index` file, looking for an index-position pair with the largest offset that is less than or equal to the target offset. The offset stored in the `index` file is relative to the base offset. That means if the base offset is 100, offset 101 is stored as 1, offset 102 is stored as 2, etc. Using the relative offset, the `index` file can use two 4-byte entries, one for the offset and the other for the position. The base offset is the number used to name the file, which we'll cover soon.

The `timeindex` is a memory-mapped file that maintains a mapping of timestamp to offset.

NOTE	A memory-mapped file is a special file in Java that stores a portion of the file in memory allowing for faster reads from the file. For a more detailed description read the excellent entry www.geeksforgeeks.org/what-is-memory-mapped-file-in-java/ from GeeksForGeeks site.

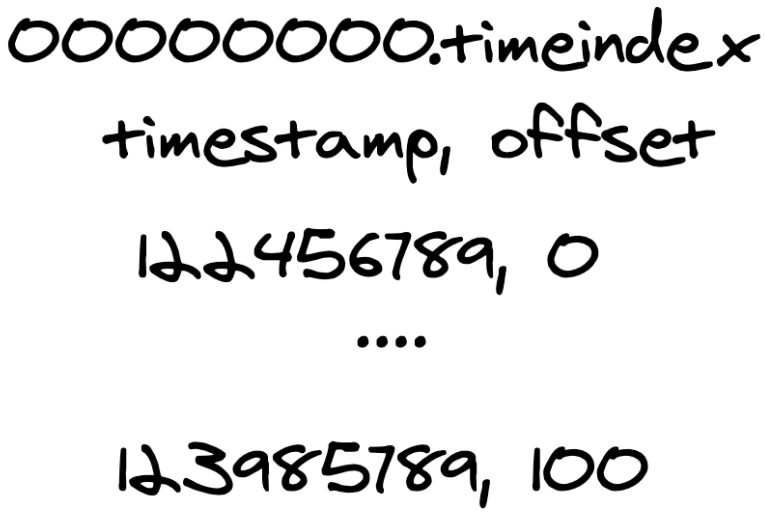

Figure 2.9 Timeindex file

The file's physical layout is an 8-byte timestamp and a 4-byte entry for the "relative" offset. The brokers search for records by looking at the timestamp of the earliest segment. If the timestamp is smaller than the target timestamp, the broker does a binary search on the `timeindex` file looking for the closest entry.

So what about the names then? The broker names these files based on the first offset contained in the `log` file. A segment in Kafka comprises the `log`, `index`, and `timeindex` files. So in our example directory listing above, there is one active segment. Once the broker creates a new segment, the directory would look something like this:

Listing 2.3 Contents of the directory after creating a new segment

```
/var/kafka/topic-data/purchases-0
 00000000000000000000.index
 00000000000000000000.log
 00000000000000000000.timeindex
 00000000000000037348.index
 00000000000000037348.log
 00000000000000037348.timeindex
```

Based on the directory structure above, the first segment contains records with offset 0-37347, and in the second segment, the offsets start at 37348.

The files stored in the topic partition directory are stored in a binary format and aren't suitable for viewing. As I mentioned before, you usually won't interact with the files on the broker, but sometimes when looking into an issue, you may need to view the files' contents.

IMPORTANT **You should never modify or directly access the files stored in the topic-partition directory. Only use the tools provided by Kafka to view the contents.**

2.6 Tiered storage

We've discussed that brokers are the storage layer in the Kafka architecture. We've also covered how the brokers store data in immutable, append-only files, and how brokers manage data growth by deleting segments when the data reaches an age exceeding the configured retention time. But as Kafka can be used for your data's central nervous system, meaning all data flows into Kafka, the disk space requirements will continue to grow. Additionally, you might want to keep the data longer but can't due to the need to make space for newly arriving records.

This situation means that Kafka users wanting to keep data longer than the required retention period need to offload data from the cluster to more scalable, long term storage. For moving the data, one could use Kafka Connect (which we'll cover in a later chapter), but long term storage requires building different applications to access that data.

There is current work underway called Tiered Storage. I'll only give a brief description here, but for more details, you can read KIP-405 (cwiki.apache.org/confluence/display/KAFKA/KIP-405%3A+Kafka+Tiered+Storage). At a high-level, the proposal is for the Kafka brokers to have a concept of local and remote storage. Local storage is the same as the brokers use today, but the remote storage would be something more scalable, say S3, for example, but the Kafka brokers still manage it.

The concept is that over time, the brokers migrate older data to the remote storage. This tiered storage approach is essential for two reasons. First, the data migration is handled by the Kafka brokers as part of normal operations. There is no need to set up a separate process to move older data. Secondly, the older data is still accessible via the Kafka brokers, so no additional applications are required to process older data. Additionally, the use of tiered storage will be seamless to client applications. They won't know or even need to know if the records consumed are local or from the tiered storage.

Using the tiered storage approach effectively gives Kafka brokers the ability to have infinite storage capabilities. Another benefit of tiered storage, which might not be evident at first blush, is the improvement in elasticity. When adding a new broker, full partitions needed to get moved across the network before tiered storage. Remember from our conversation from before, Kafka distributes topic-partitions among the brokers. So adding a new broker means calculating new assignments and moving the data accordingly. But with tiered storage, most of the segments beyond the active ones will be in the storage tier. This means there is much less data that needs to get moved around, so changing the number of brokers will be much faster.

As of the writing of this book (November 2020), tiered storage for Apache Kafka is currently underway. Still, given the project's scope, the final delivery of the tiered storage feature isn't expected until mid-2021. Again for the reader interested in the details involved in the tiered storage feature, I encourage you to read the details found in KIP-405 KIP-405 (cwiki.apache.org/confluence/display/KAFKA/KIP-405%3A+Kafka+Tiered+Storage).

2.7 Cluster Metadata

Kafka is a distributed system, and to manage all activity and state in the cluster, it requires metadata. But the metadata is external to the working brokers, so it uses a metadata server. Having a metadata server to keep this state is integral to Kafka's architecture. As of the writing of this book, Kafka uses ZooKeeper for metadata management. It's through the storage and use of metadata that enables Kafka to have leader brokers and to do such things as track the replication of topics.

The use of metadata in a cluster is involved in the following aspects of Kafka operations:

- *Cluster membership* — Joining a cluster and maintaining membership in a cluster. If a broker becomes unavailable, ZooKeeper removes the broker from cluster membership.
- *Topic configuration* — Keeping track of the topics in a cluster, which broker is the leader for a topic, how many partitions there are for a topic, and any specific configuration overrides for a topic.
- *Access control* — Identifying which users (a person or other software) can read from and write to particular topics.

NOTE **The term metadata manager is a bit generic. Up until the writing of this book, Kafka used ZooKeeper zookeeper.apache.org for metadata management. There is an effort underway to remove ZooKeeper and use Kafka itself to store the cluster metadata. KIP - 500 cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum describes the details. This blog post,**

) describes the details. This blog post, www.confluent.io/blog/removing-zookeeper-dependency-in-kafka/, describes the process of how and when the changes to Kafka occur. Since most users don't work at the level of cluster metadata, I feel that some knowledge of *how* Kafka uses metadata is sufficient.

This has been a quick overview of how Kafka manages metadata. I don't want to go into too much detail about metadata management as my approach to this book is more from the developer's point of view and not someone who will manage a Kafka cluster. Now that we've briefly discussed Kafka's need for metadata and how it's used let's resume our discussion on leaders and followers and their role in replication.

2.8 Leaders and followers

So far, we've discussed the role topics play in Kafka and how and why topics have partitions. You've seen that partitions aren't all located on one machine but are spread out on brokers throughout the cluster. Now it's time to look at how Kafka provides data availability in the face of machine failures.

In the Kafka cluster for each topic-partition, one broker is the *leader*, and the rest are followers.

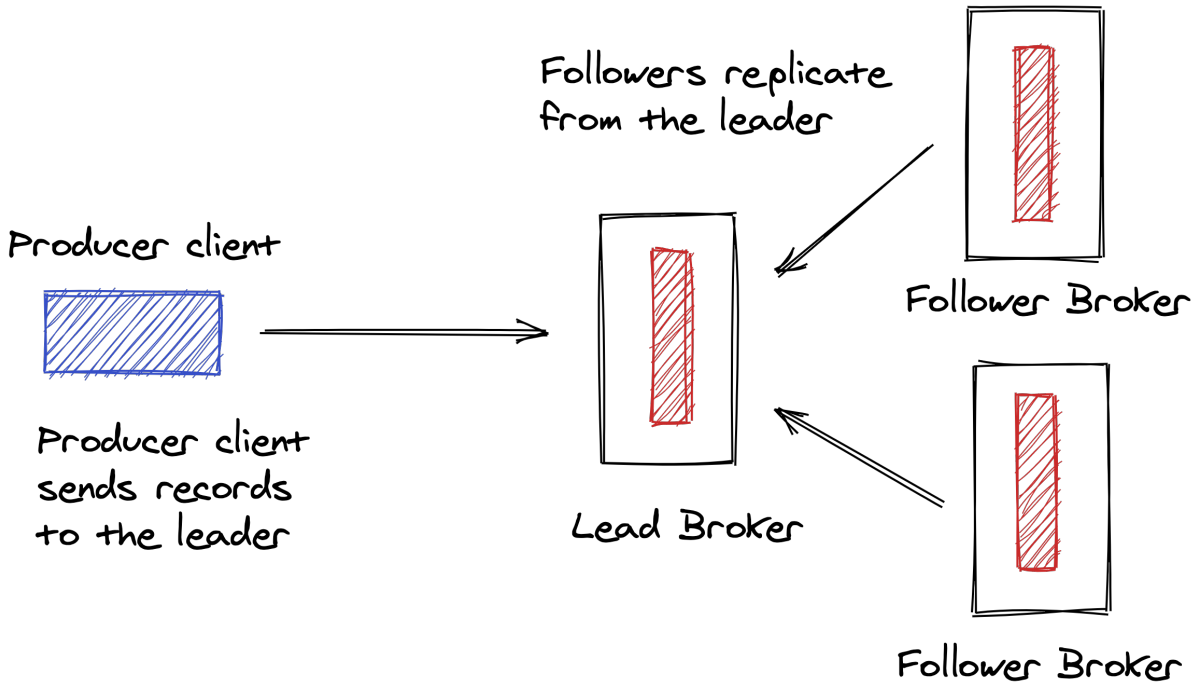

Figure 2.10 Leader and follower example

In figure 10 above, we have a simplified view of the leader and follower concept. The lead broker for a topic-partition handles all of the produce and consume requests (although it is possible to have consumers work with followers, and we'll cover that in the chapter on clients). The following brokers replicate records from the leader for a given topic partition. Kafka uses this leader and follower relationship for data integrity. It's important to remember the leadership for the topic- partitions are spread around the cluster. No single broker is the leader for all partitions of a given topic.

But before we discuss how leaders, followers, and replication work, we need to consider what Kafka does to achieve this.

2.8.1 Replication

I mentioned in the leaders and followers section that topic-partitions have a leader broker and one or more followers. Illustration 10 above shows this concept. Once the leader adds records to its log, the followers read from the leader.

Kafka replicates records among brokers to ensure data availability, should a broker in the cluster fail. Figure 11 below demonstrates the replication flow between brokers. A user configuration determines the replication level, but it's recommended to use a setting of three. With a replication factor of three, the lead broker is considered a replica one, and two followers are replica two and three.

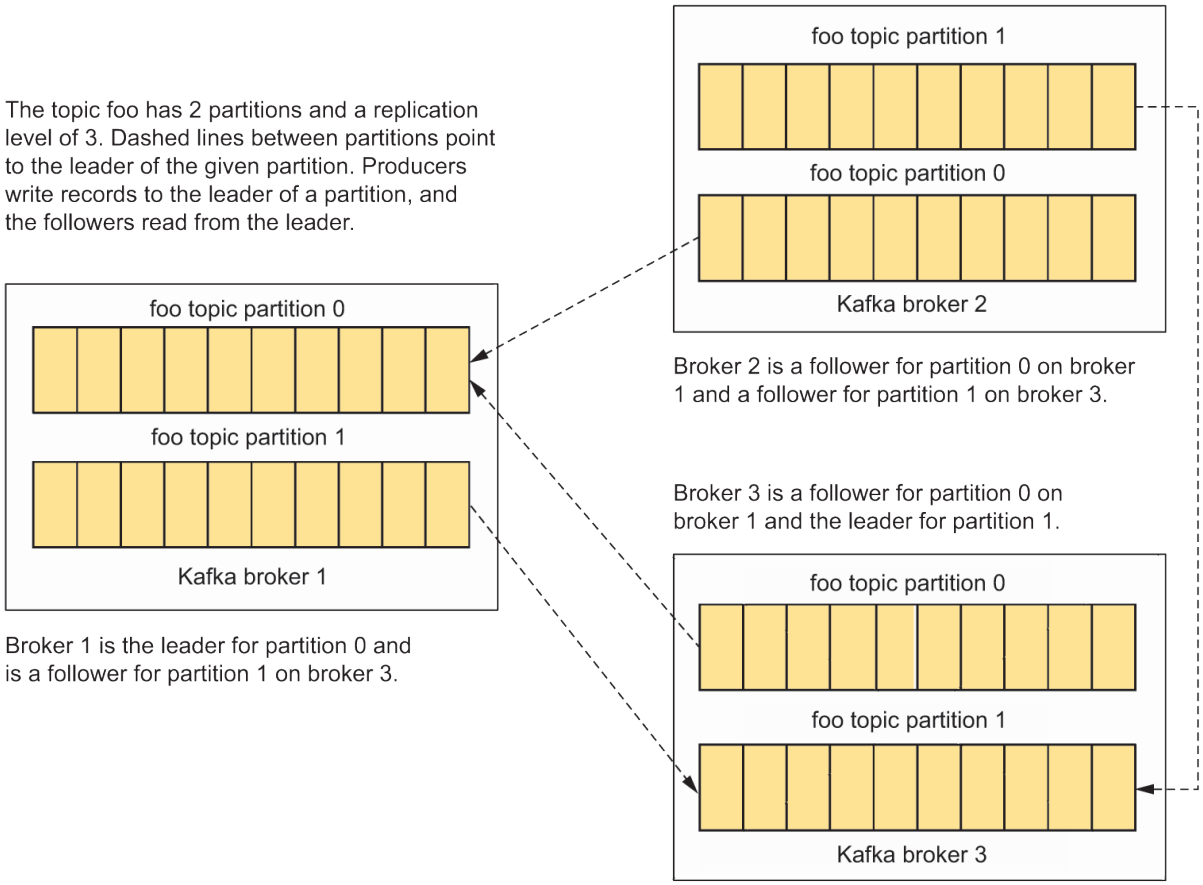

Figure 2.11 The Kafka replication process

The Kafka replication process is straightforward. Brokers following a topic-partition consume messages from the topic-partition leader. After the leader appends new records to its log, followers consume from the leader and append the new records to their log. After the followers have completed adding the records, their logs replicate the leader's log with the same data and offsets. When fully caught up to the leader, these following brokers are considered an in-sync replica or ISR.

When a producer sends a batch of records, the leader must first append those records before the

followers can replicate them. There is a small window of time where the leader will be ahead of the followers. This illustration demonstrates this concept:

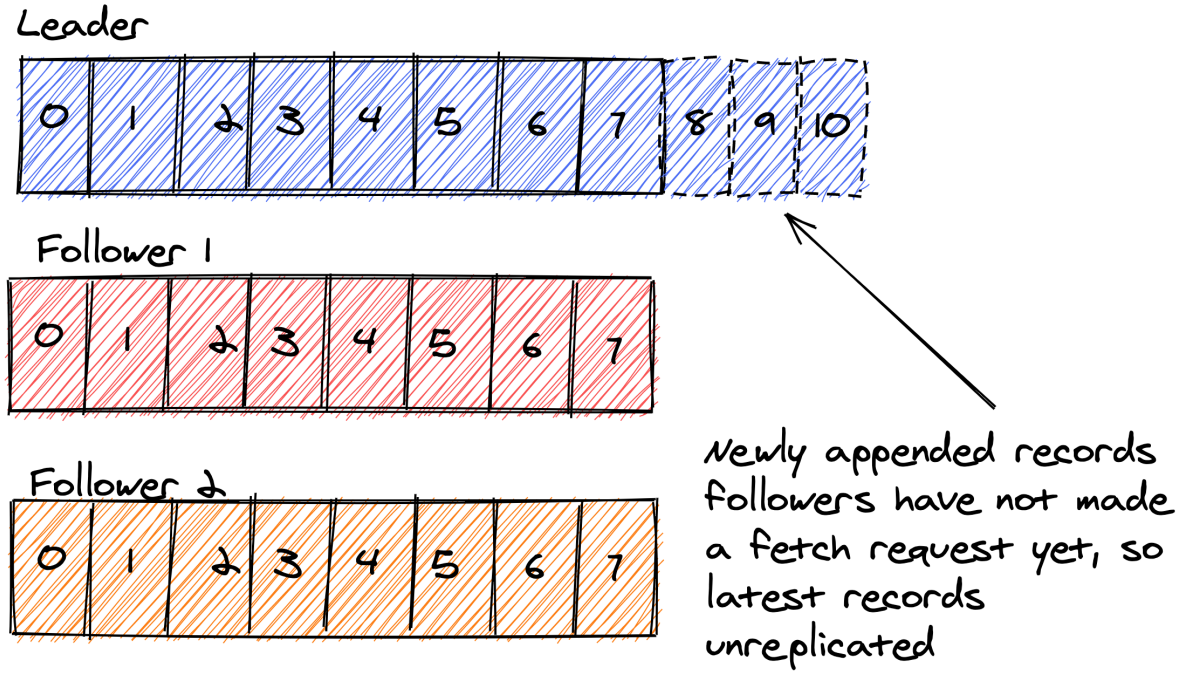

Figure 2.12 The leader may have a few unreplicated messages in its topic-partition

In practical terms, this small lag of replication records is no issue. But, we have to ensure that it must not fall too far behind, as this could indicate an issue with the follower. So how do we determine what's not too far behind? Kafka brokers have a configuration `replica.lag.time.max.ms`.

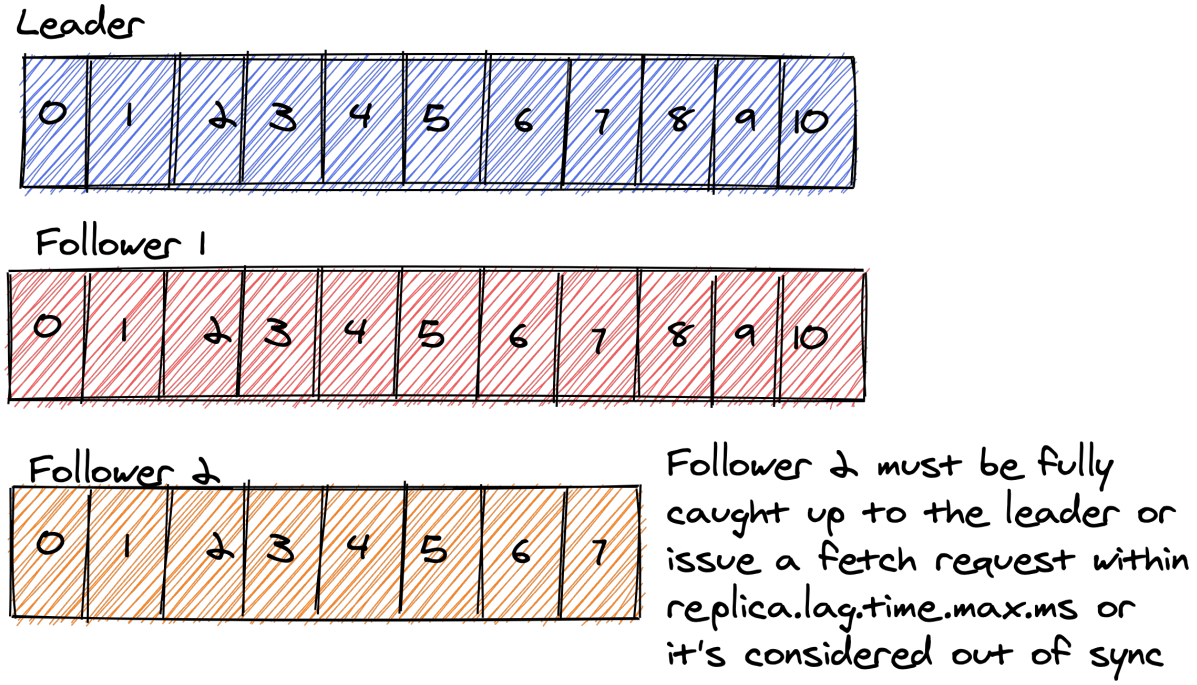

Figure 2.13 Followers must issue a fetch request or be caught up withing lag time configuration

The replica lag time configuration sets an upper-bound how long followers have to either issue a fetch request or be entirely caught-up for the leader's log. Followers failing to do so within the configured time are considered too far behind and removed from the in-sync replica (ISR) list.

As I stated above, follower brokers who are caught up with their leader broker are considered an in-sync replica or ISR. ISR brokers are eligible to be elected leader should the current leader fail or become unavailable.[5]

In Kafka, consumers never see records that haven't been written by all ISRs. The offset of the latest record stored by all replicas is known as the high-water mark, and it represents the highest offset accessible to consumers. This property of Kafka means that consumers don't worry about recently read records disappearing. As an example, consider the situation in illustration 11 above. Since offsets 8-10 haven't been written to all the replicas, 7 is the highest offset available to consumers of that topic.

Should the lead broker become unavailable or die before records 8-10 are persisted, that means an acknowledgment isn't sent to the producer, and it will retry sending the records. There's a little more to this scenario, and we'll talk about it more in the chapter on clients.

If the leader for a topic-partition fails, a follower has a complete replica of the leader's log. But we should explore the relationship between leaders, followers, and replicas.

REPLICATION AND ACKNOWLEDGMENTS

When writing records to Kafka, the producer can wait for acknowledgment of record persistence of none, some, or all for in-sync replicas. These different settings allow for the producer to trade-off latency for data durability. But there is a crucial point to consider.

The leader of a topic-partition is considered a replica itself. The configuration `min.insync.replicas` specifies how many replicas must be in-sync to consider a record committed. The default setting for `min.insync.replicas` is one. Assuming a broker cluster size of three and a replication-factor of three with a setting of `acks=all`, only the leader must acknowledge the record. The following illustration demonstrates this scenario:

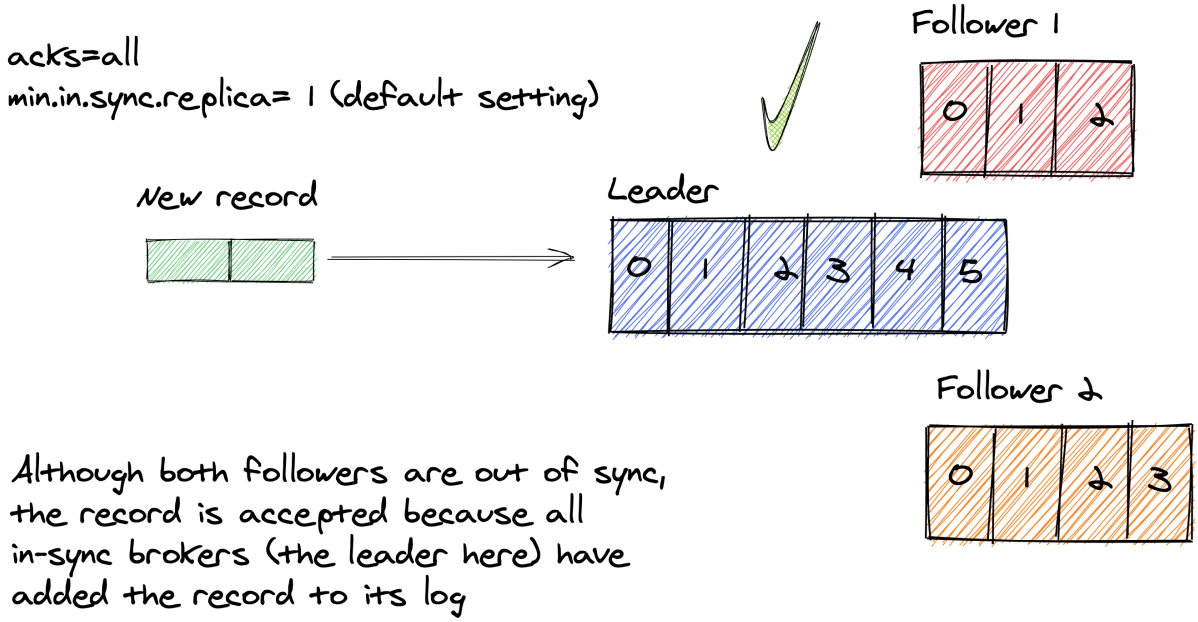

Figure 2.14 Acks set to "all" with default in-sync replicas

How can something like the above happen? Imagine that the two followers temporarily lag enough for the controller to remove them from the ISR. This means that even with setting `acks=all` on the producer, there is a potential for data loss should the leader fail before the followers have a chance to recover and become in sync again.

To prevent such a scenario, you need to set the `min.insync.replicas=2` Setting the min in-sync replicas configuration to two means that the leader checks the number of in-sync replicas before appending a new record to its log. If the required number of in-sync replicas isn't met at this point, the leader doesn't process the produce request. Instead, the leader throws a `NotEnoughReplicasException`, and the producer will retry the request.

Let's look at another illustration to help get a clear idea of what is going on:

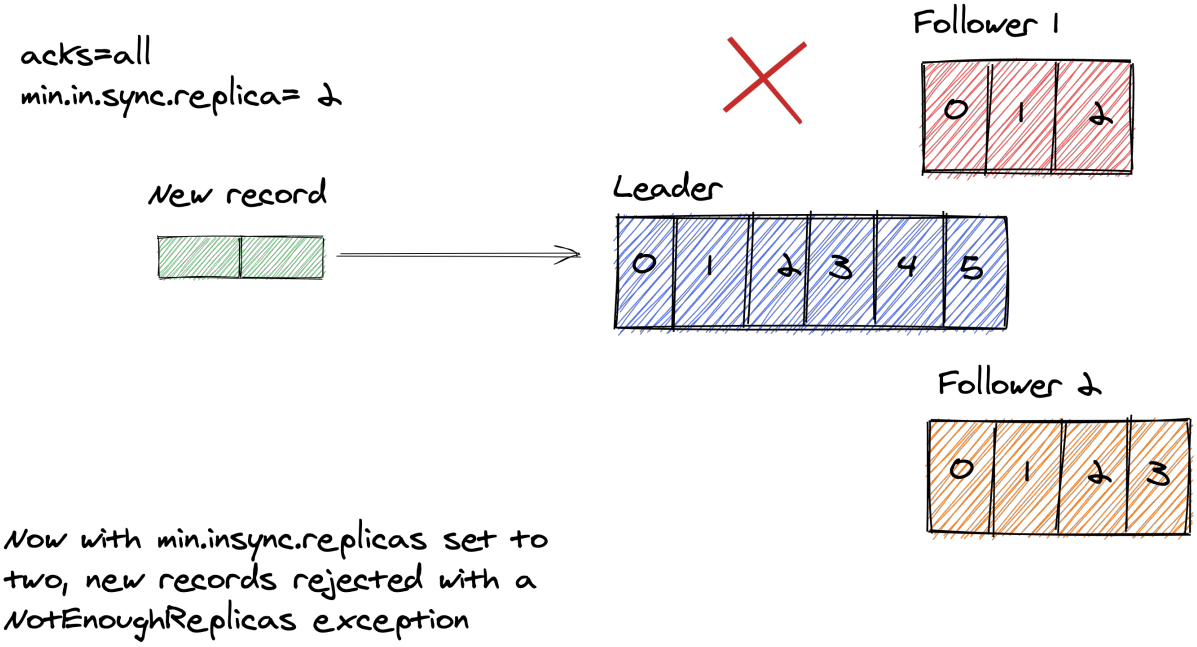

Figure 2.15 Setting Min ISR to a value greater than one increases data durability

As you can see in figure 14, a batch of records arrives. But the leader won't append them because there aren't enough in-sync replicas. By doing so, your data durability increases as the produce request won't succeed until there are enough in-sync replicas. This discussion of message acknowledgments and in-sync replicas is broker-centric. In chapter 4, when we discuss clients, we'll revisit this idea from the producer client's perspective to discuss the performance trade-offs.

2.9 Checking for a healthy broker

At the beginning of the chapter, we covered how a Kafka broker handles requests from clients and process them in the order of their arrival. Kafka brokers handle several types of requests, for example:

- Produce - A request to append records to the log
- Fetch - A request to consume records from a given offset
- Metadata - A request for the cluster's current state - broker leaders for topic-partitions, topic partitions available, etc.

These are a small subset of all possible requests made to the broker. The broker processes requests in first-in-first-out processing order, passing them off to the appropriate handler based on the request type.

Simply put, a client makes a request, and the broker responds. If they come in faster than the broker can reply, the requests queue up. Internally, Kafka has a thread-pool dedicated to handling the incoming requests. This process leads us to the first line of checking for issues should your Kafka cluster performance suffer.

With a distributed system, you need to embrace failure as a way of life. However, this doesn't mean that the system should shut down at the first sign of an issue. Network partitions are not uncommon in a distributed system, and frequently they resolve quickly. So it makes sense to have a notion of retryable errors vs. fatal errors. If you are experiencing issues with your Kafka installation, timeouts for producing or consuming records, for example, where's the first place to look?

2.9.1 Request handler idle percentage

When you are experiencing issues with a Kafka based application, a good first check is to examine the `RequestHandlerAvgIdlePercent` JMX metric.

The `RequestHandlerAvgIdlePercent` metric provides the average fraction of time the threads handling requests are idle, with a number between 0 and 1. Under normal conditions, you'd expect to see an idle ratio of .7 - .9, indicating that the broker handles requests quickly. If the request-idle number hits zero, there are no threads left for processing incoming requests, which means the request queue continues to increase. A massive request queue is problematic, as that means longer response times and possible timeouts.

2.9.2 Network handler idle percentage

The `NetworkProcessorAvgIdlePercent` JMX metric is analogous to the request-idle metric. The network-idle metric measures the average amount of time the network processors are busy. In the best scenarios, you want to see the number above 0.5 if it's ***consistently*** below 0.5 that indicates a problem.

2.9.3 Under replicated partitions

The `UnderReplicatedPartitions` JMX metric represents the number of partitions belonging to a broker removed from the ISR (in-sync replicas). We discussed ISR and replication in the `Replication` section. A value higher than zero means a Kafka broker is not keeping up with replicating for assigned following topic-partitions. Causes of a non-zero `UnderReplicatedPartitions` metric could indicate network issues, or the broker is overloaded and can't keep up. Note that you always want to see the URP number at zero.

2.10 Summary

- The Kafka broker is the storage layer and also handles requests from clients for producing (writing) and consuming (reading) records
- Kafka brokers receive records as bytes, stores them in the same format, and sends them out for consume requests in byte format as well
- Kafka brokers durably store records in topics.
- Topics represent a directory on the file system and are partitioned, meaning the records in a topic are placed in different buckets
- Kafka uses partitions for throughput and for distributing the load as topic-partitions are spread out on different brokers
- Kafka brokers replicate data from each other for durable storage