CONFLUENT

The Best Apache Kafka® Tips and Tricks That Every Kafka Developer Should Know

Bill Bejeck

Table of Contents

Introduction

Apache Kafka® is an event streaming platform used by more than 30% of the Fortune 500. There are numerous features of Kafka that make it the de facto standard for an event streaming platform. Gaining a deeper understanding of just a handful of these features can enable you to quickly improve the performance of your applications, as well as the efficiency of your development process.

In this white paper, you'll learn about five Kafka elements that deserve your closer attention, either because they significantly improve upon the behavior of their predecessors, because they are easy to overlook or to make assumptions about, or simply because they are extremely useful!

Message Delivery and Durability Guarantees

Acks Options

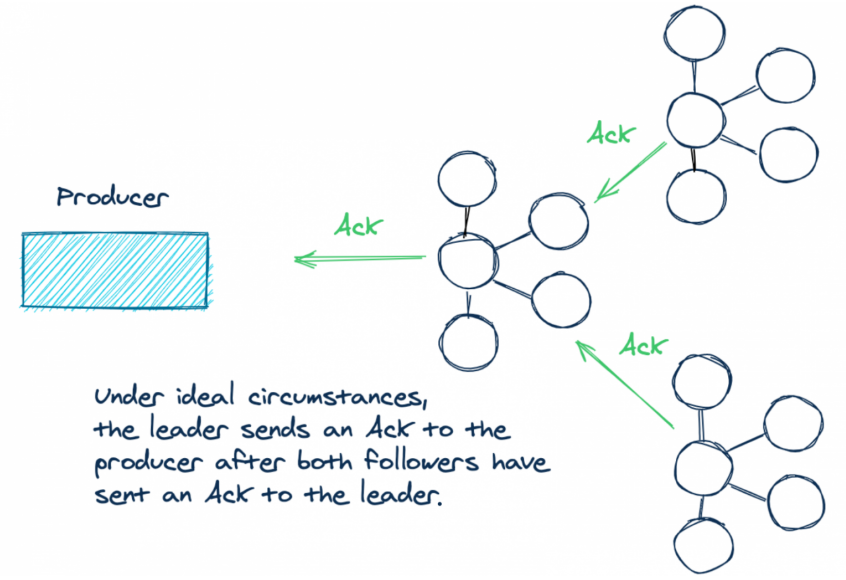
For data durability, the KafkaProducer has the configuration setting `acks`. The `acks` setting specifies how many acknowledgments the producer must receive in order to consider a record delivered to the broker. The available values are:

- `none`: The producer considers the record successfully delivered once it has sent the record to the broker. This is basically "fire and forget."
- `one`: The producer waits for the lead broker to acknowledge that it has written the record to its log.
- `all`: The producer waits for an acknowledgment from the lead broker and from the follower brokers that they have successfully written the record to their logs.

As you can see, there is a trade-off to make here. This is by design, because different applications have different requirements. You can opt for higher throughput with a chance of data loss, or you may prefer a very high data durability guarantee at the cost of lower throughput.

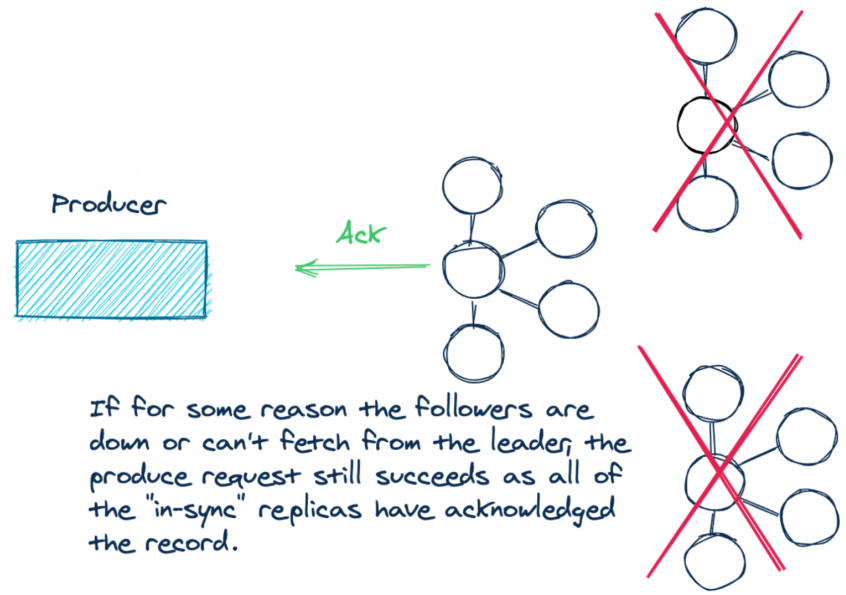
Let's take a second to discuss the `acks=all` scenario. If you set `acks` to `all` and produce records to a cluster of three Kafka brokers, then under ideal conditions, Kafka will contain three replicas of your data: one for the lead broker and one each for the two followers. When the logs of each of these replicas all have the same record offsets, they are considered to be *in sync*. These in-sync replicas have the same content for a given topic partition.

Take a look at the following illustration to better understand what's going on:



But there's some subtlety to the use of the `acks=all` configuration. What it doesn't specify is *how many* replicas need to be in sync. The lead broker will always be in sync with itself. But you could have a situation where the two follower brokers can't keep up due to network partitions, record load, etc. So when a producer has a successful send, the actual number of acknowledgments could have come from only one broker! If the two followers are not in sync, the producer still receives the required number of `acks`, but in this case, it's only from the leader.

For example:



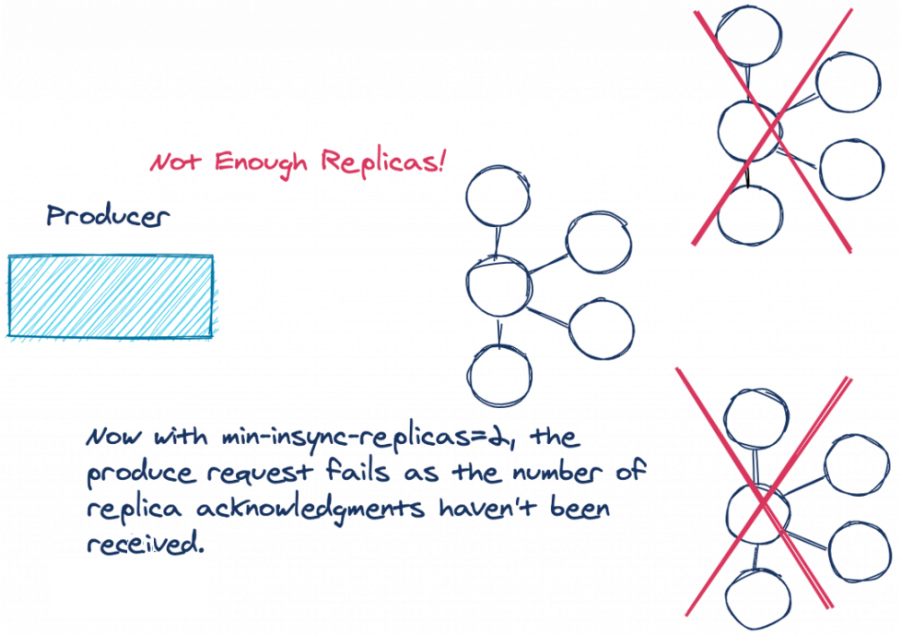
By setting `acks=all`, you are placing a premium on the durability of your data. So if the replicas aren't keeping up, it stands to reason that you want to raise an exception for new records until the replicas are caught up.

In a nutshell, having only one in-sync replica follows the "letter of the law" but not the "spirit of the law." What we need is a guarantee when using the `acks=all` setting. A successful send involves at least a majority of the available in-sync brokers.

Setting a Minimum

There just so happens to be such a guaranteeing setting: `min.insync.replicas`. The `min.insync.replicas` configuration setting enforces the number of replicas that must be in sync for the write to proceed. The `min.insync.replicas` setting is set at the broker or topic level, and is not part of the producer configuration. The default value for `min.insync.replicas` is one. So to avoid the scenario described above, in a three-broker cluster, you'd want to increase the value to two.

Let's revisit our previous example and see the difference:



If the number of in-sync replicas is below the configured amount, the lead broker won't attempt to append the record to its log. Instead, the leader throws either a `NotEnoughReplicasException` or `NotEnoughReplicasAfterAppendException`, forcing the producer to retry the write. Having replicas out of sync with the leader is considered a retryable error, so the producer will continue to retry and send the records up to the configured delivery timeout.

So by setting the `min.insync.replicas` and producer `acks` configuration settings to work together in

this way, you've increased the durability of your data.

Now let's move on to the next item in our list: the Kafka clients, i.e. the Kafka Producer and Consumer APIs.

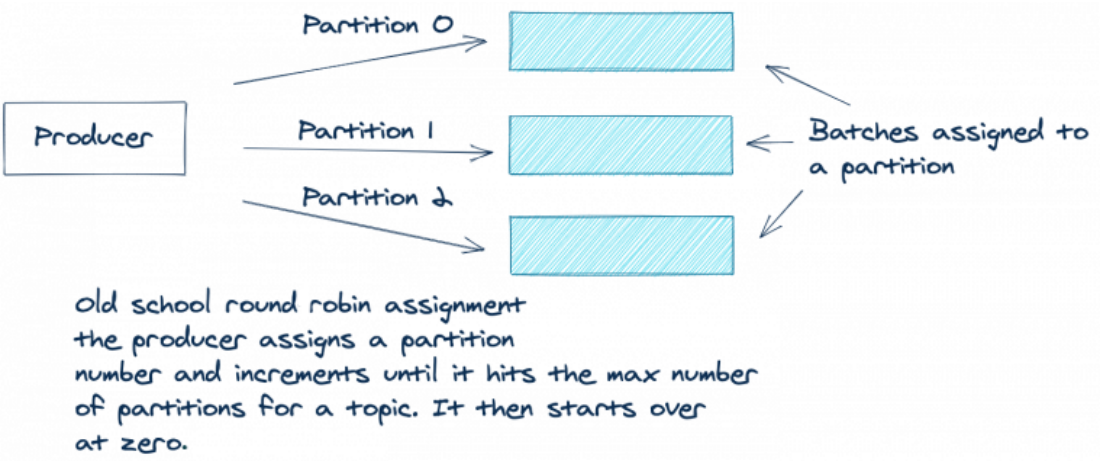
The Sticky Partitioner in the Producer API

Kafka uses partitions to increase throughput and spread the load of messages to all brokers in a cluster. Kafka records are in a key/value format, where the keys can be null. Producers don't immediately send records, but place them in partition-specific batches to be sent later. This batching is an effective means of increasing network utilization.

There are three ways in which the partitioner determines the partition to use for records:

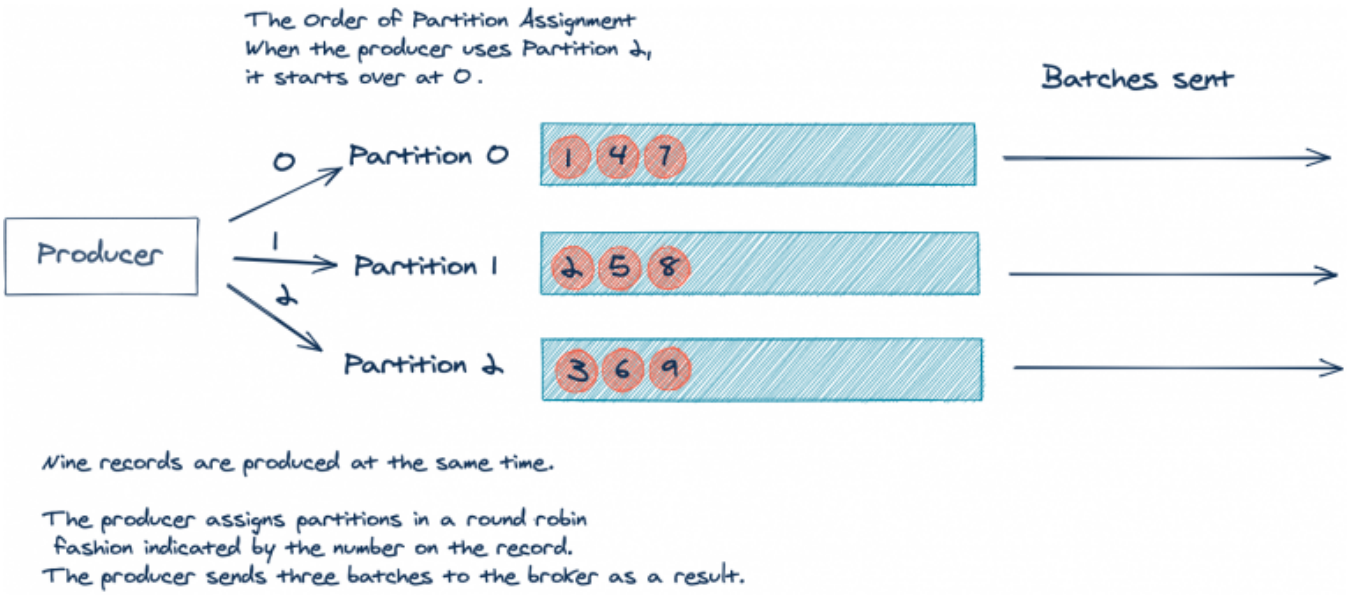
1. The partition can be explicitly provided in the ProducerRecord object, via the overloaded ProducerRecord constructor. In this case, the producer always uses this partition.
2. If no partition is provided, and the ProducerRecord has a key, the producer takes the hash of the key modulo the number of partitions. The resulting number from this calculation is the number of the partition that the producer will use.
3. In the past, if there was no key and no partition was present in the ProducerRecord, Kafka used a round robin approach to assign messages across partitions. The producer would assign the first record in the batch to partition zero, the second to partition one, and so on, until reaching the end of the partitions. The producer would then start over with partition zero and repeat the entire process for all of the remaining records.

The following illustration depicts this process:



The round robin approach works well for even distribution of records across partitions, but there's one drawback. Due to this "fair" round robin approach, you can end up sending multiple sparsely populated batches. It's more efficient to send fewer batches, with more records in each batch. Fewer batches mean less queuing of produce requests, resulting in less load on the brokers.

To demonstrate this, let's look at a simplified example where you have a topic with three partitions. For the sake of simplicity, we'll assume that your application has produced nine records with no key, and that all of the records arrive at the same time:

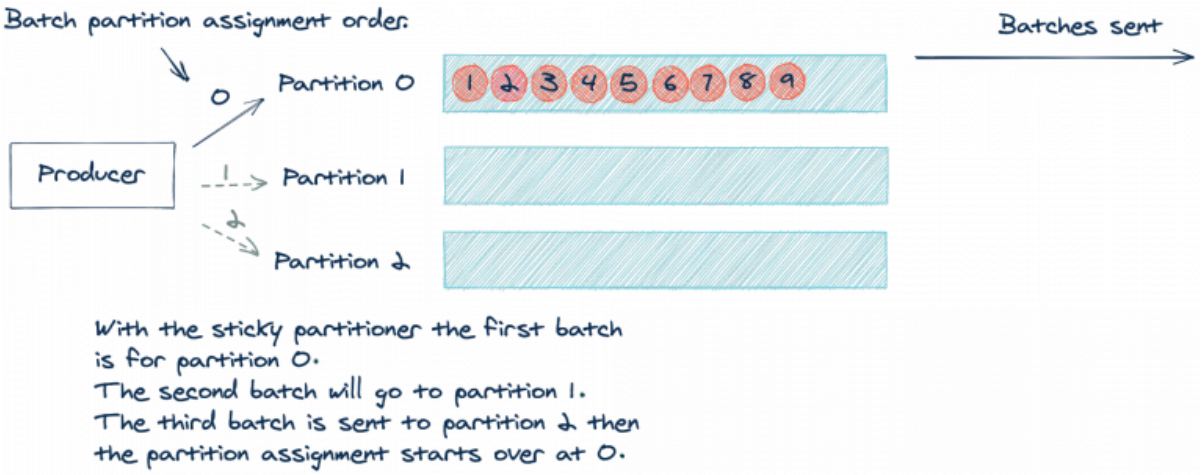


As you can see in the illustration, the nine incoming records will result in three batches of three records. But it would be better if we could send one batch of nine records. As mentioned before, fewer batches result in less network traffic and less load on the brokers.

Sending Full Batches

Apache Kafka 2.4.0 added the sticky partitioner approach, which now makes this possible. Instead of using a round robin approach per record, the sticky partitioner assigns records to the same partition until the batch is sent. Then, after sending the batch, the sticky partitioner increments the partition number to use for the next batch.

Let's revisit our last illustration, but this time see what happens when we use the sticky partitioner:



By using the same partition until a batch is full or otherwise completed, we'll send fewer produce requests, which reduces the load on the request queue and reduces latency of the system as well. It's worth noting that the sticky partitioner approach still results in an even distribution of records; the even distribution occurs over time, as the partitioner sends a batch to each partition. You can think of it as a "per-batch" round robin, or "eventually even" approach.

To learn more about the sticky partitioner, you can read the Apache Kafka Producer Improvements with the Sticky Partitioner blog post and the related KIP-480 design document.

Now let's move on to the Consumer changes.

How to Avoid "Stop-the-World" Consumer Group Rebalances

Kafka is a distributed system, and one of the key requirements of a distributed system is to be able to deal with failures and disruptions—not just to anticipate failures, but to fully embrace them. A great example of how Kafka handles this expected disruption is the consumer group protocol, which manages

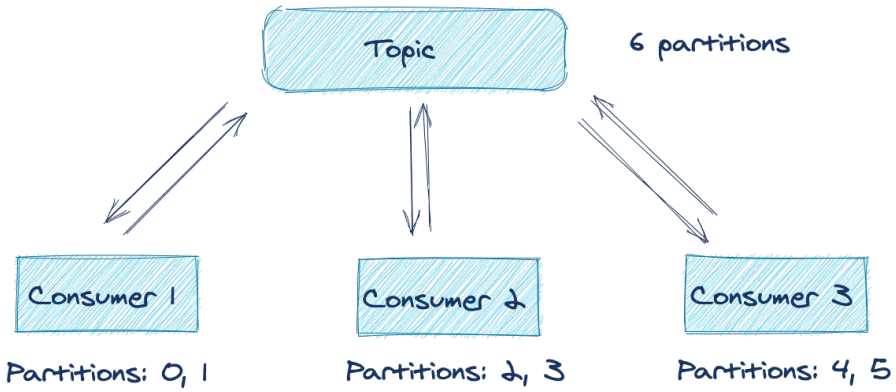
multiple instances of a consumer for a single logical application. If an instance of a consumer stops—by design or otherwise—Kafka will *rebalance* and make sure that another instance of the consumer takes over the work.

In version 2.4, Kafka introduced a new rebalance protocol: cooperative rebalancing. Before we dive into the new protocol, let's look in more detail at the consumer group basics.

Default Rebalancing Approach

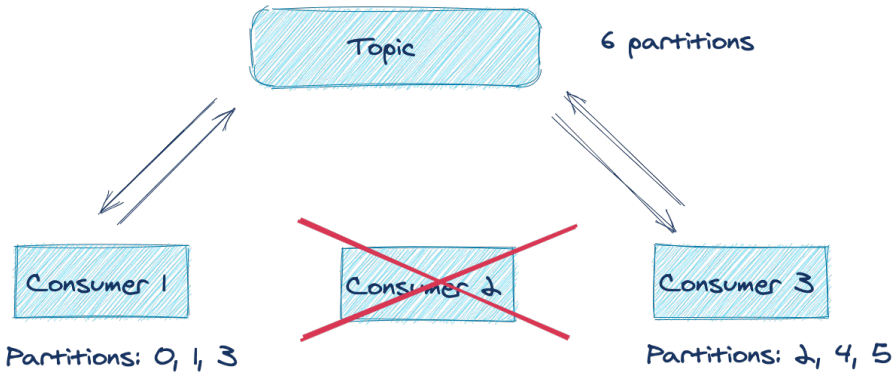
Let's assume you have a distributed application with several consumers subscribed to a topic. Any set of consumers configured with the same group.id form one logical consumer, which is called a "consumer group." Each consumer in the consumer group is responsible for consuming from one or more partitions of the subscribed topic(s). These partitions are assigned by the leader of the consumer group.

Here's an illustration demonstrating this concept:



From this illustration, you can see that under optimal conditions, all three consumers are processing records from two partitions each. But what happens if one of the applications suffers an error or can't connect to the network any more? Does processing for those topic partitions stop until you can restore the application in question? Fortunately, the answer is no, thanks to the consumer rebalancing protocol.

Here's another illustration, showing the consumer group protocol in action:



As you can see, Consumer 2 fails for some reason and either misses a poll or triggers a session timeout. The group coordinator removes it from the group and triggers what is known as a rebalance. A rebalance is a mechanism that attempts to evenly distribute (or balance) the workload across all available members of a consumer group. In this case, since Consumer 2 left the group, the rebalance assigns its previously owned partitions to the other active members of the group. So as you can see, losing a consumer application for a particular group ID doesn't result in a loss of processing on those topic partitions.

There is, however, a drawback of the default rebalancing approach. Each consumer gives up its entire assignment of topic partitions, and no processing takes place until the topic partitions are reassigned. This is sometimes referred to as a "stop-the-world" rebalance. To compound the issue, depending on the instance of the `ConsumerPartitionAssignor` used, consumers may be simply reassigned the same topic partitions that they owned prior to the rebalance. The net effect of this is that there is no need to pause work on those partitions.

This implementation of the rebalance protocol is called ***eager rebalancing,*** because it prioritizes the importance of ensuring that no consumers in the same group claim ownership over the same topic partitions. Ownership of the same topic partition by two consumers in the same group would result in undefined behavior.

The Cooperative Approach

While it is critical to keep any two consumers from claiming ownership over the same topic partition, it turns out that there is a better approach that provides safety without compromising on time spent processing: incremental cooperative rebalancing. First introduced to Kafka Connect in Apache Kafka 2.3, this has now also been implemented for the consumer group protocol. With the cooperative approach, consumers don't automatically give up ownership of all topic partitions at the start of the rebalance. Instead, all members encode their current assignment and forward the information to the group leader. The group leader then determines which partitions need to change ownership, instead of producing an

entirely new assignment from scratch.

Now a second rebalance is issued, but this time, only the topic partitions that need to change ownership are involved. It could be revoking topic partitions that are no longer assigned or adding new topic partitions. For the topic partitions that are in both the new and old assignments, nothing has to change, which means continued processing for topic partitions that aren't moving.

The bottom line is that eliminating the "stop-the-world" approach to rebalancing, and only stopping the topic partitions involved, means less costly rebalances, thus reducing the total time to complete a rebalance. Even long rebalances are less painful now that they don't stop all processing. This positive change in rebalancing is made possible by the `CooperativeStickyAssignor`. The `CooperativeStickyAssignor` makes the trade-off of having a second rebalance in exchange for a faster return to normal operations.

To enable this new rebalance protocol, you need to set the `partition.assignment.strategy` to use the new `CooperativeStickyAssignor`. Be aware that this change is entirely on the client side; to take advantage of the new rebalance protocol, you only need to update your client version. If you're a Kafka Streams user, there is even better news: Kafka Streams enables the cooperative rebalance protocol by default, so you have nothing else to do!

The Command Line Tools

The Apache Kafka binary installation includes several tools located in the `bin` directory. While you'll find several tools in that directory, the four tools that will have the most impact on your day-to-day work are `console-consumer`, `console-producer`, `dump-log`, and `delete-records`.

Kafka Console Producer

The console producer allows you to produce records to a topic directly from the command line. Producing from the command line is a great way to quickly test new consumer applications when you aren't producing data to the topics yet. To start the console producer, run this command:

```
kafka-console-producer --topic  \
                       --broker-list <broker-host:port>
```

After you execute the command, there's an empty prompt waiting for your input. Just type in some characters and hit Enter to produce a message.

Using the command line producer in this way does not send any keys, only values. Luckily, there is a way to send keys as well. You just have to update the command to include the necessary flags:

```
kafka-console-producer --topic  \
                       --broker-list <broker-host:port> \
                       --property parse.key=true \
                       --property key.separator=":"
```

The choice of the `key.separator` property is arbitrary; you can use any character. And now, you can send full key/value pairs from the command line! In case you are using Confluent Schema Registry, there are command line producers available for sending records in the Apache Avro, Protocol Buffers (Protobuf), and JSON Schema formats.

Now let's take a look at the other side of the coin: consuming records from the command line.

Kafka Console Consumer

The console consumer gives you the ability to consume records from a Kafka topic directly from the command line. Being able to quickly start a consumer can be an invaluable tool in prototyping or debugging. Consider the case of building a new microservice: to quickly confirm that your producer application is sending messages, you can simply run this command:

```
kafka-console-consumer --topic  \
                         --bootstrap-server <broker-host:port>
```

After you run the command, you'll start seeing records scroll across your screen (so long as data is currently being produced to the topic). If you want to see all of the records from the start, you can add a `--from-beginning` flag to the command, and you'll see all records produced to that topic.

```
kafka-console-consumer --topic <topic-name> \
                         --bootstrap-server <broker-host:port> \
                         --from-beginning
```

In case you are using Schema Registry, there are command line consumers available for records encoded

in the Avro, Protobuf, and JSON Schema formats. The Schema Registry command line consumers are intended for working with records in Avro, Protobuf, or JSON Schema, while the plain consumers work with records of primitive Java types: `String`, `Long`, `Double`, `Integer`, etc. For the plain console consumer, the default format expected for keys and values is the `String` type. If the keys or values are not strings, you'll need to provide the deserializers; add the command line flags `--key-deserializer` and `--value-deserializer` and supply the fully qualified class names of the respective deserializers.

You may well have noticed that by default, the console consumer only prints the value component of the messages to the screen. If you want to see the keys as well, you can do so by including the necessary flags:

```
kafka-console-consumer --topic  \
                         --bootstrap-server <broker-host:port> \
                         --property print.key=true
                         --property key.separator=":"
```

As with the producer, the value used for the key separator is arbitrary, so you can choose any character that you want to use.

Dump Log

Sometimes when you're working with Kafka, you may find yourself needing to manually inspect the underlying logs of a topic. Whether you're just curious about Kafka internals or you need to debug an issue and verify the content, the `kafka-dump-log` command is your friend. Here's a command used to view the log of an example topic aptly named `example`:

```
kafka-dump-log \
--print-data-log \
--files  ./var/lib/kafka/data/example-0/00000000000000000000.log
```

- The `--print-data-log` flag specifies that we want to print the data in the log.
- The `--files` flag is required. (This can also be a comma-separated list of files.)

For a full list of options and a description of what each one does, run `kafka-dump-log` with the `--help` flag.

Running the command to print the log data yields something like this:

```
Dumping ./var/lib/kafka/data/example-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional:
false isControl: false position: 0 CreateTime: 1599775774460 size: 81 magic:
2 compresscodec: NONE crc: 3162584294 isvalid: true
| offset: 0 CreateTime: 1599775774460 keysize: 3 valuesize: 10 sequence: -1
headerKeys: [] key: 887 payload: -2.1510235
baseOffset: 1 lastOffset: 9 count: 9 baseSequence: -1 lastSequence: -1
producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional:
false isControl: false position: 81 CreateTime: 1599775774468 size: 252
magic: 2 compresscodec: NONE crc: 2796351311 isvalid: true
| offset: 1 CreateTime: 1599775774463 keysize: 1 valuesize: 9 sequence: -1
headerKeys: [] key: 5 payload: 33.440664
| offset: 2 CreateTime: 1599775774463 keysize: 8 valuesize: 9 sequence: -1
headerKeys: [] key: 60024247 payload: 9.1408728
| offset: 3 CreateTime: 1599775774463 keysize: 1 valuesize: 9 sequence: -1
headerKeys: [] key: 1 payload: 45.348946
| offset: 4 CreateTime: 1599775774464 keysize: 6 valuesize: 10 sequence: -1
headerKeys: [] key: 241795 payload: -63.786373
| offset: 5 CreateTime: 1599775774465 keysize: 8 valuesize: 9 sequence: -1
headerKeys: [] key: 53596698 payload: 69.431393
| offset: 6 CreateTime: 1599775774465 keysize: 8 valuesize: 9 sequence: -1
headerKeys: [] key: 33219463 payload: 88.307875
| offset: 7 CreateTime: 1599775774466 keysize: 1 valuesize: 9 sequence: -1
headerKeys: [] key: 0 payload: 39.940350
| offset: 8 CreateTime: 1599775774467 keysize: 5 valuesize: 9 sequence: -1
headerKeys: [] key: 78496 payload: 74.180098
| offset: 9 CreateTime: 1599775774468 keysize: 8 valuesize: 9 sequence: -1
headerKeys: [] key: 89866187 payload: 79.459314
```

There's a lot of information available here. You can clearly see the `key`, `payload` (value), offset, and timestamp for each record. Keep in mind that this data is from a demo topic that contains only 10 messages, so with a real topic, there will be substantially more data. Also note that in this example, the keys and values for the topic are strings. To run the `dump-log` tool with key or value types other than strings, you'll need to use the `--key-decoder-class` or `--value-decoder-class` flags.

Delete Records

Kafka stores records for topics on disk and retains that data even once consumers have read it. Records aren't stored in a single monolithic file, but are broken up into segments by partition, where the offset order is continuous across segments for the same topic partition. Because servers do not have infinite amounts of storage, Kafka provides settings to control how much data is retained, based on time and size:

- The time configuration setting controlling data retention,`log.retention.hours`, defaults to 168 hours (one week)
- The size configuration setting, `log.retention.bytes`, controls how large segments can grow before they are eligible for deletion

However, the default setting for `log.retention.bytes` is -1, which allows the log segment size to be unlimited. If you're not careful and haven't configured the retention size as well as the retention time, you could have a situation where you will run out of disk space. Remember, we *never* want to go into the filesystem and manually delete files. Instead, we want a controlled and supported way to delete records from a topic in order to free up space. Fortunately, Kafka ships with a tool that can delete data as required.

The `kafka-delete-records` command has two required parameters:

- `--bootstrap-server` indicates the broker(s) to connect to for bootstrapping
- `--offset-json-file` specifies a JSON file that contains the deletion settings

Here's an example of a JSON file used as the `--offset-json-file`:

```
{
   "partitions": [
                   {"topic": "example", "partition": 0, "offset": -1}
                  ],
                  "version":1
 }
```

As you can see, the file follows a simple format. It's an array of JSON objects, each of which has three properties:

1. **Topic**: the topic to delete from

2. **Partition**: the partition to delete from
3. **Offset**: the offset for the delete to start from (moving backward to lower offsets)

In this example, we're reusing the same topic that we used to demonstrate the dump-log tool, so this is a very simple JSON file. If you had more partitions or topics, you would simply expand on the JSON configuration file above.

Let's discuss how to choose the `offset` in this configuration file. Because the example topic contains only 10 records, we can easily calculate the offset we would use to start the deletion process. But in practice, you most likely won't immediately know what offset to use. Also bear in mind that the offset is not a message number; you can't just delete from "message 42." If you supply a `-1`, then the tool uses the offset of the *high watermark*, which means that it will delete all of the data currently in the topic. The high watermark is the highest available offset for consumption (the offset of the last successfully replicated message, plus one).

Now run the command:

```
kafka-delete-records --bootstrap-server <broker-host:port> \
                     --offset-json-file offsets.json
```

After running this command, you should see something like this on the console:

```
Executing records delete operation
Records delete operation completed:
partition: example-0  low_watermark: 10
```

The results of the command show that Kafka deleted all records from the topic partition `example-0`. The `low_watermark` value of 10 indicates the *lowest* offset available to consumers. Because there were only 10 records in the `example topic`, we know that the offsets ranged from 0 to 9, and no consumer can read those records again. For more background on how deletions are implemented, you can read KIP-107 and KIP-204.

The Power of Record Headers

Apache Kafka 0.11 introduced the concept of record headers. Record headers give you the ability to add metadata about the Kafka record, without adding any extra information to the key/value pair of the record itself.

Consider a case where you wanted to embed some information in a message—maybe an identifier for the system from which the data originated. Perhaps you want this for lineage and audit purposes and in order to facilitate routing of the data downstream.

Why not just append this information to the key? Then you could extract the part needed and you would be able to route data accordingly. Well, adding artificial data to the key poses two potential problems:

1. If you are using a compacted topic, adding information to the key would make the record incorrectly appear as unique. Thus, compaction would not function as intended.
2. If one particular system identifier dominates in the records sent, you could have significant key skew. Depending on how you are consuming from the partitions, the uneven distribution of keys could have an impact on processing by increasing latency.

These are two situations where you might want to use record headers. The original KIP proposing headers provides some additional use cases as well:

1. Automated routing of messages based on header information between clusters
2. "Magic" transaction IDs used by enterprise APM tools (e.g., AppDynamics or Dynatrace) in order to provide end-to-end transaction flow monitoring
3. Recording audit metadata with messages (for example, recording the client-id that produced a record)
4. A business payload that must be encrypted end to end and signed without tampering, while allowing ecosystem components to access the metadata so that they can achieve their tasks

Now that we've seen the case for using headers, let's walk through how you can add headers to your Kafka records.

Adding Headers to Kafka Records

Here's the Java code to add headers to a `ProducerRecord`:

```
ProducerRecord<String, String> producerRecord = new
ProducerRecord<>("bizops", "value");

producerRecord.headers().add("client-id",
"2334".getBytes(StandardCharsets.UTF_8));
producerRecord.headers().add("data-file", "incoming-
data.txt".getBytes(StandardCharsets.UTF_8));

// Details left out for clarity
producer.send(producerRecord);
```

This code performs the following steps:

1. Create an instance of the `ProducerRecord` class
2. Call the `ProducerRecord.headers()` method and add the key and value for the header
3. Add another header

There are a few things we need to point out in this code example. The header interface expects a `String` for the key and a byte array for the value. Even though you're providing a key, you can add many headers with the same key if needed. Duplicate keys will *not* overwrite previous entries with the same key.

There are also overloaded `ProducerRecord` constructors that accept an `Iterable<Header>`. You could create your own concrete class that implements the `Header` interface and passes in a collection that implements the `Iterable` interface. However, in practice, the simple method shown here should suffice.

Now that you know how to add headers, let's take a look at how you can access headers from the consumer side.

Retrieving Headers

Here is the Java code for accessing headers when consuming records:

```
//Details omitted for clarity

ConsumerRecords<String, String> consumerRecords =
consumer.poll(Duration.ofSeconds(1));

for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
    for (Header header : consumerRecord.headers()) {
        System.out.println("header key " + header.key() + "header value " +
new String(header.value()));
    }
}
```

This code performs the following steps:

1. Iterate over the `ConsumerRecords`
2. For each `ConsumerRecord`, iterate over the headers
3. Process each header

From the code above, you can see that you simply use the `ConsumerRecord.headers()` method to return the headers. In this example, we just print the headers out to the console. Once you have access to the headers, you can process them as needed. For reading headers from the command line, KIP-431 added support for optionally printing headers from the console consumer, and this was made available in Kafka 2.7.0.

You can also use kafkacat to view headers from the command line. Here's an example command:

```
kafkacat -b kafka-broker:9092 -t my_topic_name -C \
  -f '\nKey (%K bytes): %k
  Value (%S bytes): %s
  Timestamp: %T
  Partition: %p
  Offset: %o
  Headers: %h\n'
```

Conclusion

Now that you are familiar with record headers, the `min.insync.replicas` configuration setting, incremental cooperative rebalancing, sticky partitioning, and the most impactful command-line Kafka tools, you should be able to better reason about the behavior of your applications, as well as improve your development process.

Instead of spinning up a local Kafka cluster, you can use Confluent Cloud to try out any of the features in this white paper. When you sign up, you'll receive $200 of free usage each month for your first three months. You can also use the promo code `C50INTEG` to get an additional $50 of free usage (details).

To learn more, visit Confluent Developer and Kafka Tutorials.