CHAPTER 1

Meet Kafka

Every enterprise is powered by data. We take information in, analyze it, manipulate it, and create more as output. Every application creates data, whether it is log messages, metrics, user activity, outgoing messages, or something else. Every byte of data has a story to tell, something of importance that will inform the next thing to be done. In order to know what that is, we need to get the data from where it is created to where it can be analyzed. We see this every day on websites like Amazon, where our clicks on items of interest to us are turned into recommendations that are shown to us a little later.

The faster we can do this, the more agile and responsive our organizations can be. The less effort we spend on moving data around, the more we can focus on the core business at hand. This is why the pipeline is a critical component in the data-driven enterprise. How we move the data becomes nearly as important as the data itself.

> Any time scientists disagree, it's because we have insufficient data. Then we can agree on what kind of data to get; we get the data; and the data solves the problem. Either I'm right, or you're right, or we're both wrong. And we move on.
>
> —Neil deGrasse Tyson

Publish/Subscribe Messaging

Before discussing the specifics of Apache Kafka, it is important for us to understand the concept of publish/subscribe messaging and why it is important. *Publish/subscribe messaging* is a pattern that is characterized by the sender (publisher) of a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of messages. Pub/sub systems often have a broker, a central point where messages are published, to facilitate this.

How It Starts

Many use cases for publish/subscribe start out the same way: with a simple message queue or interprocess communication channel. For example, you create an application that needs to send monitoring information somewhere, so you write in a direct connection from your application to an app that displays your metrics on a dashboard, and push metrics over that connection, as seen in Figure 1-1.

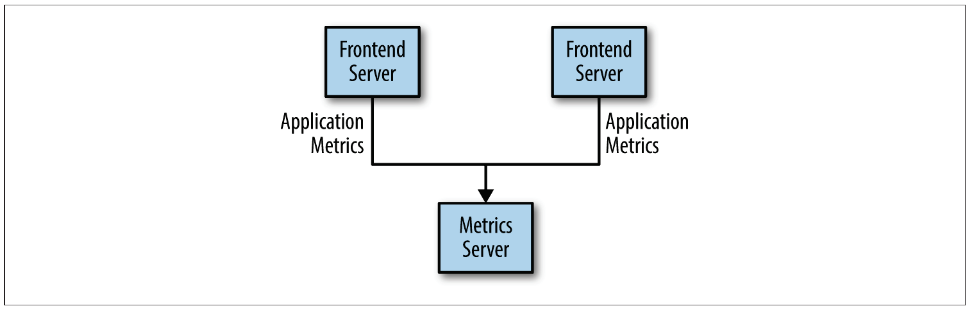

Figure 1-1. A single, direct metrics publisher

This is a simple solution to a simple problem that works when you are getting started with monitoring. Before long, you decide you would like to analyze your metrics over a longer term, and that doesn't work well in the dashboard. You start a new service that can receive metrics, store them, and analyze them. In order to support this, you modify your application to write metrics to both systems. By now you have three more applications that are generating metrics, and they all make the same connections to these two services. Your coworker thinks it would be a good idea to do active polling of the services for alerting as well, so you add a server on each of the applications to provide metrics on request. After a while, you have more applications that are using those servers to get individual metrics and use them for various purposes. This architecture can look much like Figure 1-2, with connections that are even harder to trace.

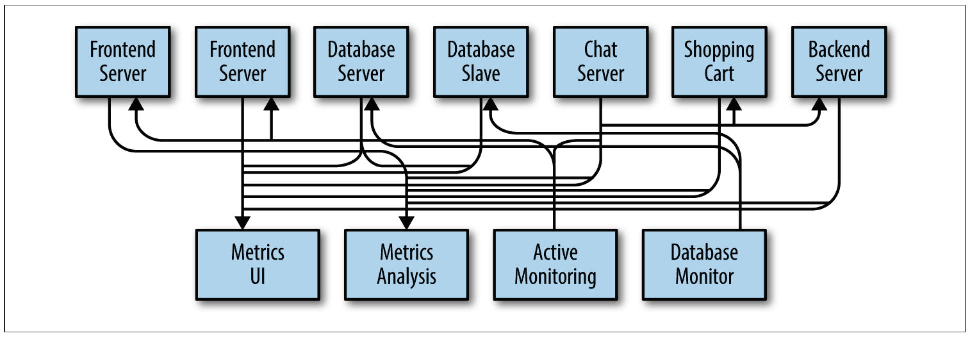

Figure 1-2. Many metrics publishers, using direct connections

The technical debt built up here is obvious, so you decide to pay some of it back. You set up a single application that receives metrics from all the applications out there, and provide a server to query those metrics for any system that needs them. This reduces the complexity of the architecture to something similar to Figure 1-3. Congratulations, you have built a publish-subscribe messaging system!

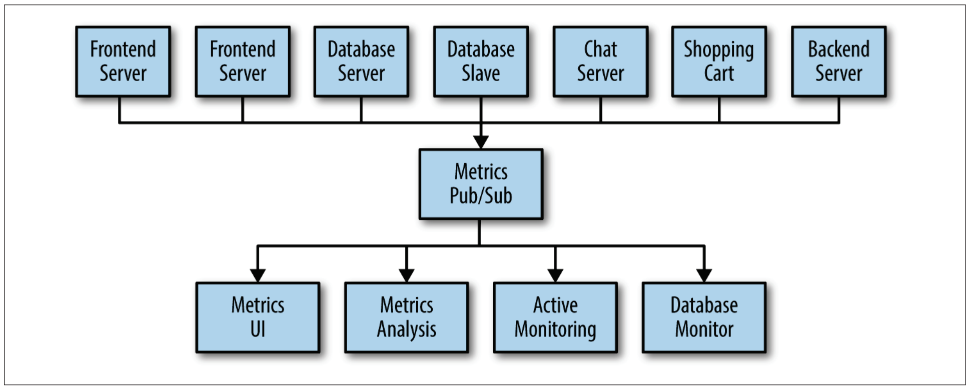

Figure 1-3. A metrics publish/subscribe system

Individual Queue Systems

At the same time that you have been waging this war with metrics, one of your coworkers has been doing similar work with log messages. Another has been working on tracking user behavior on the frontend website and providing that information to developers who are working on machine learning, as well as creating some reports for management. You have all followed a similar path of building out systems that decouple the publishers of the information from the subscribers to that information. Figure 1-4 shows such an infrastructure, with three separate pub/sub systems.

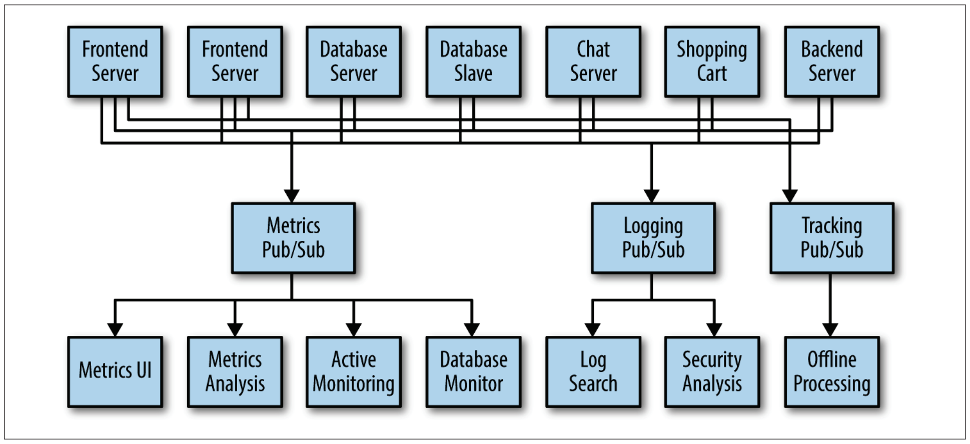

Figure 1-4. Multiple publish/subscribe systems

This is certainly a lot better than utilizing point-to-point connections (as in Figure 1-2), but there is a lot of duplication. Your company is maintaining multiple systems for queuing data, all of which have their own individual bugs and limitations. You also know that there will be more use cases for messaging coming soon. What you would like to have is a single centralized system that allows for publishing generic types of data, which will grow as your business grows.

Enter Kafka

Apache Kafka is a publish/subscribe messaging system designed to solve this problem. It is often described as a "distributed commit log" or more recently as a "distributing streaming platform." A filesystem or database commit log is designed to provide a durable record of all transactions so that they can be replayed to consistently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

Messages and Batches

The unit of data within Kafka is called a *message*. If you are approaching Kafka from a database background, you can think of this as similar to a *row* or a *record*. A message is simply an array of bytes as far as Kafka is concerned, so the data contained within it does not have a specific format or meaning to Kafka. A message can have an optional bit of metadata, which is referred to as a *key*. The key is also a byte array and, as with the message, has no specific meaning to Kafka. Keys are used when messages are to be written to partitions in a more controlled manner. The simplest such scheme is to generate a consistent hash of the key, and then select the partition number for that message by taking the result of the hash modulo, the total number of partitions in the topic. This assures that messages with the same key are always written to the same partition. Keys are discussed in more detail in Chapter 3.

For efficiency, messages are written into Kafka in batches. A *batch* is just a collection of messages, all of which are being produced to the same topic and partition. An individual roundtrip across the network for each message would result in excessive overhead, and collecting messages together into a batch reduces this. Of course, this is a tradeoff between latency and throughput: the larger the batches, the more messages that can be handled per unit of time, but the longer it takes an individual message to propagate. Batches are also typically compressed, providing more efficient data transfer and storage at the cost of some processing power.

Schemas

While messages are opaque byte arrays to Kafka itself, it is recommended that additional structure, or schema, be imposed on the message content so that it can be easily understood. There are many options available for message *schema*, depending on your application's individual needs. Simplistic systems, such as Javascript Object Notation (JSON) and Extensible Markup Language (XML), are easy to use and human-readable. However, they lack features such as robust type handling and compatibility between schema versions. Many Kafka developers favor the use of Apache Avro, which is a serialization framework originally developed for Hadoop. Avro provides a compact serialization format; schemas that are separate from the message payloads and that do not require code to be generated when they change; and strong data typing and schema evolution, with both backward and forward compatibility.

A consistent data format is important in Kafka, as it allows writing and reading messages to be decoupled. When these tasks are tightly coupled, applications that subscribe to messages must be updated to handle the new data format, in parallel with the old format. Only then can the applications that publish the messages be updated to utilize the new format. By using well-defined schemas and storing them in a common repository, the messages in Kafka can be understood without coordination. Schemas and serialization are covered in more detail in Chapter 3.

Topics and Partitions

Messages in Kafka are categorized into *topics*. The closest analogies for a topic are a database table or a folder in a filesystem. Topics are additionally broken down into a number of *partitions*. Going back to the "commit log" description, a partition is a single log. Messages are written to it in an append-only fashion, and are read in order from beginning to end. Note that as a topic typically has multiple partitions, there is no guarantee of message time-ordering across the entire topic, just within a single partition. Figure 1-5 shows a topic with four partitions, with writes being appended to the end of each one. Partitions are also the way that Kafka provides redundancy and scalability. Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers to provide performance far beyond the ability of a single server.

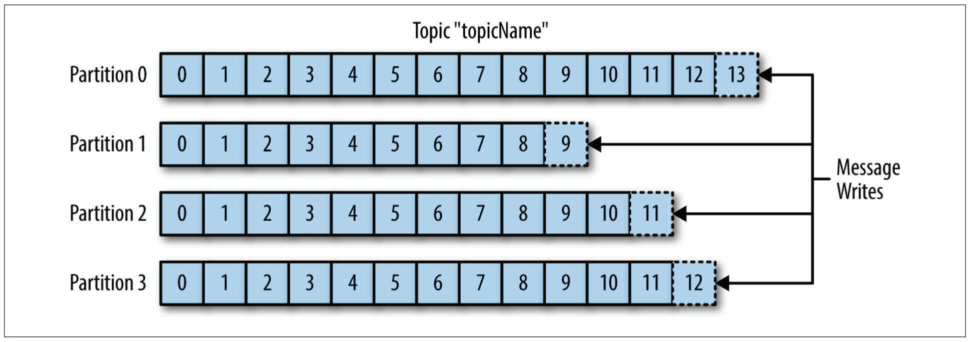

Figure 1-5. Representation of a topic with multiple partitions

The term *stream* is often used when discussing data within systems like Kafka. Most often, a stream is considered to be a single topic of data, regardless of the number of partitions. This represents a single stream of data moving from the producers to the consumers. This way of referring to messages is most common when discussing stream processing, which is when frameworks—some of which are Kafka Streams, Apache Samza, and Storm—operate on the messages in real time. This method of operation can be compared to the way offline frameworks, namely Hadoop, are designed to work on bulk data at a later time. An overview of stream processing is provided in Chapter 11.

Producers and Consumers

Kafka clients are users of the system, and there are two basic types: producers and consumers. There are also advanced client APIs—Kafka Connect API for data integration and Kafka Streams for stream processing. The advanced clients use producers and consumers as building blocks and provide higher-level functionality on top.

Producers create new messages. In other publish/subscribe systems, these may be called *publishers* or *writers*. In general, a message will be produced to a specific topic. By default, the producer does not care what partition a specific message is written to and will balance messages over all partitions of a topic evenly. In some cases, the producer will direct messages to specific partitions. This is typically done using the message key and a partitioner that will generate a hash of the key and map it to a specific partition. This assures that all messages produced with a given key will get written to the same partition. The producer could also use a custom partitioner that follows other business rules for mapping messages to partitions. Producers are covered in more detail in Chapter 3.

Consumers read messages. In other publish/subscribe systems, these clients may be called *subscribers* or *readers*. The consumer subscribes to one or more topics and reads the messages in the order in which they were produced. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages. The *offset* is another bit of metadata—an integer value that continually increases—that Kafka adds to each message as it is produced. Each message in a given partition has a unique offset. By storing the offset of the last consumed message for each partition, either in Zookeeper or in Kafka itself, a consumer can stop and restart without losing its place.

Consumers work as part of a *consumer group*, which is one or more consumers that work together to consume a topic. The group assures that each partition is only consumed by one member. In Figure 1-6, there are three consumers in a single group consuming a topic. Two of the consumers are working from one partition each, while the third consumer is working from two partitions. The mapping of a consumer to a partition is often called *ownership* of the partition by the consumer.

In this way, consumers can horizontally scale to consume topics with a large number of messages. Additionally, if a single consumer fails, the remaining members of the group will rebalance the partitions being consumed to take over for the missing member. Consumers and consumer groups are discussed in more detail in Chapter 4.

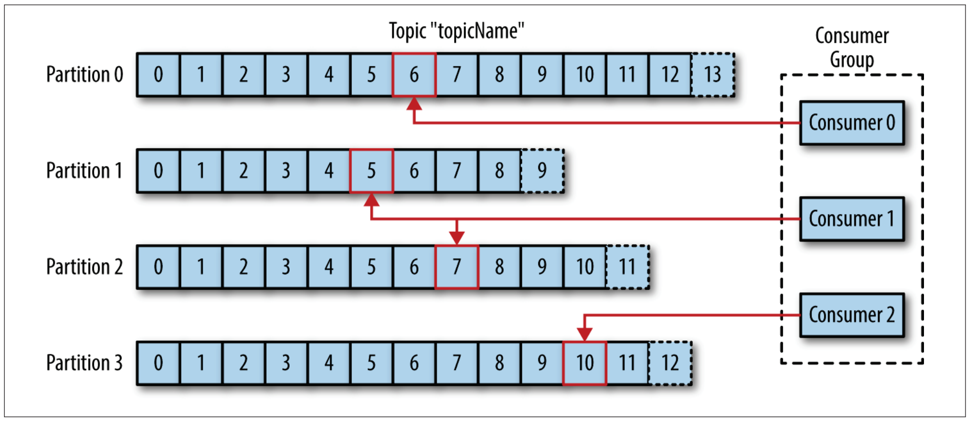

Figure 1-6. A consumer group reading from a topic

Brokers and Clusters

A single Kafka server is called a *broker*. The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been committed to disk. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second.

Kafka brokers are designed to operate as part of a *cluster*. Within a cluster of brokers, one broker will also function as the cluster *controller* (elected automatically from the live members of the cluster). The controller is responsible for administrative operations, including assigning partitions to brokers and monitoring for broker failures. A partition is owned by a single broker in the cluster, and that broker is called the *leader* of the partition. A partition may be assigned to multiple brokers, which will result in the partition being replicated (as seen in Figure 1-7). This provides redundancy of messages in the partition, such that another broker can take over leadership if there is a broker failure. However, all consumers and producers operating on that partition must connect to the leader. Cluster operations, including partition replication, are covered in detail in Chapter 6.

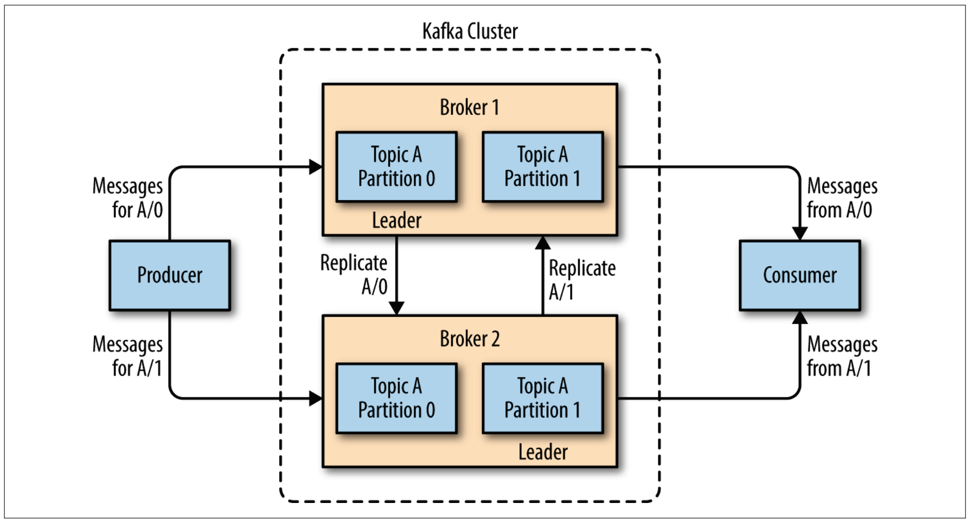

Figure 1-7. Replication of partitions in a cluster

A key feature of Apache Kafka is that of *retention*, which is the durable storage of messages for some period of time. Kafka brokers are configured with a default retention setting for topics, either retaining messages for some period of time (e.g., 7 days) or until the topic reaches a certain size in bytes (e.g., 1 GB). Once these limits are reached, messages are expired and deleted so that the retention configuration is a minimum amount of data available at any time. Individual topics can also be configured with their own retention settings so that messages are stored for only as long as they are useful. For example, a tracking topic might be retained for several days, whereas application metrics might be retained for only a few hours. Topics can also be configured as *log compacted*, which means that Kafka will retain only the last message produced with a specific key. This can be useful for changelog-type data, where only the last update is interesting.

Multiple Clusters

As Kafka deployments grow, it is often advantageous to have multiple clusters. There are several reasons why this can be useful:

- Segregation of types of data
- Isolation for security requirements
- Multiple datacenters (disaster recovery)

When working with multiple datacenters in particular, it is often required that messages be copied between them. In this way, online applications can have access to user activity at both sites. For example, if a user changes public information in their profile, that change will need to be visible regardless of the datacenter in which search results are displayed. Or, monitoring data can be collected from many sites into a single central location where the analysis and alerting systems are hosted. The replication mechanisms within the Kafka clusters are designed only to work within a single cluster, not between multiple clusters.

The Kafka project includes a tool called *MirrorMaker*, used for this purpose. At its core, MirrorMaker is simply a Kafka consumer and producer, linked together with a queue. Messages are consumed from one Kafka cluster and produced for another. Figure 1-8 shows an example of an architecture that uses MirrorMaker, aggregating messages from two local clusters into an aggregate cluster, and then copying that cluster to other datacenters. The simple nature of the application belies its power in creating sophisticated data pipelines, which will be detailed further in Chapter 7.

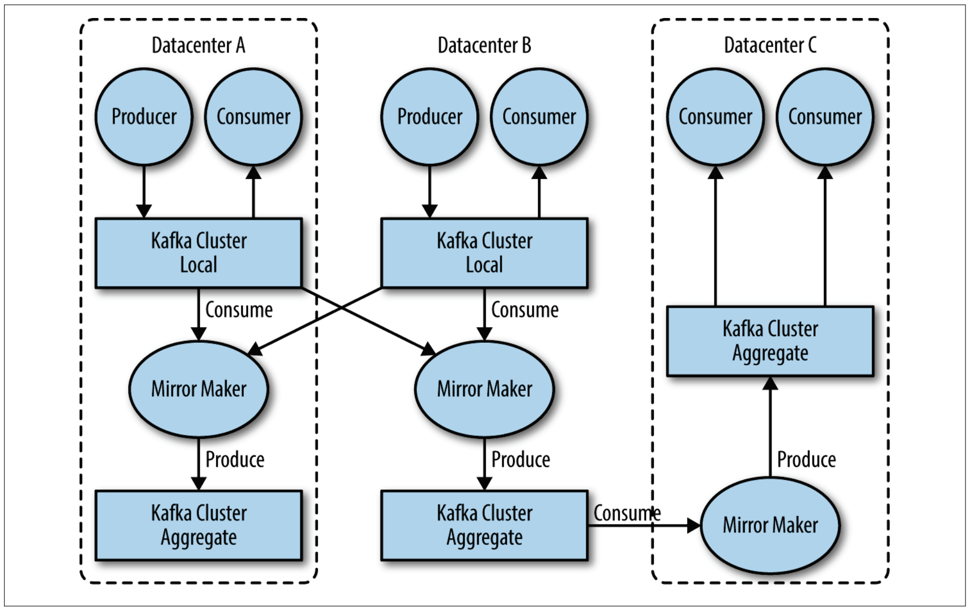

Figure 1-8. Multiple datacenter architecture

Why Kafka?

There are many choices for publish/subscribe messaging systems, so what makes Apache Kafka a good choice?

Multiple Producers

Kafka is able to seamlessly handle multiple producers, whether those clients are using many topics or the same topic. This makes the system ideal for aggregating data from many frontend systems and making it consistent. For example, a site that serves content to users via a number of microservices can have a single topic for page views that all services can write to using a common format. Consumer applications can then receive a single stream of page views for all applications on the site without having to coordinate consuming from multiple topics, one for each application.

Multiple Consumers

In addition to multiple producers, Kafka is designed for multiple consumers to read any single stream of messages without interfering with each other. This is in contrast to many queuing systems where once a message is consumed by one client, it is not available to any other. Multiple Kafka consumers can choose to operate as part of a group and share a stream, assuring that the entire group processes a given message only once.

Disk-Based Retention

Not only can Kafka handle multiple consumers, but durable message retention means that consumers do not always need to work in real time. Messages are committed to disk, and will be stored with configurable retention rules. These options can be selected on a per-topic basis, allowing for different streams of messages to have different amounts of retention depending on the consumer needs. Durable retention means that if a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data. It also means that maintenance can be performed on consumers, taking applications offline for a short period of time, with no concern about messages backing up on the producer or getting lost. Consumers can be stopped, and the messages will be retained in Kafka. This allows them to restart and pick up processing messages where they left off with no data loss.

Scalable

Kafka's flexible scalability makes it easy to handle any amount of data. Users can start with a single broker as a proof of concept, expand to a small development cluster of three brokers, and move into production with a larger cluster of tens or even hundreds of brokers that grows over time as the data scales up. Expansions can be performed while the cluster is online, with no impact on the availability of the system as a whole. This also means that a cluster of multiple brokers can handle the failure of an individual broker, and continue servicing clients. Clusters that need to tolerate more simultaneous failures can be configured with higher replication factors. Replication is discussed in more detail in Chapter 6.

High Performance

All of these features come together to make Apache Kafka a publish/subscribe messaging system with excellent performance under high load. Producers, consumers, and brokers can all be scaled out to handle very large message streams with ease. This can be done while still providing subsecond message latency from producing a message to availability to consumers.

The Data Ecosystem

Many applications participate in the environments we build for data processing. We have defined inputs in the form of applications that create data or otherwise introduce it to the system. We have defined outputs in the form of metrics, reports, and other data products. We create loops, with some components reading data from the system, transforming it using data from other sources, and then introducing it back into the data infrastructure to be used elsewhere. This is done for numerous types of data, with each having unique qualities of content, size, and usage.

Apache Kafka provides the circulatory system for the data ecosystem, as shown in Figure 1-9. It carries messages between the various members of the infrastructure, providing a consistent interface for all clients. When coupled with a system to provide message schemas, producers and consumers no longer require tight coupling or direct connections of any sort. Components can be added and removed as business cases are created and dissolved, and producers do not need to be concerned about who is using the data or the number of consuming applications.

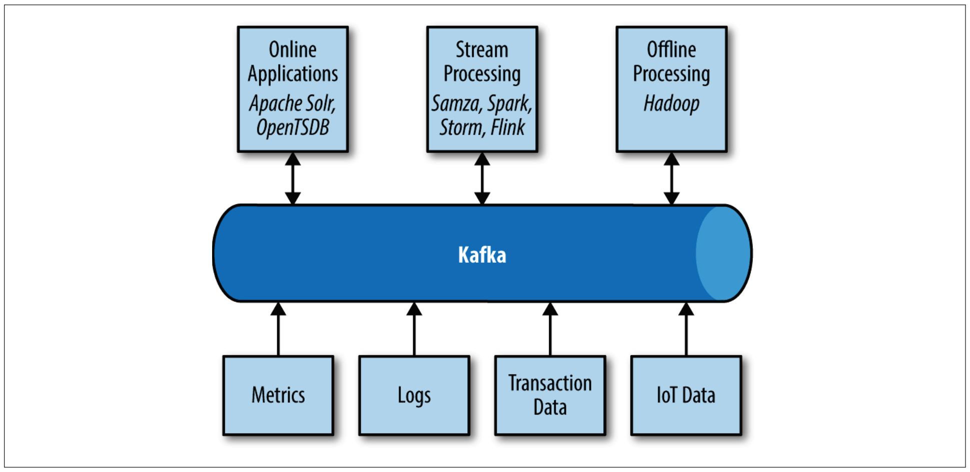

Figure 1-9. A big data ecosystem

Use Cases

Activity tracking

The original use case for Kafka, as it was designed at LinkedIn, is that of user activity tracking. A website's users interact with frontend applications, which generate messages regarding actions the user is taking. This can be passive information, such as page views and click tracking, or it can be more complex actions, such as information that a user adds to their profile. The messages are published to one or more topics, which are then consumed by applications on the backend. These applications may be generating reports, feeding machine learning systems, updating search results, or performing other operations that are necessary to provide a rich user experience.

Messaging

Kafka is also used for messaging, where applications need to send notifications (such as emails) to users. Those applications can produce messages without needing to be concerned about formatting or how the messages will actually be sent. A single application can then read all the messages to be sent and handle them consistently, including:

- Formatting the messages (also known as decorating) using a common look and feel
- Collecting multiple messages into a single notification to be sent
- Applying a user's preferences for how they want to receive messages

Using a single application for this avoids the need to duplicate functionality in multiple applications, as well as allows operations like aggregation which would not otherwise be possible.

Metrics and logging

Kafka is also ideal for collecting application and system metrics and logs. This is a use case in which the ability to have multiple applications producing the same type of message shines. Applications publish metrics on a regular basis to a Kafka topic, and those metrics can be consumed by systems for monitoring and alerting. They can also be used in an offline system like Hadoop to perform longer-term analysis, such as growth projections. Log messages can be published in the same way, and can be routed to dedicated log search systems like Elastisearch or security analysis applications. Another added benefit of Kafka is that when the destination system needs to change (e.g., it's time to update the log storage system), there is no need to alter the frontend applications or the means of aggregation.

Commit log

Since Kafka is based on the concept of a commit log, database changes can be published to Kafka and applications can easily monitor this stream to receive live updates as they happen. This changelog stream can also be used for replicating database updates to a remote system, or for consolidating changes from multiple applications into a single database view. Durable retention is useful here for providing a buffer for the changelog, meaning it can be replayed in the event of a failure of the consuming applications. Alternately, log-compacted topics can be used to provide longer retention by only retaining a single change per key.

Stream processing

Another area that provides numerous types of applications is stream processing. While almost all usage of Kafka can be thought of as stream processing, the term is typically used to refer to applications that provide similar functionality to map/reduce processing in Hadoop. Hadoop usually relies on aggregation of data over a long time frame, either hours or days. Stream processing operates on data in real time, as quickly as messages are produced. Stream frameworks allow users to write small applications to operate on Kafka messages, performing tasks such as counting metrics, partitioning messages for efficient processing by other applications, or transforming messages using data from multiple sources. Stream processing is covered in Chapter 11.

Kafka's Origin

Kafka was created to address the data pipeline problem at LinkedIn. It was designed to provide a high-performance messaging system that can handle many types of data and provide clean, structured data about user activity and system metrics in real time.

> Data really powers everything that we do.
>
> —Jeff Weiner, *CEO of LinkedIn*

LinkedIn's Problem

Similar to the example described at the beginning of this chapter, LinkedIn had a system for collecting system and application metrics that used custom collectors and open source tools for storing and presenting data internally. In addition to traditional metrics, such as CPU usage and application performance, there was a sophisticated request-tracing feature that used the monitoring system and could provide introspection into how a single user request propagated through internal applications. The monitoring system had many faults, however. This included metrics collection based on polling, large intervals between metrics, and no ability for application owners to manage their own metrics. The system was high-touch, requiring human intervention for most simple tasks, and inconsistent, with differing metric names for the same measurement across different systems.

At the same time, there was a system created for tracking user activity information. This was an HTTP service that frontend servers would connect to periodically and publish a batch of messages (in XML format) to the HTTP service. These batches were then moved to offline processing, which is where the files were parsed and collated. This system had many faults. The XML formatting was inconsistent, and parsing it was computationally expensive. Changing the type of user activity that was tracked required a significant amount of coordinated work between frontends and offline processing. Even then, the system would break constantly due to changing schemas. Tracking was built on hourly batching, so it could not be used in real-time.

Monitoring and user-activity tracking could not use the same backend service. The monitoring service was too clunky, the data format was not oriented for activity tracking, and the polling model for monitoring was not compatible with the push model for tracking. At the same time, the tracking service was too fragile to use for metrics, and the batch-oriented processing was not the right model for real-time monitoring and alerting. However, the monitoring and tracking data shared many traits, and correlation of the information (such as how specific types of user activity affected application performance) was highly desirable. A drop in specific types of user activity could indicate problems with the application that serviced it, but hours of delay in processing activity batches meant a slow response to these types of issues.

At first, existing off-the-shelf open source solutions were thoroughly investigated to find a new system that would provide real-time access to the data and scale out to handle the amount of message traffic needed. Prototype systems were set up using ActiveMQ, but at the time it could not handle the scale. It was also a fragile solution for the way LinkedIn needed to use it, discovering many flaws in ActiveMQ that would cause the brokers to pause. This would back up connections to clients and interfere with the ability of the applications to serve requests to users. The decision was made to move forward with a custom infrastructure for the data pipeline.

The Birth of Kafka

The development team at LinkedIn was led by Jay Kreps, a principal software engineer who was previously responsible for the development and open source release of Voldemort, a distributed key-value storage system. The initial team also included Neha Narkhede and, later, Jun Rao. Together, they set out to create a messaging system that could meet the needs of both the monitoring and tracking systems, and scale for the future. The primary goals were to:

- Decouple producers and consumers by using a push-pull model
- Provide persistence for message data within the messaging system to allow multiple consumers
- Optimize for high throughput of messages
- Allow for horizontal scaling of the system to grow as the data streams grew

The result was a publish/subscribe messaging system that had an interface typical of messaging systems but a storage layer more like a log-aggregation system. Combined with the adoption of Apache Avro for message serialization, Kafka was effective for handling both metrics and user-activity tracking at a scale of billions of messages per day. The scalability of Kafka has helped LinkedIn's usage grow in excess of one trillion messages produced (as of August 2015) and over a petabyte of data consumed daily.

Open Source

Kafka was released as an open source project on GitHub in late 2010. As it started to gain attention in the open source community, it was proposed and accepted as an Apache Software Foundation incubator project in July of 2011. Apache Kafka graduated from the incubator in October of 2012. Since then, it has continuously been worked on and has found a robust community of contributors and committers outside of LinkedIn. Kafka is now used in some of the largest data pipelines in the world. In the fall of 2014, Jay Kreps, Neha Narkhede, and Jun Rao left LinkedIn to found Confluent, a company centered around providing development, enterprise support, and training for Apache Kafka. The two companies, along with ever-growing contributions from others in the open source community, continue to develop and maintain Kafka, making it the first choice for big data pipelines.

The Name

People often ask how Kafka got its name and if it has anything to do with the application itself. Jay Kreps offered the following insight:

> I thought that since Kafka was a system optimized for writing, using a writer's name would make sense. I had taken a lot of lit classes in college and liked Franz Kafka. Plus the name sounded cool for an open source project.
>
> So basically there is not much of a relationship.

Getting Started with Kafka

Now that we know all about Kafka and its history, we can set it up and build our own data pipeline. In the next chapter, we will explore installing and configuring Kafka. We will also cover selecting the right hardware to run Kafka on, and some things to keep in mind when moving to production operations.

CHAPTER 3

Kafka Producers: Writing Messages to Kafka

Whether you use Kafka as a queue, message bus, or data storage platform, you will always use Kafka by writing a producer that writes data to Kafka, a consumer that reads data from Kafka, or an application that serves both roles.

For example, in a credit card transaction processing system, there will be a client application, perhaps an online store, responsible for sending each transaction to Kafka immediately when a payment is made. Another application is responsible for immediately checking this transaction against a rules engine and determining whether the transaction is approved or denied. The approve/deny response can then be written back to Kafka and the response can propagate back to the online store where the transaction was initiated. A third application can read both transactions and the approval status from Kafka and store them in a database where analysts can later review the decisions and perhaps improve the rules engine.

Apache Kafka ships with built-in client APIs that developers can use when developing applications that interact with Kafka.

In this chapter we will learn how to use the Kafka producer, starting with an overview of its design and components. We will show how to create `KafkaProducer` and `ProducerRecord` objects, how to send records to Kafka, and how to handle the errors that Kafka may return. We'll then review the most important configuration options used to control the producer behavior. We'll conclude with a deeper look at how to use different partitioning methods and serializers, and how to write your own serializers and partitioners.

In Chapter 4 we will look at Kafka's consumer client and reading data from Kafka.

Third-Party Clients

In addition to the built-in clients, Kafka has a binary wire protocol. This means that it is possible for applications to read messages from Kafka or write messages to Kafka simply by sending the correct byte sequences to Kafka's network port. There are multiple clients that implement Kafka's wire protocol in different programming languages, giving simple ways to use Kafka not just in Java applications but also in languages like C++, Python, Go, and many more. Those clients are not part of Apache Kafka project, but a list of non-Java clients is maintained in the project wiki. The wire protocol and the external clients are outside the scope of the chapter.

Producer Overview

There are many reasons an application might need to write messages to Kafka: recording user activities for auditing or analysis, recording metrics, storing log messages, recording information from smart appliances, communicating asynchronously with other applications, buffering information before writing to a database, and much more.

Those diverse use cases also imply diverse requirements: is every message critical, or can we tolerate loss of messages? Are we OK with accidentally duplicating messages? Are there any strict latency or throughput requirements we need to support?

In the credit card transaction processing example we introduced earlier, we can see that it is critical to never lose a single message nor duplicate any messages. Latency should be low but latencies up to 500ms can be tolerated, and throughput should be very high—we expect to process up to a million messages a second.

A different use case might be to store click information from a website. In that case, some message loss or a few duplicates can be tolerated; latency can be high as long as there is no impact on the user experience. In other words, we don't mind if it takes a few seconds for the message to arrive at Kafka, as long as the next page loads immediately after the user clicked on a link. Throughput will depend on the level of activity we anticipate on our website.

The different requirements will influence the way you use the producer API to write messages to Kafka and the configuration you use.

While the producer APIs are very simple, there is a bit more that goes on under the hood of the producer when we send data. Figure 3-1 shows the main steps involved in sending data to Kafka.

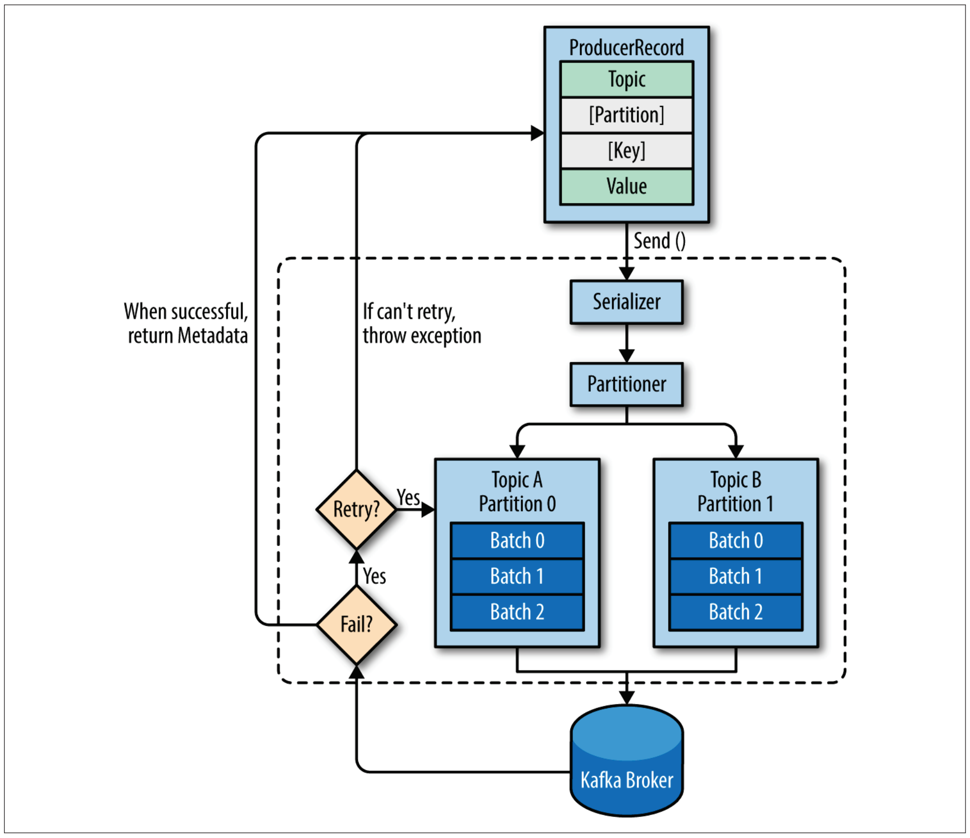

Figure 3-1. High-level overview of Kafka producer components

We start producing messages to Kafka by creating a `ProducerRecord`, which must include the topic we want to send the record to and a value. Optionally, we can also specify a key and/or a partition. Once we send the `ProducerRecord`, the first thing the producer will do is serialize the key and value objects to ByteArrays so they can be sent over the network.

Next, the data is sent to a partitioner. If we specified a partition in the `ProducerRecord`, the partitioner doesn't do anything and simply returns the partition we specified. If we didn't, the partitioner will choose a partition for us, usually based on the `ProducerRecord` key. Once a partition is selected, the producer knows which topic and partition the record will go to. It then adds the record to a batch of records that will also be sent to the same topic and partition. A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers.

When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a `RecordMetadata` object with the topic, partition, and the offset of the record within the partition. If the broker failed to write the messages, it will return an error. When the producer receives an error, it may retry sending the message a few more times before giving up and returning an error.

Constructing a Kafka Producer

The first step in writing messages to Kafka is to create a producer object with the properties you want to pass to the producer. A Kafka producer has three mandatory properties:

`bootstrap.servers`
: List of `host:port` pairs of brokers that the producer will use to establish initial connection to the Kafka cluster. This list doesn't need to include all brokers, since the producer will get more information after the initial connection. But it is recommended to include at least two, so in case one broker goes down, the producer will still be able to connect to the cluster.

`key.serializer`
: Name of a class that will be used to serialize the keys of the records we will produce to Kafka. Kafka brokers expect byte arrays as keys and values of messages. However, the producer interface allows, using parameterized types, any Java object to be sent as a key and value. This makes for very readable code, but it also means that the producer has to know how to convert these objects to byte arrays. `key.serializer` should be set to a name of a class that implements the `org.apache.kafka.common.serialization.Serializer` interface. The producer will use this class to serialize the key object to a byte array. The Kafka client package includes `ByteArraySerializer` (which doesn't do much), `StringSerializer`, and `IntegerSerializer`, so if you use common types, there is no need to implement your own serializers. Setting `key.serializer` is required even if you intend to send only values.

`value.serializer`
: Name of a class that will be used to serialize the values of the records we will produce to Kafka. The same way you set `key.serializer` to a name of a class that will serialize the message key object to a byte array, you set `value.serializer` to a class that will serialize the message value object.

The following code snippet shows how to create a new producer by setting just the mandatory parameters and using defaults for everything else:

```
private Properties kafkaProps = new Properties(); ❶
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
      "org.apache.kafka.common.serialization.StringSerializer"); ❷
    kafkaProps.put("value.serializer",
      "org.apache.kafka.common.serialization.StringSerializer");

    producer = new KafkaProducer<String, String>(kafkaProps); ❸
```

❶ We start with a `Properties` object.

❷ Since we plan on using strings for message key and value, we use the built-in `StringSerializer`.

❸ Here we create a new producer by setting the appropriate key and value types and passing the `Properties` object.

With such a simple interface, it is clear that most of the control over producer behavior is done by setting the correct configuration properties. Apache Kafka documentation covers all the configuration options, and we will go over the important ones later in this chapter.

Once we instantiate a producer, it is time to start sending messages. There are three primary methods of sending messages:

Fire-and-forget
: We send a message to the server and don't really care if it arrives succesfully or not. Most of the time, it will arrive successfully, since Kafka is highly available and the producer will retry sending messages automatically. However, some messages will get lost using this method.

Synchronous send
: We send a message, the `send()` method returns a Future object, and we use `get()` to wait on the future and see if the `send()` was successful or not.

Asynchronous send
: We call the `send()` method with a callback function, which gets triggered when it receives a response from the Kafka broker.

In the examples that follow, we will see how to send messages using these methods and how to handle the different types of errors that might occur.

While all the examples in this chapter are single threaded, a producer object can be used by multiple threads to send messages. You will probably want to start with one producer and one thread. If you need better throughput, you can add more threads that use the same producer. Once this ceases to increase throughput, you can add more producers to the application to achieve even higher throughput.

Sending a Message to Kafka

The simplest way to send a message is as follows:

```
ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision Products",
          "France"); ❶
try {
  producer.send(record); ❷
} catch (Exception e) {
        e.printStackTrace(); ❸
}
```

❶ The producer accepts `ProducerRecord` objects, so we start by creating one. `ProducerRecord` has multiple constructors, which we will discuss later. Here we use one that requires the name of the topic we are sending data to, which is always a string, and the key and value we are sending to Kafka, which in this case are also strings. The types of the key and value must match our `serializer` and `producer` objects.

❷ We use the producer object `send()` method to send the `ProducerRecord`. As we've seen in the producer architecture diagram in Figure 3-1, the message will be placed in a buffer and will be sent to the broker in a separate thread. The `send()` method returns a Java Future object with `RecordMetadata`, but since we simply ignore the returned value, we have no way of knowing whether the message was sent successfully or not. This method of sending messages can be used when dropping a message silently is acceptable. This is not typically the case in production applications.

❸ While we ignore errors that may occur while sending messages to Kafka brokers or in the brokers themselves, we may still get an exception if the producer encountered errors before sending the message to Kafka. Those can be a `SerializationException` when it fails to serialize the message, a `BufferExhaustedException` or `TimeoutException` if the buffer is full, or an InterruptException if the sending thread was interrupted.

Sending a Message Synchronously

The simplest way to send a message synchronously is as follows:

```
ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision Products", "France");
try {
        producer.send(record).get(); ❶
} catch (Exception e) {
```

```
            e.printStackTrace(); ❷
}
```

❶ Here, we are using `Future.get()` to wait for a reply from Kafka. This method will throw an exception if the record is not sent successfully to Kafka. If there were no errors, we will get a `RecordMetadata` object that we can use to retrieve the offset the message was written to.

❷ If there were any errors before sending data to Kafka, while sending, if the Kafka brokers returned a nonretriable exceptions or if we exhausted the available retries, we will encounter an exception. In this case, we just print any exception we ran into.

`KafkaProducer` has two types of errors. *Retriable* errors are those that can be resolved by sending the message again. For example, a connection error can be resolved because the connection may get reestablished. A "no leader" error can be resolved when a new leader is elected for the partition. `KafkaProducer` can be configured to retry those errors automatically, so the application code will get retriable exceptions only when the number of retries was exhausted and the error was not resolved. Some errors will not be resolved by retrying. For example, "message size too large." In those cases, `KafkaProducer` will not attempt a retry and will return the exception immediately.

Sending a Message Asynchronously

Suppose the network roundtrip time between our application and the Kafka cluster is 10ms. If we wait for a reply after sending each message, sending 100 messages will take around 1 second. On the other hand, if we just send all our messages and not wait for any replies, then sending 100 messages will barely take any time at all. In most cases, we really don't need a reply—Kafka sends back the topic, partition, and offset of the record after it was written, which is usually not required by the sending app. On the other hand, we do need to know when we failed to send a message completely so we can throw an exception, log an error, or perhaps write the message to an "errors" file for later analysis.

In order to send messages asynchronously and still handle error scenarios, the producer supports adding a callback when sending a record. Here is an example of how we use a callback:

```
private class DemoProducerCallback implements Callback { ❶
        @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
     if (e != null) {
         e.printStackTrace(); ❷
        }
    }
}

ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");
❸
producer.send(record, new DemoProducerCallback()); ❹
```

❶ To use callbacks, you need a class that implements the `org.apache.kafka.clients.producer.Callback` interface, which has a single function—`onCompletion()`.

❷ If Kafka returned an error, `onCompletion()` will have a nonnull exception. Here we "handle" it by printing, but production code will probably have more robust error handling functions.

❸ The records are the same as before.

❹ And we pass a `Callback` object along when sending the record.

Configuring Producers

So far we've seen very few configuration parameters for the producers—just the mandatory `bootstrap.servers` URI and serializers.

The producer has a large number of configuration parameters; most are documented in Apache Kafka documentation and many have reasonable defaults so there is no reason to tinker with every single parameter. However, some of the parameters have a significant impact on memory use, performance, and reliability of the producers. We will review those here.

acks

The `acks` parameter controls how many partition replicas must receive the record before the producer can consider the write successful. This option has a significant impact on how likely messages are to be lost. There are three allowed values for the `acks` parameter:

- If `acks=0`, the producer will not wait for a reply from the broker before assuming the message was sent successfully. This means that if something went wrong and the broker did not receive the message, the producer will not know about it and the message will be lost. However, because the producer is not waiting for any response from the server, it can send messages as fast as the network will support, so this setting can be used to achieve very high throughput.

- If `acks=1`, the producer will receive a success response from the broker the moment the leader replica received the message. If the message can't be written to the leader (e.g., if the leader crashed and a new leader was not elected yet), the producer will receive an error response and can retry sending the message, avoiding potential loss of data. The message can still get lost if the leader crashes and a replica without this message gets elected as the new leader (via unclean leader election). In this case, throughput depends on whether we send messages synchronously or asynchronously. If our client code waits for a reply from the server (by calling the `get()` method of the `Future` object returned when sending a message) it will obviously increase latency significantly (at least by a network roundtrip). If the client uses callbacks, latency will be hidden, but throughput will be limited by the number of in-flight messages (i.e., how many messages the producer will send before receiving replies from the server).
- If `acks=all`, the producer will receive a success response from the broker once all in-sync replicas received the message. This is the safest mode since you can make sure more than one broker has the message and that the message will survive even in the case of crash (more information on this in Chapter 5). However, the latency we discussed in the `acks=1` case will be even higher, since we will be waiting for more than just one broker to receive the message.

buffer.memory

This sets the amount of memory the producer will use to buffer messages waiting to be sent to brokers. If messages are sent by the application faster than they can be delivered to the server, the producer may run out of space and additional `send()` calls will either block or throw an exception, based on the `block.on.buffer.full` parameter (replaced with `max.block.ms` in release 0.9.0.0, which allows blocking for a certain time and then throwing an exception).

compression.type

By default, messages are sent uncompressed. This parameter can be set to `snappy`, `gzip`, or `lz4`, in which case the corresponding compression algorithms will be used to compress the data before sending it to the brokers. Snappy compression was invented by Google to provide decent compression ratios with low CPU overhead and good performance, so it is recommended in cases where both performance and bandwidth are a concern. Gzip compression will typically use more CPU and time but result in better compression ratios, so it recommended in cases where network bandwidth is more restricted. By enabling compression, you reduce network utilization and storage, which is often a bottleneck when sending messages to Kafka.

retries

When the producer receives an error message from the server, the error could be transient (e.g., a lack of leader for a partition). In this case, the value of the `retries` parameter will control how many times the producer will retry sending the message before giving up and notifying the client of an issue. By default, the producer will wait 100ms between retries, but you can control this using the `retry.backoff.ms` parameter. We recommend testing how long it takes to recover from a crashed broker (i.e., how long until all partitions get new leaders) and setting the number of retries and delay between them such that the total amount of time spent retrying will be longer than the time it takes the Kafka cluster to recover from the crash—otherwise, the producer will give up too soon. Not all errors will be retried by the producer. Some errors are not transient and will not cause retries (e.g., "message too large" error). In general, because the producer handles retries for you, there is no point in handling retries within your own application logic. You will want to focus your efforts on handling nonretriable errors or cases where retry attempts were exhausted.

batch.size

When multiple records are sent to the same partition, the producer will batch them together. This parameter controls the amount of memory in bytes (not messages!) that will be used for each batch. When the batch is full, all the messages in the batch will be sent. However, this does not mean that the producer will wait for the batch to become full. The producer will send half-full batches and even batches with just a single message in them. Therefore, setting the batch size too large will not cause delays in sending messages; it will just use more memory for the batches. Setting the batch size too small will add some overhead because the producer will need to send messages more frequently.

linger.ms

`linger.ms` controls the amount of time to wait for additional messages before sending the current batch. `KafkaProducer` sends a batch of messages either when the current batch is full or when the `linger.ms` limit is reached. By default, the producer will send messages as soon as there is a sender thread available to send them, even if there's just one message in the batch. By setting `linger.ms` higher than 0, we instruct the producer to wait a few milliseconds to add additional messages to the batch before sending it to the brokers. This increases latency but also increases throughput (because we send more messages at once, there is less overhead per message).

client.id

This can be any string, and will be used by the brokers to identify messages sent from the client. It is used in logging and metrics, and for quotas.

max.in.flight.requests.per.connection

This controls how many messages the producer will send to the server without receiving responses. Setting this high can increase memory usage while improving throughput, but setting it too high can reduce throughput as batching becomes less efficient. Setting this to 1 will guarantee that messages will be written to the broker in the order in which they were sent, even when retries occur.

timeout.ms, request.timeout.ms, and metadata.fetch.timeout.ms

These parameters control how long the producer will wait for a reply from the server when sending data (`request.timeout.ms`) and when requesting metadata such as the current leaders for the partitions we are writing to (`metadata.fetch.timeout.ms`). If the timeout is reached without reply, the producer will either retry sending or respond with an error (either through exception or the send callback). `timeout.ms` controls the time the broker will wait for in-sync replicas to acknowledge the message in order to meet the `acks` configuration—the broker will return an error if the time elapses without the necessary acknowledgments.

max.block.ms

This parameter controls how long the producer will block when calling `send()` and when explicitly requesting metadata via `partitionsFor()`. Those methods block when the producer's send buffer is full or when metadata is not available. When `max.block.ms` is reached, a timeout exception is thrown.

max.request.size

This setting controls the size of a produce request sent by the producer. It caps both the size of the largest message that can be sent and the number of messages that the producer can send in one request. For example, with a default maximum request size of 1 MB, the largest message you can send is 1 MB or the producer can batch 1,000 messages of size 1 K each into one request. In addition, the broker has its own limit on the size of the largest message it will accept (`message.max.bytes`). It is usually a good idea to have these configurations match, so the producer will not attempt to send messages of a size that will be rejected by the broker.

receive.buffer.bytes and send.buffer.bytes

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to -1, the OS defaults will be used. It is a good idea to increase those when producers or consumers communicate with brokers in a different datacenter because those network links typically have higher latency and lower bandwidth.

Ordering Guarantees

Apache Kafka preserves the order of messages within a partition. This means that if messages were sent from the producer in a specific order, the broker will write them to a partition in that order and all consumers will read them in that order. For some use cases, order is very important. There is a big difference between depositing $100 in an account and later withdrawing it, and the other way around! However, some use cases are less sensitive.

Setting the `retries` parameter to nonzero and the `max.in.flights.requests.per.session` to more than one means that it is possible that the broker will fail to write the first batch of messages, succeed to write the second (which was already in-flight), and then retry the first batch and succeed, thereby reversing the order.

Usually, setting the number of retries to zero is not an option in a reliable system, so if guaranteeing order is critical, we recommend setting `in.flight.requests.per.session=1` to make sure that while a batch of messages is retrying, additional messages will not be sent (because this has the potential to reverse the correct order). This will severely limit the throughput of the producer, so only use this when order is important.

Serializers

As seen in previous examples, producer configuration includes mandatory serializers. We've seen how to use the default String serializer. Kafka also includes serializers for integers and `ByteArrays`, but this does not cover most use cases. Eventually, you will want to be able to serialize more generic records.

We will start by showing how to write your own serializer and then introduce the Avro serializer as a recommended alternative.

Custom Serializers

When the object you need to send to Kafka is not a simple string or integer, you have a choice of either using a generic serialization library like Avro, Thrift, or Protobuf to create records, or creating a custom serialization for objects you are already using. We highly recommend using a generic serialization library. In order to understand how the serializers work and why it is a good idea to use a serialization library, let's see what it takes to write your own custom serializer.

Suppose that instead of recording just the customer name, you create a simple class to represent customers:

```
public class Customer {
        private int customerID;
        private String customerName;

        public Customer(int ID, String name) {
                this.customerID = ID;
                this.customerName = name;
        }

  public int getID() {
    return customerID;
  }

  public String getName() {
   return customerName;
  }
}
```

Now suppose we want to create a custom serializer for this class. It will look something like this:

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerSerializer implements Serializer<Customer> {

        @Override
  public void configure(Map configs, boolean isKey) {
   // nothing to configure
  }

  @Override
  /**
  We are serializing Customer as:
  4 byte int representing customerId
  4 byte int representing length of customerName in UTF-8 bytes (0 if name is
Null)
  N bytes representing customerName in UTF-8
  */
  public byte[] serialize(String topic, Customer data) {
   try {
                  byte[] serializedName;
                  int stringSize;
      if (data == null)
        return null;
      else {
                                if (data.getName() != null) {
         serializeName = data.getName().getBytes("UTF-8");
         stringSize = serializedName.length;
                                } else {
                                        serializedName = new byte[0];
                                        stringSize = 0;
                                }
                  }

      ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + stringSize);
      buffer.putInt(data.getID());
      buffer.putInt(stringSize);
      buffer.put(serializedName);

      return buffer.array();
          } catch (Exception e) {
     throw new SerializationException("Error when serializing Customer to
byte[] " + e);
     }
  }

  @Override
  public void close() {
         // nothing to close
  }
}
```

Configuring a producer with this `CustomerSerializer` will allow you to define `ProducerRecord<String, Customer>`, and send `Customer` data and pass `Customer` objects directly to the producer. This example is pretty simple, but you can see how fragile the code is. If we ever have too many customers, for example, and need to change customerID to `Long`, or if we ever decide to add a `startDate` field to `Customer`, we will have a serious issue in maintaining compatibility between old and new messages. Debugging compatibility issues between different versions of serializers and deserializers is fairly challenging—you need to compare arrays of raw bytes. To make matters even worse, if multiple teams in the same company end up writing `Customer` data to Kafka, they will all need to use the same serializers and modify the code at the exact same time.

For these reasons, we recommend using existing serializers and deserializers such as JSON, Apache Avro, Thrift, or Protobuf. In the following section we will describe Apache Avro and then show how to serialize Avro records and send them to Kafka.

Serializing Using Apache Avro

Apache Avro is a language-neutral data serialization format. The project was created by Doug Cutting to provide a way to share data files with a large audience.

Avro data is described in a language-independent schema. The schema is usually described in JSON and the serialization is usually to binary files, although serializing to JSON is also supported. Avro assumes that the schema is present when reading and writing files, usually by embedding the schema in the files themselves.

One of the most interesting features of Avro, and what makes it a good fit for use in a messaging system like Kafka, is that when the application that is writing messages switches to a new schema, the applications reading the data can continue processing messages without requiring any change or update.

Suppose the original schema was:

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string""},
     {"name": "faxNumber", "type": ["null", "string"], "default": "null"} ❶
 ]
}
```

❶ id and name fields are mandatory, while fax number is optional and defaults to null.

We used this schema for a few months and generated a few terabytes of data in this format. Now suppose that we decide that in the new version, we will upgrade to the twenty-first century and will no longer include a fax number field and will instead use an email field.

The new schema would be:

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string"},
     {"name": "email", "type": ["null", "string"], "default": "null"}
 ]
}
```

Now, after upgrading to the new version, old records will contain "faxNumber" and new records will contain "email." In many organizations, upgrades are done slowly and over many months. So we need to consider how preupgrade applications that still use the fax numbers and postupgrade applications that use email will be able to handle all the events in Kafka.

The reading application will contain calls to methods similar to `getName()`, `getId()`, and `getFaxNumber`. If it encounters a message written with the new schema, `get`

`Name()` and `getId()` will continue working with no modification, but `getFaxNumber()` will return `null` because the message will not contain a fax number.

Now suppose we upgrade our reading application and it no longer has the `getFaxNumber()` method but rather `getEmail()`. If it encounters a message written with the old schema, `getEmail()` will return `null` because the older messages do not contain an email address.

This example illustrates the benefit of using Avro: even though we changed the schema in the messages without changing all the applications reading the data, there will be no exceptions or breaking errors and no need for expensive updates of existing data.

However, there are two caveats to this scenario:

- The schema used for writing the data and the schema expected by the reading application must be compatible. The Avro documentation includes compatibility rules.
- The deserializer will need access to the schema that was used when writing the data, even when it is different than the schema expected by the application that accesses the data. In Avro files, the writing schema is included in the file itself, but there is a better way to handle this for Kafka messages. We will look at that next.

Using Avro Records with Kafka

Unlike Avro files, where storing the entire schema in the data file is associated with a fairly reasonable overhead, storing the entire schema in each record will usually more than double the record size. However, Avro still requires the entire schema to be present when reading the record, so we need to locate the schema elsewhere. To achieve this, we follow a common architecture pattern and use a *Schema Registry*. The Schema Registry is not part of Apache Kafka but there are several open source options to choose from. We'll use the Confluent Schema Registry for this example. You can find the Schema Registry code on GitHub, or you can install it as part of the Confluent Platform. If you decide to use the Schema Registry, then we recommend checking the documentation.

The idea is to store all the schemas used to write data to Kafka in the registry. Then we simply store the identifier for the schema in the record we produce to Kafka. The consumers can then use the identifier to pull the record out of the schema registry and deserialize the data. The key is that all this work—storing the schema in the registry and pulling it up when required—is done in the serializers and deserializers. The code that produces data to Kafka simply uses the Avro serializer just like it would any other serializer. Figure 3-2 demonstrates this process.

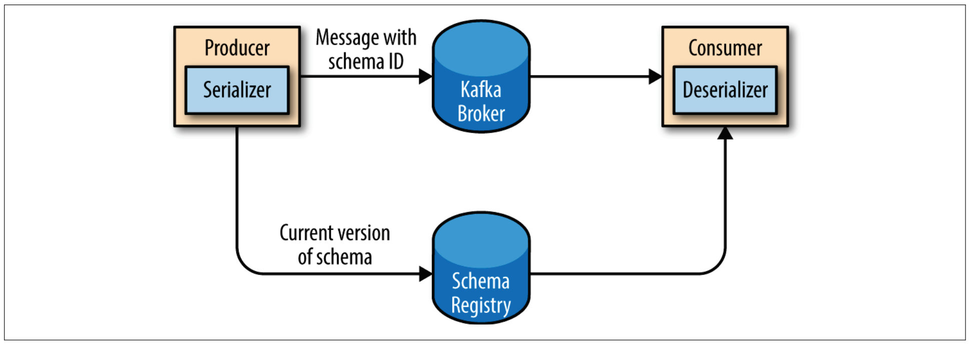

Figure 3-2. Flow diagram of serialization and deserialization of Avro records

Here is an example of how to produce generated Avro objects to Kafka (see the Avro Documentation for how to use code generation with Avro):

```
Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("value.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("schema.registry.url", schemaUrl); ❷

String topic = "customerContacts";
int wait = 500;

Producer<String, Customer> producer = new KafkaProducer<String,
   Customer>(props); ❸

// We keep producing new events until someone ctrl-c
while (true) {
    Customer customer = CustomerGenerator.getNext();
    System.out.println("Generated customer " +
       customer.toString());
    ProducerRecord<String, Customer> record =
                        new ProducerRecord<>(topic, customer.getId(), cus-
tomer); ❹
    producer.send(record); ❺
}
```

❶ We use the `KafkaAvroSerializer` to serialize our objects with Avro. Note that the `AvroSerializer` can also handle primitives, which is why we can later use `String` as the record key and our `Customer` object as the value.

❷ `schema.registry.url` is a new parameter. This simply points to where we store the schemas.

❸ `Customer` is our generated object. We tell the producer that our records will contain `Customer` as the value.

❹ We also instantiate `ProducerRecord` with `Customer` as the value type, and pass a `Customer` object when creating the new record.

❺ That's it. We send the record with our `Customer` object and `KafkaAvroSerializer` will handle the rest.

What if you prefer to use generic Avro objects rather than the generated Avro objects? No worries. In this case, you just need to provide the schema:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("value.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", url); ❷

    String schemaString = "{\"namespace\": \"customerManagement.avro\",

\"type\": \"record\", " + ❸
                           "\"name\": \"Customer\"," +
                           "\"fields\": [" +
                            "{\"name\": \"id\", \"type\": \"int\"}," +
                            "{\"name\": \"name\", \"type\": \"string\"}," +
                            "{\"name\": \"email\", \"type\": [\"null\",\"string
\"], \"default\":\"null\" }" +
                           "]}";
    Producer<String, GenericRecord> producer =
       new KafkaProducer<String, GenericRecord>(props); ❹

    Schema.Parser parser = new Schema.Parser();
    Schema schema = parser.parse(schemaString);

    for (int nCustomers = 0; nCustomers < customers; nCustomers++) {
      String name = "exampleCustomer" + nCustomers;
      String email = "example " + nCustomers + "@example.com"

      GenericRecord customer = new GenericData.Record(schema); ❺
      customer.put("id", nCustomer);
      customer.put("name", name);
      customer.put("email", email);

      ProducerRecord<String, GenericRecord> data =
                                 new ProducerRecord<String,
                                    GenericRecord>("customerContacts",
name, customer);
      producer.send(data);
    }
  }
```

❶ We still use the same `KafkaAvroSerializer`.

❷ And we provide the URI of the same schema registry.

❸ But now we also need to provide the Avro schema, since it is not provided by the Avro-generated object.

❹ Our object type is an Avro `GenericRecord`, which we initialize with our schema and the data we want to write.

❺ Then the value of the `ProducerRecord` is simply a `GenericRecord` that countains our schema and data. The serializer will know how to get the schema from this record, store it in the schema registry, and serialize the object data.

Partitions

In previous examples, the `ProducerRecord` objects we created included a topic name, key, and value. Kafka messages are key-value pairs and while it is possible to create a `ProducerRecord` with just a topic and a value, with the key set to `null` by default, most applications produce records with keys. Keys serve two goals: they are additional information that gets stored with the message, and they are also used to decide which one of the topic partitions the message will be written to. All messages with the same key will go to the same partition. This means that if a process is reading only a subset of the partitions in a topic (more on that in Chapter 4), all the records for a single key will be read by the same process. To create a key-value record, you simply create a `ProducerRecord` as follows:

```
ProducerRecord<Integer, String> record =
        new ProducerRecord<>("CustomerCountry", "Laboratory Equipment", "USA");
```

When creating messages with a null key, you can simply leave the key out:

```
ProducerRecord<Integer, String> record =
        new ProducerRecord<>("CustomerCountry", "USA"); ❶
```

❶ Here, the key will simply be set to `null`, which may indicate that a customer name was missing on a form.

When the key is `null` and the default partitioner is used, the record will be sent to one of the available partitions of the topic at random. A round-robin algorithm will be used to balance the messages among the partitions.

If a key exists and the default partitioner is used, Kafka will hash the key (using its own hash algorithm, so hash values will not change when Java is upgraded), and use the result to map the message to a specific partition. Since it is important that a key is always mapped to the same partition, we use all the partitions in the topic to calculate the mapping—not just the available partitions. This means that if a specific partition is unavailable when you write data to it, you might get an error. This is fairly rare, as you will see in Chapter 6 when we discuss Kafka's replication and availability.

The mapping of keys to partitions is consistent only as long as the number of partitions in a topic does not change. So as long as the number of partitions is constant, you can be sure that, for example, records regarding user 045189 will always get written to partition 34. This allows all kinds of optimization when reading data from partitions. However, the moment you add new partitions to the topic, this is no longer guaranteed—the old records will stay in partition 34 while new records will get written to a different partition. When partitioning keys is important, the easiest solution is to create topics with sufficient partitions (Chapter 2 includes suggestions for how to determine a good number of partitions) and never add partitions.

Implementing a custom partitioning strategy

So far, we have discussed the traits of the default partitioner, which is the one most commonly used. However, Kafka does not limit you to just hash partitions, and sometimes there are good reasons to partition data differently. For example, suppose that you are a B2B vendor and your biggest customer is a company that manufactures handheld devices called Bananas. Suppose that you do so much business with customer "Banana" that over 10% of your daily transactions are with this customer. If you use default hash partitioning, the Banana records will get allocated to the same partition as other accounts, resulting in one partition being about twice as large as the rest. This can cause servers to run out of space, processing to slow down, etc. What we really want is to give Banana its own partition and then use hash partitioning to map the rest of the accounts to partitions.

Here is an example of a custom partitioner:

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;

public class BananaPartitioner implements Partitioner {

        public void configure(Map<String, ?> configs) {} ❶

        public int partition(String topic, Object key, byte[] keyBytes,
                                   Object value, byte[] valueBytes,
                                     Cluster cluster) {
                List<PartitionInfo> partitions =
                  cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if ((keyBytes == null) || (!(key instanceOf String))) ❷
         throw new InvalidRecordException("We expect all messages
           to have customer name as key")

        if (((String) key).equals("Banana"))
         return numPartitions; // Banana will always go to last
                                  partition

        // Other records will get hashed to the rest of the
           partitions
        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1))
        }

        public void close() {}
}
```

❶ Partitioner interface includes `configure`, `partition`, and `close` methods. Here we only implement `partition`, although we really should have passed the special customer name through `configure` instead of hard-coding it in `partition`.

❷ We only expect String keys, so we throw an exception if that is not the case.

Old Producer APIs

In this chapter we've discussed the Java producer client that is part of the `org.apache.kafka.clients` package. However, Apache Kafka still has two older clients written in Scala that are part of the `kafka.producer` package and the core Kafka module. These producers are called `SyncProducers` (which, depending on the value of the `acks` parameter, may wait for the server to `ack` each message or batch of messages before sending additional messages) and `AsyncProducer` (which batches messages in the background, sends them in a separate thread, and does not provide feedback regarding success to the client).

Because the current producer supports both behaviors and provides much more reliability and control to the developer, we will not discuss the older APIs. If you are interested in using them, think twice and then refer to Apache Kafka documentation to learn more.

Summary

We began this chapter with a simple example of a producer—just 10 lines of code that send events to Kafka. We added to the simple example by adding error handling and experimenting with synchronous and asynchronous producing. We then explored the most important producer configuration parameters and saw how they modify the behavior of the producers. We discussed serializers, which let us control the format of the events we write to Kafka. We looked in-depth at Avro, one of many ways to serialize events, but one that is very commonly used with Kafka. We concluded the chapter with a discussion of partitioning in Kafka and an example of an advanced custom partitioning technique.

Now that we know how to write events to Kafka, in Chapter 4 we'll learn all about consuming events from Kafka.

CHAPTER 4

Kafka Consumers: Reading Data from Kafka

Applications that need to read data from Kafka use a `KafkaConsumer` to subscribe to Kafka topics and receive messages from these topics. Reading data from Kafka is a bit different than reading data from other messaging systems, and there are few unique concepts and ideas involved. It is difficult to understand how to use the consumer API without understanding these concepts first. We'll start by explaining some of the important concepts, and then we'll go through some examples that show the different ways consumer APIs can be used to implement applications with varying requirements.

Kafka Consumer Concepts

In order to understand how to read data from Kafka, you first need to understand its consumers and consumer groups. The following sections cover those concepts.

Consumers and Consumer Groups

Suppose you have an application that needs to read messages from a Kafka topic, run some validations against them, and write the results to another data store. In this case your application will create a consumer object, subscribe to the appropriate topic, and start receiving messages, validating them and writing the results. This may work well for a while, but what if the rate at which producers write messages to the topic exceeds the rate at which your application can validate them? If you are limited to a single consumer reading and processing the data, your application may fall farther and farther behind, unable to keep up with the rate of incoming messages. Obviously there is a need to scale consumption from topics. Just like multiple producers can write to the same topic, we need to allow multiple consumers to read from the same topic, splitting the data between them.

Kafka consumers are typically part of a `consumer group`. When multiple consumers are subscribed to a topic and belong to the same consumer group, each consumer in the group will receive messages from a different subset of the partitions in the topic.

Let's take topic T1 with four partitions. Now suppose we created a new consumer, C1, which is the only consumer in group G1, and use it to subscribe to topic T1. Consumer C1 will get all messages from all four t1 partitions. See Figure 4-1.

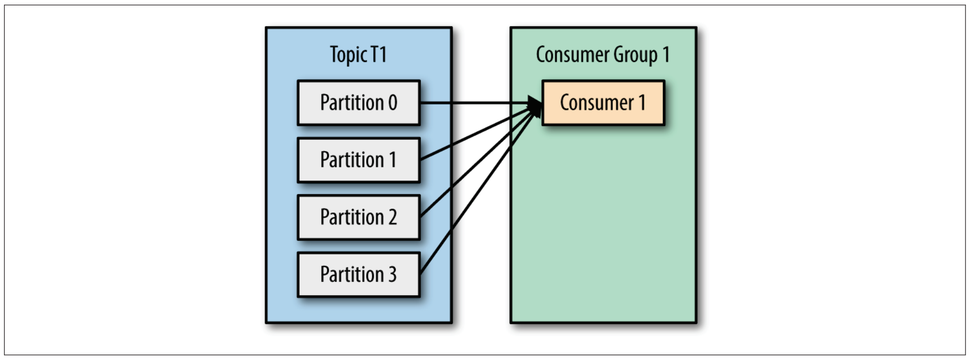

Figure 4-1. One Consumer group with four partitions

If we add another consumer, C2, to group G1, each consumer will only get messages from two partitions. Perhaps messages from partition 0 and 2 go to C1 and messages from partitions 1 and 3 go to consumer C2. See Figure 4-2.

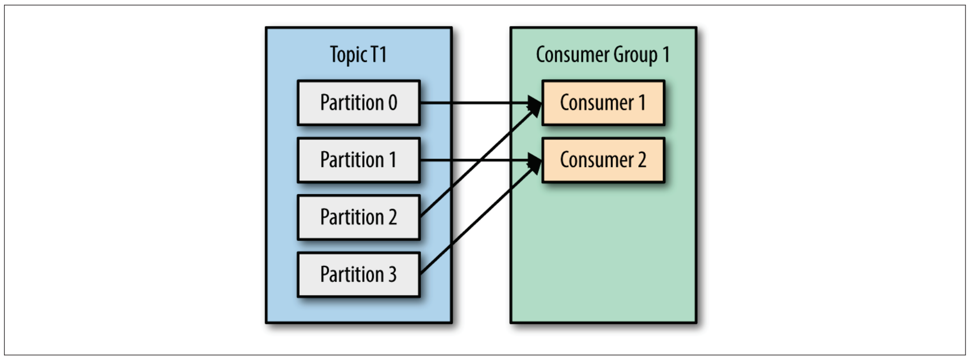

Figure 4-2. Four partitions split to two consumer groups

If G1 has four consumers, then each will read messages from a single partition. See Figure 4-3.

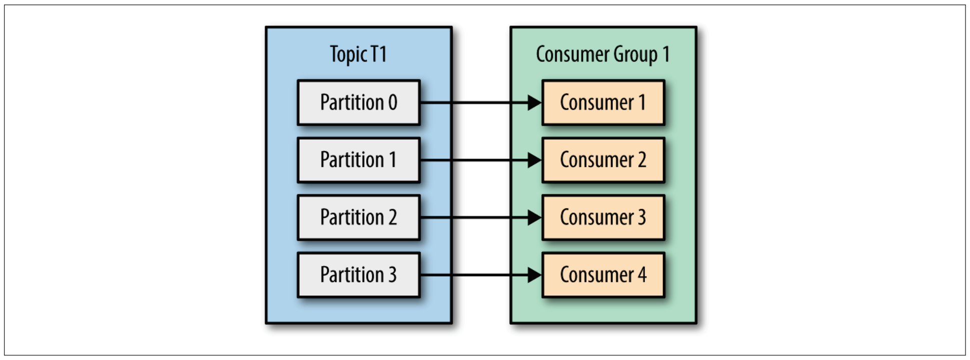

Figure 4-3. Four consumer groups to one partition each

If we add more consumers to a single group with a single topic than we have partitions, some of the consumers will be idle and get no messages at all. See Figure 4-4.

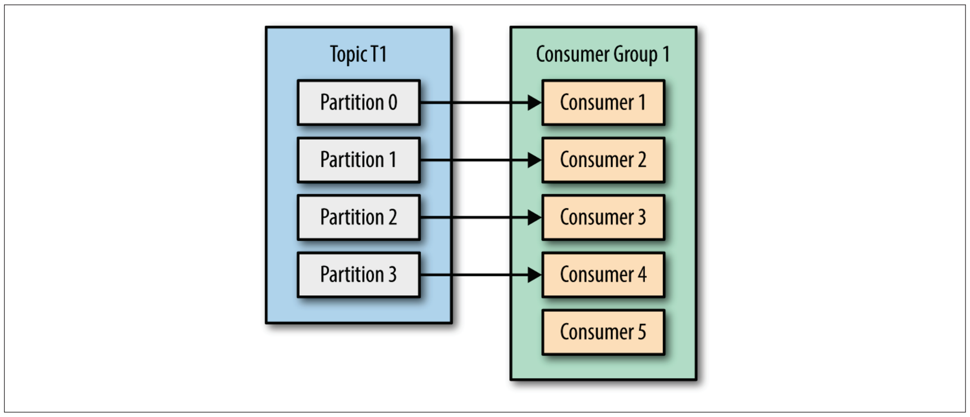

Figure 4-4. More consumer groups than partitions means missed messages

The main way we scale data consumption from a Kafka topic is by adding more consumers to a consumer group. It is common for Kafka consumers to do high-latency operations such as write to a database or a time-consuming computation on the data. In these cases, a single consumer can't possibly keep up with the rate data flows into a topic, and adding more consumers that share the load by having each consumer own just a subset of the partitions and messages is our main method of scaling. This is a good reason to create topics with a large number of partitions—it allows adding more consumers when the load increases. Keep in mind that there is no point in adding more consumers than you have partitions in a topic—some of the consumers will just be idle. Chapter 2 includes some suggestions on how to choose the number of partitions in a topic.

In addition to adding consumers in order to scale a single application, it is very common to have multiple applications that need to read data from the same topic. In fact, one of the main design goals in Kafka was to make the data produced to Kafka topics available for many use cases throughout the organization. In those cases, we want each application to get all of the messages, rather than just a subset. To make sure an application gets all the messages in a topic, ensure the application has its own consumer group. Unlike many traditional messaging systems, Kafka scales to a large number of consumers and consumer groups without reducing performance.

In the previous example, if we add a new consumer group G2 with a single consumer, this consumer will get all the messages in topic T1 independent of what G1 is doing. G2 can have more than a single consumer, in which case they will each get a subset of partitions, just like we showed for G1, but G2 as a whole will still get all the messages regardless of other consumer groups. See Figure 4-5.

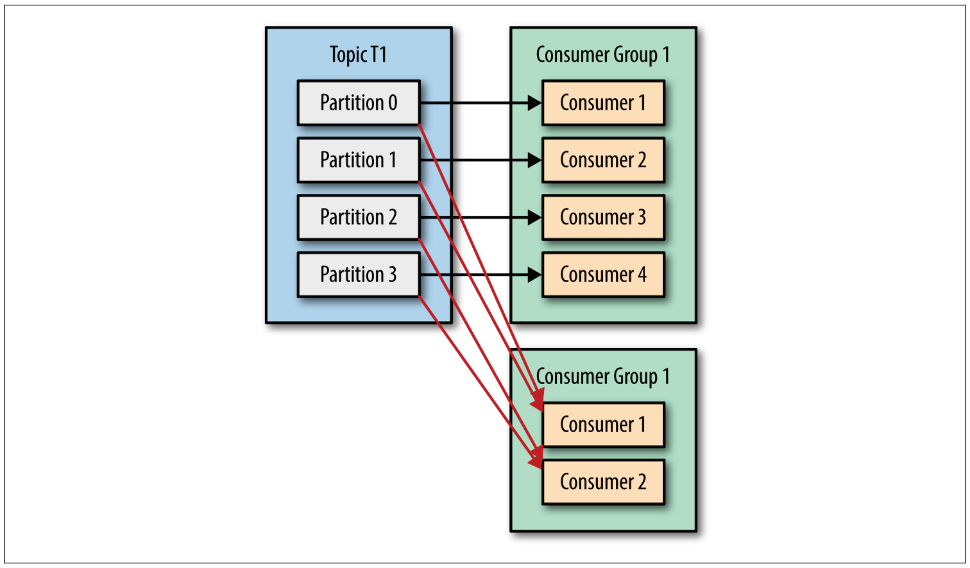

Figure 4-5. Adding a new consumer group ensures no messages are missed

To summarize, you create a new consumer group for each application that needs all the messages from one or more topics. You add consumers to an existing consumer group to scale the reading and processing of messages from the topics, so each additional consumer in a group will only get a subset of the messages.

Consumer Groups and Partition Rebalance

As we saw in the previous section, consumers in a consumer group share ownership of the partitions in the topics they subscribe to. When we add a new consumer to the group, it starts consuming messages from partitions previously consumed by another consumer. The same thing happens when a consumer shuts down or crashes; it leaves the group, and the partitions it used to consume will be consumed by one of the remaining consumers. Reassignment of partitions to consumers also happen when the topics the consumer group is consuming are modified (e.g., if an administrator adds new partitions).

Moving partition ownership from one consumer to another is called a *rebalance*. Rebalances are important because they provide the consumer group with high availability and scalability (allowing us to easily and safely add and remove consumers), but in the normal course of events they are fairly undesirable. During a rebalance, consumers can't consume messages, so a rebalance is basically a short window of unavailability of the entire consumer group. In addition, when partitions are moved from one consumer to another, the consumer loses its current state; if it was caching any data, it will need to refresh its caches—slowing down the application until the consumer sets up its state again. Throughout this chapter we will discuss how to safely handle rebalances and how to avoid unnecessary ones.

The way consumers maintain membership in a consumer group and ownership of the partitions assigned to them is by sending *heartbeats* to a Kafka broker designated as the *group coordinator* (this broker can be different for different consumer groups). As long as the consumer is sending heartbeats at regular intervals, it is assumed to be alive, well, and processing messages from its partitions. Heartbeats are sent when the consumer polls (i.e., retrieves records) and when it commits records it has consumed.

If the consumer stops sending heartbeats for long enough, its session will time out and the group coordinator will consider it dead and trigger a rebalance. If a consumer crashed and stopped processing messages, it will take the group coordinator a few seconds without heartbeats to decide it is dead and trigger the rebalance. During those seconds, no messages will be processed from the partitions owned by the dead consumer. When closing a consumer cleanly, the consumer will notify the group coordinator that it is leaving, and the group coordinator will trigger a rebalance immediately, reducing the gap in processing. Later in this chapter we will discuss configuration options that control heartbeat frequency and session timeouts and how to set those to match your requirements.

Changes to Heartbeat Behavior in Recent Kafka Versions

In release 0.10.1, the Kafka community introduced a separate heartbeat thread that will send heartbeats in between polls as well. This allows you to separate the heartbeat frequency (and therefore how long it takes for the consumer group to detect that a consumer crashed and is no longer sending heartbeats) from the frequency of polling (which is determined by the time it takes to process the data returned from the brokers). With newer versions of Kafka, you can configure how long the application can go without polling before it will leave the group and trigger a rebalance. This configuration is used to prevent a *livelock*, where the application did not crash but fails to make progress for some reason. This configuration is separate from `session.timeout.ms`, which controls the time it takes to detect a consumer crash and stop sending heartbeats.

The rest of the chapter will discuss some of the challenges with older behaviors and how the programmer can handle them. This chapter includes discussion about how to handle applications that take longer to process records. This is less relevant to readers running Apache Kafka 0.10.1 or later. If you are using a new version and need to handle records that take longer to process, you simply need to tune `max.poll.interval.ms` so it will handle longer delays between polling for new records.

How Does the Process of Assigning Partitions to Brokers Work?

When a consumer wants to join a group, it sends a `JoinGroup` request to the group coordinator. The first consumer to join the group becomes the group *leader*. The leader receives a list of all consumers in the group from the group coordinator (this will include all consumers that sent a heartbeat recently and which are therefore considered alive) and is responsible for assigning a subset of partitions to each consumer. It uses an implementation of `PartitionAssignor` to decide which partitions should be handled by which consumer.

Kafka has two built-in partition assignment policies, which we will discuss in more depth in the configuration section. After deciding on the partition assignment, the consumer leader sends the list of assignments to the `GroupCoordinator`, which sends this information to all the consumers. Each consumer only sees his own assignment—the leader is the only client process that has the full list of consumers in the group and their assignments. This process repeats every time a rebalance happens.

Creating a Kafka Consumer

The first step to start consuming records is to create a `KafkaConsumer` instance. Creating a `KafkaConsumer` is very similar to creating a `KafkaProducer`—you create a Java `Properties` instance with the properties you want to pass to the consumer. We will discuss all the properties in depth later in the chapter. To start we just need to use the three mandatory properties: `bootstrap.servers`, `key.deserializer`, and `value.deserializer`.

The first property, `bootstrap.servers`, is the connection string to a Kafka cluster. It is used the exact same way as in `KafkaProducer` (you can refer to Chapter 3 for details on how this is defined). The other two properties, `key.deserializer` and `value.deserializer`, are similar to the `serializers` defined for the producer, but rather than specifying classes that turn Java objects to byte arrays, you need to specify classes that can take a byte array and turn it into a Java object.

There is a fourth property, which is not strictly mandatory, but for now we will pretend it is. The property is `group.id` and it specifies the consumer group the `KafkaConsumer` instance belongs to. While it is possible to create consumers that do not belong to any consumer group, this is uncommon, so for most of the chapter we will assume the consumer is part of a group.

The following code snippet shows how to create a `KafkaConsumer`:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
  "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
  "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<String,
String>(props);
```

Most of what you see here should be familiar if you've read Chapter 3 on creating producers. We assume that the records we consume will have `String` objects as both the key and the value of the record. The only new property here is `group.id`, which is the name of the consumer group this consumer belong to.

Subscribing to Topics

Once we create a consumer, the next step is to subscribe to one or more topics. The `subcribe()` method takes a list of topics as a parameter, so it's pretty simple to use:

```
consumer.subscribe(Collections.singletonList("customerCountries")); ❶
```

❶ Here we simply create a list with a single element: the topic name `customerCountries`.

It is also possible to call `subscribe` with a regular expression. The expression can match multiple topic names, and if someone creates a new topic with a name that matches, a rebalance will happen almost immediately and the consumers will start consuming from the new topic. This is useful for applications that need to consume from multiple topics and can handle the different types of data the topics will contain. Subscribing to multiple topics using a regular expression is most commonly used in applications that replicate data between Kafka and another system.

To subscribe to all test topics, we can call:

```
consumer.subscribe("test.*");
```

The Poll Loop

At the heart of the consumer API is a simple loop for polling the server for more data. Once the consumer subscribes to topics, the poll loop handles all details of coordination, partition rebalances, heartbeats, and data fetching, leaving the developer with a clean API that simply returns available data from the assigned partitions. The main body of a consumer will look as follows:

```
try {
  while (true) { ❶
      ConsumerRecords<String, String> records = consumer.poll(100); ❷
      for (ConsumerRecord<String, String> record : records) ❸
      {
          log.debug("topic = %s, partition = %s, offset = %d,
             customer = %s, country = %s\n",
             record.topic(), record.partition(), record.offset(),
             record.key(), record.value());

          int updatedCount = 1;
          if (custCountryMap.countainsValue(record.value())) {
              updatedCount = custCountryMap.get(record.value()) + 1;
          }
          custCountryMap.put(record.value(), updatedCount)

          JSONObject json = new JSONObject(custCountryMap);
          System.out.println(json.toString(4)) ❹
      }
  }
} finally {
  consumer.close(); ❺
}
```

❶ This is indeed an infinite loop. Consumers are usually long-running applications that continuously poll Kafka for more data. We will show later in the chapter how to cleanly exit the loop and close the consumer.

❷ This is the most important line in the chapter. The same way that sharks must keep moving or they die, consumers must keep polling Kafka or they will be considered dead and the partitions they are consuming will be handed to another consumer in the group to continue consuming. The parameter we pass, `poll()`, is a timeout interval and controls how long `poll()` will block if data is not available in the consumer buffer. If this is set to 0, `poll()` will return immediately; otherwise, it will wait for the specified number of milliseconds for data to arrive from the broker.

❸ `poll()` returns a list of records. Each record contains the topic and partition the record came from, the offset of the record within the partition, and of course the key and the value of the record. Typically we want to iterate over the list and process the records individually. The `poll()` method takes a timeout parameter. This specifies how long it will take `poll` to return, with or without data. The value is typically driven by application needs for quick responses—how fast do you want to return control to the thread that does the polling?

❹ Processing usually ends in writing a result in a data store or updating a stored record. Here, the goal is to keep a running count of customers from each county, so we update a hashtable and print the result as JSON. A more realistic example would store the updates result in a data store.

❺ Always `close()` the consumer before exiting. This will close the network connections and sockets. It will also trigger a rebalance immediately rather than wait for the group coordinator to discover that the consumer stopped sending heartbeats and is likely dead, which will take longer and therefore result in a longer period of time in which consumers can't consume messages from a subset of the partitions.

The `poll` loop does a lot more than just get data. The first time you call `poll()` with a new consumer, it is responsible for finding the `GroupCoordinator`, joining the consumer group, and receiving a partition assignment. If a rebalance is triggered, it will be handled inside the poll loop as well. And of course the heartbeats that keep consumers alive are sent from within the poll loop. For this reason, we try to make sure that whatever processing we do between iterations is fast and efficient.

Thread Safety

You can't have multiple consumers that belong to the same group in one thread and you can't have multiple threads safely use the same consumer. One consumer per thread is the rule. To run multiple consumers in the same group in one application, you will need to run each in its own thread. It is useful to wrap the consumer logic in its own object and then use Java's `ExecutorService` to start multiple threads each with its own consumer. The Confluent blog has a tutorial that shows how to do just that.

Configuring Consumers

So far we have focused on learning the consumer API, but we've only looked at a few of the configuration properties—just the mandatory `bootstrap.servers`, `group.id`, `key.deserializer`, and `value.deserializer`. All the consumer configuration is documented in Apache Kafka documentation. Most of the parameters have reasonable defaults and do not require modification, but some have implications on the performance and availability of the consumers. Let's take a look at some of the more important properties.

fetch.min.bytes

This property allows a consumer to specify the minimum amount of data that it wants to receive from the broker when fetching records. If a broker receives a request for records from a consumer but the new records amount to fewer bytes than `min.fetch.bytes`, the broker will wait until more messages are available before sending the records back to the consumer. This reduces the load on both the consumer and the broker as they have to handle fewer back-and-forth messages in cases where the topics don't have much new activity (or for lower activity hours of the day). You will want to set this parameter higher than the default if the consumer is using too much CPU when there isn't much data available, or reduce load on the brokers when you have large number of consumers.

fetch.max.wait.ms

By setting `fetch.min.bytes`, you tell Kafka to wait until it has enough data to send before responding to the consumer. `fetch.max.wait.ms` lets you control how long to wait. By default, Kafka will wait up to 500 ms. This results in up to 500 ms of extra latency in case there is not enough data flowing to the Kafka topic to satisfy the minimum amount of data to return. If you want to limit the potential latency (usually due to SLAs controlling the maximum latency of the application), you can set `fetch.max.wait.ms` to a lower value. If you set `fetch.max.wait.ms` to 100 ms and `fetch.min.bytes` to 1 MB, Kafka will recieve a fetch request from the consumer and will respond with data either when it has 1 MB of data to return or after 100 ms, whichever happens first.

max.partition.fetch.bytes

This property controls the maximum number of bytes the server will return per partition. The default is 1 MB, which means that when `KafkaConsumer.poll()` returns `ConsumerRecords`, the record object will use at most `max.partition.fetch.bytes` per partition assigned to the consumer. So if a topic has 20 partitions, and you have 5 consumers, each consumer will need to have 4 MB of memory available for `Consumer`

`Records`. In practice, you will want to allocate more memory as each consumer will need to handle more partitions if other consumers in the group fail. `max.partition.fetch.bytes` must be larger than the largest message a broker will accept (determined by the `max.message.size` property in the broker configuration), or the broker may have messages that the consumer will be unable to consume, in which case the consumer will hang trying to read them. Another important consideration when setting `max.partition.fetch.bytes` is the amount of time it takes the consumer to process data. As you recall, the consumer must call `poll()` frequently enough to avoid session timeout and subsequent rebalance. If the amount of data a single `poll()` returns is very large, it may take the consumer longer to process, which means it will not get to the next iteration of the poll loop in time to avoid a session timeout. If this occurs, the two options are either to lower `max.partition.fetch.bytes` or to increase the session timeout.

session.timeout.ms

The amount of time a consumer can be out of contact with the brokers while still considered alive defaults to 3 seconds. If more than `session.timeout.ms` passes without the consumer sending a heartbeat to the group coordinator, it is considered dead and the group coordinator will trigger a rebalance of the consumer group to allocate partitions from the dead consumer to the other consumers in the group. This property is closely related to `heartbeat.interval.ms`. `heartbeat.interval.ms` controls how frequently the `KafkaConsumer poll()` method will send a heartbeat to the group coordinator, whereas `session.timeout.ms` controls how long a consumer can go without sending a heartbeat. Therefore, those two properties are typically modified together—`heatbeat.interval.ms` must be lower than `session.timeout.ms`, and is usually set to one-third of the timeout value. So if `session.timeout.ms` is 3 seconds, `heartbeat.interval.ms` should be 1 second. Setting `session.timeout.ms` lower than the default will allow consumer groups to detect and recover from failure sooner, but may also cause unwanted rebalances as a result of consumers taking longer to complete the poll loop or garbage collection. Setting `session.timeout.ms` higher will reduce the chance of accidental rebalance, but also means it will take longer to detect a real failure.

auto.offset.reset

This property controls the behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset or if the committed offset it has is invalid (usually because the consumer was down for so long that the record with that offset was already aged out of the broker). The default is "latest," which means that lacking a valid offset, the consumer will start reading from the newest records (records that were written after the consumer started running). The alternative is "earliest," which means that lacking a valid offset, the consumer will read all the data in the partition, starting from the very beginning.

enable.auto.commit

We discussed the different options for committing offsets earlier in this chapter. This parameter controls whether the consumer will commit offsets automatically, and defaults to `true`. Set it to `false` if you prefer to control when offsets are committed, which is necessary to minimize duplicates and avoid missing data. If you set `enable.auto.commit` to `true`, then you might also want to control how frequently offsets will be committed using `auto.commit.interval.ms`.

partition.assignment.strategy

We learned that partitions are assigned to consumers in a consumer group. A `PartitionAssignor` is a class that, given consumers and topics they subscribed to, decides which partitions will be assigned to which consumer. By default, Kafka has two assignment strategies:

Range
: Assigns to each consumer a consecutive subset of partitions from each topic it subscribes to. So if consumers C1 and C2 are subscribed to two topics, T1 and T2, and each of the topics has three partitions, then C1 will be assigned partitions 0 and 1 from topics T1 and T2, while C2 will be assigned partition 2 from those topics. Because each topic has an uneven number of partitions and the assignment is done for each topic independently, the first consumer ends up with more partitions than the second. This happens whenever Range assignment is used and the number of consumers does not divide the number of partitions in each topic neatly.

RoundRobin
: Takes all the partitions from all subscribed topics and assigns them to consumers sequentially, one by one. If C1 and C2 described previously used RoundRobin assignment, C1 would have partitions 0 and 2 from topic T1 and partition 1 from topic T2. C2 would have partition 1 from topic T1 and partitions 0 and 2 from topic T2. In general, if all consumers are subscribed to the same topics (a very common scenario), RoundRobin assignment will end up with all consumers having the same number of partitions (or at most 1 partition difference).

The `partition.assignment.strategy` allows you to choose a partition-assignment strategy. The default is `org.apache.kafka.clients.consumer.RangeAssignor`, which implements the Range strategy described above. You can replace it with `org.apache.kafka.clients.consumer.RoundRobinAssignor`. A more advanced option is to implement your own assignment strategy, in which case `partition.assignment.strategy` should point to the name of your class.

client.id

This can be any string, and will be used by the brokers to identify messages sent from the client. It is used in logging and metrics, and for quotas.

max.poll.records

This controls the maximum number of records that a single call to `poll()` will return. This is useful to help control the amount of data your application will need to process in the polling loop.

receive.buffer.bytes and send.buffer.bytes

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to -1, the OS defaults will be used. It can be a good idea to increase those when producers or consumers communicate with brokers in a different datacenter, because those network links typically have higher latency and lower bandwidth.

Commits and Offsets

Whenever we call `poll()`, it returns records written to Kafka that consumers in our group have not read yet. This means that we have a way of tracking which records were read by a consumer of the group. As discussed before, one of Kafka's unique characteristics is that it does not track acknowledgments from consumers the way many JMS queues do. Instead, it allows consumers to use Kafka to track their position (offset) in each partition.

We call the action of updating the current position in the partition a `commit`.

How does a consumer commit an offset? It produces a message to Kafka, to a special `__consumer_offsets` topic, with the committed offset for each partition. As long as all your consumers are up, running, and churning away, this will have no impact. However, if a consumer crashes or a new consumer joins the consumer group, this will *trigger a rebalance*. After a rebalance, each consumer may be assigned a new set of partitions than the one it processed before. In order to know where to pick up the work, the consumer will read the latest committed offset of each partition and continue from there.

If the committed offset is smaller than the offset of the last message the client processed, the messages between the last processed offset and the committed offset will be processed twice. See Figure 4-6.

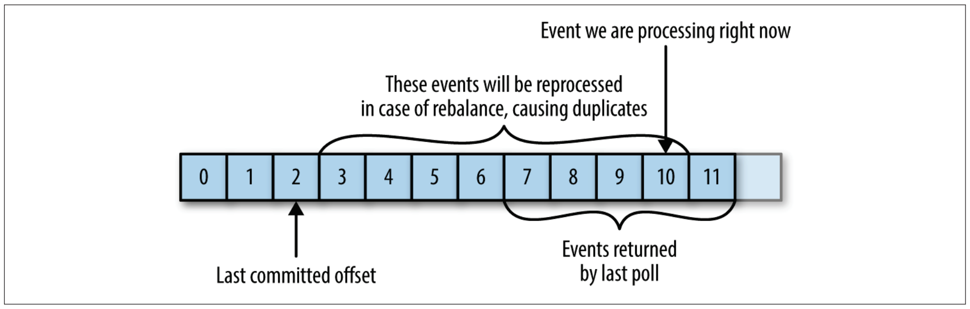
Event we are processing right now
These events will be reprocessed in case of rebalance, causing duplicates
0
1
2
3
4
5
6
7
8
9
10
11
Last committed offset
Events returned by last poll

Figure 4-6. Re-processed messages

If the committed offset is larger than the offset of the last message the client actually processed, all messages between the last processed offset and the committed offset will be missed by the consumer group. See Figure 4-7.

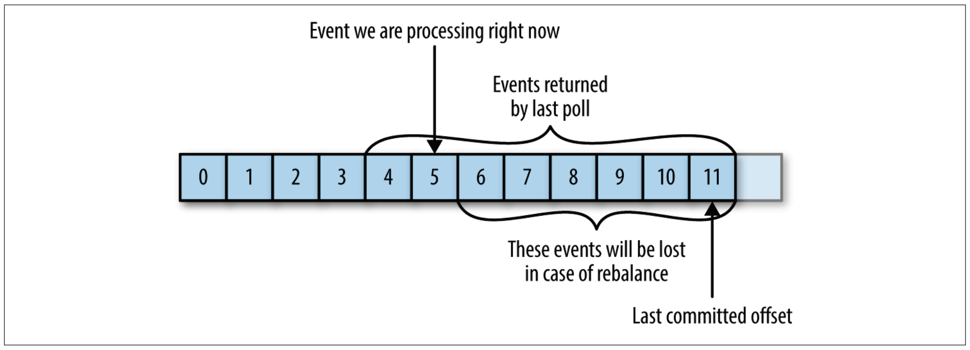
Event we are processing right now
Events returned by last poll
0
1
2
3
4
5
6
7
8
9
10
11
These events will be lost in case of rebalance
Last committed offset

Figure 4-7. Missed messages between offsets

Clearly, managing offsets has a big impact on the client application. The `KafkaConsumer` API provides multiple ways of committing offsets:

Automatic Commit

The easiest way to commit offsets is to allow the consumer to do it for you. If you configure `enable.auto.commit=true`, then every five seconds the consumer will commit the largest offset your client received from `poll()`. The five-second interval is the default and is controlled by setting `auto.commit.interval.ms`. Just like everything else in the consumer, the automatic commits are driven by the poll loop. Whenever you poll, the consumer checks if it is time to commit, and if it is, it will commit the offsets it returned in the last poll.

Before using this convenient option, however, it is important to understand the consequences.

Consider that, by default, automatic commits occur every five seconds. Suppose that we are three seconds after the most recent commit and a rebalance is triggered. After the rebalancing, all consumers will start consuming from the last offset committed. In this case, the offset is three seconds old, so all the events that arrived in those three seconds will be processed twice. It is possible to configure the commit interval to commit more frequently and reduce the window in which records will be duplicated, but it is impossible to completely eliminate them.

With autocommit enabled, a call to poll will always commit the last offset returned by the previous poll. It doesn't know which events were actually processed, so it is critical to always process all the events returned by `poll()` before calling `poll()` again. (Just like `poll()`, `close()` also commits offsets automatically.) This is usually not an issue, but pay attention when you handle exceptions or exit the poll loop prematurely.

Automatic commits are convenient, but they don't give developers enough control to avoid duplicate messages.

Commit Current Offset

Most developers exercise more control over the time at which offsets are committed —both to eliminate the possibility of missing messages and to reduce the number of messages duplicated during rebalancing. The consumer API has the option of committing the current offset at a point that makes sense to the application developer rather than based on a timer.

By setting `auto.commit.offset=false`, offsets will only be committed when the application explicitly chooses to do so. The simplest and most reliable of the commit APIs is `commitSync()`. This API will commit the latest offset returned by `poll()` and return once the offset is committed, throwing an exception if commit fails for some reason.

It is important to remember that `commitSync()` will commit the latest offset returned by `poll()`, so make sure you call `commitSync()` after you are done processing all the records in the collection, or you risk missing messages as described previously. When rebalance is triggered, all the messages from the beginning of the most recent batch until the time of the rebalance will be processed twice.

Here is how we would use `commitSync` to commit offsets after we finished processing the latest batch of messages:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset =
          %d, customer = %s, country = %s\n",
             record.topic(), record.partition(),
               record.offset(), record.key(), record.value()); ❶
    }
    try {
      consumer.commitSync(); ❷
    } catch (CommitFailedException e) {
        log.error("commit failed", e) ❸
    }
}
```

❶ Let's assume that by printing the contents of a record, we are done processing it. Your application will likely do a lot more with the records—modify them, enrich them, aggregate them, display them on a dashboard, or notify users of important events. You should determine when you are "done" with a record according to your use case.

❷ Once we are done "processing" all the records in the current batch, we call `commitSync` to commit the last offset in the batch, before polling for additional messages.

❸ `commitSync` retries committing as long as there is no error that can't be recovered. If this happens, there is not much we can do except log an error.

Asynchronous Commit

One drawback of manual commit is that the application is blocked until the broker responds to the commit request. This will limit the throughput of the application. Throughput can be improved by committing less frequently, but then we are increasing the number of potential duplicates that a rebalance will create.

Another option is the asynchronous commit API. Instead of waiting for the broker to respond to a commit, we just send the request and continue on:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s,
        offset = %d, customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(); ❶
}
```

❶ Commit the last offset and carry on.

The drawback is that while `commitSync()` will retry the commit until it either succeeds or encounters a nonretriable failure, `commitAsync()` will not retry. The reason it does not retry is that by the time `commitAsync()` receives a response from the server, there may have been a later commit that was already successful. Imagine that we sent a request to commit offset 2000. There is a temporary communication problem, so the broker never gets the request and therefore never responds. Meanwhile, we processed another batch and successfully committed offset 3000. If `commitAsync()` now retries the previously failed commit, it might succeed in committing offset 2000 *after* offset 3000 was already processed and committed. In the case of a rebalance, this will cause more duplicates.

We mention this complication and the importance of correct order of commits, because `commitAsync()` also gives you an option to pass in a callback that will be triggered when the broker responds. It is common to use the callback to log commit errors or to count them in a metric, but if you want to use the callback for retries, you need to be aware of the problem with commit order:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s,
        offset = %d, customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
        OffsetAndMetadata> offsets, Exception exception) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
      }); ❶
}
```

❶ We send the commit and carry on, but if the commit fails, the failure and the offsets will be logged.

Retrying Async Commits

A simple pattern to get commit order right for asynchronous retries is to use a monotonically increasing sequence number. Increase the sequence number every time you commit and add the sequence number at the time of the commit to the `commitAsync` callback. When you're getting ready to send a retry, check if the commit sequence number the callback got is equal to the instance variable; if it is, there was no newer commit and it is safe to retry. If the instance sequence number is higher, don't retry because a newer commit was already sent.

Combining Synchronous and Asynchronous Commits

Normally, occasional failures to commit without retrying are not a huge problem because if the problem is temporary, the following commit will be successful. But if we know that this is the last commit before we close the consumer, or before a rebalance, we want to make extra sure that the commit succeeds.

Therefore, a common pattern is to combine `commitAsync()` with `commitSync()` just before shutdown. Here is how it works (we will discuss how to commit just before rebalance when we get to the section about rebalance listeners):

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(); ❷
    } finally {
        consumer.close();
    }
}
```

❶ While everything is fine, we use `commitAsync`. It is faster, and if one commit fails, the next commit will serve as a retry.

❷ But if we are closing, there is no "next commit." We call `commitSync()`, because it will retry until it succeeds or suffers unrecoverable failure.

Commit Specified Offset

Committing the latest offset only allows you to commit as often as you finish processing batches. But what if you want to commit more frequently than that? What if `poll()` returns a huge batch and you want to commit offsets in the middle of the batch to avoid having to process all those rows again if a rebalance occurs? You can't just call `commitSync()` or `commitAsync()`—this will commit the last offset returned, which you didn't get to process yet.

Fortunately, the consumer API allows you to call `commitSync()` and `commitAsync()` and pass a map of partitions and offsets that you wish to commit. If you are in the middle of processing a batch of records, and the last message you got from partition 3 in topic "customers" has offset 5000, you can call `commitSync()` to commit offset 5000 for partition 3 in topic "customers." Since your consumer may be consuming more than a single partition, you will need to track offsets on all of them, which adds complexity to your code.

Here is what a commit of specific offsets looks like:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ❶
int count = 0;

....

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset = %d,
        customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value()); ❷
        currentOffsets.put(new TopicPartition(record.topic(),
        record.partition()), new
        OffsetAndMetadata(record.offset()+1, "no metadata")); ❸
        if (count % 1000 == 0)   ❹
            consumer.commitAsync(currentOffsets, null); ❺
        count++;
    }
}
```

❶ This is the map we will use to manually track offsets.

❷ Remember, `println` is a stand-in for whatever processing you do for the records you consume.

❸ After reading each record, we update the offsets map with the offset of the next message we expect to process. This is where we'll start reading next time we start.

❹ Here, we decide to commit current offsets every 1,000 records. In your application, you can commit based on time or perhaps content of the records.

❺ I chose to call `commitAsync()`, but `commitSync()` is also completely valid here. Of course, when committing specific offsets you still need to perform all the error handling we've seen in previous sections.

Rebalance Listeners

As we mentioned in the previous section about committing offsets, a consumer will want to do some cleanup work before exiting and also before partition rebalancing.

If you know your consumer is about to lose ownership of a partition, you will want to commit offsets of the last event you've processed. If your consumer maintained a buffer with events that it only processes occasionally (e.g., the `currentRecords` map we used when explaining `pause()` functionality), you will want to process the events you accumulated before losing ownership of the partition. Perhaps you also need to close file handles, database connections, and such.

The consumer API allows you to run your own code when partitions are added or removed from the consumer. You do this by passing a `ConsumerRebalanceListener` when calling the `subscribe()` method we discussed previously. `ConsumerRebalanceListener` has two methods you can implement:

`public void onPartitionsRevoked(Collection<TopicPartition> partitions)`
: Called before the rebalancing starts and after the consumer stopped consuming messages. This is where you want to commit offsets, so whoever gets this partition next will know where to start.

`public void onPartitionsAssigned(Collection<TopicPartition> partitions)`
: Called after partitions have been reassigned to the broker, but before the consumer starts consuming messages.

This example will show how to use `onPartitionsRevoked()` to commit offsets before losing ownership of a partition. In the next section we will show a more involved example that also demonstrates the use of `onPartitionsAssigned()`:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
  new HashMap<>();

private class HandleRebalance implements ConsumerRebalanceListener { (1)
    public void onPartitionsAssigned(Collection<TopicPartition>
      partitions) { (2)
    }

    public void onPartitionsRevoked(Collection<TopicPartition>
      partitions) {
        System.out.println("Lost partitions in rebalance.
          Committing current
        offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets); (3)
    }
}

try {
    consumer.subscribe(topics, new HandleRebalance()); ❹

    while (true) {
        ConsumerRecords<String, String> records =
          consumer.poll(100);
        for (ConsumerRecord<String, String> record : records)
        {
            System.out.printf("topic = %s, partition = %s, offset = %d,
             customer = %s, country = %s\n",
             record.topic(), record.partition(), record.offset(),
             record.key(), record.value());
             currentOffsets.put(new TopicPartition(record.topic(),
             record.partition()), new
             OffsetAndMetadata(record.offset()+1, "no metadata"));
        }
        consumer.commitAsync(currentOffsets, null);
    }
} catch (WakeupException e) {
    // ignore, we're closing
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(currentOffsets);
    } finally {
        consumer.close();
        System.out.println("Closed consumer and we are done");
    }
  }
```

❶ We start by implementing a `ConsumerRebalanceListener`.

❷ In this example we don't need to do anything when we get a new partition; we'll just start consuming messages.

❸ However, when we are about to lose a partition due to rebalancing, we need to commit offsets. Note that we are committing the latest offsets we've processed, not the latest offsets in the batch we are still processing. This is because a partition could get revoked while we are still in the middle of a batch. We are committing offsets for all partitions, not just the partitions we are about to lose—because the offsets are for events that were already processed, there is no harm in that. And we are using `commitSync()` to make sure the offsets are committed before the rebalance proceeds.

❹ The most important part: pass the `ConsumerRebalanceListener` to the `subscribe()` method so it will get invoked by the consumer.

Consuming Records with Specific Offsets

So far we've seen how to use `poll()` to start consuming messages from the last committed offset in each partition and to proceed in processing all messages in sequence. However, sometimes you want to start reading at a different offset.

If you want to start reading all messages from the beginning of the partition, or you want to skip all the way to the end of the partition and start consuming only new messages, there are APIs specifically for that: `seekToBeginning(TopicPartition tp)` and `seekToEnd(TopicPartition tp)`.

However, the Kafka API also lets you seek a specific offset. This ability can be used in a variety of ways; for example, to go back a few messages or skip ahead a few messages (perhaps a time-sensitive application that is falling behind will want to skip ahead to more relevant messages). The most exciting use case for this ability is when offsets are stored in a system other than Kafka.

Think about this common scenario: Your application is reading events from Kafka (perhaps a clickstream of users in a website), processes the data (perhaps remove records that indicate clicks from automated programs rather than users), and then stores the results in a database, NoSQL store, or Hadoop. Suppose that we really don't want to lose any data, nor do we want to store the same results in the database twice.

In these cases, the consumer loop may look a bit like this:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        currentOffsets.put(new TopicPartition(record.topic(),
        record.partition()),
        record.offset());
        processRecord(record);
        storeRecordInDB(record);
        consumer.commitAsync(currentOffsets);
    }
}
```

In this example, we are very paranoid, so we commit offsets after processing each record. However, there is still a chance that our application will crash after the record was stored in the database but before we committed offsets, causing the record to be processed again and the database to contain duplicates.

This could be avoided if there was a way to store both the record and the offset in one atomic action. Either both the record and the offset are committed, or neither of them are committed. As long as the records are written to a database and the offsets to Kafka, this is impossible.

But what if we wrote both the record and the offset to the database, in one transaction? Then we'll know that either we are done with the record and the offset is committed or we are not and the record will be reprocessed.

Now the only problem is if the record is stored in a database and not in Kafka, how will our consumer know where to start reading when it is assigned a partition? This is exactly what `seek()` can be used for. When the consumer starts or when new partitions are assigned, it can look up the offset in the database and `seek()` to that location.

Here is a skeleton example of how this may work. We use `ConsumerRebalanceLister` and `seek()` to make sure we start processing at the offsets stored in the database:

```
public class SaveOffsetsOnRebalance implements
  ConsumerRebalanceListener {

    public void onPartitionsRevoked(Collection<TopicPartition>
      partitions) {
              commitDBTransaction(); ❶
        }

    public void onPartitionsAssigned(Collection<TopicPartition>
      partitions) {
        for(TopicPartition partition: partitions)
            consumer.seek(partition, getOffsetFromDB(partition)); ❷
       }
   }
}


  consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));
  consumer.poll(0);

  for (TopicPartition partition: consumer.assignment())
    consumer.seek(partition, getOffsetFromDB(partition));   ❸

  while (true) {
      ConsumerRecords<String, String> records =
        consumer.poll(100);
      for (ConsumerRecord<String, String> record : records)
      {
          processRecord(record);
          storeRecordInDB(record);
          storeOffsetInDB(record.topic(), record.partition(),
            record.offset()); ❹
      }
      commitDBTransaction();
  }
```

❶ We use an imaginary method here to commit the transaction in the database. The idea here is that the database records and offsets will be inserted to the database as we process the records, and we just need to commit the transactions when we are about to lose the partition to make sure this information is persisted.

❷ We also have an imaginary method to fetch the offsets from the database, and then we `seek()` to those records when we get ownership of new partitions.

❸ When the consumer first starts, after we subscribe to topics, we call `poll()` once to make sure we join a consumer group and get assigned partitions, and then we immediately `seek()` to the correct offset in the partitions we are assigned to. Keep in mind that `seek()` only updates the position we are consuming from, so the next `poll()` will fetch the right messages. If there was an error in `seek()` (e.g., the offset does not exist), the exception will be thrown by `poll()`.

❹ Another imaginary method: this time we update a table storing the offsets in our database. Here we assume that updating records is fast, so we do an update on every record, but commits are slow, so we only commit at the end of the batch. However, this can be optimized in different ways.

There are many different ways to implement exactly-once semantics by storing offsets and data in an external store, but all of them will need to use the `ConsumerRebalanceListener` and `seek()` to make sure offsets are stored in time and that the consumer starts reading messages from the correct location.

But How Do We Exit?

Earlier in this chapter, when we discussed the poll loop, I told you not to worry about the fact that the consumer polls in an infinite loop and that we would discuss how to exit the loop cleanly. So, let's discuss how to exit cleanly.

When you decide to exit the poll loop, you will need another thread to call `consumer.wakeup()`. If you are running the consumer loop in the main thread, this can be done from `ShutdownHook`. Note that `consumer.wakeup()` is the only consumer method that is safe to call from a different thread. Calling wakeup will cause `poll()` to exit with `WakeupException`, or if `consumer.wakeup()` was called while the thread was not waiting on poll, the exception will be thrown on the next iteration when `poll()` is called. The `WakeupException` doesn't need to be handled, but before exiting the thread, you must call `consumer.close()`. Closing the consumer will commit offsets if needed and will send the group coordinator a message that the consumer is leaving the group. The consumer coordinator will trigger rebalancing immediately and you won't need to wait for the session to time out before partitions from the consumer you are closing will be assigned to another consumer in the group.

Here is what the exit code will look like if the consumer is running in the main application thread. This example is a bit truncated, but you can view the full example at *http://bit.ly/2u47e9A*.

```
Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                System.out.println("Starting exit...");
                consumer.wakeup(); ❶
                try {
                    mainThread.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

...

try {
            // looping until ctrl-c, the shutdown hook will
               cleanup on exit
            while (true) {
                ConsumerRecords<String, String> records =
                  movingAvg.consumer.poll(1000);
                System.out.println(System.currentTimeMillis() + "
                   --  waiting for data...");
                for (ConsumerRecord<String, String> record :
                  records) {
                    System.out.printf("offset = %d, key = %s,
                      value = %s\n",
                      record.offset(), record.key(),
                      record.value());
                }
                for (TopicPartition tp: consumer.assignment())
                    System.out.println("Committing offset at
                      position:" +
                      consumer.position(tp));
                movingAvg.consumer.commitSync();
            }
        } catch (WakeupException e) {
            // ignore for shutdown ❷
        } finally {
            consumer.close(); ❸
            System.out.println("Closed consumer and we are done");
        }
    }
```

❶ `ShutdownHook` runs in a seperate thread, so the only safe action we can take is to call `wakeup` to break out of the `poll` loop.

❷ Another thread calling `wakeup` will cause poll to throw a `WakeupException`. You'll want to catch the exception to make sure your application doesn't exit unexpectedly, but there is no need to do anything with it.

❸ Before exiting the consumer, make sure you close it cleanly.

Deserializers

As discussed in the previous chapter, Kafka producers require *serializers* to convert objects into byte arrays that are then sent to Kafka. Similarly, Kafka consumers require *deserializers* to convert byte arrays recieved from Kafka into Java objects. In previous examples, we just assumed that both the key and the value of each message are strings and we used the default `StringDeserializer` in the consumer configuration.

In Chapter 3 about the Kafka producer, we saw how to serialize custom types and how to use Avro and `AvroSerializers` to generate Avro objects from schema definitions and then serialize them when producing messages to Kafka. We will now look at how to create custom deserializers for your own objects and how to use Avro and its deserializers.

It should be obvious that the serializer used to produce events to Kafka must match the deserializer that will be used when consuming events. Serializing with `IntSerializer` and then deserializing with `StringDeserializer` will not end well. This means that as a developer you need to keep track of which serializers were used to write into each topic, and make sure each topic only contains data that the deserializers you use can interpret. This is one of the benefits of using Avro and the Schema Repository for serializing and deserializing—the `AvroSerializer` can make sure that all the data written to a specific topic is compatible with the schema of the topic, which means it can be deserialized with the matching deserializer and schema. Any errors in compatibility—on the producer or the consumer side—will be caught easily with an appropriate error message, which means you will not need to try to debug byte arrays for serialization errors.

We will start by quickly showing how to write a custom deserializer, even though this is the less common method, and then we will move on to an example of how to use Avro to deserialize message keys and values.

Custom deserializers

Let's take the same custom object we serialized in Chapter 3, and write a deserializer for it:

```
public class Customer {
        private int customerID;
        private String customerName;

        public Customer(int ID, String name) {
                this.customerID = ID;
                this.customerName = name;
        }

  public int getID() {
    return customerID;
  }

  public String getName() {
   return customerName;
  }
}
```

The custom deserializer will look as follows:

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerDeserializer implements
  Deserializer<Customer> { ❶

        @Override
  public void configure(Map configs, boolean isKey) {
   // nothing to configure
  }

  @Override
  public Customer deserialize(String topic, byte[] data) {

    int id;
    int nameSize;
    String name;

    try {
      if (data == null)
        return null;
      if (data.length < 8)
        throw new SerializationException("Size of data received by
          IntegerDeserializer is shorter than expected");

      ByteBuffer buffer = ByteBuffer.wrap(data);
```

```
        id = buffer.getInt();
        String nameSize = buffer.getInt();

        byte[] nameBytes = new Array[Byte](nameSize);
        buffer.get(nameBytes);
        name = new String(nameBytes, 'UTF-8');

        return new Customer(id, name); ❷

      } catch (Exception e) {
       throw new SerializationException("Error when serializing
         Customer
         to byte[] " + e);
      }
    }

    @Override
    public void close() {
            // nothing to close
    }
  }
```

❶ The consumer also needs the implementation of the `Customer` class, and both the class and the serializer need to match on the producing and consuming applications. In a large organization with many consumers and producers sharing access to the data, this can become challenging.

❷ We are just reversing the logic of the serializer here—we get the customer ID and name out of the byte array and use them to construct the object we need.

The consumer code that uses this serializer will look similar to this example:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
   "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
   "org.apache.kafka.common.serialization.CustomerDeserializer");

KafkaConsumer<String, Customer> consumer =
  new KafkaConsumer<>(props);

consumer.subscribe("customerCountries")

while (true) {
    ConsumerRecords<String, Customer> records =
      consumer.poll(100);
    for (ConsumerRecord<String, Customer> record : records)
    {
    System.out.println("current customer Id: " +
            record.value().getId() + " and
             current customer name: " + record.value().getName());
        }
    }
```

Again, it is important to note that implementing a custom serializer and deserializer is not recommended. It tightly couples producers and consumers and is fragile and error-prone. A better solution would be to use a standard message format such as JSON, Thrift, Protobuf, or Avro. We'll now see how to use Avro deserializers with the Kafka consumer. For background on Apache Avro, its schemas, and schema-compatibility capabilities, refer back to Chapter 3.

Using Avro deserialization with Kafka consumer

Let's assume we are using the implementation of the `Customer` class in Avro that was shown in Chapter 3. In order to consume those objects from Kafka, you want to implement a consuming application similar to this:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.serializer",
   "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer",
   "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts"

KafkaConsumer consumer = new
   KafkaConsumer(createConsumerConfig(brokers, groupId, url));
consumer.subscribe(Collections.singletonList(topic));

System.out.println("Reading topic:" + topic);

while (true) {
    ConsumerRecords<String, Customer> records =
      consumer.poll(1000); ❸

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Current customer name is: " +
           record.value().getName()); ❹
    }
    consumer.commitSync();
}
```

❶ We use `KafkaAvroDeserializer` to deserialize the Avro messages.

❷ `schema.registry.url` is a new parameter. This simply points to where we store the schemas. This way the consumer can use the schema that was registered by the producer to deserialize the message.

3 We specify the generated class, `Customer`, as the type for the record value.

4 `record.value()` is a `Customer` instance and we can use it accordingly.

Standalone Consumer: Why and How to Use a Consumer Without a Group

So far, we have discussed consumer groups, which are where partitions are assigned automatically to consumers and are rebalanced automatically when consumers are added or removed from the group. Typically, this behavior is just what you want, but in some cases you want something much simpler. Sometimes you know you have a single consumer that always needs to read data from all the partitions in a topic, or from a specific partition in a topic. In this case, there is no reason for groups or rebalances—just assign the consumer-specific topic and/or partitions, consume messages, and commit offsets on occasion.

When you know exactly which partitions the consumer should read, you don't *subscribe* to a topic—instead, you *assign* yourself a few partitions. A consumer can either subscribe to topics (and be part of a consumer group), or assign itself partitions, but not both at the same time.

Here is an example of how a consumer can assign itself all partitions of a specific topic and consume from them:

```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); 1

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
          partition.partition()));
    consumer.assign(partitions); 2

    while (true) {
        ConsumerRecords<String, String> records =
          consumer.poll(1000);

        for (ConsumerRecord<String, String> record: records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
              customer = %s, country = %s\n",
              record.topic(), record.partition(), record.offset(),
              record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```

1. We start by asking the cluster for the partitions available in the topic. If you only plan on consuming a specific partition, you can skip this part.

2. Once we know which partitions we want, we call `assign()` with the list.

Other than the lack of rebalances and the need to manually find the partitions, everything else is business as usual. Keep in mind that if someone adds new partitions to the topic, the consumer will not be notified. You will need to handle this by checking `consumer.partitionsFor()` periodically or simply by bouncing the application whenever partitions are added.

Older Consumer APIs

In this chapter we discussed the Java `KafkaConsumer` client that is part of the `org.apache.kafka.clients` package. At the time of writing, Apache Kafka still has two older clients written in Scala that are part of the `kafka.consumer` package, which is part of the core Kafka module. These consumers are called `SimpleConsumer` (which is not very simple). `SimpleConsumer` is a thin wrapper around the Kafka APIs that allows you to consume from specific partitions and offsets. The other old API is called high-level consumer or `ZookeeperConsumerConnector`. The high-level consumer is somewhat similar to the current consumer in that it has consumer groups and it rebalances partitions, but it uses Zookeeper to manage consumer groups and does not give you the same control over commits and rebalances as we have now.

Because the current consumer supports both behaviors and provides much more reliability and control to the developer, we will not discuss the older APIs. If you are interested in using them, please think twice and then refer to Apache Kafka documentation to learn more.

Summary

We started this chapter with an in-depth explanation of Kafka's consumer groups and the way they allow multiple consumers to share the work of reading events from topics. We followed the theoretical discussion with a practical example of a consumer subscribing to a topic and continuously reading events. We then looked into the most important consumer configuration parameters and how they affect consumer behavior. We dedicated a large part of the chapter to discussing offsets and how consumers keep track of them. Understanding how consumers commit offsets is critical when writing reliable consumers, so we took time to explain the different ways this can be done. We then discussed additional parts of the consumer APIs, handling rebalances and closing the consumer.

We concluded by discussing the deserializers used by consumers to turn bytes stored in Kafka into Java objects that the applications can process. We discussed Avro deserializers in some detail, even though they are just one type of deserializer you can use, because these are most commonly used with Kafka.

Now that you know how to produce and consume events with Kafka, the next chapter explains some of the internals of a Kafka implementation.

CHAPTER 11

Stream Processing

Kafka was traditionally seen as a powerful message bus, capable of delivering streams of events but without processing or transformation capabilities. Kafka's reliable stream delivery capabilities make it a perfect source of data for stream-processing systems. Apache Storm, Apache Spark Streaming, Apache Flink, Apache Samza, and many more stream-processing systems were built with Kafka often being their only reliable data source.

Industry analysts sometimes claim that all those stream-processing systems are just like the complex event processing (CEP) systems that have been around for 20 years. We think stream processing became more popular because it was created after Kafka and therefore could use Kafka as a reliable source of event streams to process. With the increased popularity of Apache Kafka, first as a simple message bus and later as a data integration system, many companies had a system containing many streams of interesting data, stored for long amounts of time and perfectly ordered, just waiting for some stream-processing framework to show up and process them. In other words, in the same way that data processing was significantly more difficult before databases were invented, stream processing was held back by lack of a stream-processing platform.

Starting from version 0.10.0, Kafka does more than provide a reliable source of data streams to every popular stream-processing framework. Now Kafka includes a powerful stream-processing library as part of its collection of client libraries. This allows developers to consume, process, and produce events in their own apps, without relying on an external processing framework.

We'll begin the chapter by explaining what we mean by stream processing (since this term is frequently misunderstood), then discuss some of the basic concepts of stream processing and the design patterns that are common to all stream-processing systems. We'll then dive into Apache Kafka's stream-processing library—its goals and architecture. We'll give a small example of how to use Kafka Streams to calculate a moving average of stock prices. We'll then discuss other examples for good stream-processing use cases and finish off the chapter by providing a few criteria you can use when choosing which stream-processing framework (if any) to use with Apache Kafka. This chapter is intended as a brief introduction to stream processing and will not cover every Kafka Streams feature or attempt to discuss and compare every stream-processing framework in existence—those topics deserve entire books on their own, possibly several.

What Is Stream Processing?

There is a lot of confusion about what stream processing means. Many definitions mix up implementation details, performance requirements, data models, and many other aspects of software engineering. I've seen the same thing play out in the world of relational databases—the abstract definitions of the relational model are getting forever entangled in the implementation details and specific limitations of the popular database engines.

The world of stream processing is still evolving, and just because a specific popular implementation does things in specific ways or has specific limitations doesn't mean that those details are an inherent part of processing streams of data.

Let's start at the beginning: What is a data stream (also called an *event stream* or *streaming data*)? First and foremost, a *data stream* is an abstraction representing an unbounded dataset. *Unbounded* means infinite and ever growing. The dataset is unbounded because over time, new records keep arriving. This definition is used by Google, Amazon, and pretty much everyone else.

Note that this simple model (a stream of events) can be used to represent pretty much every business activity we care to analyze. We can look at a stream of credit card transactions, stock trades, package deliveries, network events going through a switch, events reported by sensors in manufacturing equipment, emails sent, moves in a game, etc. The list of examples is endless because pretty much everything can be seen as a sequence of events.

There are few other attributes of event streams model, in addition to their unbounded nature:

Event streams are ordered
: There is an inherent notion of which events occur before or after other events. This is clearest when looking at financial events. A sequence in which I first put money in my account and later spend the money is very different from a sequence at which I first spend the money and later cover my debt by depositing money back. The latter will incur overdraft charges while the former will not. Note that this is one of the differences between an event stream and a database

table—records in a table are always considered unordered and the "order by" clause of SQL is not part of the relational model; it was added to assist in reporting.

Immutable data records
: Events, once occured, can never be modified. A financial transaction that is cancelled does not disapear. Instead, an additional event is written to the stream, recording a cancellation of previous transaction. When a customer returns merchandise to a shop, we don't delete the fact that the merchandise was sold to him earlier, rather we record the return as an additional event. This is another difference between a data stream and a database table—we can delete or update records in a table, but those are all additional transactions that occur in the database, and as such can be recorded in a stream of events that records all transactions. If you are familiar with binlogs, WALs, or redo logs in databases you can see that if we insert a record into a table and later delete it, the table will no longer contain the record, but the redo log will contain two transactions—the insert and the delete.

Event streams are replayable
: This is a desirable property. While it is easy to imagine nonreplayable streams (TCP packets streaming through a socket are generally nonreplayable), for most business applications, it is critical to be able to replay a raw stream of events that occured months (and sometimes years) earlier. This is required in order to correct errors, try new methods of analysis, or perform audits. This is the reason we believe Kafka made stream processing so successful in modern businesses—it allows capturing and replaying a stream of events. Without this capability, stream processing would not be more than a lab toy for data scientists.

It is worth noting that neither the definition of event streams nor the attributes we later listed say anything about the data contained in the events or the number of events per second. The data differs from system to system—events can be tiny (sometimes only a few bytes) or very large (XML messages with many headers); they can also be completely unstructured, key-value pairs, semi-structured JSON, or structured Avro or Protobuf messages. While it is often assumed that data streams are "big data" and involve millions of events per second, the same techniques we'll discuss apply equally well (and often better) to smaller streams of events with only a few events per second or minute.

Now that we know what event streams are, it's time to make sure we understand stream processing. Stream processing refers to the ongoing processing of one or more event streams. Stream processing is a programming paradigm—just like request-response and batch processing. Let's look at how different programming paradigms compare to get a better understanding of how stream processing fits into software architectures:

Request-response

This is the lowest latency paradigm, with response times ranging from submilliseconds to a few milliseconds, usually with the expectation that response times will be highly consistent. The mode of processing is usually blocking—an app sends a request and waits for the processing system to respond. In the database world, this paradigm is known as *online transaction processing* (OLTP). Point-of-sale systems, credit card processing, and time-tracking systems typically work in this paradigm.

Batch processing

This is the high-latency/high-throughput option. The processing system wakes up at set times—every day at 2:00 A.M., every hour on the hour, etc. It reads all required input (either all data available since last execution, all data from beginning of month, etc.), writes all required output, and goes away until the next time it is scheduled to run. Processing times range from minutes to hours and users expect to read stale data when they are looking at results. In the database world, these are the data warehouse and business intelligence systems—data is loaded in huge batches once a day, reports are generated, and users look at the same reports until the next data load occurs. This paradigm often has great efficiency and economy of scale, but in recent years, businesses need the data available in shorter timeframes in order to make decision-making more timely and efficient. This puts huge pressure on systems that were written to exploit economy of scale —not to provide low-latency reporting.

Stream processing

This is a contentious and nonblocking option. Filling the gap between the request-response world where we wait for events that take two milliseconds to process and the batch processing world where data is processed once a day and takes eight hours to complete. Most business processes don't require an immediate response within milliseconds but can't wait for the next day either. Most business processes happen continuously, and as long as the business reports are updated continuously and the line of business apps can continuously respond, the processing can proceed without anyone waiting for a specific response within milliseconds. Business processes like alerting on suspicious credit transactions or network activity, adjusting prices in real-time based on supply and demand, or tracking deliveries of packages are all natural fit for continuous but nonblocking processing.

It is important to note that the definition doesn't mandate any specific framework, API, or feature. As long as you are continuously reading data from an unbounded dataset, doing something to it, and emitting output, you are doing stream processing. But the processing has to be continuous and ongoing. A process that starts every day at 2:00 A.M., reads 500 records from the stream, outputs a result, and goes away doesn't quite cut it as far as stream processing goes.

Stream-Processing Concepts

Stream processing is very similar to any type of data processing—you write code that receives data, does something with the data—a few transformations, aggregates, enrichments, etc.—and then place the result somewhere. However, there are some key concepts that are unique to stream processing and often cause confusion when someone who has data processing experience first attempts to write stream-processing applications. Let's take a look at a few of those concepts.

Time

Time is probably the most important concept in stream processing and often the most confusing. For an idea of how complex time can get when discussing distributed systems, we recommend Justin Sheehy's excellent "There is No Now" paper. In the context of stream processing, having a common notion of time is critical because most stream applications perform operations on time windows. For example, our stream application might calculate a moving five-minute average of stock prices. In that case, we need to know what to do when one of our producers goes offline for two hours due to network issues and returns with two hours worth of data—most of the data will be relevant for five-minute time windows that have long passed and for which the result was already calculated and stored.

Stream-processing systems typically refer to the following notions of time:

Event time
: This is the time the events we are tracking occurred and the record was created—the time a measurement was taken, an item at was sold at a shop, a user viewed a page on our website, etc. In versions 0.10.0 and later, Kafka automatically adds the current time to producer records at the time they are created. If this does not match your application's notion of *event time*, such as in cases where the Kafka record is created based on a database record some time after the event occurred, you should add the event time as a field in the record itself. Event time is usually the time that matters most when processing stream data.

Log append time
: This is the time the event arrived to the Kafka broker and was stored there. In versions 0.10.0 and higher, Kafka brokers will automatically add this time to records they receive if Kafka is configured to do so or if the records arrive from older producers and contain no timestamps. This notion of time is typically less relevant for stream processing, since we are usually interested in the times the events occurred. For example, if we calculate number of devices produced per day, we want to count devices that were actually produced on that day, even if there were network issues and the event only arrived to Kafka the following day. However, in cases where the real event time was not recorded, log append time

can still be used consistently because it does not change after the record was created.

Processing time
: This is the time at which a stream-processing application received the event in order to perform some calculation. This time can be milliseconds, hours, or days after the event occurred. This notion of time assigns different timestamps to the same event depending on exactly when each stream processing application happened to read the event. It can even differ for two threads in the same application! Therefore, this notion of time is highly unreliable and best avoided.

Mind the Time Zone

When working with time, it is important to be mindful of time zones. The entire data pipeline should standardize on a single time zones; otherwise, results of stream operations will be confusing and often meaningless. If you must handle data streams with different time zones, you need to make sure you can convert events to a single time zone before performing operations on time windows. Often this means storing the time zone in the record itself.

State

As long as you only need to process each event individually, stream processing is a very simple activity. For example, if all you need to do is read a stream of online shopping transactions from Kafka, find the transactions over $10,000 and email the relevant salesperson, you can probably write this in just few lines of code using a Kafka consumer and SMTP library.

Stream processing becomes really interesting when you have operations that involve multiple events: counting the number of events by type, moving averages, joining two streams to create an enriched stream of information, etc. In those cases, it is not enough to look at each event by itself; you need to keep track of more information—how many events of each type did we see this hour, all events that require joining, sums, averages, etc. We call the information that is stored between events a *state*.

It is often tempting to store the state in variables that are local to the stream-processing app, such as a simple hash-table to store moving counts. In fact, we did just that in many examples in this book. However, this is not a reliable approach for managing state in stream processing because when the stream-processing application is stopped, the state is lost, which changes the results. This is usually not the desired outcome, so care should be taken to persist the most recent state and recover it when starting the application.

Stream processing refers to several types of state:

Local or internal state

State that is accessible only by a specific instance of the stream-processing application. This state is usually maintained and managed with an embedded, in-memory database running within the application. The advantage of local state is that it is extremely fast. The disadvantage is that you are limited to the amount of memory available. As a result, many of the design patterns in stream processing focus on ways to partition the data into substreams that can be processed using a limited amount of local state.

External state

State that is maintained in an external datastore, often a NoSQL system like Cassandra. The advantages of an external state are its virtually unlimited size and the fact that it can be accessed from multiple instances of the application or even from different applications. The downside is the extra latency and complexity introduced with an additional system. Most stream-processing apps try to avoid having to deal with an external store, or at least limit the latency overhead by caching information in the local state and communicating with the external store as rarely as possible. This usually introduces challenges with maintaining consistency between the internal and external state.

Stream-Table Duality

We are all familiar with database tables. A table is a collection of records, each identified by its primary key and containing a set of attributes as defined by a schema. Table records are mutable (i.e., tables allow update and delete operations). Querying a table allows checking the state of the data at a specific point in time. For example, by querying the CUSTOMERS_CONTACTS table in a database, we expect to find current contact details for all our customers. Unless the table was specifically designed to include history, we will not find their past contacts in the table.

Unlike tables, streams contain a history of changes. Streams are a string of events wherein each event caused a change. A table contains a current state of the world, which is the result of many changes. From this description, it is clear that streams and tables are two sides of the same coin—the world always changes, and sometimes we are interested in the events that caused those changes, whereas other times we are interested in the current state of the world. Systems that allow you to transition back and forth between the two ways of looking at data are more powerful than systems that support just one.

In order to convert a table to a stream, we need to capture the changes that modify the table. Take all those `insert`, `update`, and `delete` events and store them in a stream. Most databases offer change data capture (CDC) solutions for capturing these changes and there are many Kafka connectors that can pipe those changes into Kafka where they will be available for stream processing.

In order to convert a stream to a table, we need to apply all the changes that the stream contains. This is also called *materializing* the stream. We create a table, either in memory, in an internal state store, or in an external database, and start going over all the events in the stream from beginning to end, changing the state as we go. When we finish, we have a table representing a state at a specific time that we can use.

Suppose we have a store selling shoes. A stream representation of our retail activity can be a stream of events:

"Shipment arrived with red, blue, and green shoes"

"Blue shoes sold"

"Red shoes sold"

"Blue shoes returned"

"Green shoes sold"

If we want to know what our inventory contains right now or how much money we made until now, we need to materialize the view. Figure 11-1 shows that we currently have blue and yellow shoes and $170 in the bank. If we want to know how busy the store is, we can look at the entire stream and see that there were five transactions. We may also want to investigate why the blue shoes were returned.

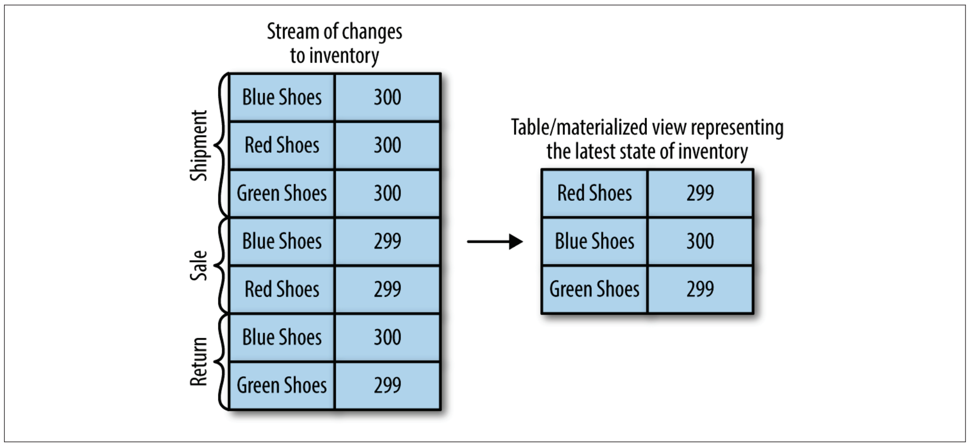

Figure 11-1. Materializing inventory changes

Time Windows

Most operations on streams are windowed operations—operating on slices of time: moving averages, top products sold this week, 99th percentile load on the system, etc. Join operations on two streams are also windowed—we join events that occurred at the same slice of time. Very few people stop and think about the type of window they want for their operations. For example, when calculating moving averages, we want to know:

- Size of the window: do we want to calculate the average of all events in every five-minute window? Every 15-minute window? Or the entire day? Larger windows are smoother but they lag more—if price increases, it will take longer to notice than with a smaller window.
- How often the window moves (*advance interval*): five-minute averages can update every minute, second, or every time there is a new event. When the *advance interval* is equal to the window size, this is sometimes called a *tumbling window*. When the window moves on every record, this is sometimes called a *sliding window*.
- How long the window remains updatable: our five-minute moving average calculated the average for 00:00-00:05 window. Now an hour later, we are getting a few more results with their *event time* showing 00:02. Do we update the result for the 00:00-00:05 period? Or do we let bygones be bygones? Ideally, we'll be able to define a certain time period during which events will get added to their respective time-slice. For example, if the events were up to four hours late, we should recalculate the results and update. If events arrive later than that, we can ignore them.

Windows can be aligned to clock time—i.e., a five-minute window that moves every minute will have the first slice as 00:00-00:05 and the second as 00:01-00:06. Or it can be unaligned and simply start whenever the app started and then the first slice can be 03:17-03:22. Sliding windows are never aligned because they move whenever there is a new record. See Figure 11-2 for the difference between two types of these windows.

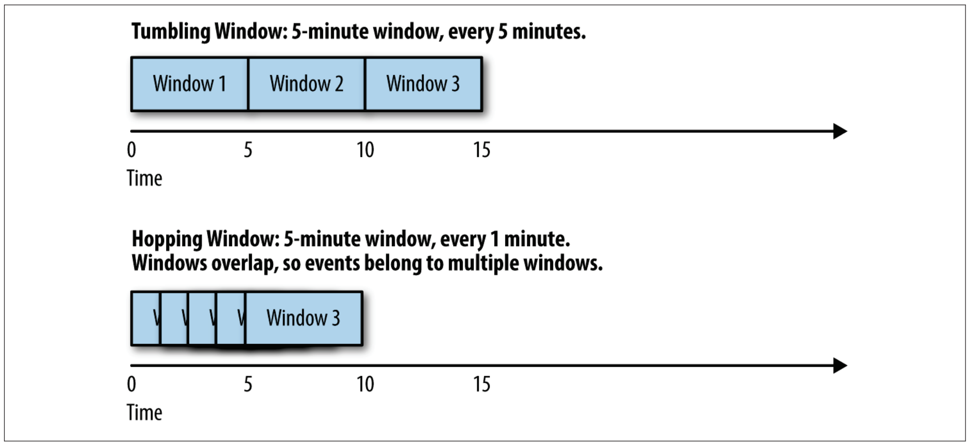
Tumbling Window: 5-minute window, every 5 minutes.
Window 1
Window 2
Window 3
0
5
10
15
Time
Hopping Window: 5-minute window, every 1 minute.
Windows overlap, so events belong to multiple windows.
Window 3
0
5
10
15
Time

Figure 11-2. Tumbling window versus Hopping window

Stream-Processing Design Patterns

Every stream-processing system is different—from the basic combination of a consumer, processing logic, and producer to involved clusters like Spark Streaming with its machine learning libraries, and much in between. But there are some basic design patterns, which are known solutions to common requirements of stream-processing architectures. We'll review a few of those well-known patterns and show how they are used with a few examples.

Single-Event Processing

The most basic pattern of stream processing is the processing of each event in isolation. This is also known as a map/filter pattern because it is commonly used to filter unnecessary events from the stream or transform each event. (The term "map" is based on the map/reduce pattern in which the map stage transforms events and the reduce stage aggregates them.)

In this pattern, the stream-processing app consumes events from the stream, modifies each event, and then produces the events to another stream. An example is an app that reads log messages from a stream and writes `ERROR` events into a high-priority stream and the rest of the events into a low-priority stream. Another example is an application that reads events from a stream and modifies them from JSON to Avro. Such applications need to maintain state within the application because each event can be handled independently. This means that recovering from app failures or load-balancing is incredibly easy as there is no need to recover state; you can simply hand off the events to another instance of the app to process.

This pattern can be easily handled with a simple producer and consumer, as seen in Figure 11-3.

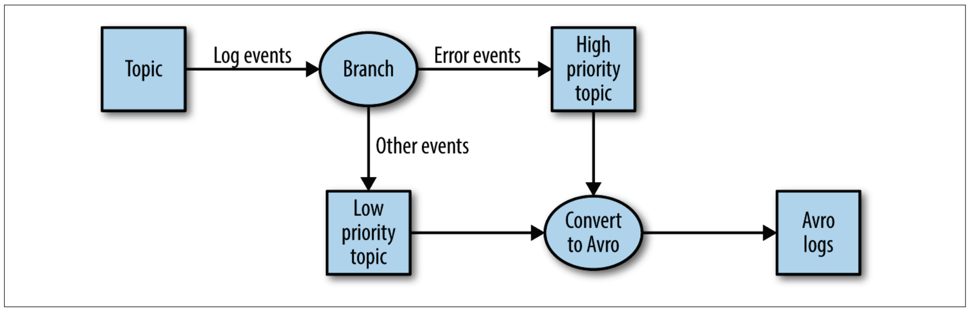

Figure 11-3. Single-event processing topology

Processing with Local State

Most stream-processing applications are concerned with aggregating information, especially time-window aggregation. An example of this is finding the minimum and maximum stock prices for each day of trading and calculating a moving average.

These aggregations require maintaining a *state* for the stream. In our example, in order to calculate the minimum and average price each day, we need to store the minimum and maximum values we've seen up until the current time and compare each new value in the stream to the stored minimum and maximum.

All these can be done using *local* state (rather than a shared state) because each operation in our example is a *group by* aggregate. That is, we perform the aggregation per stock symbol, not on the entire stock market in general. We use a Kafka partitioner to make sure that all events with the same stock symbol are written to the same partition. Then, each instance of the application will get all the events from the partitions that are assigned to it (this is a Kafka consumer guarantee). This means that each instance of the application can maintain state for the subset of stock symbols that are written to the partitions that are assigned to it. See Figure 11-4.

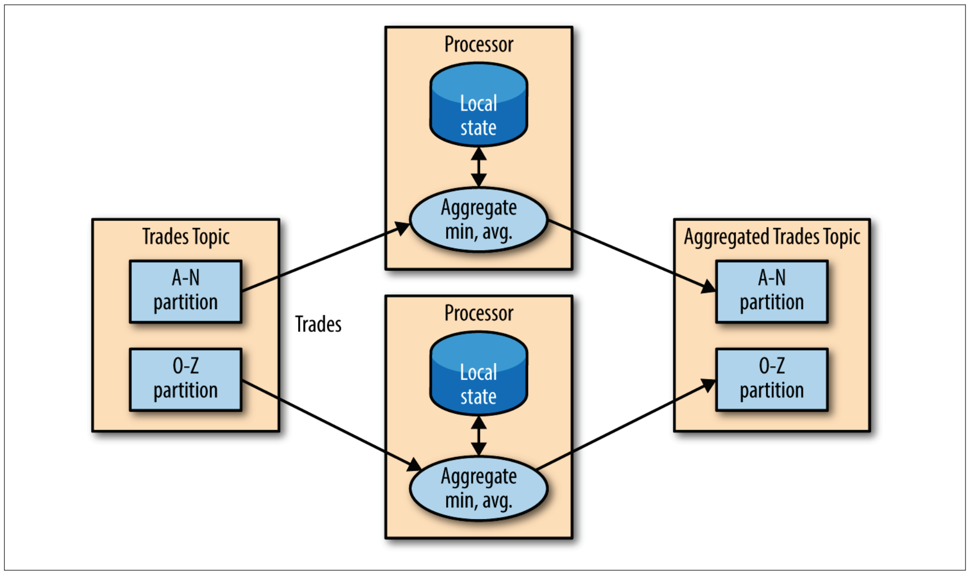

Figure 11-4. Topology for event processing with local state

Stream-processing applications become significantly more complicated when the application has local state and there are several issues a stream-processing application must address:

Memory usage
: The local state must fit into the memory available to the application instance.

Persistence
: We need to make sure the state is not lost when an application instance shuts down, and that the state can be recovered when the instance starts again or is replaced by a different instance. This is something that Kafka Streams handles very well—local state is stored in-memory using embedded RocksDB, which also persists the data to disk for quick recovery after restarts. But all the changes to the local state are also sent to a Kafka topic. If a stream's node goes down, the local state is not lost—it can be easily recreated by rereading the events from the Kafka topic. For example, if the local state contains "current minimum for IBM=167.19," we store this in Kafka, so that later we can repopulate the local cache from this data. Kafka uses log compaction for these topics to make sure they don't grow endlessly and that recreating the state is always feasible.

Rebalancing
: Partitions sometimes get reassigned to a different consumer. When this happens, the instance that loses the partition must store the last good state, and the instance that receives the partition must know to recover the correct state.

Stream-processing frameworks differ in how much they help the developer manage the local state they need. If your application requires maintaining local state, be sure to check the framework and its guarantees. We'll include a short comparison guide at the end of the chapter, but as we all know, software changes quickly and stream-processing frameworks doubly so.

Multiphase Processing/Repartitioning

Local state is great if you need a *group by* type of aggregate. But what if you need a result that uses all available information? For example, suppose we want to publish the top 10 stocks each day—the 10 stocks that gained the most from opening to closing during each day of trading. Obviously, nothing we do locally on each application instance is enough because all the top 10 stocks could be in partitions assigned to other instances. What we need is a two-phase approach. First, we calculate the daily gain/loss for each stock symbol. We can do this on each instance with a local state. Then we write the results to a new topic with a single partition. This partition will be read by a single application instance that can then find the top 10 stocks for the day. The second topic, which contains just the daily summary for each stock symbol, is obviously much smaller with significantly less traffic than the topics that contain the trades themselves, and therefore it can be processed by a single instance of the application. Sometimes more steps are needed to produce the result. See Figure 11-5.

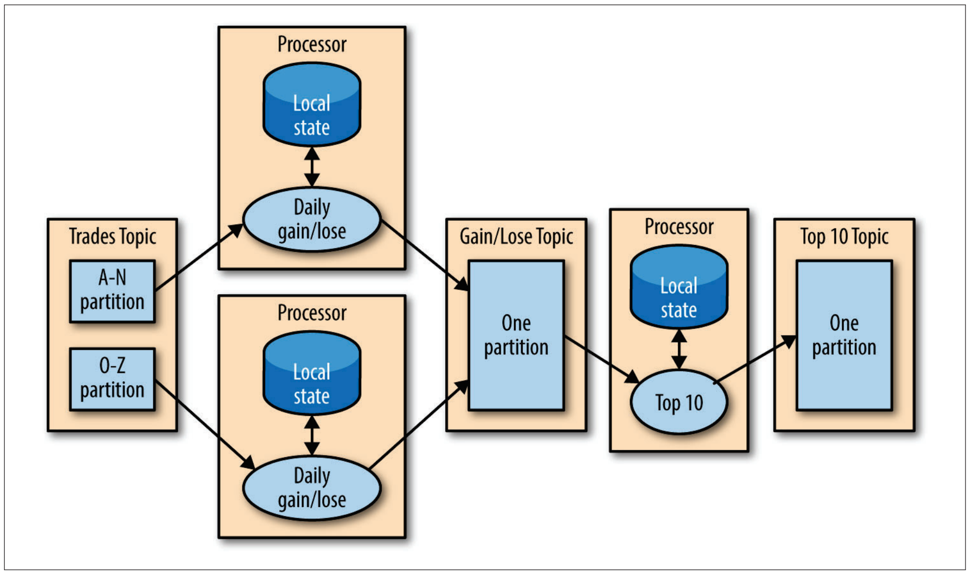

Figure 11-5. Topology that includes both local state and repartitioning steps

This type of multiphase processing is very familiar to those who write map-reduce code, where you often have to resort to multiple reduce phases. If you've ever written map-reduce code, you'll remember that you needed a separate app for each reduce step. Unlike MapReduce, most stream-processing frameworks allow including all steps in a single app, with the framework handling the details of which application instance (or worker) will run reach step.

Processing with External Lookup: Stream-Table Join

Sometimes stream processing requires integration with data external to the stream—validating transactions against a set of rules stored in a database, or enriching click-stream information with data about the users who clicked.

The obvious idea on how to perform an external lookup for data enrichment is something like this: for every click event in the stream, look up the user in the profile database and write an event that includes the original click plus the user age and gender to another topic. See Figure 11-6.

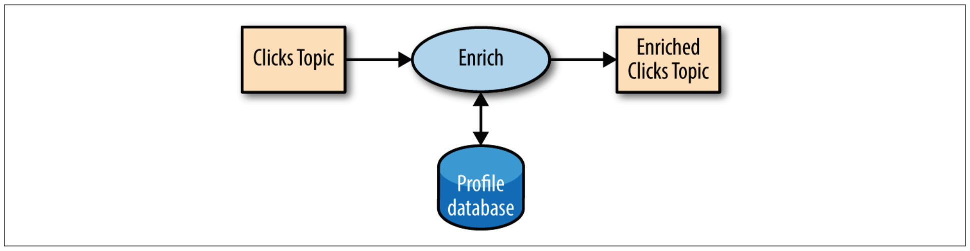

Figure 11-6. Stream processing that includes an external data source

The problem with this obvious idea is that an external lookup adds significant latency to the processing of every record—usually between 5-15 milliseconds. In many cases, this is not feasible. Often the additional load this places on the external datastore is also not acceptable—stream-processing systems can often handle 100K-500K events per second, but the database can only handle perhaps 10K events per second at reasonable performance. We want a solution that scales better.

In order to get good performance and scale, we need to cache the information from the database in our stream-processing application. Managing this cache can be challenging though—how do we prevent the information in the cache from getting stale? If we refresh events too often, we are still hammering the database and the cache isn't helping much. If we wait too long to get new events, we are doing stream processing with stale information.

But if we can capture all the changes that happen to the database table in a stream of events, we can have our stream-processing job listen to this stream and update the cache based on database change events. Capturing changes to the database as events in a stream is known as CDC, and if you use Kafka Connect you will find multiple connectors capable of performing CDC and converting database tables to a stream of change events. This allows you to keep your own private copy of the table, and you will be notified whenever there is a database change event so you can update your own copy accordingly. See Figure 11-7.

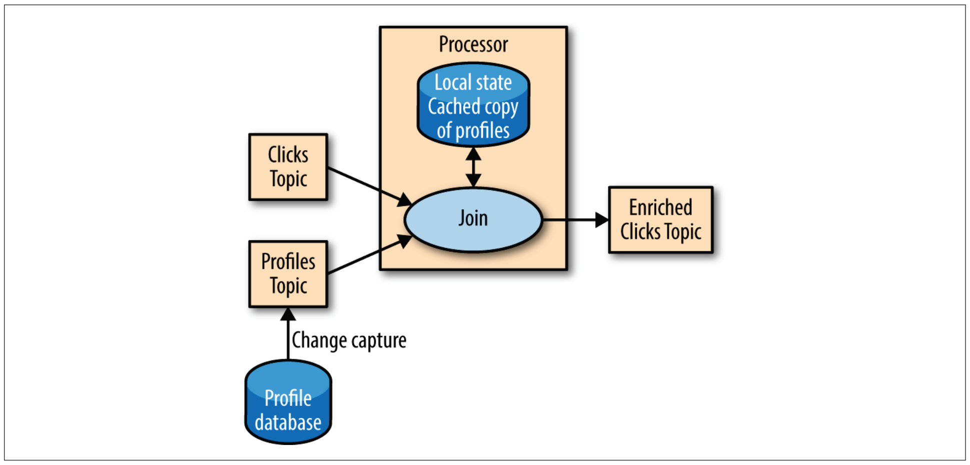

Figure 11-7. Topology joining a table and a stream of events, removing the need to involve an external data source in stream processing

Then, when you get click events, you can look up the user_id at your local cache and enrich the event. And because you are using a local cache, this scales a lot better and will not affect the database and other apps using it.

We refer to this as a *stream-table join* because one of the streams represents changes to a locally cached table.

Streaming Join

Sometimes you want to join two real event streams rather than a stream with a table. What makes a stream "real"? If you recall the discussion at the beginning of the chapter, streams are unbounded. When you use a stream to represent a table, you can ignore most of the history in the stream because you only care about the current state in the table. But when you join two streams, you are joining the entire history, trying to match events in one stream with events in the other stream that have the same key and happened in the same time-windows. This is why a streaming-join is also called a *windowed-join*.

For example, let's say that we have one stream with search queries that people entered into our website and another stream with clicks, which include clicks on search results. We want to match search queries with the results they clicked on so that we will know which result is most popular for which query. Obviously we want to match results based on the search term but only match them within a certain time-window. We assume the result is clicked seconds after the query was entered into our search engine. So we keep a small, few-seconds-long window on each stream and match the results from each window. See Figure 11-8.

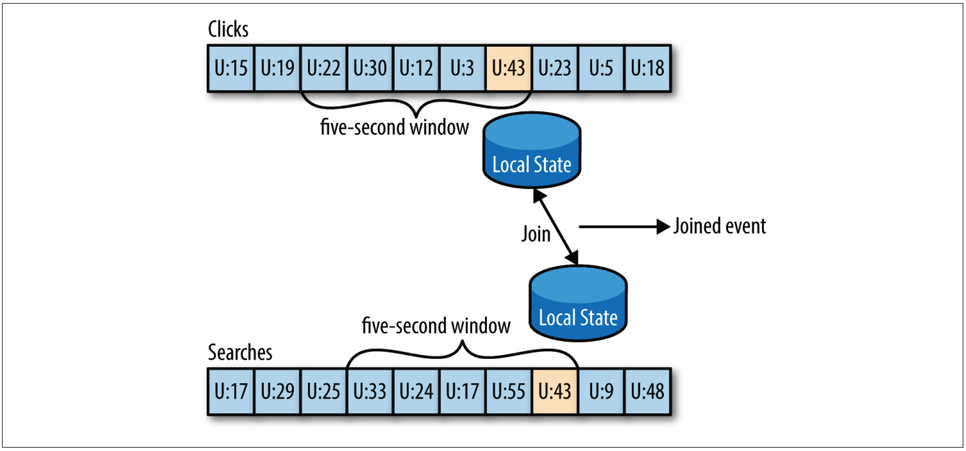

Figure 11-8. Joining two streams of events; these joins always involve a moving time window

The way this works in Kafka Streams is that both streams, queries and clicks, are partitioned on the same keys, which are also the join keys. This way, all the click events from user_id:42 end up in partition 5 of the clicks topic, and all the search events for user_id:42 end up in partition 5 of the search topic. Kafka Streams then makes sure that partition 5 of both topics is assigned to the same task. So this task sees all the relevant events for user_id:42. It maintains the join-window for both topics in its embedded RocksDB cache, and this is how it can perform the join.

Out-of-Sequence Events

Handling events that arrive at the stream at the wrong time is a challenge not just in stream processing but also in traditional ETL systems. Out-of-sequence events happen quite frequently and expectedly in IoT (Internet of Things) scenarios (Figure 11-9). For example, a mobile device loses WiFi signal for a few hours and sends a few hours' worth of events when it reconnects. This also happens when monitoring network equipment (a faulty switch doesn't send diagnostics signals until it is repaired) or manufacturing (network connectivity in plants is notoriously unreliable, especially in developing countries).

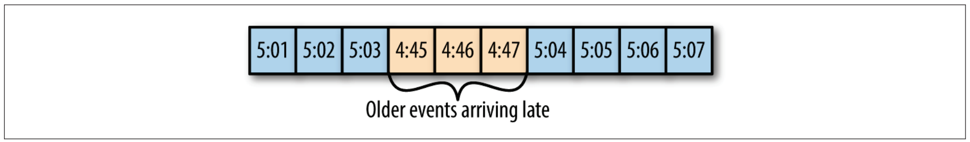

Figure 11-9. Out of sequence events

Our streams applications need to be able to handle those scenarios. This typically means the application has to do the following:

- Recognize that an event is out of sequence—this requires that the application examine the event time and discover that it is older than the current time.
- Define a time period during which it will attempt to reconcile out-of-sequence events. Perhaps a three-hour delay should be reconciled and events over three weeks old can be thrown away.
- Have an in-band capability to reconcile this event. This is the main difference between streaming apps and batch jobs. If we have a daily batch job and a few events arrived after the job completed, we can usually just rerun yesterday's job and update the events. With stream processing, there is no "rerun yesterday's job" —the same continuous process needs to handle both old and new events at any given moment.
- Be able to update results. If the results of the stream processing are written into a database, a *put* or *update* is enough to update the results. If the stream app sends results by email, updates may be trickier.

Several stream-processing frameworks, including Google's Dataflow and Kafka Streams, have built-in support for the notion of event time independent of the processing time and the ability to handle events with event times that are older or newer than the current processing time. This is typically done by maintaining multiple aggregation windows available for update in the local state and giving developers the ability to configure how long to keep those window aggregates available for updates. Of course, the longer the aggregation windows are kept available for updates, the more memory is required to maintain the local state.

The Kafka's Streams API always writes aggregation results to result topics. Those are usually `compacted topics`, which means that only the latest value for each key is preserved. In case the results of an aggregation window need to be updated as a result of a late event, Kafka Streams will simply write a new result for this aggregation window, which will overwrite the previous result.

Reprocessing

The last important pattern is processing events. There are two variants of this pattern:

- We have an improved version of our stream-processing application. We want to run the new version of the application on the same event stream as the old, produce a new stream of results that does not replace the first version, compare the results between the two versions, and at some point move clients to use the new results instead of the existing ones.
- The existing stream-processing app is buggy. We fix the bug and we want to reprocess the event stream and recalculate our results

The first use case is made simple by the fact that Apache Kafka stores the event streams in their entirety for long periods of time in a scalable datastore. This means that having two versions of a stream processing-application writing two result streams only requires the following:

- Spinning up the new version of the application as a new consumer group
- Configuring the new version to start processing from the first offset of the input topics (so it will get its own copy of all events in the input streams)
- Letting the new application continue processing and switching the client applications to the new result stream when the new version of the processing job has caught up

The second use case is more challenging—it requires "resetting" an existing app to start processing back at the beginning of the input streams, resetting the local state (so we won't mix results from the two versions of the app), and possibly cleaning the previous output stream. While Kafka Streams has a tool for resetting the state for a stream-processing app, our recommendation is to try to use the first method whenever sufficient capacity exists to run two copies of the app and generate two result streams. The first method is much safer—it allows switching back and forth between multiple versions and comparing results between versions, and doesn't risk losing critical data or introducing errors during the cleanup process.

Kafka Streams by Example

In order to demonstrate how these patterns are implemented in practice, we'll show a few examples using Apache Kafka's Streams API. We are using this specific API because it is relatively simple to use and it ships with Apache Kafka, which you already have access to. It is important to remember that the patterns can be implemented in any stream-processing framework and library—the patterns are universal but the examples are specific.

Apache Kafka has two streams APIs—a low-level Processor API and a high-level Streams DSL. We will use Kafka Streams DSL in our examples. The DSL allows you to define the stream-processing application by defining a chain of transformations to events in the streams. Transformations can be as simple as a filter or as complex as a stream-to-stream join. The lower level API allows you to create your own transformations, but as you'll see, this is rarely required.

An application that uses the DSL API always starts with using the StreamBuilder to create a processing *topology*—a directed graph (DAG) of transformations that are applied to the events in the streams. Then you create a `KafkaStreams` execution object from the topology. Starting the `KafkaStreams` object will start multiple threads, each applying the processing topology to events in the stream. The processing will conclude when you close the `KafkaStreams` object.

We'll look at few examples that use Kafka Streams to implement some of the design patterns we just discussed. A simple word count example will be used to demonstrate the map/filter pattern and simple aggregates. Then we'll move to an example where we calculate different statistics on stock market trades, which will allow us to demonstrate window aggregations. Finally we'll use ClickStream Enrichment as an example to demonstrate streaming joins.

Word Count

Let's walk through an abbreviated word count example for Kafka Streams. You can find the full example on GitHub.

The first thing you do when creating a stream-processing app is configure Kafka Streams. Kafka Streams has a large number of possible configurations, which we won't discuss here, but you can find them in the documentation. In addition, you can also configure the producer and consumer embedded in Kafka Streams by adding any producer or consumer config to the `Properties` object:

```
public class WordCountExample {

    public static void main(String[] args) throws Exception{

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
          "wordcount"); ❶
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
          "localhost:9092"); ❷
        props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
          Serdes.String().getClass().getName()); ❸
        props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
          Serdes.String().getClass().getName());
```

❶ Every Kafka Streams application must have an application ID. This is used to coordinate the instances of the application and also when naming the internal local stores and the topics related to them. This name must be unique for each Kafka Streams application working with the same Kafka cluster.

❷ The Kafka Streams application always reads data from Kafka topics and writes its output to Kafka topics. As we'll discuss later, Kafka Streams applications also use Kafka for coordination. So we had better tell our app where to find Kafka.

❸ When reading and writing data, our app will need to serialize and deserialize, so we provide default Serde classes. If needed, we can override these defaults later when building the streams topology.

Now that we have the configuration, let's build our streams topology:

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

KStream<String, String> source =
  builder.stream("wordcount-input");


final Pattern pattern = Pattern.compile("\\W+");

KStream counts  = source.flatMapValues(value->
  Arrays.asList(pattern.split(value.toLowerCase()))) ❷
        .map((key, value) -> new KeyValue<Object,
          Object>(value, value))
        .filter((key, value) -> (!value.equals("the"))) ❸
        .groupByKey() ❹
        .count("CountStore").mapValues(value->
          Long.toString(value)).toStream();❺
counts.to("wordcount-output"); ❻
```

❶ We create a `KStreamBuilder` object and start defining a stream by pointing at the topic we'll use as our input.

❷ Each event we read from the source topic is a line of words; we split it up using a regular expression into a series of individual words. Then we take each word (currently a value of the event record) and put it in the event record key so it can be used in a group-by operation.

❸ We filter out the word "the," just to show how easy filtering is.

❹ And we group by key, so we now have a collection of events for each unique word.

❺ We count how many events we have in each collection. The result of counting is a `Long` data type. We convert it to a `String` so it will be easier for humans to read the results.

❻ Only one thing left–write the results back to Kafka.

Now that we have defined the flow of transformations that our application will run, we just need to… run it:

```
        KafkaStreams streams = new KafkaStreams(builder, props); ❶

        streams.start(); ❷

        // usually the stream application would be running
           forever,
        // in this example we just let it run for some time and
           stop since the input data is finite.
        Thread.sleep(5000L);

        streams.close(); ❸

    }
}
```

❶ Define a `KafkaStreams` object based on our topology and the properties we defined.

❷ Start Kafka Streams.

❸ After a while, stop it.

Thats it! In just a few short lines, we demonstrated how easy it is to implement a single event processing pattern (we applied a map and a filter on the events). We repartitioned the data by adding a group-by operator and then maintained simple local state when we counted the number of records that have each word as a key. Then we maintained simple local state when we counted the number of times each word appeared.

At this point, we recommend running the full example. The README in the GitHub repository contains instructions on how to run the example.

One thing you'll notice is that you can run the entire example on your machine without installing anything except Apache Kafka. This is similar to the experience you may have seen when using Spark in something like *Local Mode*. The main difference is that if your input topic contains multiple partitions, you can run multiple instances of the `WordCount` application (just run the app in several different terminal tabs) and you have your first Kafka Streams processing cluster. The instances of the `WordCount` application talk to each other and coordinate the work. One of the biggest barriers to entry with Spark is that local mode is very easy to use, but then to run a production cluster, you need to install YARN or Mesos and then install Spark on all those machines, and then learn how to submit your app to the cluster. With the Kafka's Streams API, you just start multiple instances of your app—and you have a cluster. The exact same app is running on your development machine and in production.

Stock Market Statistics

The next example is more involved—we will read a stream of stock market trading events that include the stock ticker, ask price, and ask size. In stock market trades, *ask price* is what a seller is asking for whereas *bid price* is what the buyer is suggesting to pay. *Ask size* is the number of shares the seller is willing to sell at that price. For simplicity of the example, we'll ignore bids completely. We also won't include a timestamp in our data; instead, we'll rely on event time populated by our Kafka producer.

We will then create output streams that contains a few windowed statistics:

- Best (i.e., minimum) ask price for every five-second window
- Number of trades for every five-second window
- Average ask price for every five-second window

All statistics will be updated every second.

For simplicity, we'll assume our exchange only has 10 stock tickers trading in it. The setup and configuration are very similar to those we used in the "Word Count" on page 265:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stockstat");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
Constants.BROKER);
props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
TradeSerde.class.getName());
```

The main difference is the `Serde` classes used. In the "Word Count" on page 265, we used strings for both key and value and therefore used the `Serdes.String()` class as a serializer and deserializer for both. In this example, the key is still a string, but the value is a `Trade` object that contains the ticker symbol, ask price, and ask size. In order to serialize and deserialize this object (and a few other objects we used in this small app), we used the Gson library from Google to generate a JSon serializer and deserializer from our `Java` object. Then created a small wrapper that created a `Serde` object from those. Here is how we created the `Serde`:

```
static public final class TradeSerde extends WrapperSerde<Trade> {
    public TradeSerde() {
        super(new JsonSerializer<Trade>(),
          new JsonDeserializer<Trade>(Trade.class));
    }
}
```

Nothing fancy, but you need to remember to provide a Serde object for every object you want to store in Kafka—input, output, and in some cases, also intermediate results. To make this easier, we recommend generating these Serdes through projects like GSon, Avro, Protobufs, or similar.

Now that we have everything configured, it's time to build our topology:

```
KStream<TickerWindow, TradeStats> stats = source.groupByKey() ❶
        .aggregate(TradeStats::new,  ❷
            (k, v, tradestats) -> tradestats.add(v),  ❸
            TimeWindows.of(5000).advanceBy(1000), ❹
            new TradeStatsSerde(), ❺
            "trade-stats-store") ❻
        .toStream((key, value) -> new TickerWindow(key.key(),
          key.window().start())) ❼
        .mapValues((trade) -> trade.computeAvgPrice()); ❽

        stats.to(new TickerWindowSerde(), new TradeStatsSerde(),
          "stockstats-output"); ❾
```

❶ We start by reading events from the input topic and performing a `groupByKey()` operation. Despite its name, this operation does not do any grouping. Rather, it ensures that the stream of events is partitioned based on the record key. Since we wrote the data into a topic with a key and didn't modify the key before calling `groupByKey()`, the data is still partitioned by its key—so this method does nothing in this case.

❷ After we ensure correct partitioning, we start the windowed aggregation. The "aggregate" method will split the stream into overlapping windows (a five-second window every second), and then apply an aggregate method on all the events in the window. The first parameter this method takes is a new object that will contain the results of the aggregation—`Tradestats` in our case. This is an object we created to contain all the statistics we are interested in for each time window—minimum price, average price, and number of trades.

❸ We then supply a method for actually aggregating the records—in this case, an `add` method of the `Tradestats` object is used to update the minimum price, number of trades, and total prices in the window with the new record.

❹ We define the window—in this case, a window of five seconds (5,000 ms), advancing every second.

❺ Then we provide a Serde object for serializing and deserializing the results of the aggregation (the `Tradestats` object).

❻ As mentioned in "Stream-Processing Design Patterns" on page 256, windowing aggregation requires maintaining a state and a local store in which the state will be maintained. The last parameter of the aggregate method is the name of the state store. This can be any unique name.

❼ The results of the aggregation is a *table* with the ticker and the time window as the primary key and the aggregation result as the value. We are turning the table back into a stream of events and replacing the key that contains the entire time window definition with our own key that contains just the ticker and the start time of the window. This `toStream` method converts the table into a stream and also converts the key into my `TickerWindow` object.

❽ The last step is to update the average price—right now the aggregation results include the sum of prices and number of trades. We go over these records and use the existing statistics to calculate average price so we can include it in the output stream.

❾ And finally, we write the results back to the `stockstats-output` stream.

After we define the flow, we use it to generate a `KafkaStreams` object and run it, just like we did in the "Word Count" on page 265.

This example shows how to perform windowed aggregation on a stream—probably the most popular use case of stream processing. One thing to notice is how little work was needed to maintain the local state of the aggregation—just provide a Serde and name the state store. Yet this application will scale to multiple instances and automatically recover from a failure of each instance by shifting processing of some partitions to one of the surviving instances. We will see more on how it is done in "Kafka Streams: Architecture Overview" on page 272.

As usual, you can find the complete example including instructions for running it on GitHub.

Click Stream Enrichment

The last example will demonstrate streaming joins by enriching a stream of clicks on a website. We will generate a stream of simulated clicks, a stream of updates to a fictional profile database table, and a stream of web searches. We will then join all three streams to get a 360-view into each user activity. What did the users search for? What did they click as a result? Did they change their "interests" in their user profile? These kinds of joins provide a rich data collection for analytics. Product recommendations are often based on this kind of information—user searched for bikes, clicked on links for "Trek," and is interested in travel, so we can advertise bikes from Trek, helmets, and bike tours to exotic locations like Nebraska.

Since configuring the app is similar to the previous examples, let's skip this part and take a look at the topology for joining multiple streams:

```
KStream<Integer, PageView> views =
builder.stream(Serdes.Integer(),
new PageViewSerde(), Constants.PAGE_VIEW_TOPIC); ❶
KStream<Integer, Search> searches =
builder.stream(Serdes.Integer(), new SearchSerde(),
Constants.SEARCH_TOPIC);
KTable<Integer, UserProfile> profiles =
builder.table(Serdes.Integer(), new ProfileSerde(),
Constants.USER_PROFILE_TOPIC, "profile-store"); ❷

KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles, ❸
    (page, profile) -> new UserActivity(profile.getUserID(),
    profile.getUserName(), profile.getZipcode(),
    profile.getInterests(), "", page.getPage())); ❹

KStream<Integer, UserActivity> userActivityKStream =
viewsWithProfile.leftJoin(searches, ❺
    (userActivity, search) ->
    userActivity.updateSearch(search.getSearchTerms()), ❻
    JoinWindows.of(1000), Serdes.Integer(),
    new UserActivitySerde(), new SearchSerde()); ❼
```

❶ First, we create a streams objects for the two streams we want to join—clicks and searches.

❷ We also define a KTable for the user profiles. A KTable is a local cache that is updated through a stream of changes.

❸ Then we enrich the stream of clicks with user-profile information by joining the stream of events with the profile table. In a stream-table join, each event in the stream receives information from the cached copy of the profile table. We are doing a left-join, so clicks without a known user will be preserved.

❹ This is the `join` method—it takes two values, one from the stream and one from the record, and returns a third value. Unlike in databases, you get to decide how to combine the two values into one result. In this case, we created one `activity` object that contains both the user details and the page viewed.

❺ Next, we want to `join` the click information with searches performed by the same user. This is still a left `join`, but now we are joining two streams, not streaming to a table.

❻ This is the `join` method—we simply add the search terms to all the matching page views.

❼ This is the interesting part—a *stream-to-stream join* is a join with a time window. Joining all clicks and searches for each user doesn't make much sense—we want to join each search with clicks that are related to it—that is, click that occurred a short period of time after the search. So we define a join window of one second. Clicks that happen within one second of the search are considered relevant, and the search terms will be included in the activity record that contains the click and the user profile. This will allow a full analysis of searches and their results.

After we define the flow, we use it to generate a `KafkaStreams` object and run it, just like we did in the "Word Count" on page 265.

This example shows two different join patterns possible in stream processing. One joins a stream with a table to enrich all streaming events with information in the table. This is similar to joining a fact table with a dimension when running queries on a data warehouse. The second example joins two streams based on a time window. This operation is unique to stream processing.

As usual, you can find the complete example including instructions for running it on GitHub.

Kafka Streams: Architecture Overview

The examples in the previous section demonstrated how to use the Kafka Streams API to implement a few well-known stream-processing design patterns. But to understand better how Kafka's Streams library actually works and scales, we need to peek under the covers and understand some of the design principles behind the API.

Building a Topology

Every streams application implements and executes at least one *topology*. Topology (also called DAG, or directed acyclic graph, in other stream-processing frameworks) is a set of operations and transitions that every event moves through from input to output. Figure 11-10 shows the topology in the "Word Count" on page 265.

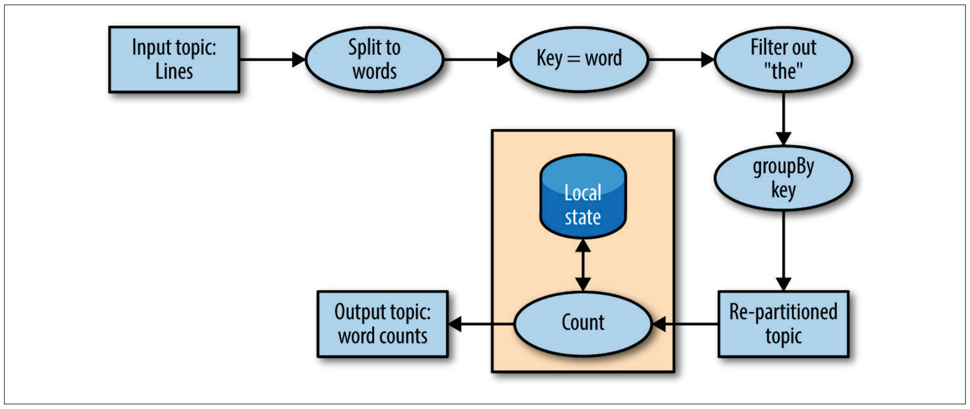

Figure 11-10. Topology for the word-count stream processing example

Even a simple app has a nontrivial topology. The topology is made up of processors—those are the nodes in the topology graph (represented by circles in our diagram). Most processors implement an operation of the data—filter, map, aggregate, etc. There are also source processors, which consume data from a topic and pass it on, and sink processors, which take data from earlier processors and produce it to a topic. A topology always starts with one or more source processors and finishes with one or more sink processors.

Scaling the Topology

Kafka Streams scales by allowing multiple threads of executions within one instance of the application and by supporting load balancing between distributed instances of the application. You can run the Streams application on one machine with multiple threads or on multiple machines; in either case, all active threads in the application will balance the work involved in data processing.

The Streams engine parallelizes execution of a topology by splitting it into tasks. The number of tasks is determined by the Streams engine and depends on the number of partitions in the topics that the application processes. Each task is responsible for a subset of the partitions: the task will subscribe to those partitions and consume events from them. For every event it consumes, the task will execute all the processing steps that apply to this partition in order before eventually writing the result to the sink. Those tasks are the basic unit of parallelism in Kafka Streams, because each task can execute independently of others. See Figure 11-11.

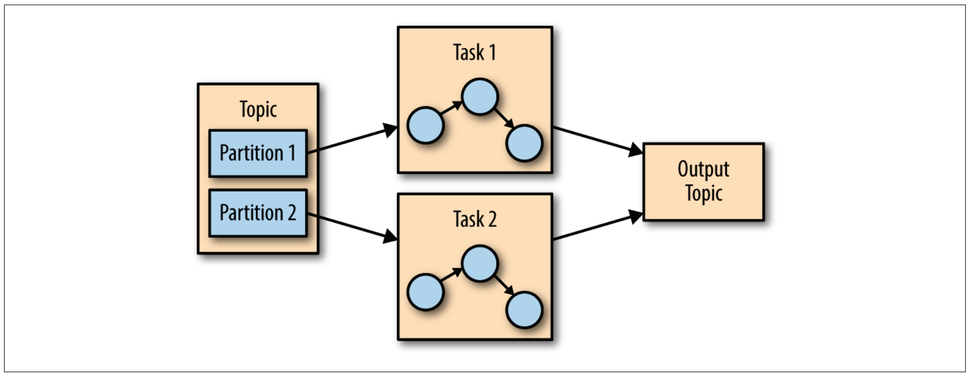

Figure 11-11. Two tasks running the same topology—one for each partition in the input topic

The developer of the application can choose the number of threads each application instance will execute. If multiple threads are available, every thread will execute a subset of the tasks that the application creates. If multiple instances of the application are running on multiple servers, different tasks will execute for each thread on each server. This is the way streaming applications scale: you will have as many tasks as you have partitions in the topics you are processing. If you want to process faster, add more threads. If you run out of resources on the server, start another instance of the application on another server. Kafka will automatically coordinate work—it will assign each task its own subset of partitions and each task will independently process events from those partitions and maintain its own local state with relevant aggregates if the topology requires this. See Figure 11-12.

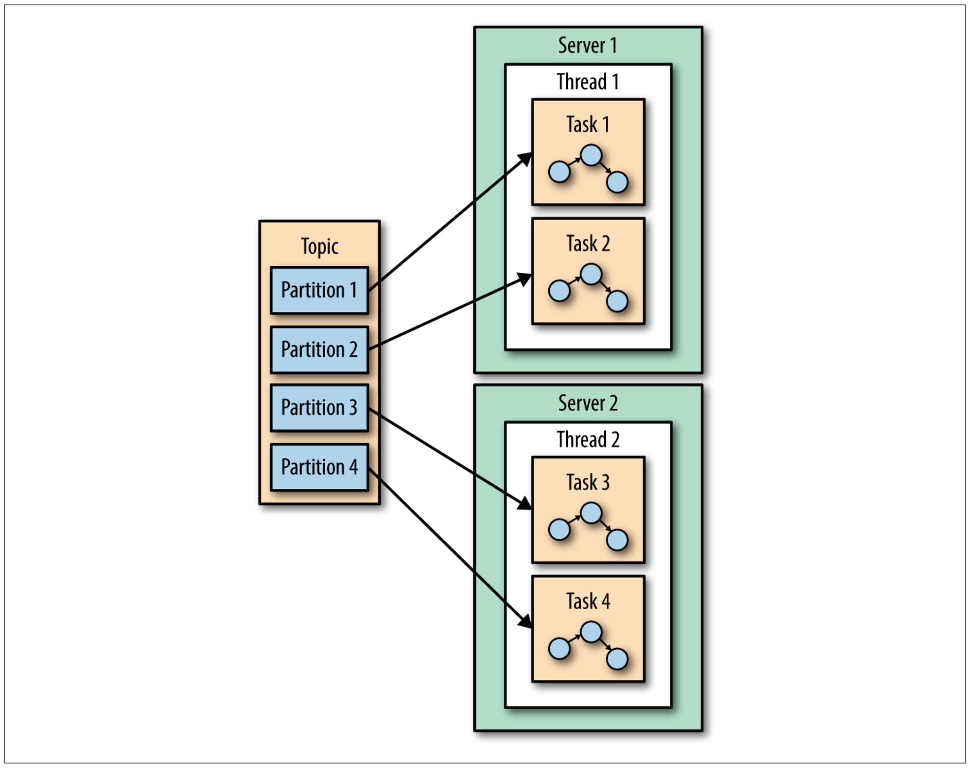

Figure 11-12. The stream processing tasks can run on multiple threads and multiple servers

You may have noticed that sometimes a processing step may require results from multiple partitions, which could create dependencies between tasks. For example, if we join two streams, as we did in the ClickStream example in "Click Stream Enrichment" on page 270, we need data from a partition in each stream before we can emit a result. Kafka Streams handles this situation by assigning all the partitions needed for one join to the same task so that the task can consume from all the relevant partitions and perform the join independently. This is why Kafka Streams currently requires that all topics that participate in a join operation will have the same number of partitions and be partitioned based on the join key.

Another example of dependencies between tasks is when our application requires repartitioning. For instance, in the ClickStream example, all our events are keyed by the user ID. But what if we want to generate statistics per page? Or per zip code? We'll need to repartition the data by the zip code and run an aggregation of the data with the new partitions. If task 1 processes the data from partition 1 and reaches a processor that repartitions the data (groupBy operation), it will need to *shuffle*, which means sending them the events—send events to other tasks to process them. Unlike other stream processor frameworks, Kafka Streams repartitions by writing the events to a new topic with new keys and partitions. Then another set of tasks reads events from the new topic and continues processing. The repartitioning steps break our topology into two subtopologies, each with its own tasks. The second set of tasks depends on the first, because it processes the results of the first subtopology. However, the first and second sets of tasks can still run independently and in parallel because the first set of tasks writes data into a topic at its own rate and the second set consumes from the topic and processes the events on its own. There is no communication and no shared resources between the tasks and they don't need to run on the same threads or servers. This is one of the more useful things Kafka does—reduce dependencies between different parts of a pipeline. See Figure 11-13.

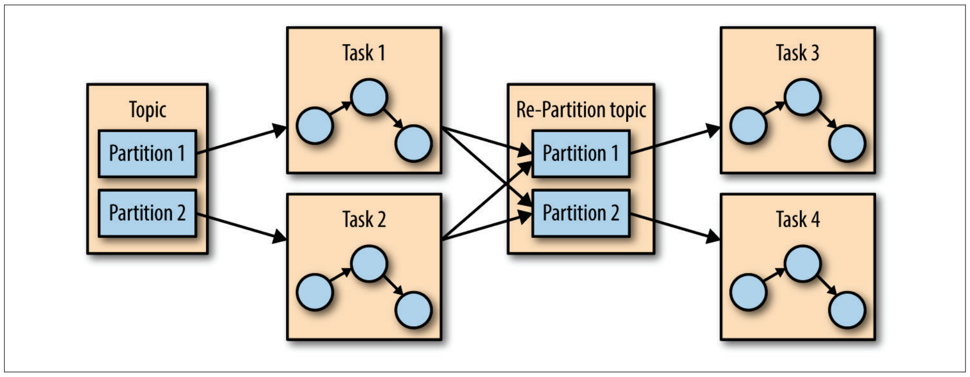

Figure 11-13. Two sets of tasks processing events with a topic for re-partitioning events between them

Surviving Failures

The same model that allows us to scale our application also allows us to gracefully handle failures. First, Kafka is highly available, and therefore the data we persist to Kafka is also highly available. So if the application fails and needs to restart, it can look up its last position in the stream from Kafka and continue its processing from the last offset it committed before failing. Note that if the local state store is lost (e.g., because we needed to replace the server it was stored on), the streams application can always re-create it from the change log it stores in Kafka.

Kafka Streams also leverages Kafka's consumer coordination to provide high availability for tasks. If a task failed but there are threads or other instances of the streams application that are active, the task will restart on one of the available threads. This is similar to how consumer groups handle the failure of one of the consumers in the group by assigning partitions to one of the remaining consumers.

Stream Processing Use Cases

Throughout this chapter we've learned how to do stream processing—from general concepts and patterns to specific examples in Kafka Streams. At this point it may be worth looking at the common stream processing use cases. As explained in the beginning of the chapter, stream processing—or continuous processing—is useful in cases where you want your events to be processed in quick order rather than wait for hours until the next batch, but also where you are not expecting a response to arrive in milliseconds. This is all true but also very abstract. Let's look at a few real scenarios that can be solved with stream processing:

Customer Service

Suppose that you just reserved a room at a large hotel chain and you expect an email confirmation and receipt. A few minutes after reserving, when the confirmation still hasn't arrived, you call customer service to confirm your reservation. Suppose the customer service desk tells you "I don't see the order in our system, but the batch job that loads the data from the reservation system to the hotels and the customer service desk only runs once a day, so please call back tomorrow. You should see the email within 2-3 business days." This doesn't sound like very good service, yet I've had this conversation more than once with a large hotel chain. What we really want is every system in the hotel chain to get an update about a new reservations seconds or minutes after the reservation is made, including the customer service center, the hotel, the system that sends email confirmations, the website, etc. You also want the customer service center to be able to immediately pull up all the details about any of your past visits to any of the hotels in the chain, and the reception desk at the hotel to know that you are a loyal customer so they can give you an upgrade. Building all those systems using stream-processing applications allows them to receive and process updates in near real time, which makes for a better customer experience. With such a system, I'd receive a confirmation email within minutes, my credit card would be charged on time, the receipt would be sent, and the service desk could immediately answer my questions regarding the reservation.

Internet of Things

Internet of Things can mean many things—from a home device for adjusting temperature and ordering refills of laundry detergent to real-time quality control of pharmaceutical manufacturing. A very common use case when applying stream processing to sensors and devices is to try to predict when preventive maintenance is needed. This is similar to application monitoring but applied to hardware and is common in many industries, including manufacturing, telecommunications (identifying faulty cellphone towers), cable TV (identifying faulty box-top devices before users complain), and many more. Every case has its own pattern, but the goal is similar: process events arriving from devices at a large scale and identify patterns that signal that a device requires maintenance. These patterns can be dropped packets for a switch, more force required to tighten screws in manufacturing, or users restarting the box more frequently for cable TV.

- **Fraud Detection**: Also known as anomaly detection, is a very wide field that focuses on catching "cheaters" or bad actors in the system. Examples of fraud-detection applications include detecting credit card fraud, stock trading fraud, video-game cheaters, and cybersecurity risks. In all these fields, there are large benefits to catching fraud as early as possible, so a near real-time system that is capable of responding to events quickly—perhaps stopping a bad transaction before it is even approved—is much preferred to a batch job that detects fraud three days after the fact when cleanup is much more complicated. This is again a problem of identifying patterns in a large-scale stream of events.

In cyber security, there is a method known as *beaconing*. When the hacker plants malware inside the organization, it will occasionally reach outside to receive commands. It can be difficult to detect this activity since it can happen at any time and any frequency. Typically, networks are well defended against external attacks but more vulnerable to someone inside the organization reaching out. By processing the large stream of network connection events and recognizing a pattern of communication as abnormal (for example, detecting that this host typically doesn't access those specific IPs), the security organization can be alerted early, before more harm is done.

How to Choose a Stream-Processing Framework

When choosing a stream-processing framework, it is important to consider the type of application you are planning on writing. Different types of applications call for different stream-processing solutions:

Ingest
: Where the goal is to get data from one system to another, with some modification to the data on how it will make it conform to the target system.

Low milliseconds actions
: Any application that requires almost immediate response. Some fraud detection use cases fall within this bucket.

Asynchronous microservices
: These microservices perform a simple action on behalf of a larger business process, such as updating the inventory of a store. These applications may need to maintain a local state caching events as a way to improve performance.

Near real-time data analytics
These streaming applications perform complex aggregations and joins in order to slice and dice the data and generate interesting business-relevant insights.

The stream-processing system you will choose will depend a lot on the problem you are solving.

- If you are trying to solve an ingest problem, you should reconsider whether you want a stream processing system or a simpler ingest-focused system like Kafka Connect. If you are sure you want a stream processing system, you need to make sure it has both a good selection of connectors and high-quality connectors for the systems you are targeting.
- If you are trying to solve a problem that requires low milliseconds actions, you should also reconsider your choice of streams. Request-response patterns are often better suited to this task. If you are sure you want a stream-processing system, then you need to opt for one that supports an event-by-event low-latency model rather than one that focuses on microbatches.
- If you are building asynchronous microservices, you need a stream processing system that integrates well with your message bus of choice (Kafka, hopefully), has change capture capabilities that easily deliver upstream changes to the microservice local caches, and has the good support of a local store that can serve as a cache or materialized view of the microservice data.
- If you are building a complex analytics engine, you also need a stream-processing system with great support for a local store—this time, not for maintenance of local caches and materialized views but rather to support advanced aggregations, windows, and joins that are otherwise difficult to implement. The APIs should include support for custom aggregations, window operations, and multiple join types.

In addition to use-case specific considerations, there are a few global considerations you should take into account:

Operability of the system
Is it easy to deploy to production? Is it easy to monitor and troubleshoot? Is it easy to scale up and down when needed? Does it integrate well with your existing infrastructure? What if there is a mistake and you need to reprocess data?

Usability of APIs and ease of debugging
I've seen orders of magnitude differences in the time it takes to write a high-quality application among different versions of the same framework. Development time and time-to-market is important so you need to choose a system that makes you efficient.

Makes hard things easy

Almost every system will claim they can do advanced windowed aggregations and maintain local caches, but the question is: do they make it easy for you? Do they handle gritty details around scale and recovery, or do they supply leaky abstractions and make you handle most of the mess? The more a system exposes clean APIs and abstractions and handles the gritty details on its own, the more productive developers will be.

Community

Most stream processing applications you consider are going to be open source, and there's no replacement for a vibrant and active community. Good community means you get new and exciting features on a regular basis, the quality is relatively good (no one wants to work on bad software), bugs get fixed quickly, and user questions get answers in timely manner. It also means that if you get a strange error and Google it, you will find information about it because other people are using this system and seeing the same issues.

Summary

We started the chapter by explaining stream processing. We gave a formal definition and discussed the common attributes of the stream-processing paradigm. We also compared it to other programming paradigms.

We then discussed important stream-processing concepts. Those concepts were demonstrated with three example applications written with Kafka Streams.

After going over all the details of these example applications, we gave an overview of the Kafka Streams architecture and explained how it works under the covers. We conclude the chapter, and the book, with several examples of stream-processing use cases and advice on how to compare different stream-processing frameworks.