Kafka Streams IN ACTION

SECOND EDITION

Bill Bejeck

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Kafka Streams in Action, Second Edition

Version 6

For more information on this and other Manning titles go to
manning.com

welcome

Hi There!

Wow, time does fly! It's hard to believe it's February 2022. While I've tried to improve the rate of MEAP releases, it still seems like life continues to throw challenges at me, impacting my delivery. But hang in there as, with this MEAP release, we're just about halfway done! As before, I'm working to improve the timing of MEAP releases.

So what's new in this installment? First of all, I've decided to change the title to *Kafka Streams in Action 2nd Edition*. After thinking about it for a while, the heart of this book is still Kafka Streams, so I wanted the title to reflect that fact. As for the content of this MEAP release, we continue with our coverage of the core Kafka Streams API, this time looking at stateful operations.

Here's a list of some of the things you'll learn in chapter 7:

1. The difference between stateless and stateful applications
2. The various types of stateful operations-- reduce, aggregations, and joins
3. The importance of keys in stateful operations in Kafka Streams

I'm excited to present this chapter to you, as I think this is where you learn how to build powerful applications to solve real-world problems.

One thing to note is that I've updated to source code to support Java 17. Given all the significant improvements available in that release, I felt it worthwhile to make the switch. So be sure to update your local Java installation when working with the source code.

I've made every attempt to make sure the book is clear and accurate. Feel free to join me on the liveBook forum at Manning.com to ask questions, offer feedback, and participate in the conversation to shape this book.

—Bill Bejeck

brief contents

1 *Welcome to the kafka event streaming platform*

This chapter covers

- Defining event streaming and events
- Introducing the Kafka event streaming platform
- Applying the platform to a concrete example

We live in a world today of unprecedented connectivity. We can watch movies on demand on an IPad, get instant notification of various accounts' status, pay bills, and deposit checks from our smartphones. If you chose to, you can receive updates on events happening around the world 24/7 by watching your social media accounts.

While this constant influx of information creates more entertainment and opportunities for the human consumer, more and more of the users of this information are software systems using other software systems. Consequently, businesses are forced to find ways to keep up with the demand and leverage the available flow of information to improve the customer experience and improve their bottom lines. For today's developer, we can sum up all this digital activity in one term: event streaming.

1.1 What is event streaming ?

In a nutshell, event streaming is capturing events generated from different sources like mobile devices, customer interaction with websites, online activity, shipment tracking, and business transactions. Event streaming is analogous to our nervous system, processing millions of events and sending signals to the appropriate parts of our body. Some signals are generated by our actions such as reaching for an apple, and other signals are handled unconsciously, as when your heart rate increases in anticipation of some exciting news. We could also see activities from machines such as sensors and inventory control as event streaming.

But event streaming doesn't stop at capturing events; it also means processing and durable storage.

The ability to process the event stream immediately is essential for making decisions based on real-time information. For example, does this purchase from customer X seem suspicious? Are the signals coming from this temperature sensor seem to indicate that something has gone wrong in a manufacturing process? Has the routing information been sent to the appropriate department of a business?

The value of the event stream is not limited to immediate information. By providing durable storage, we can go back and look at event stream data-in its raw form or perform some manipulation of the data for more insight.

1.1.1 What is an event ?

So we've defined what an event stream is, but what is an event? We'll define event very simply as "something that happens"[1]. While the term event probably brings something to mind something *notable* happening like the birth of a child, a wedding, or sporting event, we're going to focus on smaller, more constant events like a customer making a purchase (online or in-person), or clicking a link on a web-page, or a sensor transmitting data. Either people or machines can generate events. It's the sequence of events and the constant flow of them that make up an event stream.

Events conceptually contain three main components:

1. Key - an identifier for the event
2. Value - the event itself
3. timestamp - when the event occurred

Let's discuss each of these parts of an event in a little more detail. The key could be an identifier for the event, and as we'll learn in later chapters, it plays a role in routing and grouping events. Think of an online purchase, and using the customer id is an excellent example of the key. The value is the event payload itself. The event value could be a trigger such as activating a sensor

when someone opens a door or a result of some action like the item purchased in the online sale. Finally, the timestamp is the date-time when recording when the event occurred. As we go through the various chapters in this book, we'll encounter all three components of this "event trinity" regularly.

1.1.2 An event stream example

Let's say you've purchased a Flux Capacitor, and you're excited to receive your new purchase. Let's walk through the events leading up to the time you get your brand new Flux Capacitor, using the following illustration as your guide.

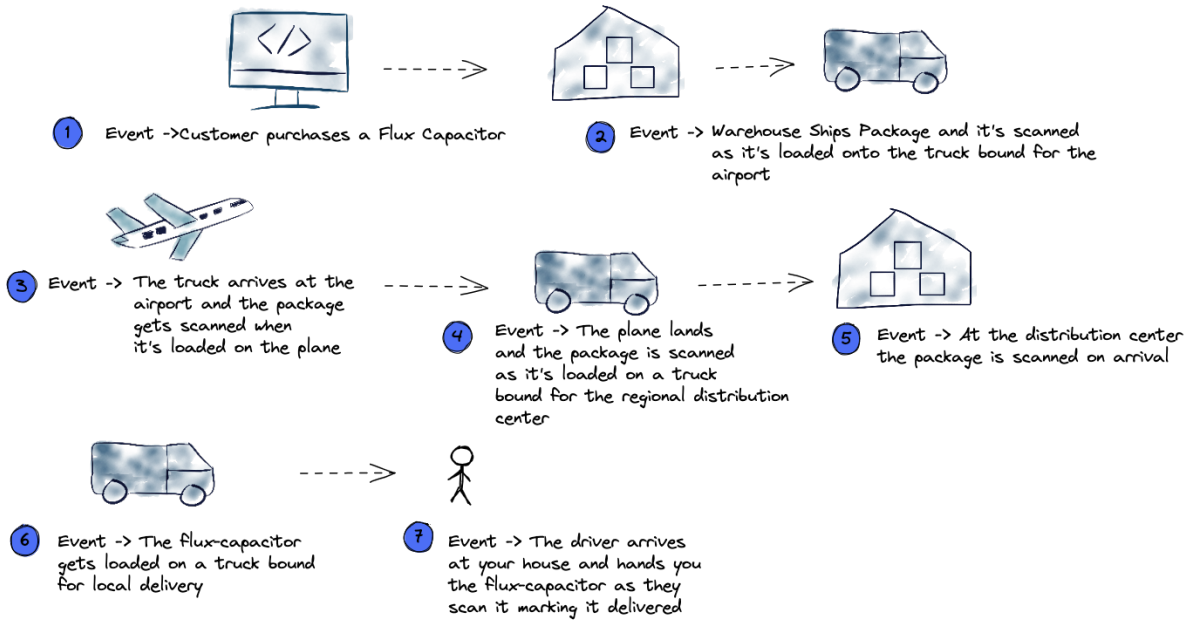


Figure 1.1 A sequence of events comprising an event stream starting with the online purchase of the flux ch01capacitor

1. You complete the purchase on the retailer's website, and the site provides a tracking number.
2. The retailer's warehouse receives the purchase event information and puts the Flux Capacitor on a shipping truck, recording the date and time your purchase left the warehouse.
3. The truck arrives at the airport, the driver loads the Flux Capacitor on a plane, and scans a barcode recording the date and time.
4. The plane lands, and the package is loaded on a truck again headed for the regional distribution center. The delivery service records the date and time when they've loaded your Flux Capacitor.
5. The truck from the airport arrives at the regional distribution center. A delivery service employee unloads the Flux Capacitor , scanning the date and time of the arrival at the distribution center.
6. Another employee takes your Flux Capacitor, scans the package saving the date and time, and loads it on a truck bound for delivery to you.
7. The driver arrives at your house, scans the package one last time, and hands it to you.

You can start building your time-traveling car!

From our example here, you can see how everyday actions create events, hence an event stream. The individual events here are the initial purchase, each time the package changes custody, and the final delivery. This scenario represents events generated by just one purchase. But if you think of the event streams generated by purchases from Amazon and the various shippers of the products, the number of events could easily number in the billions or trillions.

1.1.3 Who needs event streaming applications

Since everything in life can be considered an event, then pretty much any problem domain will benefit from using event streams. But there are some areas where it's more important to do so. Here are some typical examples

- *Credit card fraud* — A credit card owner may be unaware of unauthorized use. By reviewing purchases as they happen against established patterns (location, general spending habits), you may be able to detect a stolen credit card and alert the owner.
- *Intrusion detection* — The ability to monitor aberrant behavior in real-time is critical for the protection of sensitive data and well being of an organization.
- *The Internet of Things* - With IoT, there are sensors located in all kinds of places, and they all send back data very frequently. The ability to quickly capture this data and process it in a meaningful way is essential; anything less diminishes the effect of having these sensors deployed.
- *The financial industry* — The ability to track market prices and direction in real-time is essential for brokers and consumers to make effective decisions about when to sell or buy.
- *Sharing data in real-time* - Large organizations, like corporations or conglomerates, that have many applications need to share data in a standard, accurate, and real-time way

If the event-stream provides essential and actionable information, businesses and organizations need event-driven applications to capitalize on the information provided. In the next section, we'll break down the different components of the Kafka event streaming platform.

I've made a case for building event-streaming applications. But streaming applications aren't a fit for every situation.

Event-streaming applications become a necessity when you have data in different places or you have a large volume of events that you need to use distributed data stores to handle the volume. So if you can manage with a single database instance, then streaming is not a necessity. For example, a small e-commerce business or a local government website with mostly static data aren't good candidates for building an event-streaming solution.

1.2 Introducing the Apache Kafka® event streaming platform

The Kafka event streaming platform provides the core capabilities for you to implement your event streaming application from end-to-end. We can break down these capabilities into three main areas: publish/consume, durable storage, and processing. This move, store, and process trilogy enables Kafka to operate as the central nervous system for your data.

Before we go on, it will be useful to give you an illustration of what it means for Kafka to be the central nervous system for your data. We'll do this by showing before and after illustrations.

Let's first look at an event-streaming solution where each input source requires separate infrastructure:

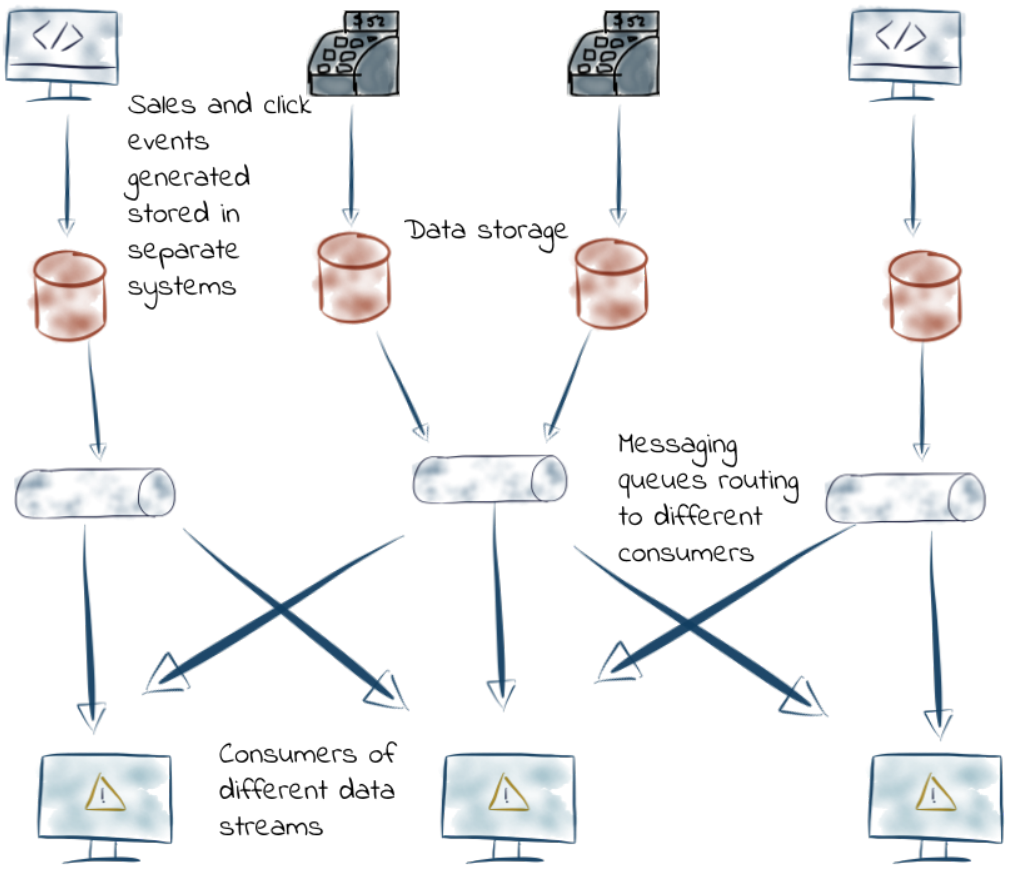


Figure 1.2 Initial event-streaming architecture leads to complexity as the different departments and data streams sources need to be aware of the other sources of events

In the above illustration, you have individual departments creating separate infrastructure to meet their requirements. But other departments may be interested in consuming the same data, which leads to a more complicated architecture to connect the various input streams.

Now let's take a look at how using the Kafka event streaming platform can change things.

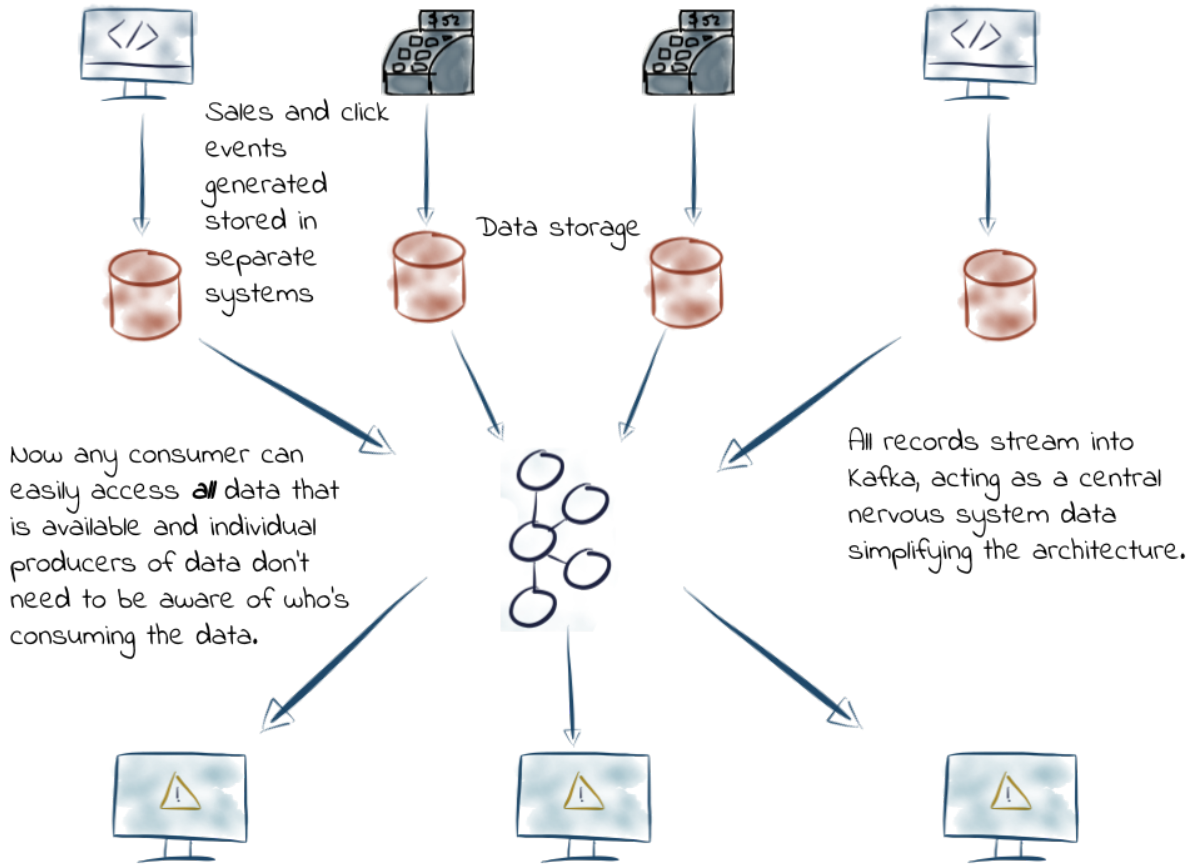


Figure 1.3 Using the Kafka event streaming platform the architecture is simplified

As you can see from this updated illustration, the architecture is greatly simplified with the Kafka event streaming platform's addition. All components now send their records to Kafka. Additionally, consumers read data from Kafka with no awareness of the producers.

At a high level, Kafka is a distributed system of servers and clients. The servers are called brokers, and the clients are record producers sending records to the brokers, and the consumer clients read records for the processing of events.

1.2.1 Kafka brokers

Kafka brokers durably ***store*** your records in contrast with traditional messaging systems (RabbitMQ or ActiveMQ) where the messages are ephemeral. The brokers store the data agnostically as the key-value pairs (and some other metadata fields) in byte format and are somewhat of a black box to the broker.

Providing storage of events has more profound implications as well concerning the difference between messages and events. You can think of messages as "tactical" communication between two machines, while events represent business-critical data that you don't want to throw away.

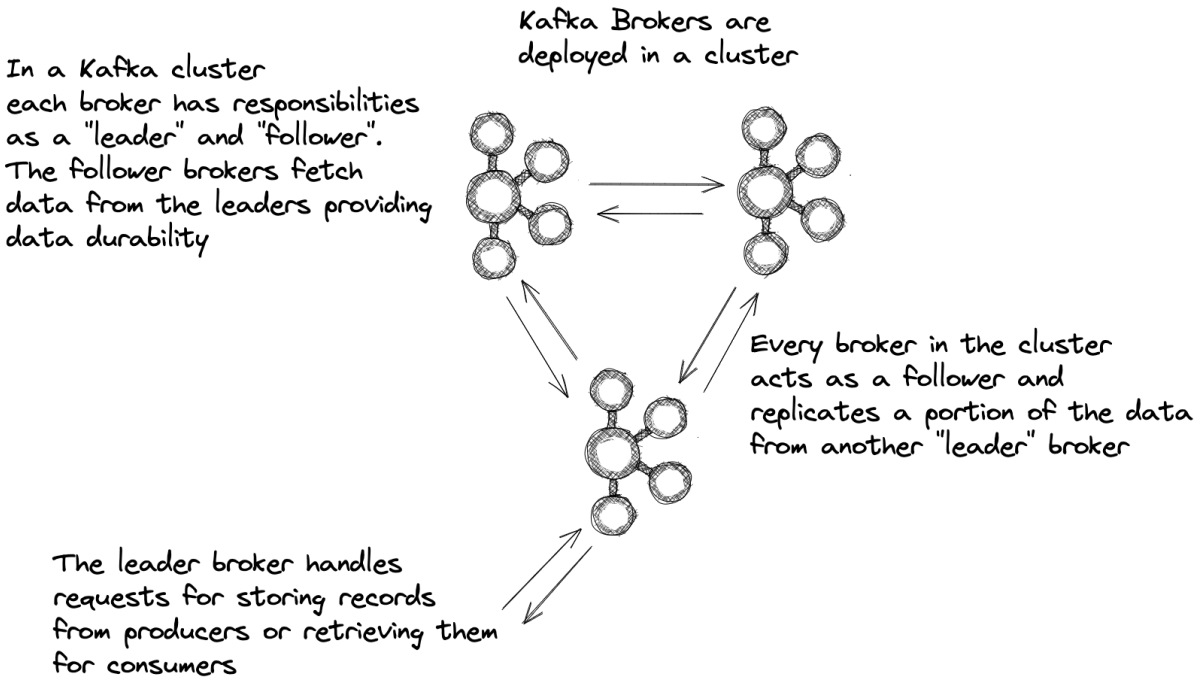


Figure 1.4 You deploy brokers in a cluster, and brokers replicate data for durable storage

From this illustration, you can see that Kafka brokers are the storage layer within the Kafka architecture and sit in the "storage" portion of the event-streaming trilogy. But in addition to acting as the storage layer, the brokers provide other essential functions such as serving requests from clients to providing coordination for consumers. We'll go into details of broker functionality in chapter 2.

1.2.2 Schema registry

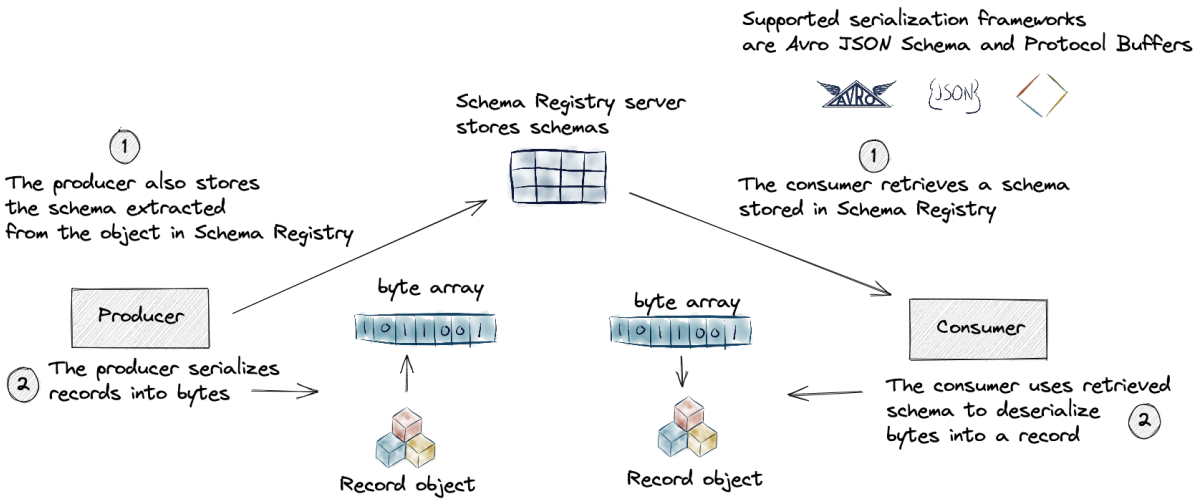


Figure 1.5 Schema registry enforces data modeling across the platform

Data governance is vital, to begin with, and its importance only increases as the size and diversity of an organization grows. Schema Registry stores schemas of the event records.

Schemas enforce a contract for data between producers and consumers. Schema Registry also provides serializers and deserializers supporting different tools that are Schema Registry aware. Providing (de)serializers means you don't have to write your serialization code. We'll cover Schema Registry in chapter 3.

1.2.3 Producer and consumer clients

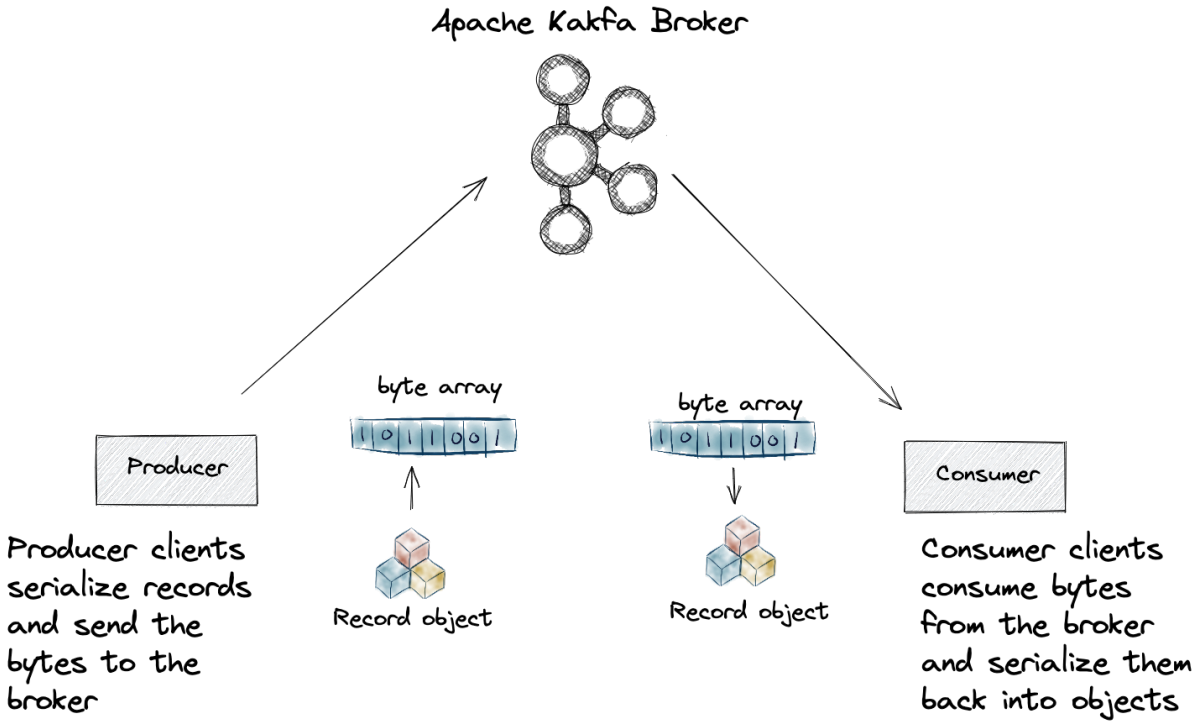


Figure 1.6 producers write records into Kafka, and consumers read records

The Producer client is responsible for sending records into Kafka. The consumer is responsible for reading records from Kafka. These two clients form the basic building blocks for creating an event-driven application and are agnostic to each other, allowing for greater scalability. The producer and consumer client also form the foundation for any higher-level abstraction working with Apache Kafka. We cover clients in chapter 4.

1.2.4 Kafka Connect

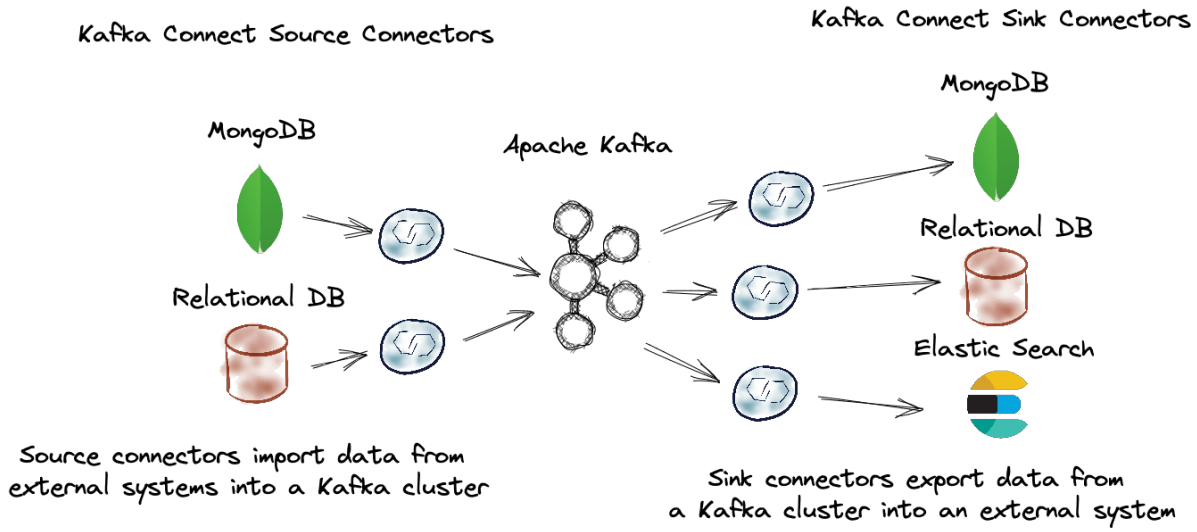


Figure 1.7 Kafka Connect bridges the gap between external systems and Apache Kafka

Kafka Connect provides an abstraction over the producer and consumer clients for importing data to and exporting data from Apache Kafka. Kafka connect is essential in connecting external data stores with Apache Kafka. It also provides an opportunity to perform light-weight transformations of data with Simple Messages Transforms when either exporting or importing data. We'll go into details of Kafka Connect in a later chapter.

1.2.5 Kafka Streams

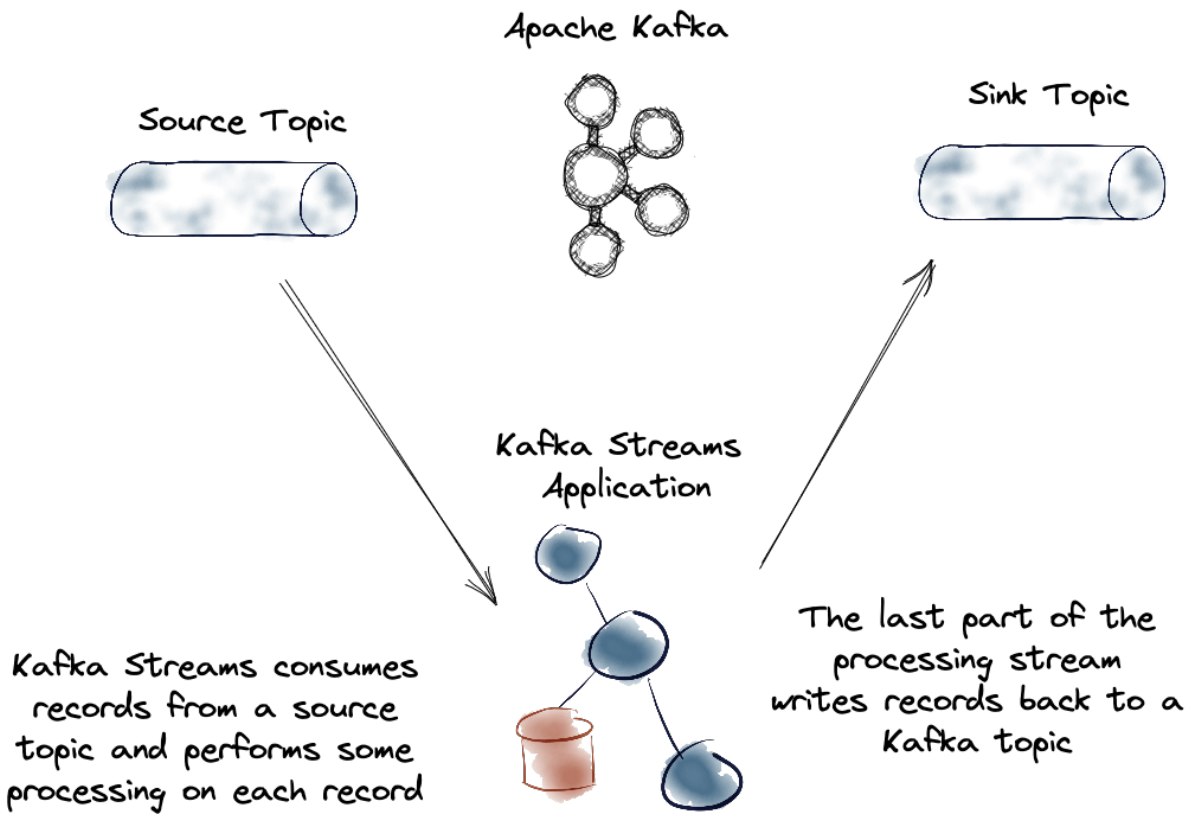


Figure 1.8 Kafka Streams is the stream processing API for Kafka

Kafka Streams is the native stream processing library for Kafka. Kafka Streams is written in the Java programming language and is used by client applications at the perimeter of a Kafka cluster; it is **not** run inside a Kafka broker. It provides support for performing operations on event data, including transformations, stateful operations like joins, and aggregations. Kafka Streams is where you'll do the heart of your work when dealing with events. Chapters 6, 7, and 8 cover Kafka Streams in detail.

1.2.6 ksqlDB

ksqlDB is an event streaming database. It does this by applying a SQL interface for event stream processing. Under the covers, ksqlDB uses Kafka Streams for performing its event streaming tasks. A key advantage of ksqlDB is that it allows you to specify your event streaming operation in SQL; no code is required. We'll discuss ksqlDB in chapters 8 and 9.

```
CREATE TABLE activePromotions AS
  SELECT rideId,
         qualifyPromotion(kmToDst) AS promotion
  FROM locations
  GROUP BY rideId
  EMIT CHANGES;


  SELECT rideId, promotion
  FROM activePromotions
  WHERE ROWKEY = '6fd0fcdb';
```

Figure 1.9 ksqlDB provides streaming database capabilities

Now that we've gone over how the Kafka event streaming platform works, including the individual components, let's apply a concrete example of a retail operation demonstrating how the Kafka event streaming platform works.

1.3 A concrete example of applying the Kafka event streaming platform

Let's say there is a consumer named Jane Doe, and she checks her email. There's one email from ZMart with a link to a page on the ZMart website containing coupons for 15% off the total purchase price. Once on the web page, Jane clicks another link to activate the coupons and print them out. While this whole sequence is just another online purchase for Jane, it represents clickstream events for ZMart.

Let's take a moment here to pause our scenario so we discuss the relationship between these simple events and how they interact with the Kafka event streaming platform.

The data generated by the initial clicks to navigate to and print the coupons create clickstream information captured and produced directly into Kafka with a producer microservice. The marketing department started a new campaign and wants to measure its effectiveness, so the clickstream events available at this point are valuable.

The first sign of a successful project is that users click on the email links to retrieve the coupons. Additionally, the data science group is interested in the pre-purchase clickstream data as well. The data science team can track customers' initial actions and later attribute purchases to those initial clicks and marketing campaigns. The amount of data from this single activity may seem

small. When you factor in a large customer base and several different marketing campaigns, you end up with a significant amount of data.

Now let's resume our shopping example.

It's late summer, and Jane has been meaning to get out shopping to get her children some back-to-school supplies. Since tonight is a rare night with no family activities, Jane decides to stop off at ZMart on her way home.

Walking through the store after grabbing everything she needs, Jane walks by the footwear section and notices some new designer shoes that would go great with her new suit. She realizes that's not what she came in for, but what the heck life is short (ZMart thrives on impulse purchases!), so Jane gets the shoes.

As Jane approaches the self-checkout aisle, she first scans her ZMart member card. After scanning all the items, she scans the coupon, which reduces the purchase by 15%. Then Jane pays for the transaction with her debit card, takes the receipt, and walks out of the store. A little later that evening, Jane checks her email, and there's a message from ZMart thanking her for her patronage, with coupons for discounts on a new line of designer clothes.

Let's dissect the purchase transaction and see this one event triggers a sequence of operations performed by the Kafka event streaming platform.

So now ZMart's sales data streams into Kafka. In this case, ZMart uses Kafka Connect to create a source connector to capture the sales as they occur and send them into Kafka. The sale transaction brings us to the first requirement, the protection of customer data. In this case, ZMart uses an SMT or Simple Message Transform to mask the credit card data as it goes into Kafka.

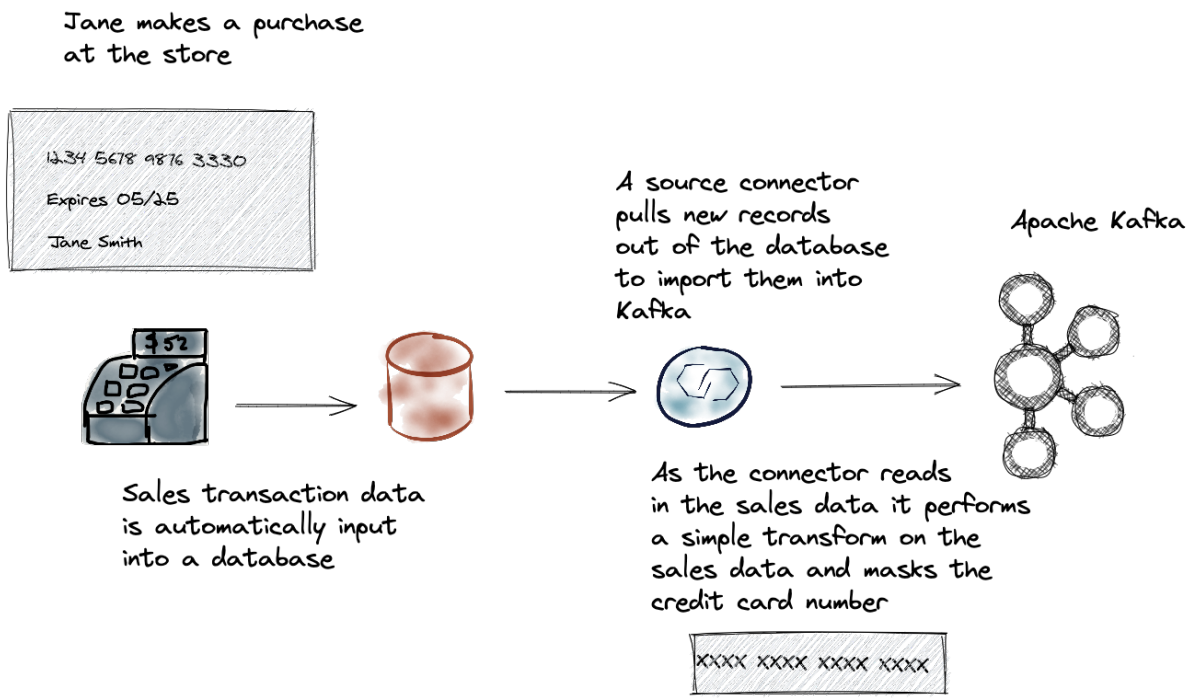


Figure 1.10 Sending all of the sales data directly into Kafka with connect masking the credit card numbers as part of the process

As connect writes records into Kafka, they are immediately consumed by different organizations within ZMart. The department in charge of promotions created an application for consuming sales data for assigning purchase rewards if they are a member of the loyalty club. If the customer reaches a threshold for earning a bonus, an email with a coupon goes out to the customer.

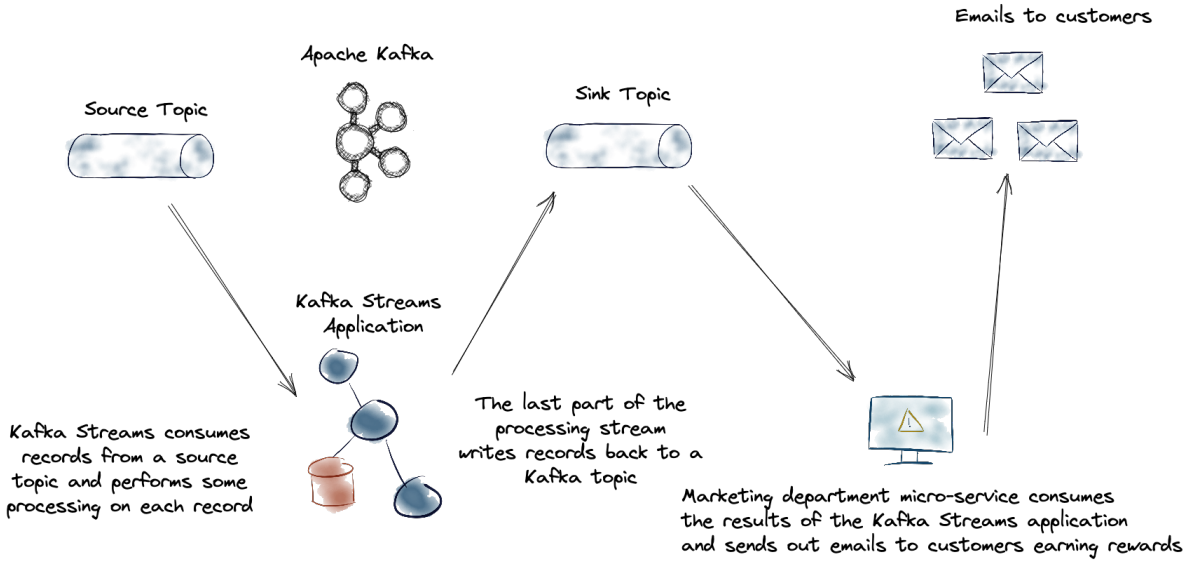


Figure 1.11 Marketing department application for processing customer points and sending out earned emails

It's important to note that ZMart processes sales records immediately after the sale. So customers

get timely emails with their rewards within a few minutes of completing their purchases. By acting on the purchase events as they happen allows ZMart a quick response time to offer customer bonuses.

The Data Science group within ZMart uses the sales data topic as well. The DS group uses a Kafka Streams application to process the sales data building up purchase patterns of what customers in different locations are purchasing the most. The Kafka Streams application crunches the data in real-time and sends the results out to a sales-trends topic.

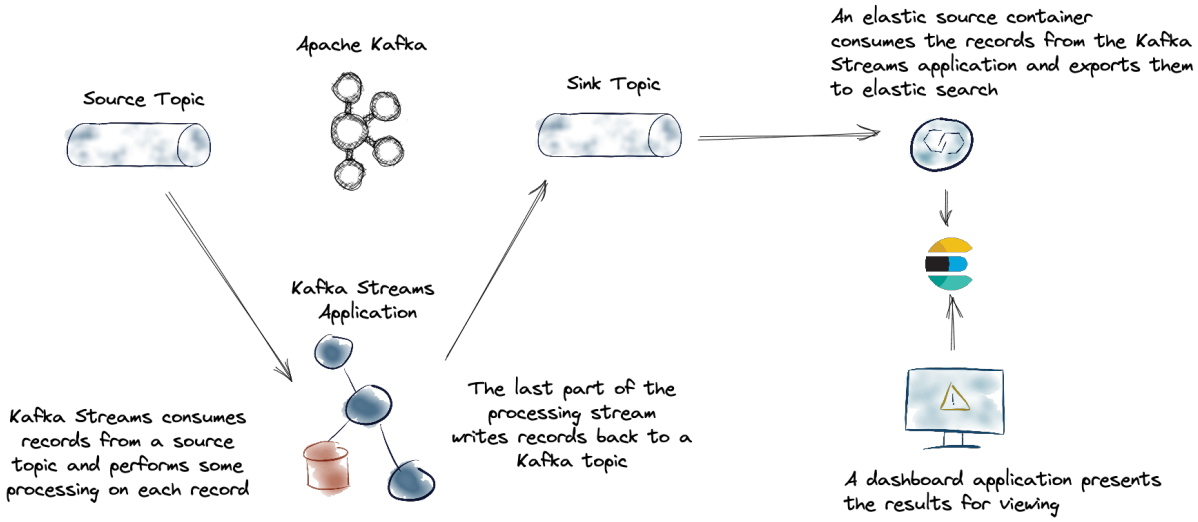


Figure 1.12 Kafka Streams application crunching sales data and connect exporting the data for a dashboard application

ZMart uses another Kafka connector to export the sales trends to an external application that publishes the results in a dashboard application. Another group also consumes from the sales topic to keep track of inventory and order new items if they drop below a given threshold, signaling the need to order more of that product.

At this point, you can see how ZMart leverages the Kafka platform. It is important to remember that with an event streaming approach, ZMart responds to data as it arrives, allowing them to make quick and efficient decisions immediately. Also, note how you write into Kafka *once*, yet multiple groups consume it at different times, independently in a way that one group's activity doesn't impede another's.

In this book, you'll learn what event-stream development is, why it's essential, and how to use the Kafka event streaming platform to build robust and responsive applications. From extract, transform, and load (ETL) applications to advanced stateful applications requiring complex transformations, we'll cover the Kafka streaming platform's components so you can solve the kinds of challenges presented earlier with an event-streaming approach. This book is suitable for any developer looking to get into building event streaming applications.

1.4 Summary

- Event streaming is capturing events generated from different sources like mobile devices, customer interaction with websites, online activity, shipment tracking, and business transactions. Event streaming is analogous to our nervous system.
- An event is "something that happens," and the ability to react immediately and review later is an essential concept of an event streaming platform
- Kafka acts as a central nervous system for your data and simplifies your event stream processing architecture
- The Kafka event streaming platform provides the core capabilities for you to implement your event streaming application from end-to-end by delivering the three main components of publish/consume, durable storage, and processing.
- Kafka broker are the storage layer and service requests from clients for writing and reading records. The brokers store records as bytes and do no touch or alter the contents.
- Schema Registry provides a way to ensure compatibility of records between producers and consumers.
- Producer clients write (produce) records to the broker. Consumer clients consume records from the broker. The producer and consumer clients are agnostic of each other. Additionally, the Kafka broker doesn’t have any knowledge of who the individual clients are, they just process the requests.
- Kafka Connect provides a mechanism for integrating existing systems such as external storage for getting data into and out of Kafka.
- Kafka Streams is the native stream processing library for Kafka. It runs at the perimeter of a Kafka cluster, not inside the brokers and provides support for transforming data including joins and stateful transformations.
- ksqlDB is an event streaming database for Kafka. It allows you to build powerful real-time systems with just a few lines of SQL.

2 Kafka brokers

This chapter covers

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

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

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

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

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

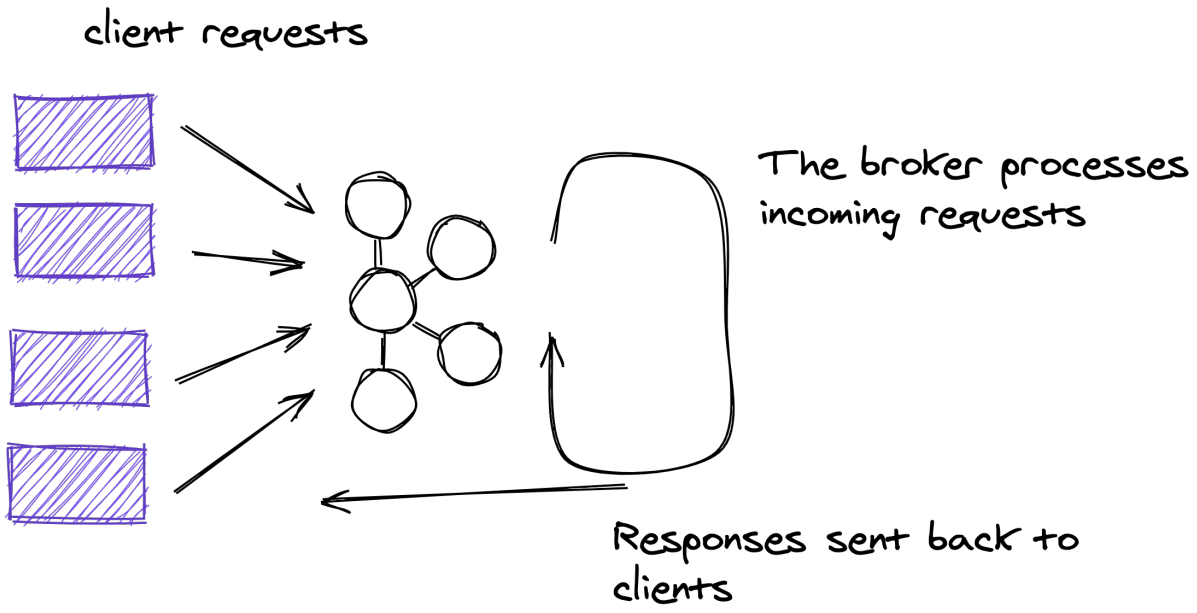


Figure 2.1 Clients communicating with brokers

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

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

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

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

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

2.1 Produce record requests

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

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

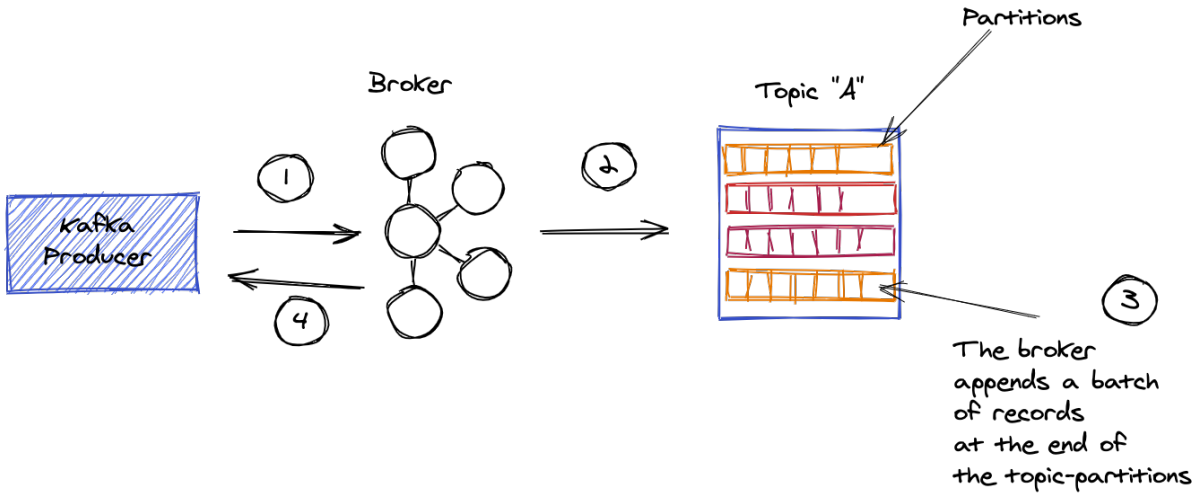


Figure 2.2 Brokers handling produce records request

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

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

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

2.2 Consume record requests

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

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

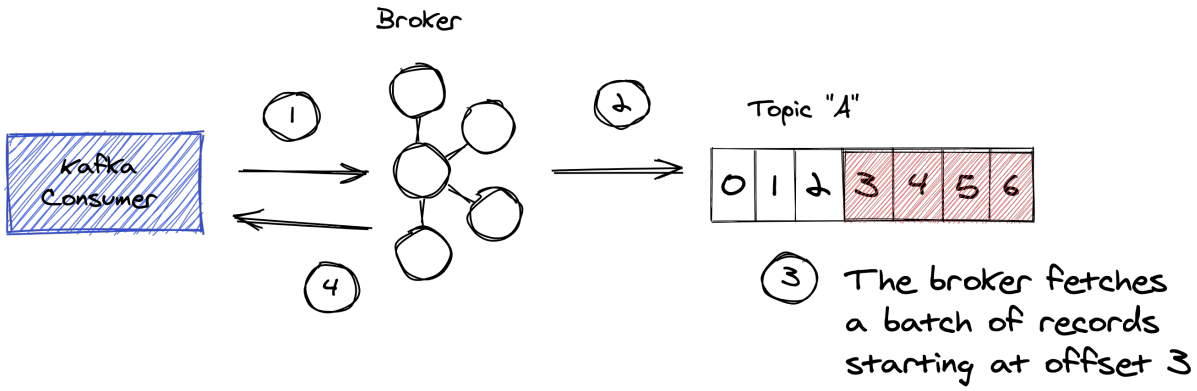


Figure 2.3 Brokers handling requests from a consumer

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

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

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

2.3 Topics and partitions

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

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

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

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

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

Listing 2.1 Topic directory structure example

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

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

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

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

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

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

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

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

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

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

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

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

2.3.1 Offsets

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

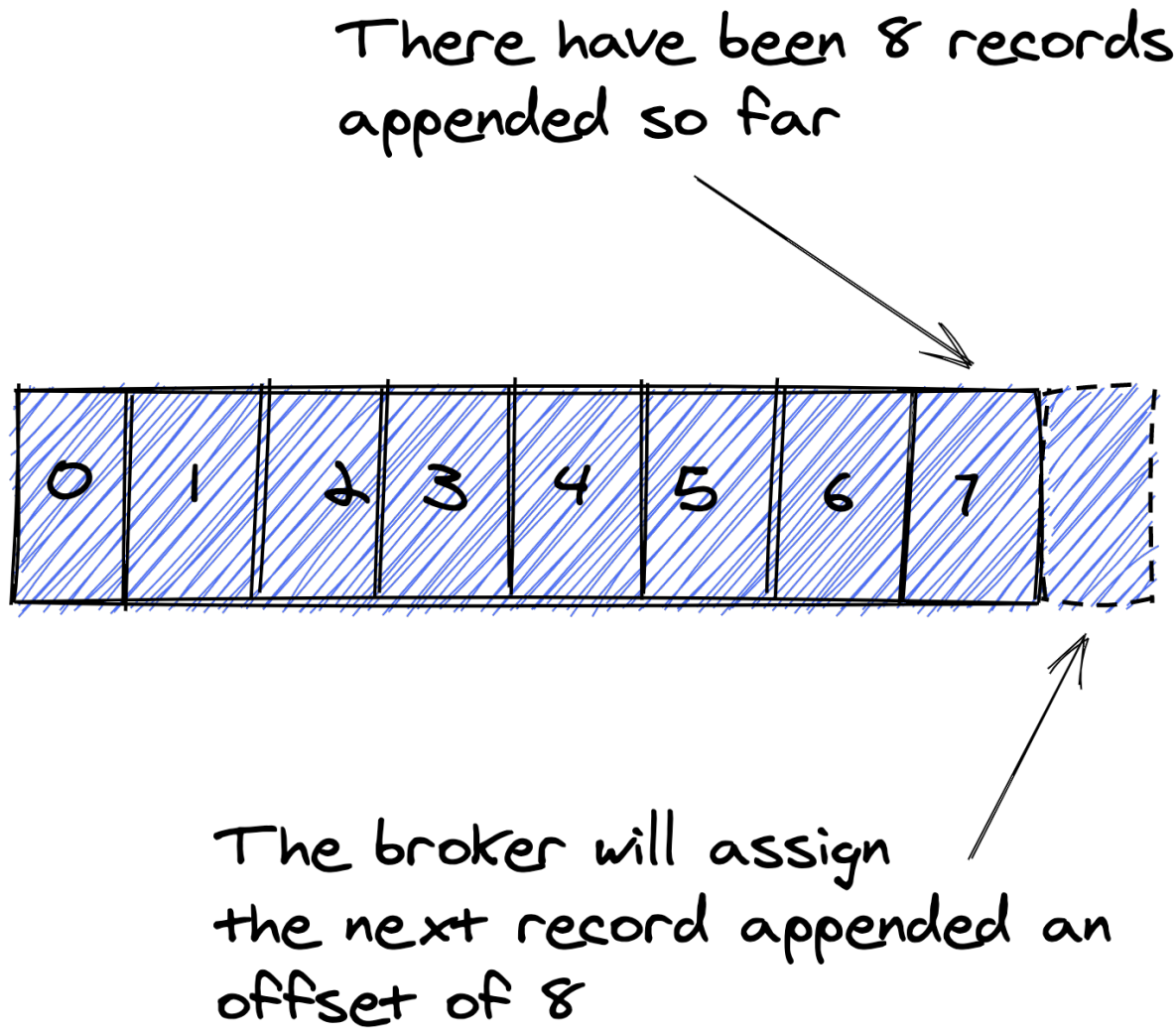


Figure 2.4 Assigning the offset to incoming records

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

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

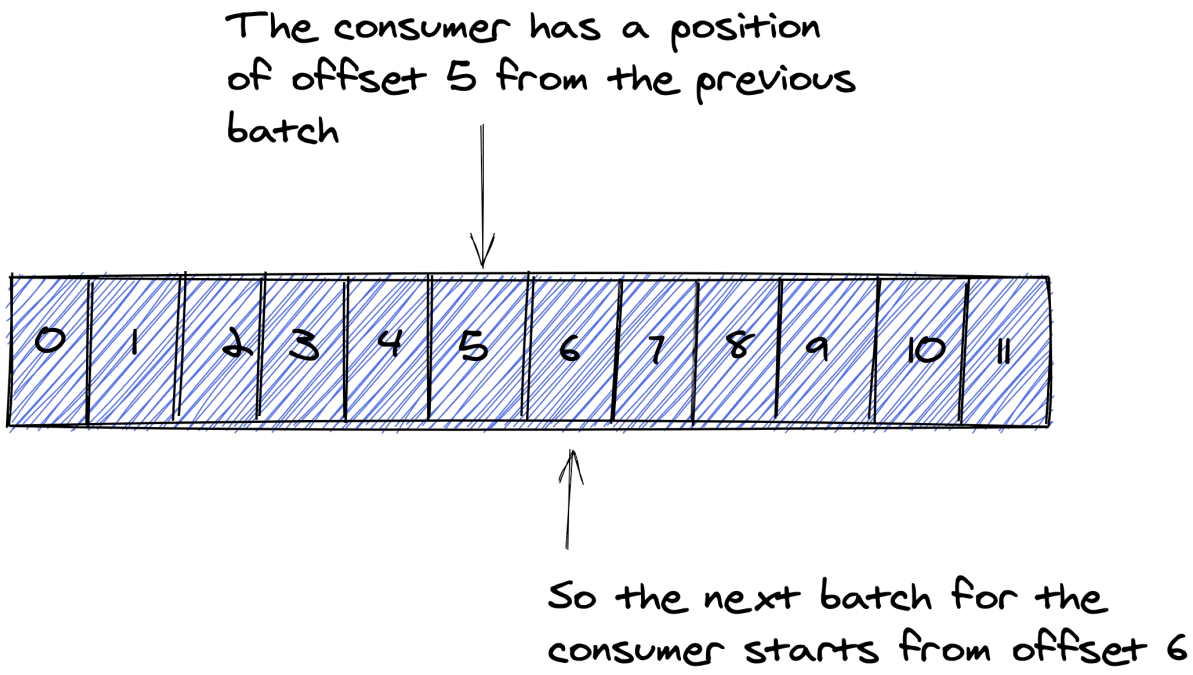


Figure 2.5 Offsets indicate where a consumer has left off reading records

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

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

2.3.2 Determining the correct number of partitions

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

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

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

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

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

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

2.4 Sending your first messages

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

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

2.4.1 Creating a topic

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

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

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

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

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

❶ The host:port to connect to the broker

❷ Specifying the replication factor

❸ The number of partitions

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

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

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

2.4.2 Producing records on the command line

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

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

❶ The topic you created in the previous step

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

❸ Specifying that you'll provide a key

❹ Specifying the separator of the key and value

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

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

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

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

2.4.3 Consuming records from the command line

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

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

❶ Specifying the topic to consume from

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

❸ Start consuming from the head of the log

❹ Print the keys

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

You should see the following output on your console:

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

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

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

2.4.4 Partitions in action

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

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

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

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

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

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

❶ Specifying the partition we'll consume from

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

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

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

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

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

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

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

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

You should see the following results:

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

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

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

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

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

2.5 Segments

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

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

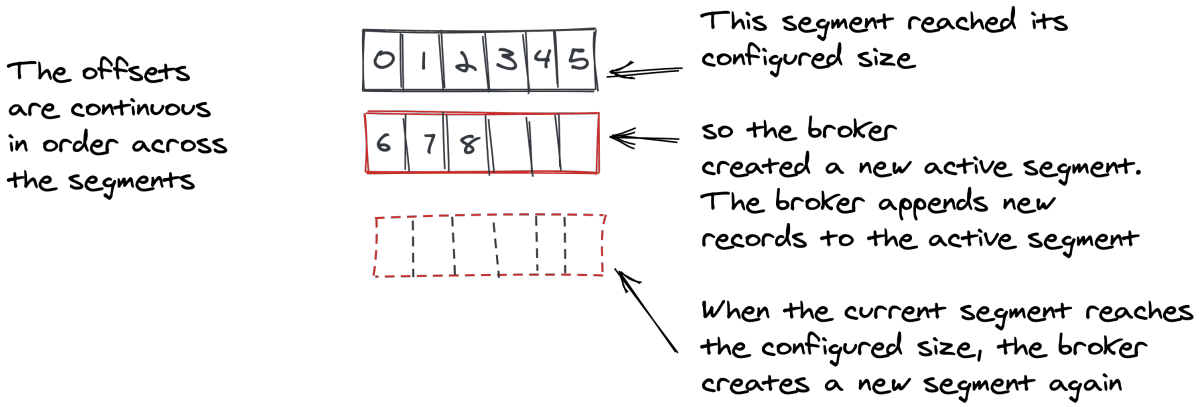


Figure 2.6 Creating new segments

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

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

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

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

2.5.1 Data retention

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

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

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

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

2.5.2 Compacted topics

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

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

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

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

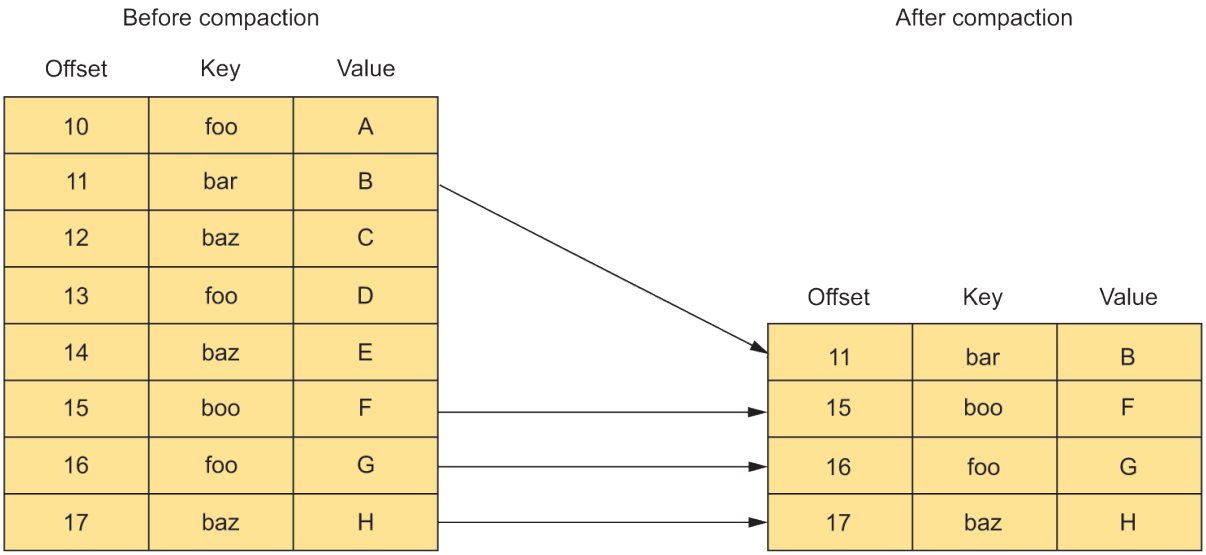


Figure 2.7 On the left is a log before compaction—you'll notice duplicate keys with different values. These duplicates are updates. On the right is after compaction—retaining the latest value for each key, but it's smaller in size.

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

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

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

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

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

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

2.5.3 Topic partition directory contents

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

Listing 2.2 Contents of topic-partition directory

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

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

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

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

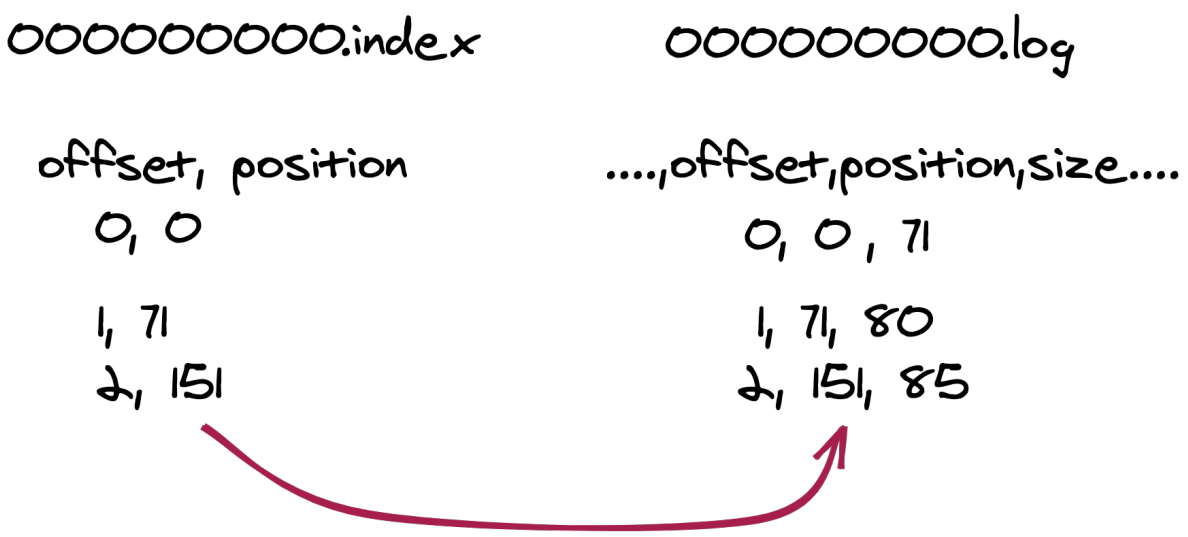


Figure 2.8 Searching for start point based on offset 2

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

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

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

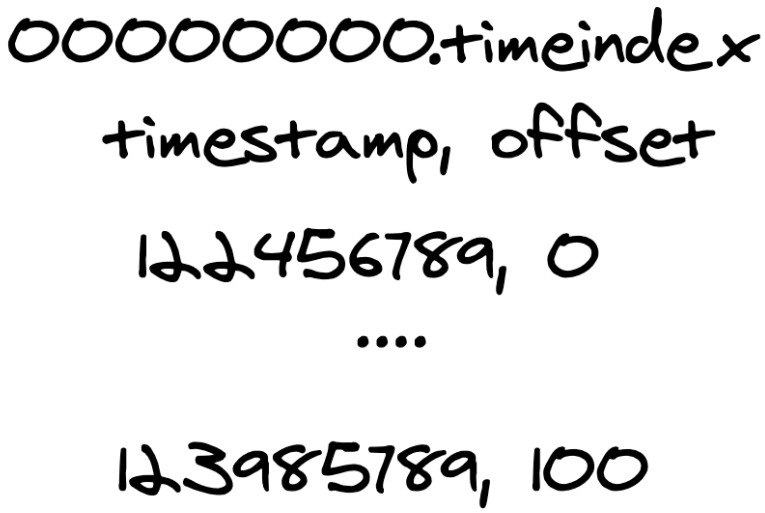


Figure 2.9 Timeindex file

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

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

Listing 2.3 Contents of the directory after creating a new segment

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

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

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

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

2.6 Tiered storage

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

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

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

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

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

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

2.7 Cluster Metadata

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

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

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

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

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

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

2.8 Leaders and followers

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

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

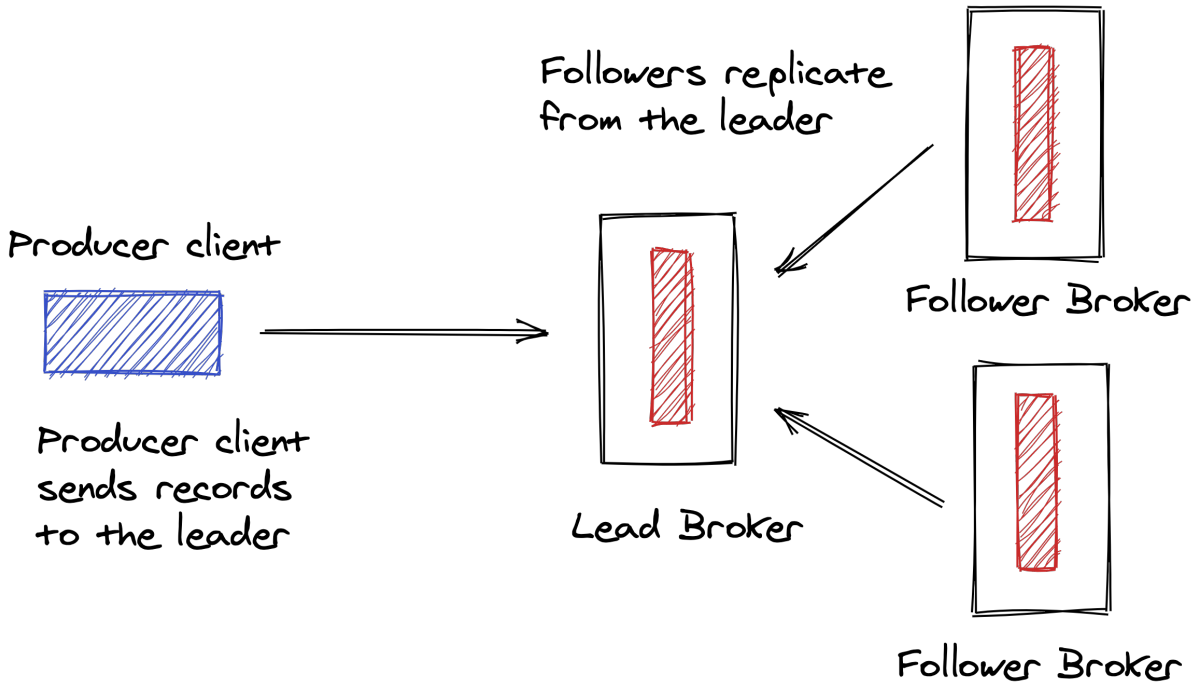


Figure 2.10 Leader and follower example

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

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

2.8.1 Replication

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

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

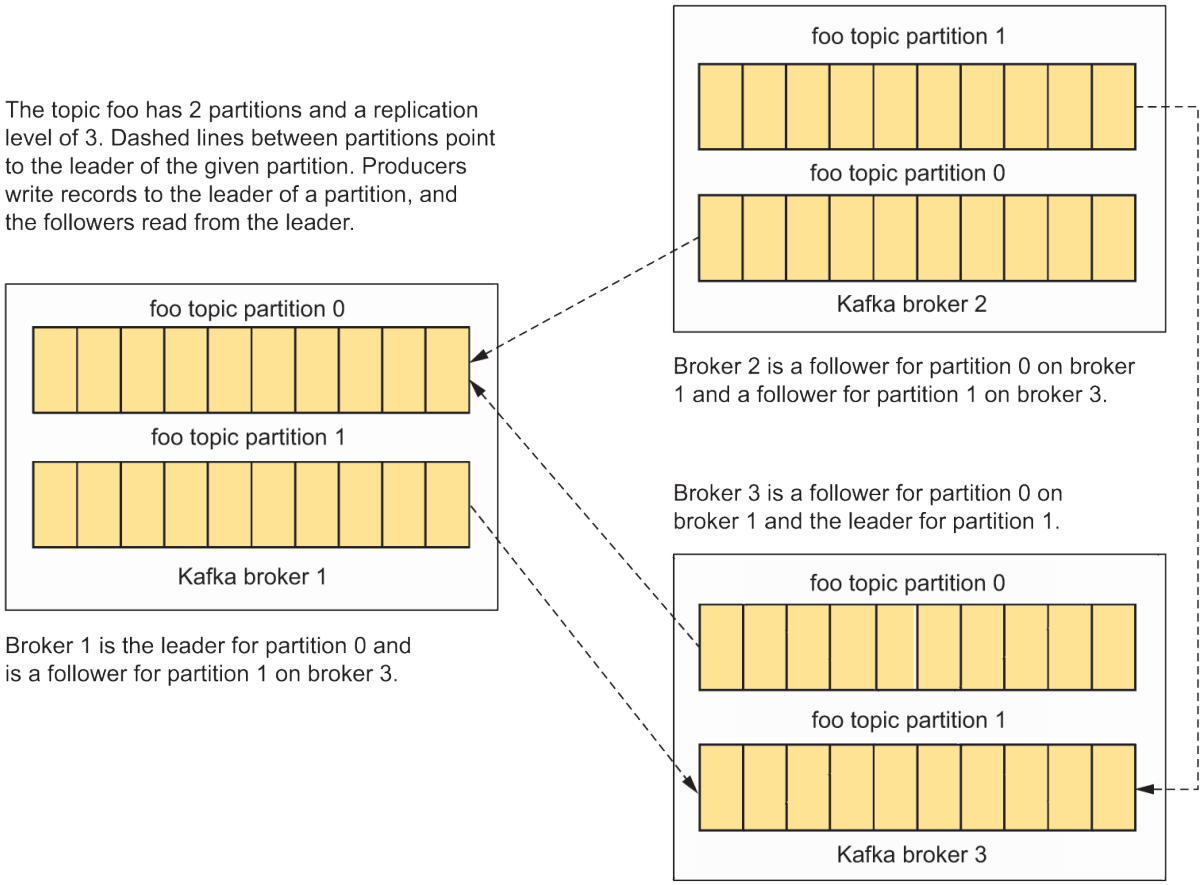


Figure 2.11 The Kafka replication process

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

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

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

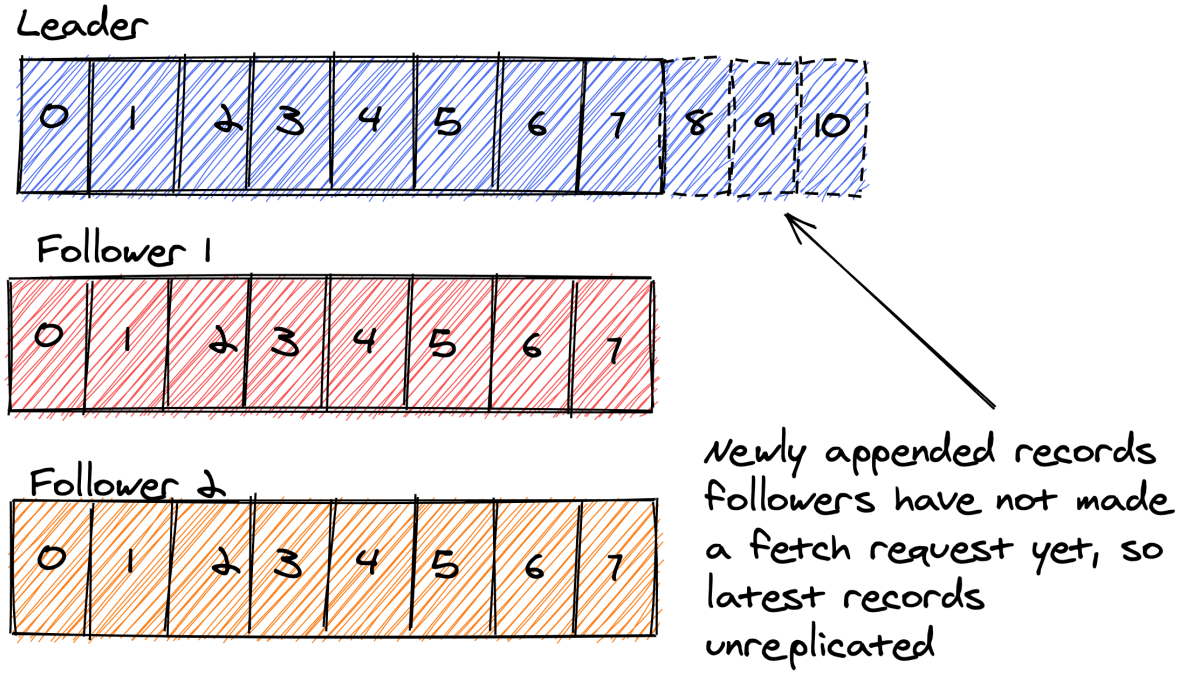


Figure 2.12 The leader may have a few unreplicated messages in its topic-partition

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

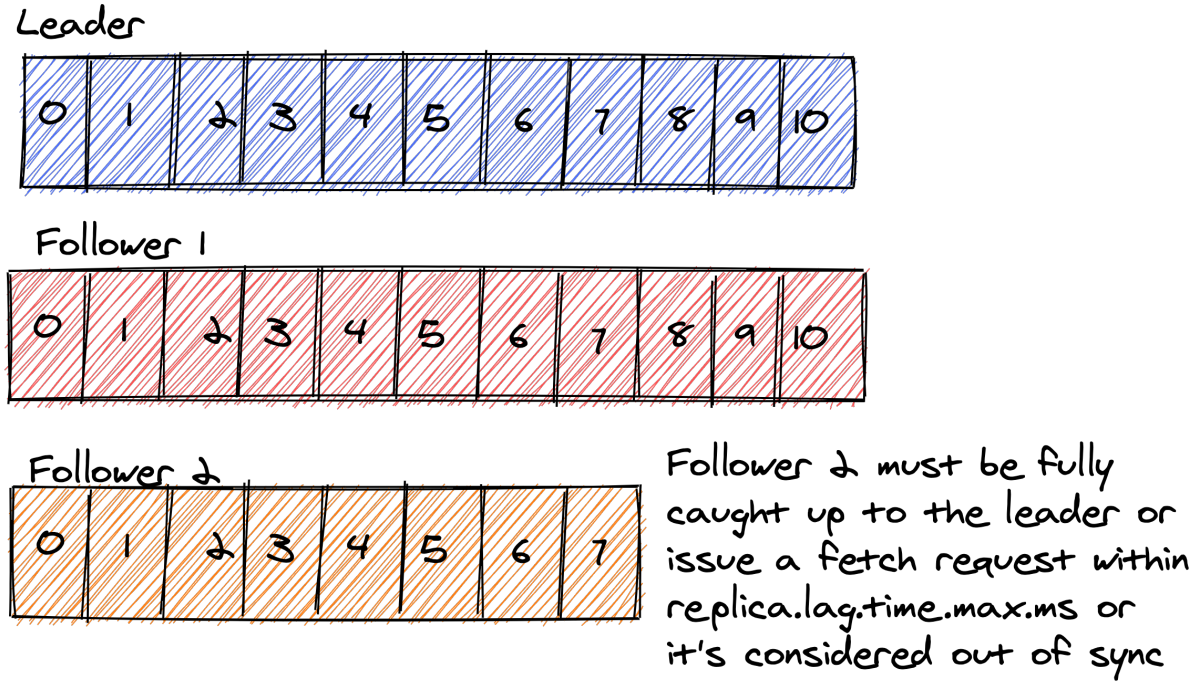


Figure 2.13 Followers must issue a fetch request or be caught up withing lag time configuration

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

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

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

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

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

REPLICATION AND ACKNOWLEDGMENTS

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

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

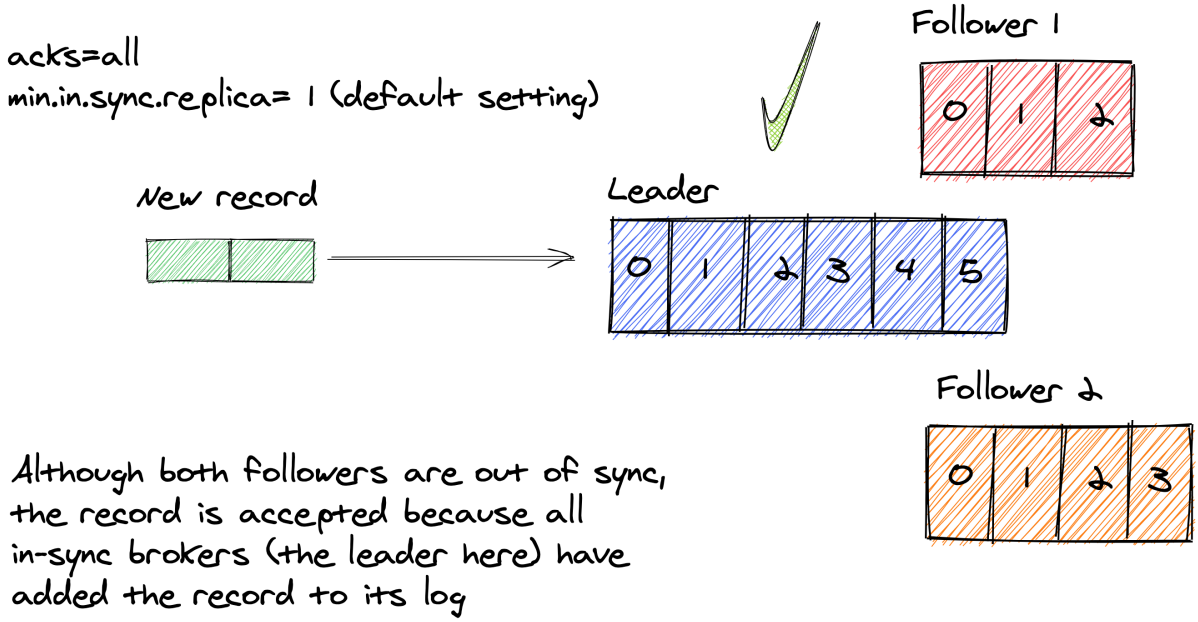


Figure 2.14 Acks set to "all" with default in-sync replicas

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

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

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

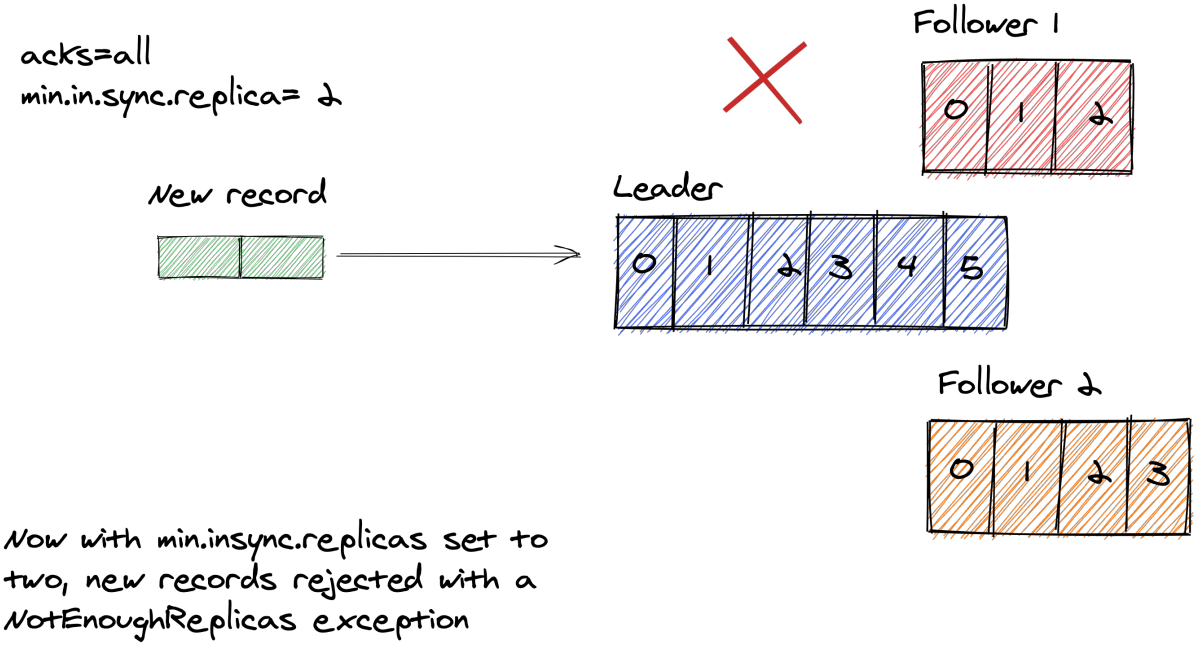


Figure 2.15 Setting Min ISR to a value greater than one increases data durability

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

2.9 Checking for a healthy broker

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

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

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

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

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

2.9.1 Request handler idle percentage

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

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

2.9.2 Network handler idle percentage

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

2.9.3 Under replicated partitions

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

2.10 Summary

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

3 Schema registry

This chapter covers

- Using bytes means serialization rules
- What is a schema and why you need to use one
- What is Schema Registry?
- Ensuring compatibility with changes - schema evolution
- Understanding subject names
- Reusing schemas with references

In chapter 2, you learned about the heart of the Kafka streaming platform, the Kafka broker. In particular, you learned how the broker is the storage layer appending incoming messages to a topic, serving as an immutable, distributed log of events. A topic represents the directory containing the log file(s).

Since the producers send messages over the network, they need to be serialized first into binary format, in other words an array of bytes. The Kafka broker does not change the messages in any way, it stores them in the same format. It's the same when the broker responds to fetch requests from consumers, it retrieves the already serialized messages and sends them over the network.

By only working with messages as arrays of bytes, the broker is completely agnostic to the data type the messages represent and completely independent of the applications that are producing and consuming the messages and the programming languages those applications use. By decoupling the broker from the data format, any client using the Kafka protocol can produce or consume messages.

While bytes are great for storage and transport over the network, developers are far more efficient working at a higher level of abstraction; the object. So where does this transformation

from object to bytes and bytes to object occur then? At the client level in the producers and consumers of messages.

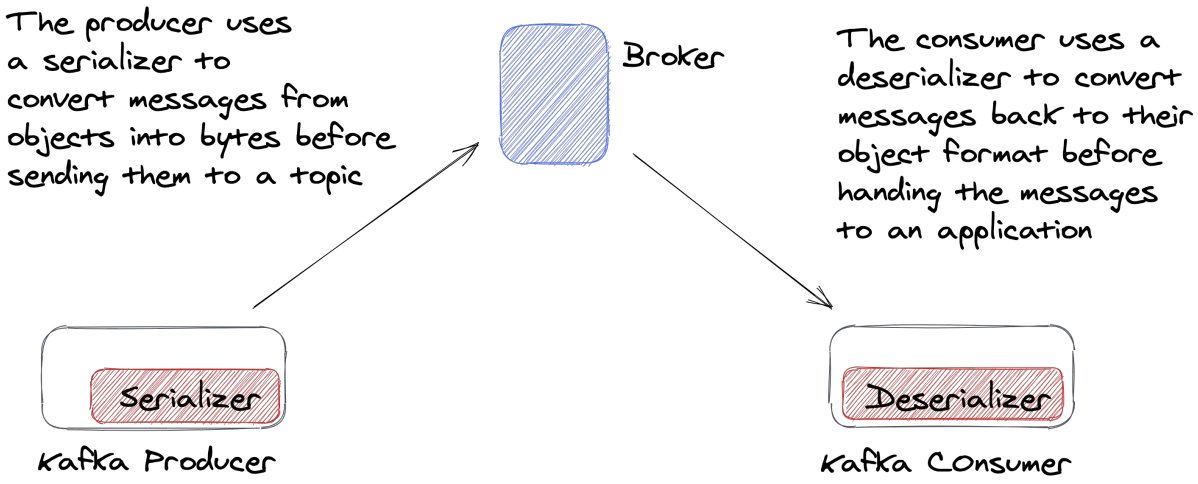


Figure 3.1 The conversion of objects to bytes and bytes to objects happens at the client level

Looking at this illustration, the message producer uses an instance of a `Serializer` to convert the message object into bytes before sending it to the topic on the broker. The message consumer does the opposite process, it receives bytes from the topic, and uses an instance of a `Deserializer` to convert the bytes back into the same object format.

The producer and consumer are decoupled from the (de)serializers; they simply call either the `serialize` or `deserialize` methods.

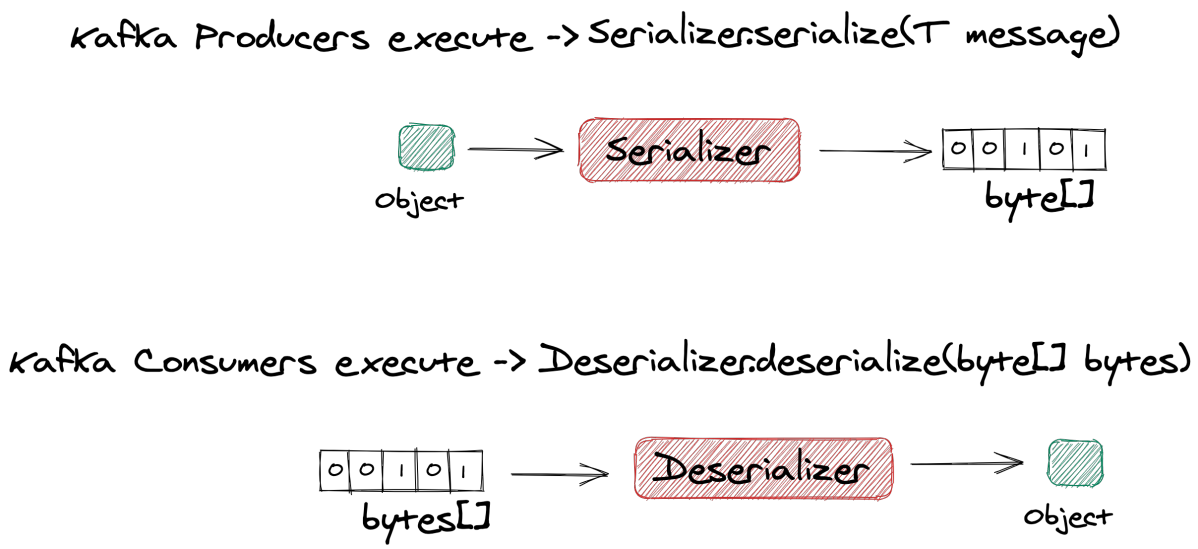


Figure 3.2 The serializer and deserializer are agnostic of the producer and consumer and perform the expected action when the serialize and deserialize methods are called

As depicted in this illustration, the producer expects to use an instance of the `Serializer` interface and just calls the `Serializer.serialize` method passing in an object of a given type

and getting back bytes. The consumer works with the `Deserializer` interface. The consumer provides an array of bytes to the `Deserializer.deserialize` method and receives an object of a given type in return.

The producer and consumer get the (de)serializers via configuration parameters and we'll see examples of this later in the chapter.

> **NOTE** I'm mentioning producers and consumers here and throughout the chapter, but we'll only go into enough detail to understand the context required for this chapter. We'll cover producer and consumer client details in the next chapter.

The point I'm trying to emphasize here is that for a given topic the object type the producer serializes is expected to be the exact same object type that a consumer deserializes. Since producers and consumers are completely agnostic of each other ***these messages or event domain objects represent an implicit contract between the producers and consumers***.

So now the question is does something exist that developers of producers and consumers can use that informs them of the proper structure of messages? The answer to that question is yes, the schema.

3.1 What is a schema and why you need to use one

When you mention the word schema to developers, there's a good chance their first thought is of database schemas. A database schema describes the structure of the database, including the names and startups of the columns in database tables and the relationship between tables. But the schema I'm referring to here, while similar in purpose, is not quite the same thing.

For our purposes what I'm referring to is a ***language agnostic description of an object, including the name, the fields on the object and the type of each field***. Here's an example of a potential schema in json format

Listing 3.1 Basic example of a schema in json format

```
{
"name":"Person",     ❶
  "fields": [        ❷
    {"name": "name", "type":"string"},   ❸
    {"name": "age", "type": "int"},
    {"name": "email", "type":"string"}
  ]
}
```

❶ The name of the object

❷ Defining the fields on the object

❸ The names of the fields and their types

Here our fictional schema describes an object named `Person` with fields we'd expect to find on such an object. Now we have a structured description of an object that producers and consumers can use as an agreement or contract on what the object should look like before and after serialization. I'll cover details on how you use schemas in message construction and (de)serialization in an upcoming section.

But for now I'd like review some key points we've established so far:

- The Kafka broker only works with messages in binary format (byte arrays)
- Kafka producers and consumers are responsible for the (de)serialization of messages. Additionally, since these two are unaware of each other, the records form a contract between them.

And we also learned that we can make the contract between producers and consumers explicit by using a schema. So we have our *why* for using a schema, but what we've defined so far is a bit abstract and we need to answer these questions for the *how* :

- How do you put schemas to use in your application development lifecyle?
- Given that serialization and deserialization is decoupled from the Kafka producers and consumers how can they use serialization that ensures messages are in the correct format?
- How do you enforce the correct version of a schema to use? After all changes are inevitable

The answer to these *how* questions is Schema Registry.

3.1.1 What is Schema Registry?

Schema Registry provides a centralized application for storing schemas, schema validation and sane schema evolution (message structure changes) procedures. Perhaps more importantly, it serves as the source of truth of schemas that producer and consumer clients can easily discover. Schema Registry provides serializers and deserializers that you can configure Kafka Producers and Kafka Consumers easing the development for applications working with Kafka.

The Schema Registry serializing code supports schemas from the serialization frameworks Avro (avro.apache.org/docs/current/) and Protocol Buffers (developers.google.com/protocol-buffers). Note that I'll refer to Protocol Buffers as "Protobuf" going forward. Additionally Schema Registry supports schemas written using the JSON Schema (json-schema.org/), but this is more of a specification vs a framework. I'll get into working with Avro, Protobuf JSON Schema as we progress through the chapter, but for now let's take a high-level view of how Schema Registry works:

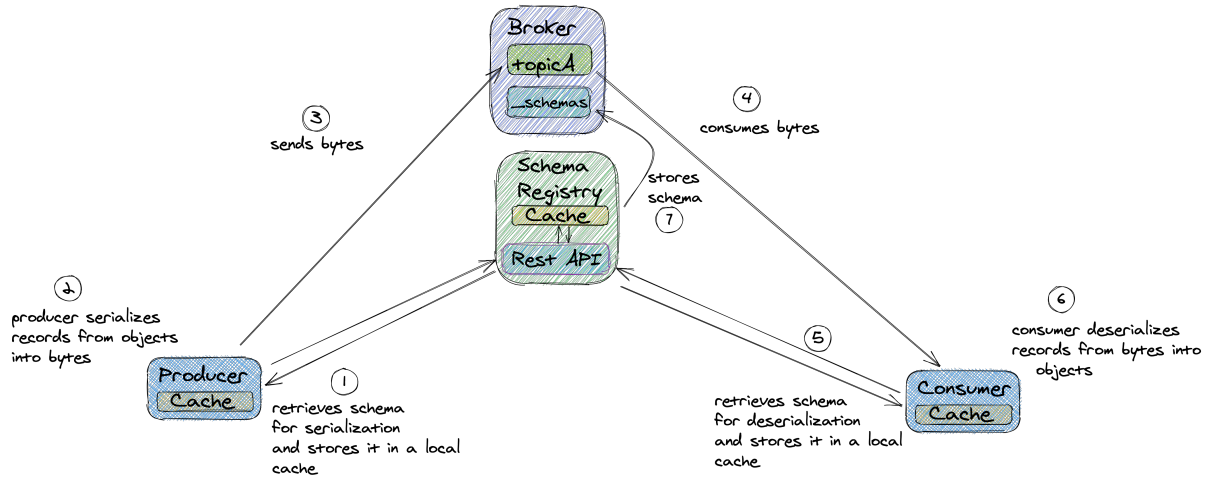


Figure 3.3 Schema registry ensures consistent data format between producers and consumers

Let's quickly walk through how Schema Registry works based on this illustration

1. As a produce calls the `serialize` method, a Schema Registry aware serializer retrieves the schema (via HTTP) and stores it in its local cache
2. The serializer embedded in the producer serializes the record
3. The producer sends the serialized message (bytes) to Kafka
4. A consumer reads in the bytes
5. The Schema Registry aware deserializer in the consumer retrieves the schema and stores it in its local cache
6. The consumer deserializes the the bytes based on the schema
7. The Schema Registry servers produces a message with the schema so that it's stored in the `__schemas` topic

TIP **While I'm presenting Schema Registry as an important part of the Kafka event streaming platform, it's not required. Remember Kafka producers and consumers are decoupled from the serializers and deserializers they use. As long as you provide a class that implements the appropriate interface, they'll work fine with the producer or consumer. But you will lose the validation checks that come from using Schema Registry. I'll cover serializing without Schema Registry at the end of this chapter.**

While the previous illustration gave you a good idea of how schema registry works, there's an important detail I'd like to point out here. While it's true that the serializer or deserializer will reach out to Schema Registry to retrieve a schema for a given record type, it only does so ***once***, the first time it encounters a record type it doesn't have the schema for. After that, the schema needed for (de)serialization operations is retrieved from local cache.

3.1.2 Getting Schema Registry

Our first step is to get Schema Registry up and running. Again you'll use docker-compose to speed up your learning and development process. We'll cover installing Schema Registry from a binary download and other options in an appendix. But for now just grab the `docker-compose.yml` file from the chapter_3 directory in the source code for the book.

This file is very similar to the `docker-compose.yml` file you used in chapter two. But in addition to the Zookeeper and Kafka images, there is an entry for a Schema Registry image as well. Go ahead and run `docker-compose up -d`. To refresh your memory about the docker commands the `-d` is for "detached" mode meaning the docker containers run in the background freeing up the terminal window you've executed the command in.

3.1.3 Architecture

Before we go into the details of how you work with Schema Registry, it would be good to get high level view of how it's designed. Schema Registry is a distributed application that lives outside the Kafka brokers. Clients communicate with Schema Registry via a REST API. A client could be a serializer (producer), deserializer (consumer), a build tool plugin, or a command line request using curl. I'll cover using build tool plugins, gradle in this case, in an upcoming section soon.

Schema Registry uses Kafka as storage (write-ahead-log) of all its schemas in `__schemas` which is a single partitioned, compacted topic. It has a primary architecture meaning there is one leader node in the deployment and the other nodes are secondary.

> **NOTE** **The double underscore characters are a Kafka topic naming convention denoting internal topics not meant for public consumption. From this point forward we'll refer to this topic simply as `schemas`.**

What this means is that only the primary node in the deployment writes to the schemas topic. Any node in the deployment will accept a request to store or update a schema, but secondary nodes forward the request to the primary node. Let's look at an illustration to demonstrate:

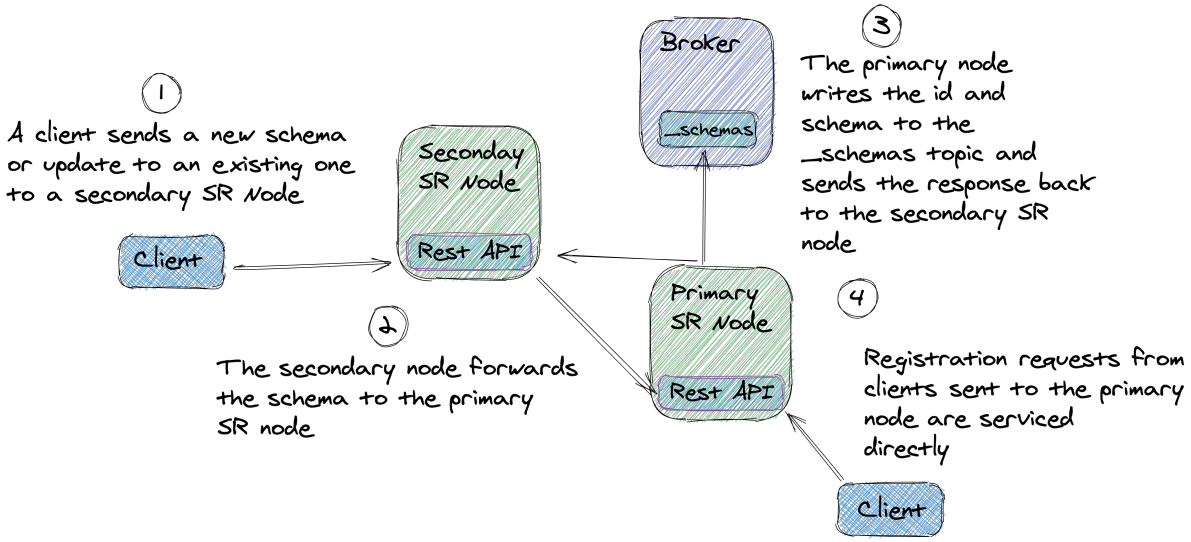


Figure 3.4 Schema Registry is a distributed application where only the primary node communicates with Kafka

Anytime a client registers or updates a schema, the primary node produces a record to the {underscore}schemas topic. Schema Registry uses a Kafka producer for writing and all the nodes use a consumer for reading updates. So you can see that Schema Registry's local state is backed up in a Kafka topic making schemas very durable.

NOTE	**When working with Schema Registry throughout all the examples in the book you'll only use a single node deployment suitable for local development.**

But all Schema Registry nodes serve read requests from clients. If any secondary nodes receive a registration or update request, it is forwarded to the primary node. Then the secondary node returns the response from the primary node. Let's take a look at an illustration of this architecture to solidify your mental model of how this works:

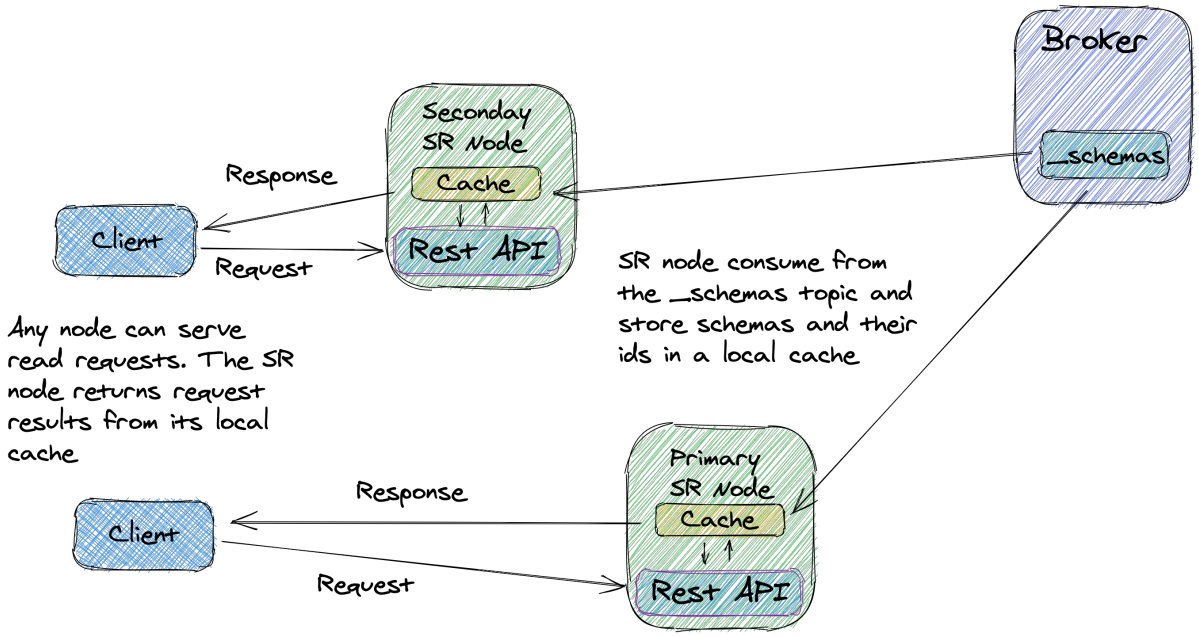


Figure 3.5 All Schema Registry nodes can serve read requests

Now that we've given an overview of the architecture, let's get to work by issuing a few basic commands using Schema Registry REST API.

3.1.4 Communication - Using Schema Registry's REST API

So far we've covered how Schema Registry works, but now it's time to see it in action by uploading a schema then running some additional commands available to get more information about your uploaded schema. For the initial commands you'll use `curl` and `jq` in a terminal window.

NOTE `curl` (curl.se/) is a command line utility for working with data via a URLs. `jq` (stedolan.github.io/jq/) is a command-line json processor. For installing jq for your platform you can visit the jq download site stedolan.github.io/jq/download/. For curl it should come installed on Windows 10+ and Mac Os. On Linux you can install via a package manager. If you are using Mac OS you can install both using homebrew - `brew.sh/`.

In later sections you'll use a `gradle` plugin for your interactions with Schema Registry. After you get an idea of how the different REST API calls work, you'll move on to using the gradle plugins and using some basic producer and consumer examples to see the serialization in action.

Typically you'll use the build tool plugins for performing Schema Registry actions. First they make he development process much faster rather than having run the API calls from the command line, and secondly they will automatically generate source code from schemas. We'll cover using build tool plugins in an upcoming section.

> **NOTE** There are Maven and Gradle plugins for working with Schema Registry, but the source code project for the book uses Gradle, so that's the plugin you'll use.

REGISTER A SCHEMA

Before we get started make sure you've run `docker-compose up -d` so that we'll have a Schema Registry instance running. But there's going to be nothing registered so your first step is to register a schema. Let's have a little fun and create a schema for Marvel Comic super heroes, the Avengers. You'll use Avro for your first schema and let's take a second now to discuss the format:

Listing 3.2 Avro schema for Avengers

```
{"namespace": "bbejeck.chapter_3",   ❶
 "type": "record",                   ❷
 "name": "Avenger",                  ❸
 "fields": [                         ❹
     {"name": "name", "type": "string"},
     {"name": "real_name", "type": "string"},          ❺
     {"name": "movies", "type":
                      {"type": "array", "items": "string"},
      "default": []    ❻
    }
  ]
}
```

❶ The namespace uniquely identifies the schema. For generated Java code the namespace is the package name.

❷ The type is `record` which is a complex type. Other complex types are `enums`, `arrays`, `maps`, `unions` and `fixed`. We'll go into more detail about Avro types later in this chapter.

❸ The name of the record

❹ Declaring the fields of the record

❺ Describing the individual fields. Fields in Avro are either simple or complex.

❻ Providing a default value. If the serialized bytes don't contain this field, Avro uses the default value when deserializing.

You define Avro schemas in JSON format. You'll use this same schema file in a upcoming section when we discuss the gradle plugin for code generation and interactions with Schema Registry. Since Schema Registry supports Protobuf and JSON Schema formats as well let's take a look at the same type in those schema formats here as well:

Listing 3.3 Protobuf schema for Avengers

```
syntax = "proto3";    ❶

package bbejeck.chapter_3.proto;    ❷

option java_outer_classname = "AvengerProto";    ❸

message Avenger {    ❹
    string name = 1;    ❺
    string real_name = 2;
    repeated string movies = 3;    ❻

}
```

❶ Defining the version of Protobuf, we're using version three in this book

❷ Declaring the package name

❸ Specifying the name of the outer class, otherwise the name of the proto file is used

❹ Defining the message

❺ Unique field number

❻ A repeated field; corresponds to a list

The Protobuf schema looks closer to regular code as the format is not JSON. Protobuf uses the numbers you see assigned to the fields to identify those fields in the message binary format. While Avro specification allows for setting default values, in Protobuf (version 3), every field is considered optional, but you don't provide a default value. Instead, Protobuf uses the type of the field to determine the default. For example the default for a numerical field is 0, for strings it's an empty string and repeated fields are an empty list.

NOTE **Protobuf is a deep subject and since this book is about the Kafka event streaming pattern, I'll only cover enough of the Protobuf specification for you to get started and feel comfortable using it. For full details you can read the language guide found here developers.google.com/protocol-buffers/docs/proto3.**

Now let's take a look at the JSON Schema version:

Listing 3.4 JSON Schema schema for Avengers

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",     ❶
  "title": "Avenger",
  "description": "A JSON schema of Avenger object",
  "type": "object",                ❷
  "javaType": "bbejeck.chapter_3.json.SimpleAvengerJson",    ❸
  "properties": {       ❹
    "name": {
      "type": "string"
    },
    "realName": {
      "type": "string"
    },
    "movies": {
      "type": "array",
      "items": {
        "type": "string"
      },
      "default": []    ❺
    }
  },
  "required": [
    "name",
    "realName"
  ]
}
```

❶ Referencing the specific schema spec

❷ Specifying the type is an object

❸ The javaType used when deserializing

❹ Listing the fields of the object

❺ Specifying a default value

The JSON Schema schema resembles the Avro version as both use JSON for the schema file. The biggest difference between the two is that in the JSON Schema you list the object fields under a `properties` element vs. a `fields` array and in the fields themselves you simply declare the name vs. having a `name` element.

NOTE **Please note there is a difference between a schema written in JSON format and one that follows the JSON Schema format. JSON Schema is "a vocabulary that allows you to annotate and validate JSON documents.". As with Avro and Protobuf, I'm going to focus on enough for you to get going using it in your projects, but for in-depth coverage you should visit json-schema.org/ for more information.**

I've shown the different schema formats here for comparison. But in the rest of the chapter, I'll usually only show one version of a schema in an example to save space. But the source code will contain examples for all three supported types.

Now that we've reviewed the schemas, let's go ahead and register one. The command to register a schema with REST API on the command-line looks like this

Listing 3.5 Register a schema on the command line

```
jq '. | {schema: tojson}' src/main/avro/avenger.avsc | \     ❶
curl -s -X POST http://localhost:8081/subjects/avro-avengers-value/versions\    ❷
         -H "Content-Type: application/vnd.schemaregistry.v1+json" \    ❸
         -d @-  \   ❹
         | jq   ❺
```

❶ Using the the jq `tojson` function to format the avenger.avsc file (new lines aren't valid json) for uploading, then pipe the result to the curl command

❷ The POST URL for adding the schema, the `-s` flag suppresses the progress info output from `curl`

❸ The content header

❹ The `-d` flag specifies the data and `@-` means read from STDIN i.e. the data provided by the jq command preceding the curl command

❺ Piping the json response through `jq` to get a nicely formatted response

The result you see from running this command should look like this:

Listing 3.6 Expected response from uploading a schema

```
{
  "id": 1
}
```

The response from the `POST` request is the id that Schema Registry assigned to the new schema. Schema Registry assigns a unique id (a monotonically increasing number) to each newly added schema. Clients use this id for storing schemas in their local cache.

Before we move on to another command I want to call your attention to annotation 2, specifically this part - `subjects/avro-avengers-value/`, it specifies the subject name for the schema. Schema Registry uses the subject name to manage the scope of any changes made to a schema. In this case it's confined to `avro-avengers-value` which means that values (in the key-value pairs) going into the `avro-avengers` topic need to be in the format of the registered schema. We'll cover subject names and the role they have in making changes in an upcoming section.

Next, let's take a look at some of the available commands you can use to retrieve information from Schema Registry.

Imagine you are working on building a new application to work with Kafka. You've heard about Schema Registry and you'd like to take a look at particular schema one of your co-workers

developed, but you can't remember the name and it's the weekend and you don't want to bother anyone. What you can do is list all the subjects of registered schemas with the following command:

Listing 3.7 Listing the subjects of registered schemas

```
curl -s "http://localhost:8081/subjects" | jq
```

The response from this command is a json array of all the subjects. Since we've only registered once schema so far the results should look like this

```
[
  "avro-avengers-value"
]
```

Great, you find here what you are looking for, the schema registered for the `avro-avengers` topic.

Now let's consider there's been some changes to the latest schema and you'd like to see what the previous version was. The problem is you don't know the version history. The next command shows you all of versions for a given schema

Listing 3.8 Getting all versions for a given schema

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/versions" | jq
```

This command returns a json array of the versions of the given schema. In our case here the results should look like this:

[1]

Now that you have the version number you need, now you can run another command to retrieve the schema at a specific version:

Listing 3.9 Retrieving a specific version of a schema

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/versions/1"\
 | jq '.'
```

After running this command you should see something resembling this:

```
{
  "subject": "avro-avengers-value",
  "version": 1,
  "id": 1,
  "schema": "{\"type\":\"record\",\"name\":\"AvengerAvro\",
      \"namespace\":\"bbejeck.chapter_3.avro\",\"fields\"
      :[{\"name\":\"name\",\"type\":\"string\"},{\"name\"
        :\"real_name\",\"type\":\"string\"},{\"name\"
          :\"movies\",\"type\":{\"type\":\"array\"
            ,\"items\":\"string\"},\"default\":[]}]}"
}
```

The value for the `schema` field is formatted as a string, so the quotes are escaped and all new-line characters are removed.

With a couple of quick commands from a console window, you've been able to find a schema, determine the version history and view the schema of a particular version.

As a side note, if you don't care about previous versions of a schema and you only want the latest one, you don't need to know the actual latest version number. You can use the following REST API call to retrieve the latest schema:

Listing 3.10 Getting the latest version of a schema

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/
  versions/latest" | jq '.'
```

I won't show the results of this command here, as it is identical to the previous command.

That has been a quick tour of some of the commands available in the REST API for Schema Registry. This just a small subset of the available commands. For a full reference go to docs.confluent.io/platform/current/schema-registry/develop/api.html#sr-api-reference.

Next we'll move on to using gradle plugins for working with Schema Registry and Avro, Protobuf and JSON Schema schemas.

3.1.5 Plugins and serialization platform tools

So far you've learned that the event objects written by producers and read by consumers represent the contract between the producer and consumer clients. You've also learned that this "implicit" contract can be a concrete one in the form of a schema. Additionally you've seen how you can use Schema Registry to store the schemas and make them available to the producer and consumer clients when the need to serialize and deserialize records.

In the upcoming sections you'll see even more functionality with Schema Registry. I'm referring to testing schemas for compatibility, different compatibility modes and how it can make changing or evolving a schema a relatively painless process for the involved producer and consumer clients.

But so far, you've only worked with a schema file and that's still a bit abstract. As I said earlier in the chapter, developers work with objects when building applications. So our next step is to see how we can convert these schema files into concrete objects you can use in an application.

Schema Registry supports schemas in Avro, Protobuf and JSON Schema format. Avro and Protobuf are serialization platforms that provide tooling for working with schemas in their respective formats. One of the most important tools is the ability to generate objects from the schemas.

Since JSON Schema is a standard and not a library or platform you'll need to use an open source tool for code generation. For this book we're using the github.com/eirnym/js2p-gradle project. For (de)serialization without Schema Registry I would recommend using `ObjectMapper` from the github.com/FasterXML/jackson-databind project.

Generating code from the schema makes your life as developer easier, as it automates the repetitive, boilerplate process of creating domain objects. Additionally since you maintain the schemas in source control (git in our case), the chance for error, such as making a field string type when it should be a long, when creating the domain objects is all but eliminated.

Also when making a change to a schema, you just commit the change and other developers pull the update and re-generate the code and everyone is unsung fairly quickly.

In this book we'll use the gradle build tool (gradle.org/) to manage the book's source code. Fortunately there are gradle plugins we can use for working with Schema Registry, Avro, Protobuf, and JSON Schema. Specifically, we'll use the following plugins

- github.com/ImFlog/schema-registry-plugin - For interacting with Schema Registry i.e. testing schema compatibility, registering schemas, and configuring schema compatibility
- github.com/davidmc24/gradle-avro-plugin - Used for Java code generation from Avro schema (`.avsc`) files.
- github.com/google/protobuf-gradle-plugin - Used for Java code generation from Protobuf schema (`.proto`) files
- github.com/eirnym/js2p-gradle - Used for Java code generation for schemas using the JSON Schema specification.

NOTE **It's important to note the distinction between schema files written in JSON such as Avro schemas and those files using the JSON Schema format (json-schema.org/). In the case of Avro files they are written as json, but follow the Avro specification. With the JSON Schema files they follow the official specification for JSON Schemas.**

By using the gradle plugins for Avro, Protobuf and JSON Schema, you don't need to learn how to use the individual tools for each component, the plugins handle all the work. We'll also use a gradle plugin for handling most of the interactions with Schema Registry.

Let's get started by uploading a schema using a gradle command instead of a REST API command in the console.

UPLOADING A SCHEMA FILE

The first thing we'll do is use gradle to register a schema. We'll use the same Avro schema from the REST API commands section. Now to upload the schema, make sure to change your current directory (`CD`) into the base directory of project and run this gradle command:

```
./gradlew streams:registerSchemasTask
```

After running this command you should see something like `BUILD SUCCESSFUL` in the console. Notice that all you needed to enter on the command line is the name of the gradle task (from the schema-registry-plugin) and the task registers all the schema inside the `register { }` block in the `streams/build.gradle` file.

Now let's take a look at the configuration of the Schema Registry plugin in the `streams/build.gradle` file.

Listing 3.11 Configuration for Schema Registry plugin in streams/build.gradle

```
schemaRegistry {     ❶
    url = 'http://localhost:8081'   ❷


register {
        subject('avro-avengers-value',   ❸
                'src/main/avro/avenger.avsc',   ❹
                'AVRO')   ❺

     //other entries left out for clarity
    }

  // other configurations left out for clarity
}
```

❶ Start of the Schema Registry configuration block in the build.gradle file

❷ Specifying the URL to connect to Schema Registry

❸ Registering a schema by subject name

❹ Specifying Avro schema file to register

❺ The type of the schema you are registering

In the `register` block you provide the same information, just in a format of a method call vs. a URL in a REST call. Under the covers the plugin code is still using the Schema Registry REST API via a `SchemaRegistryClient`. As side note, in the source code you'll notice there are several entries in the `register` block. You'll use all of them when go through the examples in the source code.

We'll cover using more gradle Schema Registry tasks soon, but let's move on to generating code from a schema.

GENERATING CODE FROM SCHEMAS

As I said earlier, one of the best advantages of using the Avro and Protobuf platforms is the code generation tools. Using the gradle plugin for these these tools takes the convenience a bit further by abstracting away the details of using the individual tools. To generate the objects represented by the schemas all you need to do is run this gradle task:

Listing 3.12 Generating the model objects

```
./gradlew clean build
```

Running this gradle command generates Java code for all the types Avro, Protobuf, and JSON Schema for the schemas in the project. Now we should talk about where you place the schemas in the project. The default locations for the Avro and Protobuf schemas are the `src/main/avro` and `src/main/proto` directories, respectively. The location for the JSON Schema schemas is the `src/main/json` directory, but you need to explicitly configure this in the `build.gradle` file:

Listing 3.13 Configure the location of JSON Schema schema files

```
jsonSchema2Pojo {

  source = files("${project.projectDir}/src/main/json")   ❶
  targetDirectory = file("${project.buildDir}/generated-main-json-java")  ❷
  // other configurations left out for clarity
}
```

❶ The `source` configuration specifies where the generation tools can locate the schemas

❷ The `targetDirectory` is were tool writes the generated Java objects

NOTE **All examples here refer to the schemas found in the `streams` sub-directory unless otherwise specified.**

Here you can see the configuration of the input and output directories for the `js2p-gradle` plugin. The Avro plugin, by default, places the generated files in a sub-directory under the `build` directory named `generated-main-avro-java`.

For Protobuf we configure the output directory to match the pattern of JSON Schema and Avro in the `Protobuf` block of the `build.gradle` file like this:

Listing 3.14 Configure Protobuf output

```
protobuf {
    generatedFilesBaseDir = "${project.buildDir}
      /generated-main-proto-java"   ❶

    protoc {
        artifact = 'com.google.protobuf:protoc:3.15.3'   ❷
    }
}
```

❶ The output directory for the Java files generated from Protobuf schema

❷ Specifying the location of the protoc compiler

I'd to take a quick second to discuss annotation two for a moment. To use Protobuf you need to have the compiler `protoc` installed. By default the plugin searches for a `protoc` executable. But we can use a pre-compiled version of `protoc` from Maven Central, which means you don't have to explicitly install it. But if you prefer to use your local install, you can specify the path inside the `protoc` block with `path = path/to/protoc/compiler`.

So we've wrapped up generating code from the schemas, now it's time to run an end-to-end example

END TO END EXAMPLE

At this point we're going to take everything you've learned so far and run a simple end-to-end example. So far, you have registered the schemas and generated the Java files you need from them. So your next steps are to:

- Create some domain objects from the generated Java files
- Produce your created objects to a Kafka topic
- Consume the objects you just sent from the same Kafka topic

While parts two and three from the list above seem to have more to do with clients than Schema Registry, I want to think about it from this perspective. You're creating instances of Java objects created from the schema files, so pay attention to fields and notice how the objects conform to the structure of the schema. Secondly, focus on the Schema Registry related configuration items, serializer or deserializer and the URL for communicating with Schema Registry.

NOTE **In this example you will use a Kafka Producer and Kafka Consumer, but I won't cover any of the details of working with them. If you're unfamiliar with the producer and consumer clients that's fine. I'll go into detail about producers and consumers in the next chapter. But for now just go through the examples as is.**

If you haven't already registered the schema files and generated the Java code, let's do so now. I'll put the steps here again and make sure you have run `docker-compose up -d` to ensure your Kafka broker and Schema Registry are running.

Listing 3.15 Register schemas and generate Java files

```
./gradlew streams:registerSchemasTask   ❶

./gradlew clean build   ❷
```

❶ Register the schema files

❷ Build the Java objects from schemas

Now let's focus on the Schema Registry specific configurations. Go to the source code and take a look at the `bbejeck.chapter_3.producer.BaseProducer` class. For now we only want to look at the following two configurations, we'll cover more configurations for the producer in the next chapter:

```
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    keySerializer);   ❶
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081");   ❷
```

❶ Specifying the serializer to use

❷ Setting the location of Schema registry

The first configuration sets the `Serializer` the producer will use. Remember, the `KafkaProducer` is decoupled from the type of the `Serializer`, it simply calls the `serialize` method and gets back an array of bytes to send. So the responsibility for providing the correct `Serializer` class is up to you.

In this case we're going to work with objects generated from an Avro schema, so you use the `KafkaAvroSerializer`. If you look at the `bbejeck.chapter_3.producer.avro.AvroProducer` class (which extends the `BaseProducer`) you see it pass the `KafkaAvroSerializer.class` to the parent object constructor. The second configuration specifies the HTTP endpoint that the `Serializer` uses for communicating with Schema Registry. These configurations enable the interactions described in the illustration "Schema registry ensures consistent data format between producers and consumers" above.

Next, let's take a quick look at creating an object:

Listing 3.16 Instantiating a an object from the generated code

```
var blackWidow = AvengerAvro.newBuilder()
                .setName("Black Widow")
                .setRealName("Natasha Romanova")
                .setMovies(List.of("Avengers", "Infinity Wars",
                  "End Game")).build();
```

OK, you're thinking now, "this code creates an object, what's the big deal?". While it could be a minor point, but it's more what you can't do here that I'm trying to drive home. You can only populate the expected fields with the correct types, enforcing the contract of producing records in the expected format. Of course you could update the schema and regenerate the code.

But by making changes, you have to register the new schema and the changes have to match the current compatibility format for the subject-name. So now can see now how Schema Registry enforces the "contract" between producers and consumers. We'll cover compatibility modes and the allowed changes in an upcoming section.

Now let's run the following gradle command to produce the objects to `avro-avengers` topic.

Listing 3.17 Running the AvroProducer

```
./gradlew streams:runAvroProducer
```

After running this command you'll see some output similar to this:

```
  DEBUG [main] bbejeck.chapter_3.producer.BaseProducer - Producing records
 [{"name": "Black Widow", "real_name": "Natasha Romanova", "movies":
["Avengers", "Infinity Wars", "End Game"]},
{"name": "Hulk", "real_name": "Dr. Bruce Banner", "movies":
["Avengers", "Ragnarok", "Infinity Wars"]},
{"name": "Thor", "real_name": "Thor", "movies":
["Dark Universe", "Ragnarok", "Avengers"]}]
```

After the application produces these few records it shuts itself down.

IMPORTANT **It's important to make sure to run this command exactly as shown here including the preceding : character. We have three different gradle modules for our Schema Registry exercises. We need to make sure the command we run are for the specific module. In this case the : executes the main module only, otherwise it will run the producer for all modules and the example will fail.**

Now running this command doesn't do anything exciting, but it demonstrate the ease of serializing by using Schema Registry. The producer retrieves the schema stores it locally and sends the records to Kafka in the correct serialized format. All without you having to write any serialization or domain model code. Congratulations you have sent serialize records to Kafka!

TIP **It could be instructive to look a the log file generated from running this command. It can be found in the `logs/` directory of the provided source code. The log4j configuration overwrites the log file with each run, so be sure to inspect it before running the next step.**

Now let's run a consumer which will deserialize the records. But as we did with the producer, we're going to focus on the configuration required for deserialization and working with Schema Registry:

Listing 3.18 Consumer configuration for using Avro

```
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    KafkaAvroDeserializer.class);   ❶
consumerProps.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG,
    true);   ❷
consumerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081");   ❸
```

❶ Using Avro deserialization

❷ Configuring to use a SpecificAvroReader

❸ The host:port for Schema Registry

You'll notice that in the second annotation you are setting the `SPECIFIC_AVRO_READER_CONFIG` to `true`. What does the `SPECIFIC_AVRO_READER_CONFIG` setting do? Well to answer that question let's take a slight detour in our conversation to discuss working with Avro, Protobuf, and JSON Schema serialized objects.

When deserializing one of the Avro, Protobuf, or JSON Schema objects there is a concept of deserializing the specific object type or a non-specific "container" object. For example, with the `SPECIFIC_AVRO_READER_CONFIG` set to true, the deserializer inside the consumer, will return an object of type `AvroAvenger` the **specific** object type.

However had you set the `SPECIFIC_AVRO_READER_CONFIG` to `false`, the deserializer returns an object of type `GenericRecord`. The returned `GenericRecrod` still follows the same schema, and has the same content, but the object itself is devoid of any type awareness, it's as the name implies simply a generic container of fields. The following example should make clear what I'm saying here:

Listing 3.19 Specific Avro records vs. GenericRecord

```
AvroAvenger avenger = // returned from consumer with
  //SPECIFIC_AVRO_READER_CONFIG=true
avenger.getName();
avenger.getRealName();     ❶
avenger.getMovies();

GenericRecord genericRecord = // returned from consumer with
  //SPECIFIC_AVRO_READER_CONFIG=false
if (genericRecord.hasField("name")) {
   genericRecord.get("name");
}

if (genericRecord.hasField("real_name")) {    ❷
    genericRecord.get("real_name");
}

if (GenericRecord.hasField("movies")) {
    genericRecord.get("movies");
}
```

❶ Accessing fields on the specific object

❷ Accessing fields on the generic object

From this simple code example, you can see the differences between the specific returned type vs. the generic. With the `AvroAvenger` object in annotation one, we can access the available properties directly, as the object is "aware" of its structure and provides methods for accessing those fields. But with the `GenericRecord` object you need to query if it contains a specific field before attempting to access it.

NOTE **The specific version of the Avro schema is not just a POJO (Plain Old Java Object) but extends the `SpecificRecordBase` class.**

Notice that with the `GenericRecord` you need to access the field exactly as its specified in the schema, while the specific version uses the more familiar camel case notation.

The difference between the two is that with the specific type you know the structure, but with the generic type, since it could represent any arbitrary type, you need to query for different fields to determine its structure. You need to work with a `GenericRecord` much like you would with a `HashMap`.

However you're not left to operate completely in the dark. You can get a list of fields from a `GenericRecord` by calling `GenericRecord.getSchema().getFields()`. Then you could iterate over the list of `Field` objects and get the names by calling the `Fields.name()`. Additionally you could get the name of the schema with `GenericRecord.getSchema().getFullName();` and presumably at that point you would know which fields the record contained.

Updating a field you'd follow a similar approach: .Updating or setting fields on specific and generic records

```
avnenger.setRealName("updated name")
genericRecord.put("real_name", "updated name")
```

So from this small example you can see that the specific object gives you the familiar setter functionality but the the generic version you need to explicitly declare the field you are updating. Again you'll notice the `HashMap` like behavior updating or setting a field with the generic version.

Protobuf provides a similar functionality for working with specific or arbitrary types. To work with an arbitrary type in Protobuf you'd us a `DynamicMessage`. As with the Avro `GenericRecord`, the `DynamicMessage` offers functions to discover the type and the fields. With JSON Schema the specific types are just the object generated from the gradle plugin, there's no framework code associated with it like Avro or Protobuf. The generic version is a type of `JsonNode` since the deserializer uses the jackson-databind (github.com/FasterXML/jackson-databind) API for serialization and deserialization.

NOTE **The source code for this chapter contain examples of working with the specific and generic types of Avro, Protobuf and JSON Schema.**

So the question is when do you use the specific type vs. the generic? In the case where you only have one type of record in a Kafka topic you'll use the specific version. On the other hand, if you have multiple event types in a topic, you'll want to use the generic version, as each consumed record could be a different type. We'll talk more about multiple event types in a single topic later in this chapter, and again in the client and Kafka Streams chapters.

The final thing to remember is that to use the specific record type, you need to set the `kafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG` to `true`. The default for the `SPECIFIC_AVRO_READER_CONFIG` is false, so the consumer returns the `GenericRecord` type if the configuration is not set.

Now with the sidebar about different record types completed, let's resume walking through your first end-to-end example using Schema Registry. You've already produced some records using the schema you uploaded previously. Now you just need to start a consumer to demonstrate deserializing those records with the schema. Again, looking at the log files should be instructive as you'll see the embedded deserializer downloading the schema for the first record only as it gets cached after the initial retrieval.

I should also note that the following example using `bbejeck.chapter_3.consumer.avro.AvroConsumer` uses both the specific class type and the `GenericRecord` type. As the example runs, the code prints out the type of the consumed record.

NOTE	There are similar examples for Protobuf and JSON Schema in the source code.

So let's run the consumer example now by executing the following command from the root of the book source code project:

Listing 3.20 Running the AvroConsumer

```
./gradlew streams:runAvroConsumer
```

IMPORTANT	Again, the same caveat here about running the command with the preceding `:` character, otherwise it will run the consumer for all modules and the example will not work.

The `AvroConsumer` prints out the consumed records and shuts down by itself. Congratulations, you've just serialized and deserialized records using Schema Registry!

So far we've covered the types of serialization frameworks supported by Schema Registry, how to write and add a schema file, and walked through a basic example using a schema. During the portion of the chapter where you uploaded a schema, I mentioned the term `subject` and how it defines the scope of schema evolution. That's what you'll learn in the next section, using the different subject name strategies.

3.2 Subject name strategies

Schema Registry uses the concept of a subject to control the scope of schema evolution. Another way to think of the subject is a namespace for a particular schema. In other words, as your business requirements evolve, you'll need to make changes to your schema files to make the appropriate changes to your domain objects. For example, with our AvroAvenger domain object, you want to remove the real (civilian) name of the hero and add a list of their powers.

Schema Registry uses the subject to lookup the existing schema and compare the changes with the new schema. It performs this check to make sure the changes are compatible with the current compatibility mode set. We'll talk about compatibility modes in an upcoming section. The subject name strategy determines the scope of where schema registry makes its compatibility checks.

There are three types of subject name strategies, `TopicNameStrategy`, `RecordNameStrategy`, and `TopicRecordNameStrategy`. You can probably infer the scope of the name-spacing implied by the strategy names, but it's worth going over the details. Let's dive in and discuss these different strategies now.

NOTE **By default all serializers will attempt to register a schema when serializing, if it doesn't find the corresponding id in its local cache. Auto registration is a great feature, but in some cases you may need to turn it off with a producer configuration setting of `auto.register.schemas=false`. One example of not wanting auto registration is when you are using an Avro union schema with references. We'll cover this in more detail later in the chapter.**

3.2.1 TopicNameStrategy

The `TopicNameStrategy` is the default subject in Schema Registry. The subject name comes from the name of the topic. You saw the `TopicNameStrategy` in action earlier in the chapter when you registered a schema with the gradle plugin. To be more precise the subject name is `topic-name-key` or `topic-name-value` as you can have different types for the key and value requiring different schemas.

The `TopicNameStrategy` ensures there is only one data type on a topic, since you can't register a schema for a different type with the same topic name. Having a single type per topic makes sense in a lot of cases. For example, if you name your topics based on the event type they store. it follows that they will contain only one record type.

Another advantage of the `TopicNameStrategy` is with the schema enforcement limited to a single topic, you can have another topic using the same record type, but using a different schema. Consider the situation where two different departments use the same record type, but use different topic names. With the `TopicNameStrategy` these departments can register completely different schemas for the same record type, since the scope of the schema is limited to a particular topic.

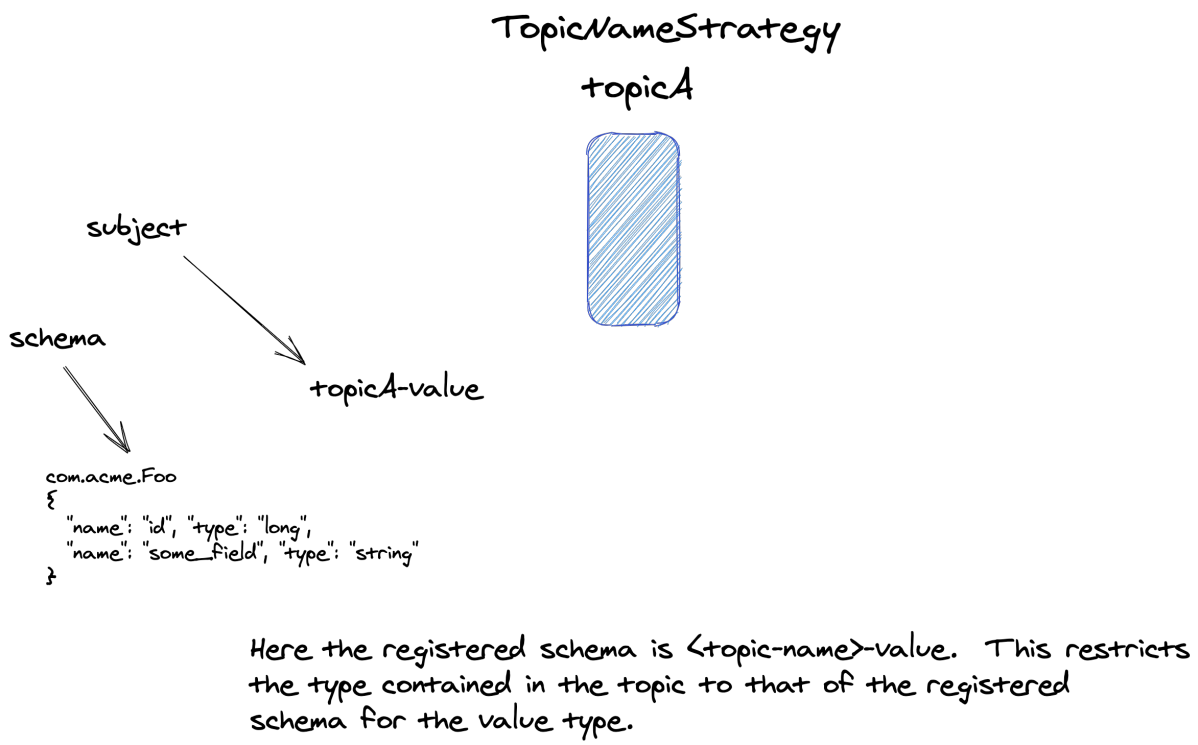


Figure 3.6 TopicNameStrategy enforces having the same type of domain object represented by the registered schema for the value and or the key

Since the `TopicNameStrategy` is the default, you don't need to specify any additional configurations. When you register schemas you'll use the format of `<topic>-value` as the subject for value schemas and `<topic>-key` as the subject for key schemas. In both cases you substitute the name of the topic for the `<topic>` token.

But there could be cases where you have closely related events and you want to produce those records into in one topic. In that case you'll want to chose a strategy that allows different types and schemas in a topic.

3.2.2 RecordNameStrategy

The `RecordNameStrategy` uses the fully qualified class name (of the Java object representation of the schema) as the subject name. By using the record name strategy you can now have multiple types of records in the same topic. But the key point is that there is a **logical** relationship between these records, it's just the physical layout of them is different.

When would you choose the `RecordNameStrategy`? Imagine you have different IoT (Internet of Things) sensors deployed. Some of sensors measure different events so they'll have different records. But you still want to have them co-located on the same topic.

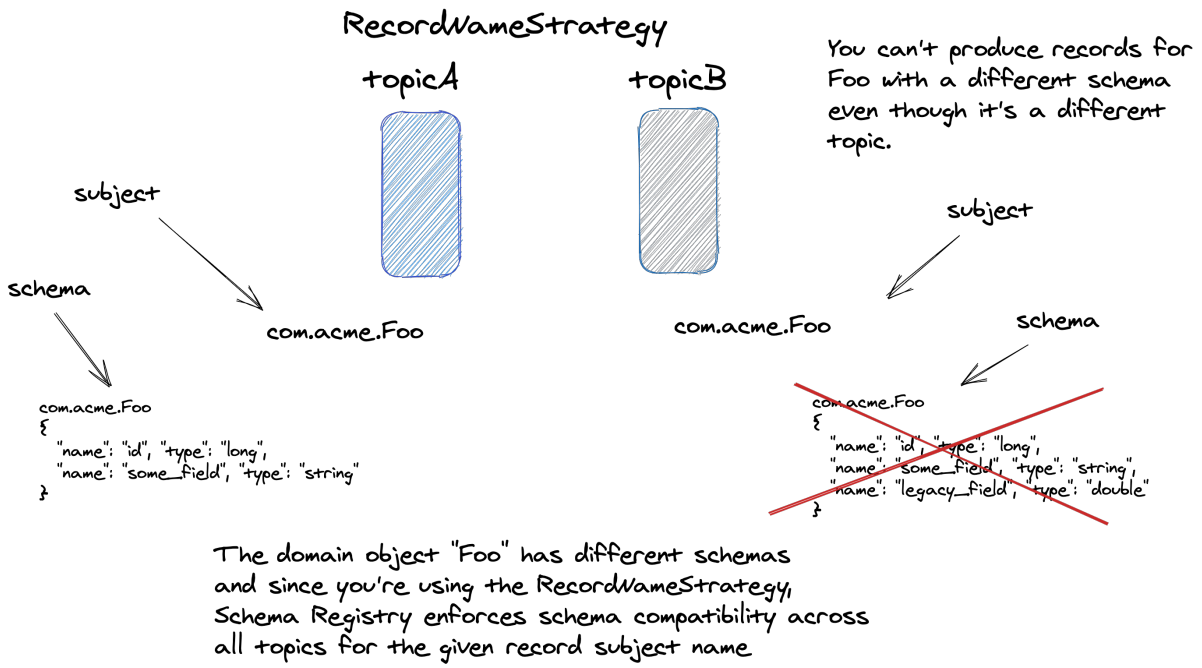


Figure 3.7 RecordNameStrategy enforces having the same schema for a domain object across different topics

Since there can be different types, the compatibility checks occur between schemas with the same record name. Additionally the compatibility check extends to all topics using a subject with the same record name.

To use the `RecordNameStrategy` you use a fully qualified class name for the subject when registering a schema for a given record type. For the `AvengerAvro` object we've used in our examples, you would configure the schema registration like this:

Listing 3.21 Schema Registry gradle plugin configuration for RecordNameStrategy

```
subject('bbejeck.chapter_3.avro.AvengerAvro','src/main/avro/avenger.avsc', 'AVRO')
```

Then you need to configure the producer and consumer with the appropriate subject name strategy. For example:

Listing 3.22 Producer configuration for RecordNameStrategy

```
Map<String, Object> producerConfig = new HashMap<>();
producerConfig.put(KafkaAvroSerializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
 RecordNameStrategy.class);
producerConfig.put(KafkaAvroSerializerConfig.KEY_SUBJECT_NAME_STRATEGY,
 RecordNameStrategy.class);
```

Listing 3.23 Consumer configuration for RecordNameStrategy

```
Map<String, Object> consumerConfig = new HashMap<>();
config.put(KafkaAvroDeserializerConfig.KEY_SUBJECT_NAME_STRATEGY,
 RecordNameStrategy.class);
config.put(KafkaAvroDeserializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
 RecordNameStrategy.class);
```

NOTE **If you are only using Avro for serializing/deserializing the values, you don't need to add the configuration for the key. Also the key and value subject name strategies do not need to match, I've only presented them that way here.**

For Protobuf use the `KafkaProtobufSerializerConfig` and `KafkaProtobufDeserializerConfig` and for JSON schema use the `KafkaJsonSchemaSerializerConfig` and `KafkaJsonSchemaDeserializerConfig`

These configurations only effect how the serializer/deserializer interact with Schema Registry for looking up schemas. Again the serialization is decoupled from producing and consuming process.

One thing to consider is that by using only the record name, all topics must use the same schema. If you want to use different records in a topic, but want to only consider the schemas for that particular topic, then you'll need to use another strategy.

3.2.3 TopicRecordNameStrategy

As you can probably infer from the name this strategy allows for having multiple record types within a topic as well. But the registered schemas for a given record are only considered within the scope of the current topic. Let's take a look at the following illustration to get a better idea of what this means.

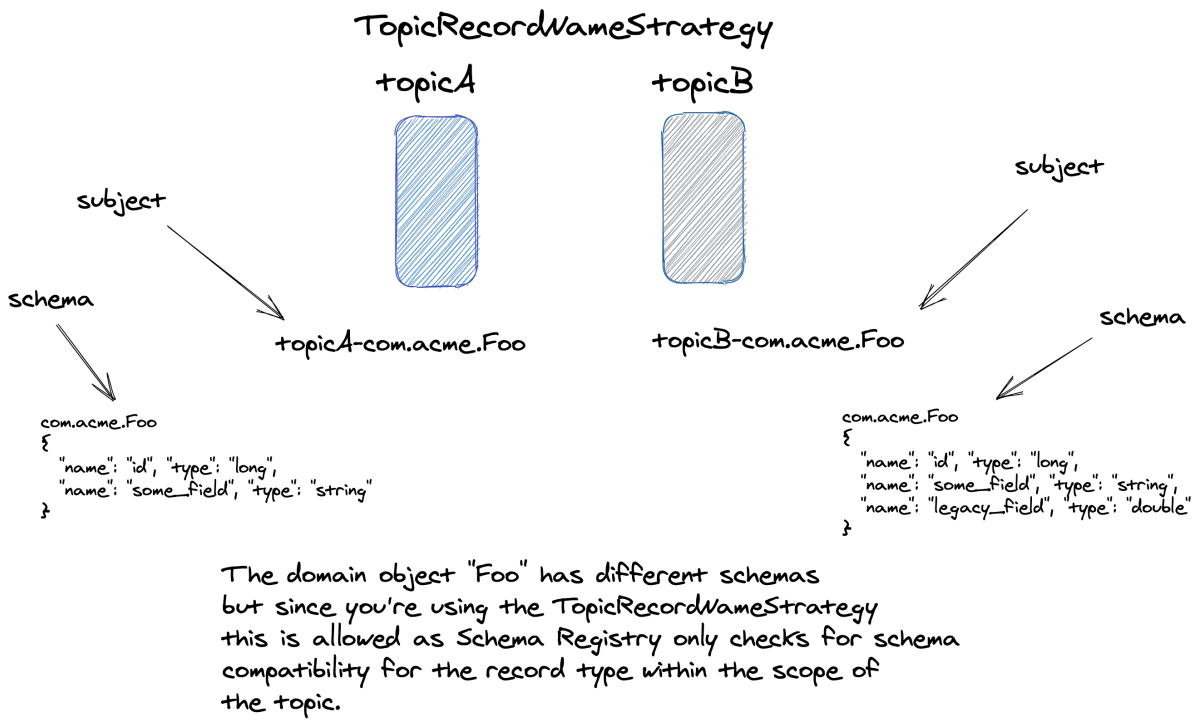


Figure 3.8 TopicRecordNameStrategy allows for having different schemas for the same domain object across different topics

As you can see from the image above `topic-A` can have a different schema for the record type `Foo` from `topic-B`. This strategy allows you to have multiple logically related types on one topic, but it's isolated from other topics where you have the same type but are using different schemas.

Why would you use the `TopicRecordNameStrategy`? For example, consider this situation:

You have one version of the `CustomerPurchaseEvent` event object in the `interactions` topic, that groups all customer event types (`CustomerSearchEvent`, `CustomerLoginEvent` etc) grouped together. But you have an older topic purchases, that also contains `CustomerPurchaseEvent` objects, but it's for a legacy system so the schema is older and contains different fields from the newer one. The `TopicRecordNameStrategy` allows for having these two topics to contain the same ***type*** but with different schema versions.

Similar to the `RecordNameStrategy` you'll need to do the following steps to configure the strategy:

Listing 3.24 Schema Registry gradle plugin configuration for TopicRecordNameStrategy

```
subject('avro-avengers-bbejeck.chapter_3.avro.AvengerAvro',
  'src/main/avro/avenger.avsc', 'AVRO')
```

Then you need to configure the producer and consumer with the appropriate subject name strategy. For example:

Listing 3.25 Producer configuration for TopicRecordNameStrategy

```
Map<String, Object> producerConfig = new HashMap<>();
producerConfig.put(KafkaAvroSerializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
 TopicRecordNameStrategy.class);
producerConfig.put(KafkaAvroSerializerConfig.KEY_SUBJECT_NAME_STRATEGY,
 TopicRecordNameStrategy.class);
```

Listing 3.26 Consumer configuration for TopicRecordNameStrategy

```
Map<String, Object> consumerConfig = new HashMap<>();
config.put(KafkaAvroDeserializerConfig.KEY_SUBJECT_NAME_STRATEGY,
 TopicRecordNameStrategy.class);
config.put(KafkaAvroDeserializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
 TopicRecordNameStrategy.class);
```

NOTE **The same caveat about registering the strategy for the key applies here as well, you would only do so if you are using a schema for the key, it's only provided here for completeness. Also the key and value subject name strategies don't need to match**

Why would you use the `TopicRecordNameStrategy` over either the `TopicNameStrategy` or the `RecordNameStrategy`? If you wanted the ability to have multiple event types in a topic, but you need the flexibility to have different schema versions for a given type across your topics.

But when considering multiple types in a topic, both the `TopicRecordNameStrategy` and the `RecordNameStrategy` don't have the ability to constrain a topic to fixed set of types. Using either of those subject name strategies opens up the topic to have an unbounded number of different types. We'll cover how to improve on this situation when we cover schema references in an upcoming section.

Here's a quick summary for you to consider when thinking of the different subject name strategies. Think of the subject name strategy as a function that accepts the topic-name and record-schema as arguments and it returns a subject-name. The `TopicNameStrategy` only uses the topic-name and ignores the record-schema. `RecordNameStrategy` does the opposite; it ignores the topic-name and only uses the record-schema. But the `TopicRecordNameStrategy` uses both of them for the subject-name.

Table 3.1 Schema strategies summary table

Strategy	Multiple types in a topic	Different versions of objects across topics
TopicNameStrategy	Maybe	Yes
RecordNameStrategy	Yes	No
TopicRecordNameStrategy	Yes	Yes

So far we've covered the subject naming strategies and how Schema Registry uses subjects for

name-spacing schemas. But there's another dimension to schema management, how to evolve changes within the schema itself. How do you handle changes like the removal or addition of a field? Do you want your clients to have forward or backward compatibility? In the next section we'll cover exactly how you handle schema compatibility.

3.3 Schema compatibility

When there are schema changes you need to consider the compatibility with the existing schema and the producer and consumer clients. If you make a change by removing a field how does this impact the producer serializing the records or the consumer deserializing this new format?

To handle these compatibility concerns, Schema Registry provides four base compatibility modes `BACKWARD`, `FORWARD`, `FULL`, and `NONE`. There are also three additional compatibility modes `BACKWARD_TRANSITIVE`, `FORWARD_TRANSITIVE`, and `FULL_TRANSITIVE` which extend on the base compatibility mode with the same name. The base compatibility modes only guarantee that a new schema is compatible with immediate previous version. The transitive compatibility specifies that the new schema is compatible with **all** previous versions of a given schema applying the compatibility mode.

You can specify a global compatibility level or a compatibility level per subject.

What follows in this chapter is a description of the valid changes for a given compatibility mode along with an illustration demonstrating the sequence of changes you'd need to make to the producers the consumers. For a hands on tutorial of making changes to a schema, see Appendix-B: Schema Compatibility Workshop.

3.3.1 Backward compatibility

Backward compatibility is the default migration setting. With backward compatibility you update the consumer code first to support the new schema. The updated consumers can read records serialized with the new schema or the immediate previous schema.

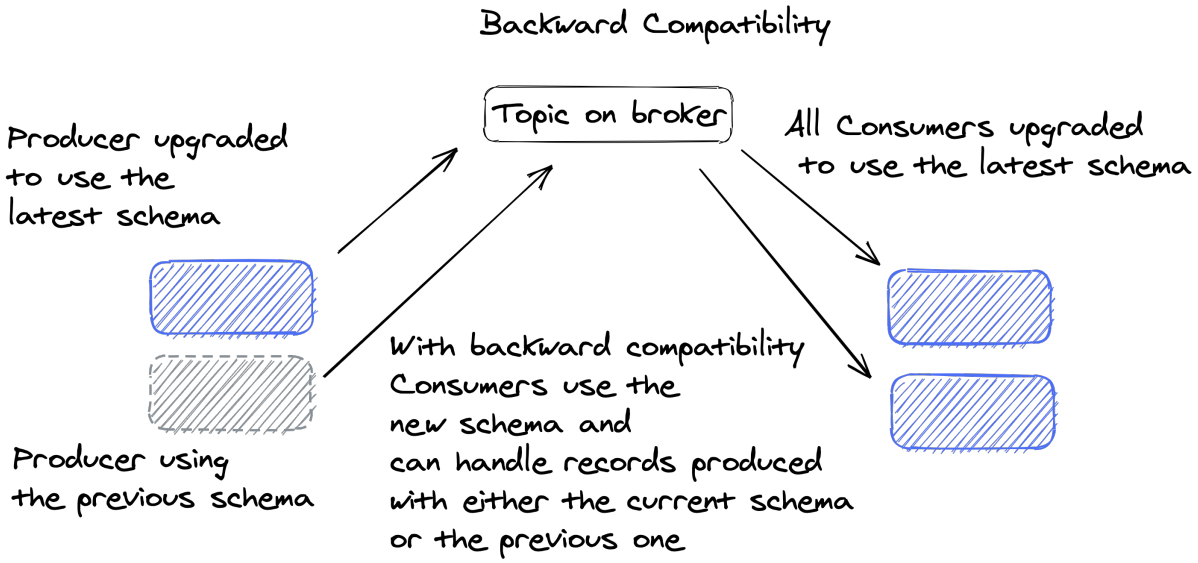


Figure 3.9 Backward compatibility updates consumers first to use the new schema then they can handle records from producers using either the new schema or the previous one

As shown in this illustration the consumer, can work with both the previous and the new schemas. The allowed changes with backwards compatibility are deleting fields or adding optional fields. An field is considered optional when the schema provides a default value. If the serialized bytes don't contain the optional field, then the deserializer uses the specified default value when deserializing the bytes back into an object.

3.3.2 Forward compatibility

Forward compatibility is a mirror image of backward compatibility regarding field changes. With forward compatibility you can add fields and delete **optional** fields.

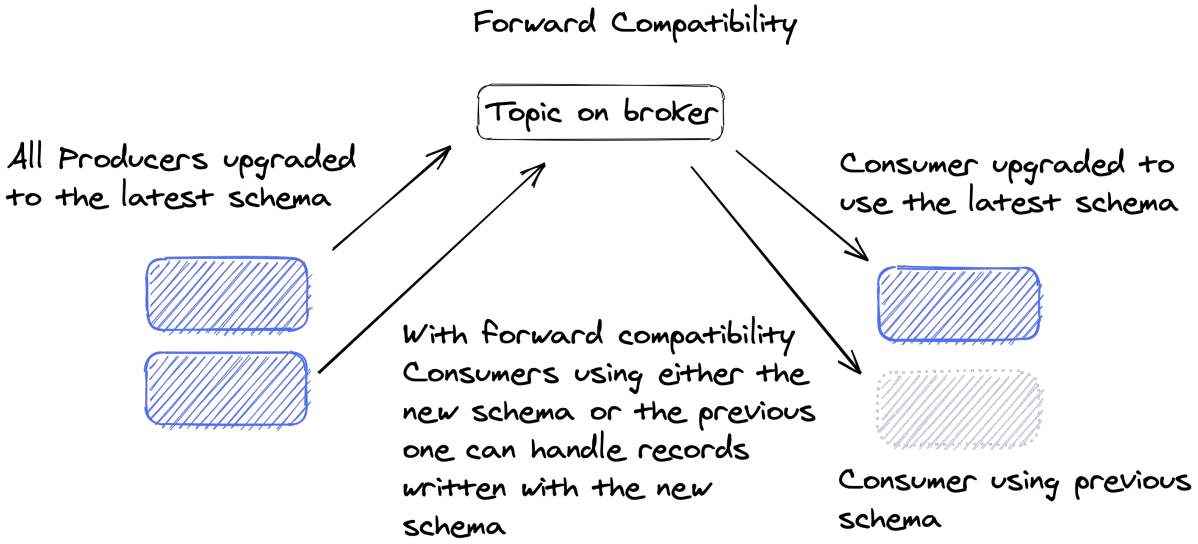


Figure 3.10 Forward compatibility updates producers first to use the new schema and consumers can handle the records either the new schema or the previous one

By upgrading the producer code first, you're making sure the new fields are properly populated and only records in the new format are available. Consumers you haven't upgraded can still work with the new schema as it will simply ignore the new fields and the deleted fields have default values.

At this point you've seen two compatibility types, backward and forward. As the compatibility name implies, you must consider record changes in one direction. In backward compatibility, you updated the consumers first as records could arrive in either the new or old format. In forward compatibility, you updated the producers first to ensure the records from that point in time are only in the new format. The last compatibility strategy to explore is the `FULL` compatibility mode.

3.3.3 Full compatibility

In full compatibility mode, you free to add or remove fields, but there is one catch. **Any changes** you make must be to **optional** fields only. To recap an optional field is one where you provide a default value in the schema definition should the original deserialized record not provide that specific field.

> **NOTE** **Both Avro and JSON Schema provide support for explicitly providing default values, with Protocol Buffers version 3 (the version used in the book) every field automatically has a default based in its type. For example number types are 0, strings are "", collections are empty etc.**

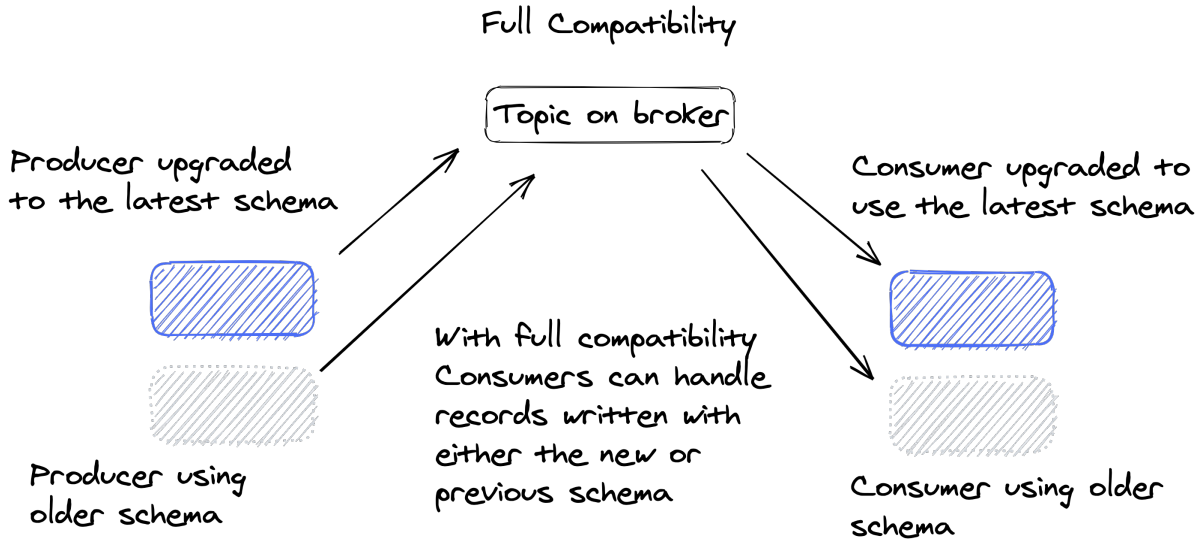


Figure 3.11 Full compatibility allows for producers to send with the previous or new schema and consumers can handle the records either the new schema or the previous one

Since the fields involved in the updated schema are optional, these changes are considered compatible for existing producer and consumer clients. This means that the upgrade order in this

case is up to you. Consumers will continue to work with records produced with the new or old schema.

3.3.4 No compatibility

Specifying a compatibility of `NONE` instructs Schema Registry to do just that, no compatibility checks. By not using any compatibility checks means that someone can add new fields, remove existing fields, or change the type of a field. Any and all changes are accepted.

Not providing any compatibility checks provides a great deal of freedom. But the trade-off is you're vulnerable to breaking changes that might go undetected until the worse possible time; in production.

It could be that every time you update a schema, you upgrade all producers and consumers at the same time. Another possibility is to create a new topic for clients to use. Applications can use the new topic without having the concerns of it containing records from the older, incompatible schema.

Now you've learned how you can migrate a schema to use a new version with changes within the different schema compatibility modes and for review here's a quick summary table of the different compatibility types

Table 3.2 Schema Compatibility Mode Summary

Mode	Changes Allowed	Client Update Order	Retro guaranteed compatibility
Backward	Delete fields, add optional fields	Consumers, Producers	Prior version
Backward Transitive	Delete fields, add optional fields	Consumers, Producers	All previous versions
Forward	Add fields, delete optional fields	Producers, Consumers	Prior version
Forward Transitive	Add fields, delete optional fields	Producers, Consumers	All previous versions
Full	Delete optional fields, add optional fields	Doesn't matter	Prior version
Full Transitive	Delete optional fields, add optional fields	Doesn't matter	All previous versions

But there's more you can do with schemas. Much like working with objects you can share common code to reduce duplication and make maintenance easier, you can do the same with schema references

3.4 Schema references

A schema reference is just what is sounds like, referring to another schema from inside the current schema. Reuse is a core principal in software engineering as the ability to leverage something you've already built solves two issues. First, you could potentially save time by not having to re-write some exiting code. Second, when you need to update the original work (which always happens) all the downstream components leveraging the original get automatically updated as well.

When would you want to use a schema reference? Let's consider you have an application providing information on commercial business and universities. To model the business you have a `Company` schema and for the universities you have a `College` schema. Now a company has executives and the college has professors. You want to represent both with a nested schema of a `Person` domain object. The schemas would look something like this:

Listing 3.27 College schema

```
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CollegeAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "professors", "type":
    {"type": "array", "items": {   ❶
      "namespace": "bbejeck.chapter_3.avro",
      "name":"PersonAvro",         ❷
      "fields": [
        {"name": "name", "type":"string"},
        {"name": "address", "type": "string"},
        {"name": "age", "type": "int"}
      ]
    }},
      "default": []
    }
  ]
}
```

❶ Array of professors

❷ The item type in array is a Person object

So you can see here you have a nested record type in your college schema, which is not uncommon. Now let's look at the company schema

Listing 3.28 Company schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CompanyAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "executives", "type":
    {"type": "array", "items": {    ❶
      "type":"record",
      "namespace": "bbejeck.chapter_3.avro",
      "name":"PersonAvro",          ❷
      "fields": [
        {"name": "name", "type":"string"},
        {"name": "address", "type": "string"},
        {"name": "age", "type": "int"}
      ]
    }},
      "default": []
    }
  ]
}
```

❶ Array of executives

❷ Item type is a PersonAvro

Again you have a nested record for the type contained in the schema array. It's natural to model the executive or professor type as a person, as it allows you to encapsulate all the details into an object. But as you can see here, there's duplication in your schemas. If you need to change the person schema you need to update every file containing the nested person definition. Additionally, as you start to add more definitions, the size and complexity of the schemas can get unwieldy quickly due to all the nesting of types.

It would be better to put a reference to the type when defining the array. So let's do that next. We'll put the nested `PersonAvro` record in its own schema file, person.avsc.

I won't show the file here, as nothing changes, we are putting the definition you see here in a separate file. Now let's take a look at how you'd update the `college.avsc` and `company.avsc` schema files:

Listing 3.29 Updated College schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CollegeAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "professors", "type":
    {"type": "array", "items": "bbejeck.chapter_3.avro.PersonAvro"},   ❶
      "default": []
    }
  ]
}
```

❶ This is the new part it's a reference to the person object

IMPORTANT **When using schema references, the referring schema you provide must be the same type. For example you can't provide a reference to an Avro schema or JSON Schema inside Protocol Buffers schema, the reference must be another Protocol Buffers schema.**

Here you've cleaned things up by using a reference to the object created by the `person.avsc` schema. Now let's look at the updated company schema as well

Listing 3.30 Updated Company schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CompanyAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "executives", "type":
      {
        "type": "array", "items": "bbejeck.chapter_3.avro.PersonAvro"},   ❶
        "default": []
       }
  ]
}
```

❶ This is the new part also it's a reference to the person object

Now both schemas refer to the same object created by the person schema file. For completeness let's take a look at how you implement a schema reference in both JSON Schema and Protocol Buffers. First we'll look at the JSON Schema version:

Listing 3.31 Company schema reference in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Exchange",
  "description": "A JSON schema of a Company using Person refs",
  "javaType": "bbejeck.chapter_3.json.CompanyJson",
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "executives": {
      "type": "array",
      "items": {
        "$ref": "person.json"   ❶
      }
    }
  }
}
```

❶ The reference to the Person object schema

The concept with references in JSON Schema is the same, but you provide an explicit `$ref` element pointing to the referenced schema file. It's assumed that the referenced file is located in the same directory as the referring schema.

Now let's take a look at the equivalent reference with Protocol Buffers:

Listing 3.32 Company Schema reference in Protocol Buffers

```
syntax = "proto3";

package bbejeck.chapter_3.proto;

import "person.proto";    ❶

option java_outer_classname = "CompanyProto";

message Company {
  string name = 1;
  repeated Person executives = 2;   ❷
}
```

❶ Import statement for the referenced schema

❷ Referring to the Person proto

With Protocol Buffers you have a very minor extra step of providing an import referring the the proto file containing the referenced object.

But now the question is how will the (de)serializers know how to serialize and deserialize the object into the correct format? You've removed the definition from inside the file, so you need to get a reference to the schema as well. Fortunately, Schema Registry provides for using schema references.

What you need to do is register a schema for the person object first, then when you register the schema for the college and company schemas, you provide a reference to the already registered person schema.

Using the gradle schema-registry plugin makes this a simple task. Here's how you would configure it for using schema references:

Listing 3.33 Gradle plugin reference configuration

```
register {

    subject('person','src/main/avro/person.avsc', 'AVRO')         ❶
    subject('college-value','src/main/avro/college.avsc', 'AVRO')
        .addReference("bbejeck.chapter_3.avro.PersonAvro", "person", 1)   ❷
    subject('company-value','src/main/avro/company.avsc', 'AVRO')
        .addReference("bbejeck.chapter_3.avro.PersonAvro", "person", 1)   ❸
    }
```

❶ Register the person schema

❷ Register the college schema and add a reference to the person schema

❸ Register the company schema and add a reference to the person schema

So you first registered the `person.avsc` file, but in this case the subject is simply `person` because in this case it's not associated directly with any one topic. Then you registered both the college and company schemas using the `<topic name> - value` pattern as the college and company schemas are tied to topics with the same names and use the default subject name strategy (TopicNameStrategy) . The `addReference` method takes three parameters:

1. A name for the reference. Since you're using Avro it's the fully qualified name of the schema. For Protobuf it's the name of the proto file and for JSON schema it's the URL in the schema.
2. The subject name for the registered schema.
3. The version number to use for the reference.

Now with the references in place, you register the schemas and your producer and consumer client will be able to properly serialize and deserialize the objects with the references.

There are examples in the source code for running a producer and consumer with the schema references in action. Since you've already run the `./gradlew streams:registerSchemasTask` for the main module, you've already set up your references. To see using schema references in action you can run the following:

Listing 3.34 Tasks for schema references in action

```
./gradlew streams:runCompanyProducer
./gradlew streams:runCompanyConsumer

./gradlew streams:runCollegeProducer
./gradlew streams:runCollegeConsumer
```

3.5 Schema references and multiple events per topic

We've covered the different subject strategies `RecordNameStrategy`, and `TopicRecordNameStrategy` and how they allow for producing records of different types to a topic. But with the `RecordNameStrategy` any topic you produce to must use the same schema version for the given type. This means that if you want to make changes or evolve the schema, all topics must use the new schema. Using the `TopicRecordNameStrategy` allows for multiple events in a topic and it scopes the schema to a single topic, allowing you to evolve the schema independent of other topics.

But with both approaches you can't control the number of different types produced to the topic. If someone wants to produce a record of a different type that is not wanted, you don't have any way to enforce this policy.

However there is a way to achieve producing multiple event types to a topic ***and*** restrict the types of records produced to the topic by using schema references. By using `TopicNameStrategy` in conjunction with schema references, it allows all records in the topic to be constrained by a single subject. In other words, schema references allow you to have multiple types, but only those types that the schema refers to. This is best understood by walking through an example scenario

Imagine you are an online retailer and you've developed system for precise tracking of packages you ship to customers. You have a fleet of trucks and planes that take packages anywhere in the country. Each time a package handled along its route its scanned into your system generating one of three possible events represented by these domain objects: - `PlaneEvent`, `TruckEvent`, or a `DeliveryEvent`.

These are distinct events, but they are closely related. Also since the order of these events is important, you want them produced to the same topic so you have all related events together and in the proper sequence of their occurrence. I'll cover more about how combining related events in a single topic helps with sequencing in chapter 4 when we cover clients. Now assuming you've already created schemas for the PlaneEvent, TruckEvent, and the DeliveryEvent you could create an schema like this to contain the different event types:

Listing 3.35 Avro schema all_events.avsc with multiple events

```
[
  "bbejeck.chapter_3.avro.TruckEvent",      ❶
  "bbejeck.chapter_3.avro.PlaneEvent",
  "bbejeck.chapter_3.avro.DeliveryEvent"
]
```

❶ An Avro union type for the different events

The `all_events.avsc` schema file is an Avro `union`, which is an array of the possible event types. You use a `union` when a field, or, in this case a schema, could be of more then one type.

Since you're defining all the expected types in a single schema, your topic can now contain multiple types, but it's limited to only those listed in the schema. When using schema references in this format with Avro, it's critical to always set `auto.register.schemas=false` and `use.latest.version=true` in you Kafka producer configuration. Here's the reason why you need to use these configurations with the given settings.

When the Avro serializer goes to serialize the object, it won't find the schema for it, since it's in the union schema. As a result it will register the schema of the individual object, overwriting the union schema. So setting the auto registration of schemas to `false` avoids the overwriting of the schema problem. In addition, by specifying `use.latest.version=true`, the serializer will

retrieve the latest version of the schema (the union schema) and use that for serialization. Otherwise it would look for the event type in the subject name, and since it won't find it, a failure will result.

> **TIP** **When using the `oneOf` field with references in Protocol Buffers, the referenced schemas are automatically registered recursively, so can go ahead and use the `auto.register.schemas` configuration set to `true`. You can also do the same with JSON Schema `oneOf` fields.**

Let's now take a look at how you'd register the schema with references:

Listing 3.36 Register the all_events schema with references

```
subject('truck_event','src/main/avro/truck_event.avsc', 'AVRO')    ❶
subject('plane_event','src/main/avro/plane_event.avsc', 'AVRO')
subject('delivery_event','src/main/avro/delivery_event.avsc', 'AVRO')

subject('inventory-events-value', 'src/main/avro/all_events.avsc','AVRO')   ❷
    .addReference("bbejeck.chapter_3.avro.TruckEvent", "truck_event", 1)   ❸
    .addReference("bbejeck.chapter_3.avro.PlaneEvent", "plane_event", 1)
    .addReference("bbejeck.chapter_3.avro.DeliveryEvent", "delivery_event", 1)
```

❶ Registering the individual schemas referenced in the all_events.avsc file

❷ Registering all_events schema

❸ Adding the references of the individual schemas

As you saw before in the schema references section, with Avro you need to register the individual schemas before the schema containing the references. After that you can register the main schema with the references to the individual schemas.

When working with Protobuf there isn't a `union` type but there is a `oneOf` which is essentially the same thing. However with Protobuf you can't have a `oneOf` at the top-level, it must exist in an Protobuf message. For your Protobuf example, consider that you want to track the following customer interactions logins, searches, and purchases as separate events. But since they are closely related and sequencing is important you want them in the same topic. Here's the Protobuf file containing the references:

Listing 3.37 Protobuf file with references

```
syntax = "proto3";

package bbejeck.chapter_3.proto;

import "purchase_event.proto";    ❶
import "login_event.proto";
import "search_event.proto";

option java_outer_classname = "EventsProto";

message Events {

  oneof type {    ❷
    PurchaseEvent purchase_event = 1;
    LoginEvent login_event = 2;
    SearchEvent search_event = 3;
  }
  string key = 4;
}
```

❶ Importing the individual Protobuf messages

❷ The oneOf field which could be one of the three types listed

You've seen a Protobuf schema earlier in the chapter so I won't go over all the parts here, but the key thing for this example is the `oneOf` field `type` which could be a `PurchaseEvent`, `LoginEvent`, or a `SearchEvent`. When when you register a Protobuf schema it has enough information present to recursively register all of the referenced schemas, so it's safe to set the `auto.register` configuration to `true`.

You can structure your Avro references in a similar manner:

Listing 3.38 Avro schema with references using an outer class

```
{
  "type": "record",
  "namespace": "bbejeck.chapter_3.avro",
  "name": "TransportationEvent",    ❶

  "fields" : [
    {"name": "event", "type"[     ❷
      "bbejeck.chapter_3.avro.TruckEvent",    ❸
      "bbejeck.chapter_3.avro.PlaneEvent",
      "bbejeck.chapter_3.avro.DeliveryEvent"
    ]}
  ]
}
```

❶ Outer class name

❷ Field named "event"

❸ Avro union for the field type

So the main difference with this Avro schema vs. the previous Avro schema with references is

this one has outer class and the references are now a field in the class. Also, when you provide an outer class with Avro references like you have done here, you can now set the `auto.register` configuration to `true`, although you still need to register the schemas for the referenced objects ahead of time as Avro, unlike Protobuf, does not have enough information to recursively register the referenced objects.

There are some additional considerations when it comes to using multiple types with producers and consumers. I'm referring to the generics you use on the Java clients and how you can determine to take the appropriate action on an object depending on its concrete class name. I think these topics are better suited to discuss when we cover clients, so we'll cover that subject in the next chapter.

At this point, you've learned about the different schema compatibility strategies, how to work with schemas and using references. In all the examples you've run you've been using the built in serializers and deserializers provided by Schema Registry. In the next section we'll cover the configuration for the (de)serializers for producers and consumers. But we'll only cover the configurations related to the (de)serializers and not general producer and consumer configuration, those we'll cover in the next chapter.

3.6 Schema Registry (de)serializers

I've covered in the beginnings of the chapter,that when producing records to Kafka you need to serialize the records for transport over the network and storage in Kafka. Conversely, when consuming records you deserialize them so you can work with objects.

You need to configure the producer and consumer with the classes required for the serialization and deserialization process. Schema Registry provides a serializer, deserializer, and a Serde (used in Kafka Streams) for all three (Avro, Protobuf, JSON) supported types.

Providing the serialization tools is a strong argument for using Schema Registry that I spoke about earlier in the chapter. Freeing developers from having to write their own serialization code speeds up development and increases standardization across an organization. Also using a standard set of serialization tools reduces errors as reduces the chance that one team implements their own serialization framework.

NOTE **What's a Serde? A Serde is a class containing both a serializer and deserializer for a given type. You will use Serdes when working with Kafka Streams because you don't have access to the embedded producer and consumer so it makes sense to provide a class containing both and Kafka Streams uses the correct serializer and deserializer accordingly. You'll see Serdes in action when we start working with Kafka Streams in a later chapter.**

In the following sections I'm going to discuss the configuration for using Schema Registry aware serializers, deserializers. One important thing to remember is you don't configure the serializers directly. You set the configuration for serializers when you configure either the `KafkaProducer` or `KafkaConsumer`. If following sections aren't entirely clear to you, that's OK because we'll cover clients (producers and consumer) in the next chapter.

3.6.1 Avro

For Avro records there is the `KafkaAvroSerializer` and `KafkaAvroDeserializer` classes for serializing and deserializing records. When configuring a consumer, you'll need to include an additional property, `KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG=true` indicating that you want the deserializer to create a `SpecificRecord` instance. Otherwise the deserializer returns a `GenericRecord`.

Let's take a look at snippets of how you add these properties to both the producer and consumer. Note the following example only shows the configurations required for the serialization. I've left out the other configurations for clarity. We'll cover configuration of producers and consumers in chapter 4.

Listing 3.39 Required configuration for Avro

```
// producer properties
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);  ❶
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaAvroSerializer.class);  ❷
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");  ❸

//consumer properties these are set separately on the consumer
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  StringDeserializer.class);  ❹
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaAvroDeserializer.class);  ❺
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG,
  true);  ❻
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");  ❼
```

❶ The serializer for the key

❷ The serializer for the value

❸ Setting the URL for the serializer

❹ The deserializer for the key

❺ The deserializer for the value

❻ Indicating to construct a specific record instance

❼ Setting the URL for the deserializer

Next, let's take a look at the configuration for working with Protobuf records

3.6.2 Protobuf

For working with Protobuf records there are the `KafkaProtobufSerializer` and `KafkaProtobufDeserializer` classes.

When using Protobuf with schema registry, it's probably a good idea to specify both the `java_outer_classname` and set `java_multiple_files` to `true` in the protobuf schema. If you end up using the `RecordNameStrategy` with protobuf then you ***must*** use these properties so the deserializer can determine the type when creating an instance from the serialized bytes.

If you remember from earlier in the chapter we discussed that when using Schema Registry aware serializers, those serializers will attempt to register a new schema. If your protobuf schema references other schemas via imports, the referenced schemas are registered as well. Only protobuf provides this capability, when using Avro or JSON referenced schemas are not loaded automatically.

Again if you don't want auto registration of schemas, you can disable it with the following configuration `auto.shema.registration = false`.

Let's look at a similar example of providing the relevant Schema Registry configurations for working with protobuf records.

Listing 3.40 Required configuration for Protobuf

```
// producer properties
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);  ❶
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaProtobufSerializer.class);  ❷
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");  ❸

// consumer properties again set separately on the consumer
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  StringDeserializer.class);  ❹
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaProtobufDeserializer.class);  ❺
props.put(KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE_TYPE,
  AvengerSimpleProtos.AvengerSimple.class);  ❻
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");  ❼
```

❶ The key serializer

❷ The protobuf value serializer

❸ Providing the URL for Schema Registry for the consumer

❹ The key deserializer

❺ The protobuf value deserializer

❻ The specific class the deserializer should instantiate

❼ The location of Schema Registry for the producer

As with the Avro deserializer, you need to instruct it to create a specific instance. But in this case you configure the actual class name instead of setting a boolean flag indicating you want a specific class. If you leave out the specific value type configuration the deserializer returns a type of `DynamicRecord`. We covered working with the `DynamicRecord` in the protobuf schema section.

The `bbejeck.chapter_3.ProtobufProduceConsumeExample` class in the book source code demonstrates the producing and consuming a protobuf record.

Now we'll move on the final example of configuration of Schema Registry's supported types, JSON schemas.

3.6.3 JSON Schema

Schema Registry provides the `KafkaJsonSchemaSerializer` and `KafkaJsonSchemaDeserializer` for working with JSON schema objects. The configuration should feel familiar to both Avro and the Protobuf configurations.

NOTE **Schema Registry also provides `KafkaJsonSerializer` and `KafkaJsonDeserializer` classes. While the names are very similar these (de)serializers are meant for working with Java objects for conversion to and from JSON, without a JSON Schema. While the names are close, make sure you are using the serializer and deserializer with `Schema` in the name. We'll talks about the generic JSON serializers in the next section.**

Listing 3.41 Required configuration for JSON Schema

```
// producer configuration
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");   ❶
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);   ❷
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaJsonSchemaSerializer.class);   ❸

// consumer configuration
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");   ❹
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  StringDeserializer.class);   ❺
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaJsonSchemaDeserializer.class);   ❻
props.put(KafkaJsonDeserializerConfig.JSON_VALUE_TYPE,
  SimpleAvengerJson.class);   ❼
```

❶ Providing the URL for Schema Registry for the producer

❷ The key serializer

❸ The JSON Schema value serializer

❹ Providing the URL for Schema Registry for the producer

❺ The key deserializer

❻ Specifying the JSON Schema value deserializer

❼ Configuring the specific classes this deserializer will create

Here you can see a similarity with the protobuf configuration in that you need to specify the class the deserializer should construct from the serialized form in annotation number 7 in this example. If you leave out the specify value type then the deserializer returns a `Map`, the generic form of a JSON schema deserialization. Just a quick note the same applies for keys. If your key is a JSON schema object, then you'll need to supply a `KafkaJsonDeserializerConfig.JSON_KEY_TYPE` configuration for the deserializer to create the exact class.

There is a simple producer and consumer example for working with JSON schema objects in the `bbejeck.chapter_3.JsonSchemaProduceConsumeExample` in the source code for the book. As with the other basic producer and consumer examples, there are sections demonstrating how to work with the specific and generic return types. We outlined the structure of the JSON schema generic type in the JSON schema section of this chapter.

Now we've covered the different serializer and deserializer for each type of serialization supported by Schema Registry. Although using Schema Registry is recommended, it's not required. In the next section we'll outline how you can serialize and deserialize your Java objects without Schema Registry.

3.7 Serialization without Schema Registry

In the beginning of this chapter, I stated that your event objects, or more specifically their schema representations, are a contract between the producers and consumers of the Kafka event streaming platform. Schema Registry provides a central repository for those schemas hence providing enforcement of this schema contracts across your organization. Additionally, the Schema Registry provided serializers and deserializers provide a convenient way of working with data without having to write your own serialization code.

Does this mean using Schema Registry is required? No not at all. In some cases, you may not have access to Schema Registry or don't want to use it. Writing your own custom serializers and deserializers isn't hard. Remember, producers and consumers are decoupled from the (de)serializer implementation, you only provide the classname as a configuration setting. Although it's good to keep in mind that by using Schema Registry you can use the same schemas across Kafka Streams, Connect and ksqlDB.

So to create your own serializer and deserializer you create classes that implement the

`org.apache.kafka.common.serialization.Serializer` and `org.apache.kafka.common.serialization.Deserializer` interfaces. With the `Serializer` interface there is only one method you ***must*** implement `serialize`. For the `Deserializer` it's the `deserialize` method. Both interfaces have additional default methods (`configure`, `close`) you can override if you need to.

Here's a section of a custom serializer using the `jackson-databind` `ObjectMapper`:

Listing 3.42 Serialize method of a custom serializer

```
// details left out for clarity
@Override
public byte[] serialize(String topic, T data) {
    if (data == null) {
        return null;
    }
    try {
        return objectMapper.writeValueAsBytes(data);   ❶
    } catch (JsonProcessingException e) {
        throw new SerializationException(e);
    }
}
```

❶ Converting the given object to a byte array

Here you call `objectMapper.writeValueAsBytes()` and it returns a serialized representation of the passed in object.

Now let's look at an example for the deserializing counterpart:

Listing 3.43 Deserialize method of a custom deserializer

```
// details left out for clarity
@Override
public T deserialize(String topic, byte[] data) {
    try {
        return objectMapper.readValue(data, objectClass);   ❶
    } catch (IOException e) {
        throw new SerializationException(e);
    }
}
```

❶ Converting the bytes back to an object specified by the objectClass parameter

The `bbejeck.serializers` package contains the serializers and deserializers shown here and additional ones for Protobuf. You can use these serializers/deserializers in any of the examples presented in this book but remember that they don't use Schema Registry. Or they can serve as examples of how to implement your own (de)serializers.

In this chapter, we've covered how event objects or more specifically, their schemas, represent contract between producers and consumers. We discussed how Schema Registry stores these schemas and enforces this implied contract across the Kafka platform. Finally we covered the

supported serialization formats of Avro, Protobuf and JSON. In the next chapter, you'll move up even further in the event streaming platform to learn about Kafka clients, the `KafkaProducer` and `KafkaConsumer`. If you think of Kafka as your central nervous system for data, then the clients are the the sensory inputs and outputs for it.

3.8 Summary

- Schemas represent a contract between producers and consumers. Even if you don't use explicit schemas, you have an implied one with your domain objects, so developing a way to enforce this contract between producers and consumers is critical.
- Schema Registry stores all your schemas enforcing data governance and it provides versioning and three different schema compatibility strategies - backward, forward and full. The compatibility strategies provide assurance that the new schema will work with it's immediate predecessor, but not necessarily older ones. For full compatibility across all versions you need to use backward-transitive, forward-transitive, and full-transitive. Schema Registry also provides a convenient REST API for uploading, view and testing schema compatibility.
- Schema Registry supports three type of serialization formats Avro, Protocol Buffers, and JSON Schema. It also provides integrated serializers and deserializers you can plug into your KafkaProducer and KafkaConsumer instances for seamless support for all three supported types. The provided (de)serializers cache schemas locally and only fetch them from Schema Registry when it can't locate a schema in the cache.
- Using code generation with tools such as Avro and Protobuf or open source plugins supporting JSON Schema help speed up development and eliminate human error. Plugins that integrate with Gradle and Maven also provide the ability to test and upload schemas in the developers normal build cycle.

4 Kafka clients

This chapter covers

- Producing records with the KafkaProducer
- Understanding message delivery semantics
- Consuming records with the KafkaConsumer
- Learning about Kafka's exactly-once streaming
- Using the Admin API for programmatic topic management
- Handling multiple event types in a single topic

This chapter is where the "rubber hits the road" and we take what you've learned over the previous two chapters and apply it here to start building event streaming applications. We'll start out by working with the producer and consumer clients individually to gain a deep understanding how each one works.

In their simplest form, clients operate like this: producers send records (in a produce request) to a broker and the broker stores them in a topic and consumers send a fetch request and the broker retrieves records from the topic to fulfill that request. When we talk about the Kafka event streaming platform, it's common to mention both producers and consumers. After all, it's a safe assumption that you are producing data for someone else to consume. But it's very important to understand that the producers and consumers are unaware of each other, there's no synchronization between these two clients.

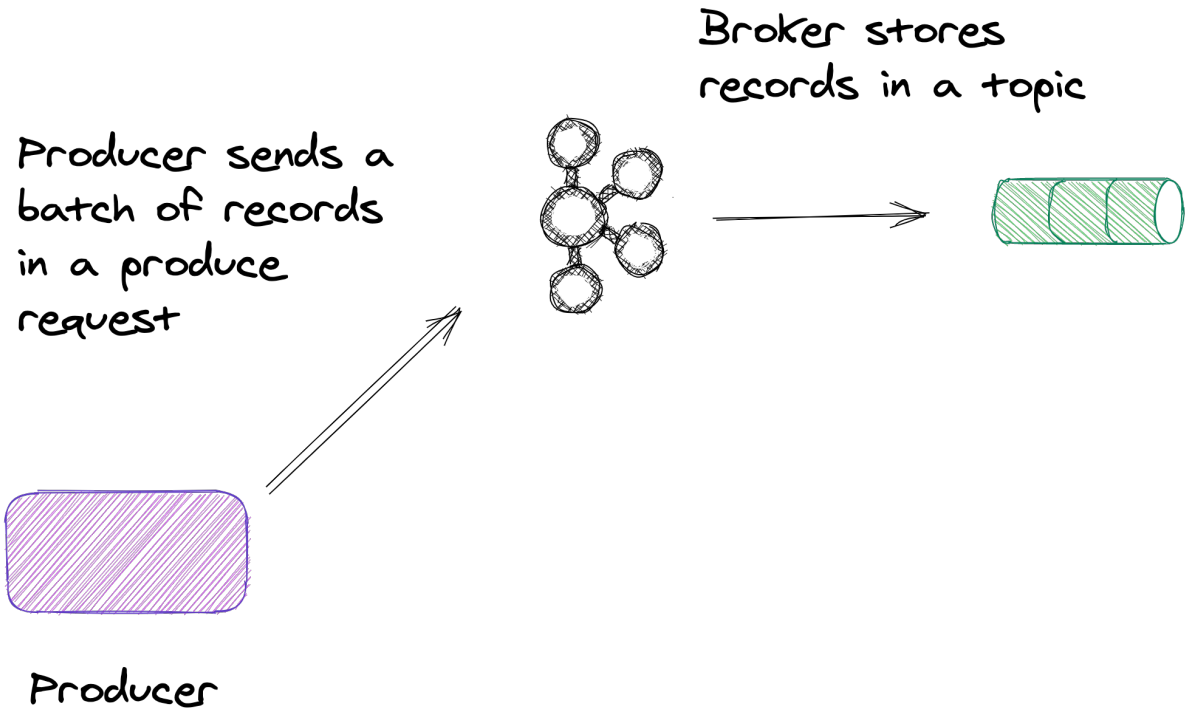


Figure 4.1 Producers send batches of records to Kafka in a produce request

The `KafkaProducer` has just one task, sending records to the broker. The records themselves contain all the information the broker needs to store them.

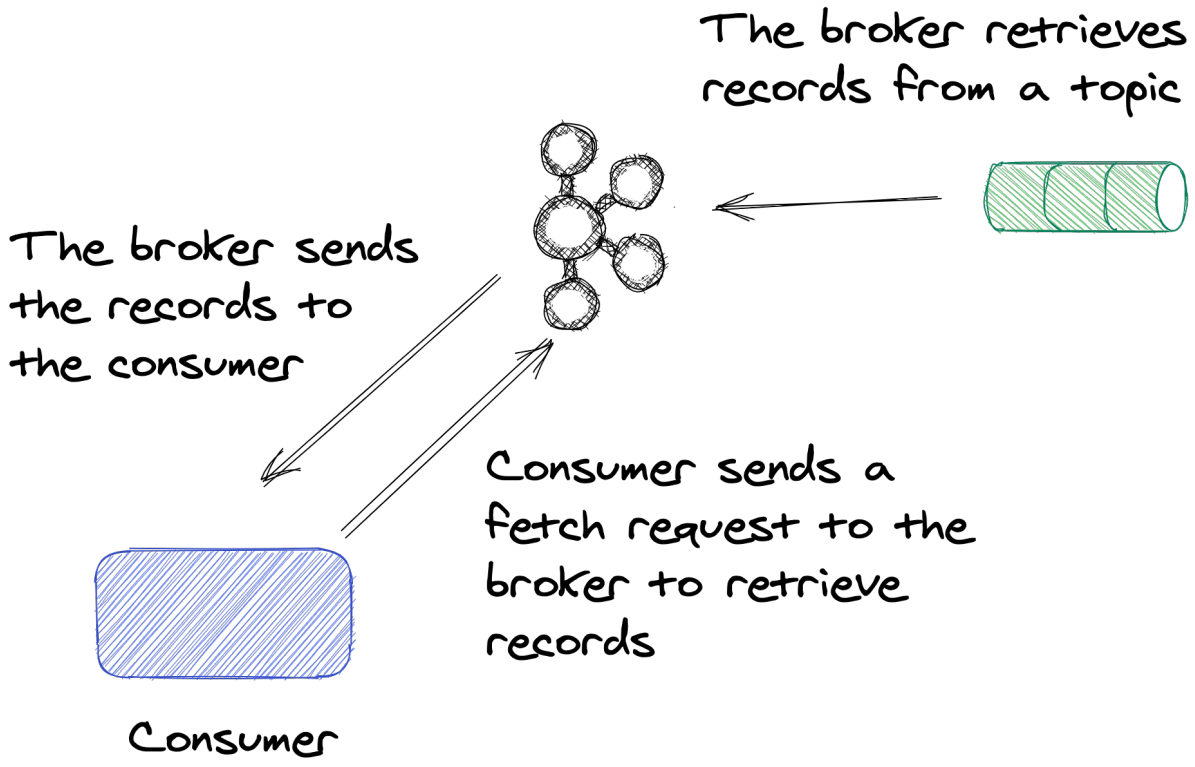


Figure 4.2 Consumers send fetch requests to consume records from a topic, the broker retrieves those records to fulfill the request

The `KafkaConsumer` on the other hand only reads or consumes records from a topic. Also, as we mentioned in the chapter covering the Kafka broker, the broker handles the storage of the records. The act of consuming records has no impact on how long the broker retains them.

In this chapter you'll take a `KafkaProducer` and dive into the essential configurations and walk through examples of producing records to the Kafka broker. Learning how the `KafkaProducer` works is important because that's the crucial starting point for building event streaming applications; getting the records into Kafka.

Next you'll move on to learning how to use the `KafkaConsumer`. Again we'll cover the vital configuration settings and from working with some examples, you'll see how an event streaming application works by continually consuming records from the Kafka broker. You've started your event streaming journey by getting your data into Kafka, but it's when you start consuming the data that you start building useful applications.

Then we'll go into working with the `Admin` interface. As the name implies, it's a client that allows you to perform administrative functions programmatically.

From there you'll get into more advanced subject matter such as the idempotent producer configuration which guarantees you per partition, exactly-once message delivery and the Kafka transnational API for exactly once delivery across multiple partitions.

When you're done with this chapter you'll know how to build event streaming applications using the `KafkaProducer` and `KafkaConsumer` clients. Additionally, you'll have a good understanding how they work so you can recognize when you have a good use-case for including them in your application. You should also come away with a good sense of how to configure the clients to make sure your applications are robust and can handle situations when things don't go as expected.

So with this overview in mind, we are going to embark on a guided tour of how the clients do their jobs. First we'll discuss the producer, then we'll cover the consumer. Along the way we'll take some time going into deeper details, then we'll come back up and continue along with the tour.

4.1 Producing records with the KafkaProducer

You've seen the `KafkaProducer` some in chapter three when we covered Schema Registry, but I didn't go into the details of how the producer works. Let's do that now.

Say you work on the data ingest team for a medium sized wholesale company. You get transaction data delivered via a point of sale service and several different departments within the company want access to the data for things such as reporting, inventory control, detecting trends

etc.

You've been tasked with providing a reliable and fast way of making that information available to anyone within the company that wants access. The company, Vandelay Industries, uses Kafka to handle all of its event streaming needs and you realize this is your opportunity to get involved. The sales data contains the following fields:

1. Product name
2. Per-unit price
3. Quantity of the order
4. The timestamp of the order
5. Customer name

At this point in your data pipeline, you don't need to do anything with the sales data other than to send it into a Kafka topic, which makes it available for anyone in the company to consume

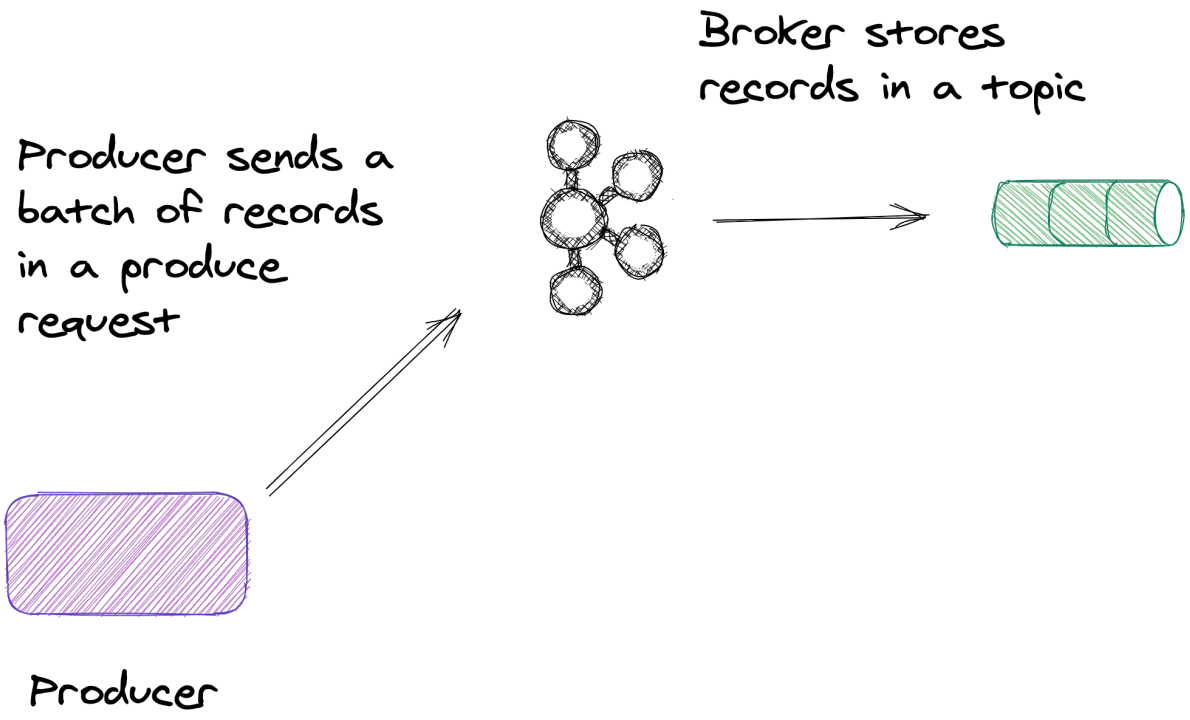


Figure 4.3 Sending the data into a Kafka Topic

To make sure everyone is on the same page with the structure of the data, you've modeled the records with a schema and published it to Schema Registry. All that's left is for you to do write the `KafkaProducer` code to take the sales records and send them into Kafka. Here's what your code looks like

Listing 4.1 A KafkaProducer the source code can be found at bbejeck.chapter_4.sales.SalesProducerClient

```
// There are some details left out for clarity here in the text
try (
Producer<String, ProductTransaction> producer = new KafkaProducer<>(
  producerConfigs)) {   ❶
   while(keepProducing) {
    Collection<ProductTransaction> purchases = salesDataSource.fetch();   ❷
     purchases.forEach(purchase -> {
        ProducerRecord<String, ProductTransaction> producerRecord =
             new ProducerRecord<>(topicName, purchase.getCustomerName(),
              purchase);   ❸
        producer.send(producerRecord,
          (RecordMetadata metadata, Exception exception) -> {      ❹
              if (exception != null) {     ❺
                  LOG.error("Error producing records ", exception);
            } else {
              LOG.info("Produced record at offset {} with timestamp {}",   ❻
                          metadata.offset(), metadata.timestamp());
              }
          });
     });
```

❶ Creating the KafkaProducer instance using a try-with-resources statement so the producer closes automatically when the code exits

❷ The data source providing the sales transaction records

❸ Creating the ProducerRecord from the incoming data

❹ Sending the record to the Kafka broker and providing a lambda for the Callback instance

❺ Logging if an exception occured with the produce request

❻ In the success case logging the offset and timestamp of the record stored in the topic

Notice at annotation one the `KafkaProducer` takes a `Map` of configuration items (In a section following this example we'll dicusss some of the more important `KafkaProducer` configurations). At annotation number 2, we're going to use a data generator to simulate the delivery of sales records. You take the list of `ProductTransaction` objects and use the Java stream API to map each object in the list into a `ProducerRecord` object.

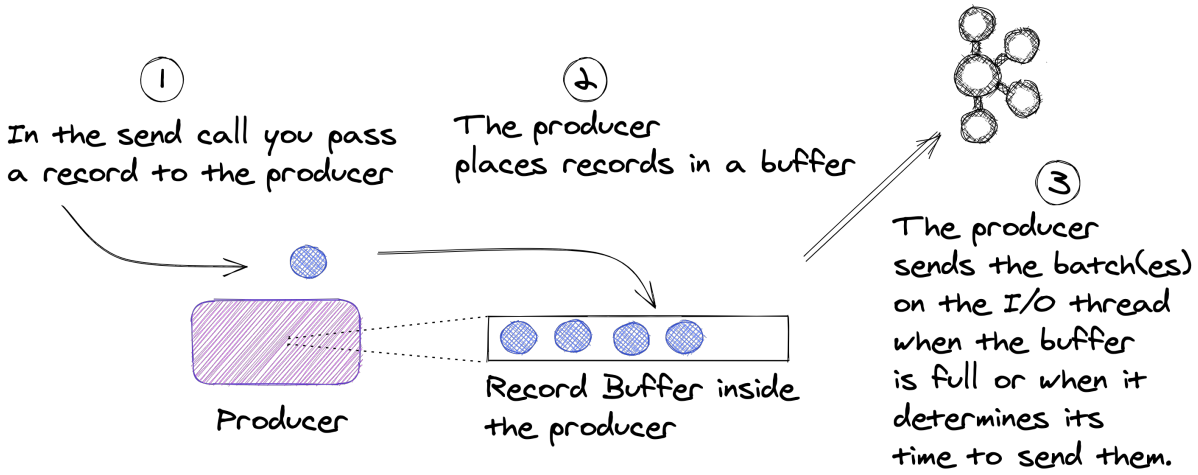


Figure 4.4 The Producer batches records and sends them to the broker when the buffer is full or it's time to send them

For each `ProducerRecord` created you pass it as a parameter to the `KafkaProducer.send()` method. However, the producer does not immediately send the record to the broker, instead it attempts to batch up records. By using batches the producer makes fewer requests which helps with performance on both the broker and the producer client. The `KafkaProducer.send()` call is asynchronous to allow for continually adding records to a batch. The producer has a separate a thread (the I/O thread) that can send records when the batch is full or when it decides it's time so transmit the batch.

There are two signatures for the `send` method. The version you are using in the code here accepts a `ProducerRecord` and `Callback` object as parameters. But since the `Callback` interface only contains one method, also known as functional interface, we can use a lambda expression instead of a concrete implementation. The producer I/O thread executes the `Callback` when the broker acknowledges the record as persisted.

The `Callback.onCompletion` method, again represented here as a lambda, accepts two parameters `RecordMetadata` and `Exception`. The `RecordMetadata` object contains metadata of the record the broker has acknowledged. Referring back to our discussion on the `acks` configuration, the `RecordMetadata.offset` field is `-1` if you have `acks=0`. The offset is `-1` because the producer doesn't wait for acknowledgment from the broker, so it can't report the offset assigned to the record. The exception parameter is non-null if an error occurred.

Since the producer I/O thread executes the callback, it's best if you don't do any heavy processing as that would hold up sending of records. The other overloaded `KafkaProducer.send()` method only accepts a `ProducerRecord` parameter and returns a `Future<RecordMetadata>`. Calling the `Future.get()` method blocks until the broker acknowledges the record (request completion). Note that if an error occurs during the send then executing the `get` method throws an exception.

Generally speaking it's better to use the `send` method with the `Callback` parameter as it's a bit cleaner to have the I/O thread handle the results of the send asynchronously vs. having to keep track of every `Future` resulting from the send calls.

At this point we've covered the fundamental behavior for a `KafkaProducer`, but before we move onto consuming records, we should take a moment to discuss other important subjects involving the producer: configurations, delivery semantics, partition assignment, and timestamps.

4.1.1 Producer configurations

- `bootstrap.servers` - One or more host:port configurations specifying a broker for the producer to connect to. Here we have a single value because this code runs against a single broker in development. In a production setting, you could list every broker in your cluster in a comma separated list.
- `key.serializer` - The serializer for converting the key into bytes. In this example, the key is a `String` so we can use the `StringSerializer` class provided with the Kafka clients. The `org.apache.kafka.common.serialization` package contains serializers for `String`, `Integer`, `Double` etc. You could also use Avro, Protobuf, or JSON Schema for the key and use the appropriate serializer.
- `value.serializer` - The serializer for the value. Here we're using object generated from an Avro schema. Since we're using Schema Registry, we'll use the `KafkaAvroSerializer` we saw from chapter 3. But the value could also be a String, Integer etc and you would use one of the serializers from the `org.apache.kafka.common.serialization` package.
- `acks` - The number of acknowledgments required to consider the produce request successful. The valid values are "0", "1", "all" the `acks` configuration is one of the most important to understand as it has a direct impact on data durability. Let's go through the different settings here.
    - Zero (`acks=0`) Using a value of 0 means the producer will not wait for any acknowledgment from the broker about persisting the records. The producer considers the send successfully immediately after transmitting it to the broker. You could think using `acks=0` as "fire and forget". Using this setting has the highest throughput, but has the lowest guarantee on data durability.
    - One (`acks=1`) A setting of one means the producer waits for notification from the lead broker for the topic-partition that it successfully persisted the record to its log. But the producer doesn't wait for acknowledgment from the leader that any of the followers persisted the record. While you have a little more assurance on the durability of the record in this case, should the lead broker fail before the followers replicate the record, it will be lost.
    - All (`acks=all`) This setting gives the highest guarantee of data durability. In this case, the producer waits for acknowledgment from the lead broker that it successfully persisted the record to its own log ***and*** the following in-sync brokers were able to persist the record as well. This setting has the lowest throughput, but the highest durability guarantees. When using the `acks=all` setting it's advised to set the `min.insync.replicas` configuration for your topics to a value higher than the default of `1`. For example with a replication factor of 3, setting `min.insyc.replicas=2` means the producer will raise an exception if there are not enough replicas available for persisting a record. We'll go into more detail on this scenario later in this chapter.
- `delivery.timeout.ms` - This is an upper bound on the amount of time you want to wait for a response after calling `KafkaProducer.send()`. Since Kafka is a distributed system, failures delivering records to the broker are going to occur. But in many cases these errors are temporary and hence re-tryable. For example the producer may encounter trouble connecting due to a network partition. But network connectivity can be a temporary issue, so the producer will re-try sending the batch and in a lot cases the re-sending of records succeeds. But after a certain point, you'll want the producer to stop

trying and throw an error as prolonged connectivity problems mean there's an issue that needs attention. Note that if the producer encounters what's considered a fatal error, then the producer will throw an exception before this timeout expires.

- `retries` - When the producer encounters an non-fatal error, it will retry sending the record batch. The producer will continue to retry until the `delivery.timeout.ms` timeout expires. The default value for `retries` is `INTEGER.MAX_VALUE`. Generally you should leave the retried configuration at the default value. If you want to limit the amount of retries a producer makes, you should reduce the amount of time for the `delivery.timeout.ms` configuration. With errors and retries it's possible that records could arrive out of order to the same partition. Consider the producer sends a batch of records but there is an error forcing a retry. But in the intervening time the producer sends a second batch that did not encounter any errors. The first batch succeeds in the subsequent retry, but now it's appended to the topic ***after*** the second batch. To avoid this issue you can set the configuration `max.in.flight.requests.per.connection=1`. Another approach to avoid the possibility of out of order batches is to use the `idempotent producer` which we'll discuss in 4.3.1 in this chapter.

Now that you have learned about the concept of retries and record acknowledgments, let's look at message delivery semantics now.

4.1.2 Kafka delivery semantics

Kafka provides three different delivery semantic types: at least once, at most once, and exactly once. Let's discuss each of them here.

- At least once - With "at least once" a records are never lost, but may be delivered more than once. From the producer's perspective this can happen when a producer sends a batch of records to the broker. The broker appends the records to the topic-partition, but the producer does not receive the acknowledgment in time. In this case the producer re-sends the batch of records. From the consumer point of view, you have processed incoming records, but before the consumer can commit, an error occurs. Your application reprocesses data from the last committed offset which includes records already processed, so there are duplicates as a result. Records are never lost, but may be delivered more than once. Kafka provides at least once delivery by default.
- At most once - records are successfully delivered, but may be lost in the event of an error. From the producer standpoint enabling `acks=0` would be an example of at most once semantics. Since the producer does not wait for any acknowledgment as soon as it sends the records it has no notion if the broker either received them or appended them to the topic. From the consumer perspective, it commits the offsets before confirming a write so in the event of an error, it will not start processing from the missed records as the consumer already committed the offsets. To achieve at "at most once" producers set `acks=0` and consumers commit offsets before doing any processing.
- Exactly once - With "exactly once" semantics records are neither delivered more than once or lost. Kafka uses transactions to achieve exactly once semantics. If a transaction is aborted, the consumers internal position gets reset to the offset prior to the start of the transaction and the stored offsets aren't visible to any consumer configured with `read_committed`.

Both of these concepts are critical elements of Kafka's design. Partitions determine the level of parallelism and allow Kafka to distribute the load of a topic's records to multiple brokers in a

cluster. The broker uses timestamps to determine which log segments it will delete. In Kafka Streams, they drive progress of records through a topology (we'll come back to timestamps in the Kafka Streams chapter).

4.1.3 Partition assignment

When it comes to assigning a partition to a record, there are three possibilities:

1. If you provide a valid partition number, then it's used when sending the record
2. If you don't give the partition number, but there is a key, then the producer sets the partition number by taking the hash of the key modulo the number of partitions.
3. Without providing a partition number or key, the `KafkaProducer` sets the partition by alternating the partition numbers for the topic. The approach to assigning partitions without keys has changed some as of the 2.4 release of Kafka and we'll discuss that change now.

Prior to Kafka 2.4, the default partitioner assigned partitions on a round-robin basis. That meant the producer assigned a partition to a record, it would increment the partition number for the next record. Following this round-robin approach, results in sending multiple, smaller batches to the broker. The following illustration will help clarify what is going on:

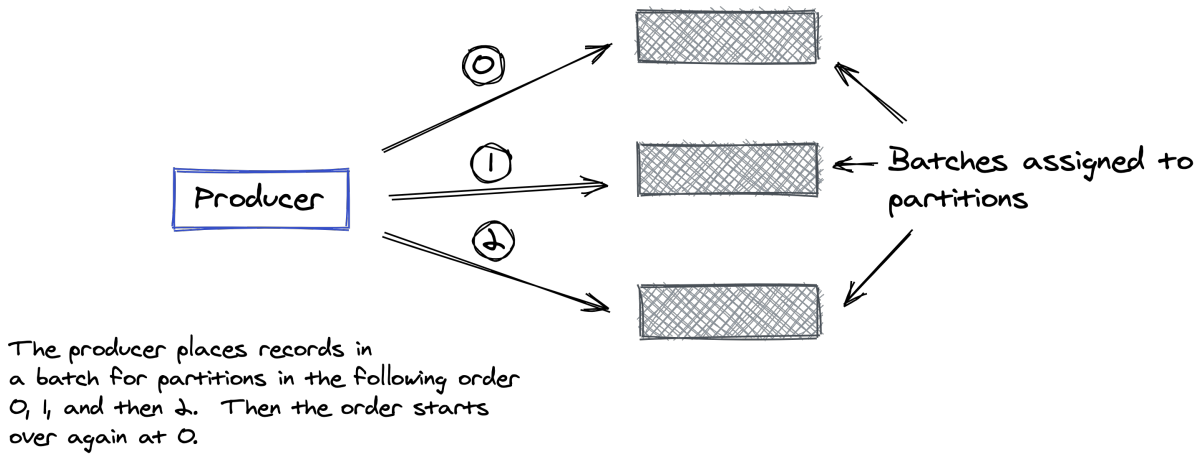


Figure 4.5 Round robin partition assignment

This approach also led to more load on the broker due to a higher number of requests.

But now when you don't provide a key or partition for the record, the partitioner assigns a partition for the record per batch. This means when the producer flushes its buffer and sends records to the broker, the batch is for single partition resulting in a single request. Let's take a look at an illustration to visualize how this works:

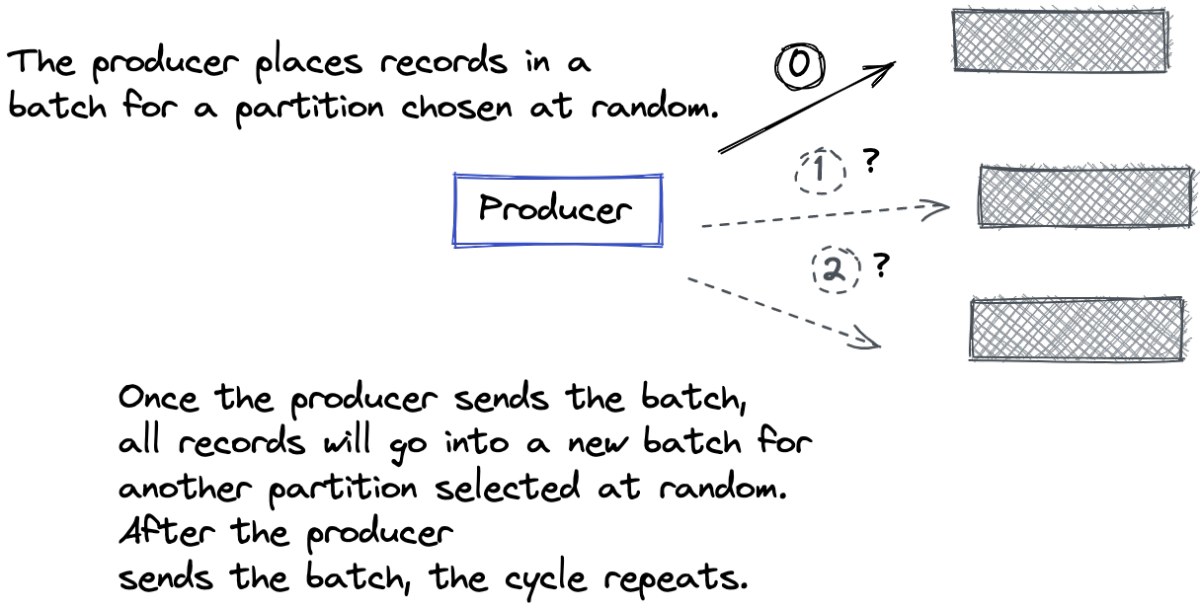


Figure 4.6 Sticky partition assignment

After sending the batch, the partitioner selects a partition at random and assigns it to the next batch. In time, there's still an even distribution of records across all partitions, but it's done one batch at a time.

Sometimes the provided partitioners may not suit your requirements and you'll need finer grained control over partition assignment. For those cases you can write your own custom partitioner.

4.1.4 Writing a custom partitioner

Let's revisit the producer application from the 4.1 section above. The key is the name of the customer, but you have some orders that don't follow the typical process and end up with a customer name of "CUSTOM" and you'd prefer to restrict those orders to a single partition 0, and have all other orders on partition 1 or higher.

So in this case, you'll need to write a custom partitioner that can look at the key and return the appropriate partition number.

The following example custom partitioner does just that. The `CustomOrderPartitioner` (from src/main/java/bbejeck/chapter_4/sales/CustomOrderPartitioner.java) examines the key to determine which partition to use.

Listing 4.2 `CustomOrderPartitioner` custom partitioner

```
public class CustomOrderPartitioner implements Partitioner {

// Some details omitted for clarity

@Override
public int partition(String topic,
                     Object key,
                     byte[] keyBytes,
                     Object value,
                     byte[] valueBytes,
                     Cluster cluster) {

    Objects.requireNonNull(key, "Key can't be null");
    int numPartitions = cluster.partitionCountForTopic(topic);   ❶
    String strKey = (String) key;
    int partition;

    if (strKey.equals("CUSTOM")) {
        partition = 0;    ❷
    } else {
        byte[] bytes = strKey.getBytes(StandardCharsets.UTF_8);
        partition = Utils.toPositive(Utils.murmur2(bytes)) %
                                     (numPartitions - 1) + 1;   ❸
    }
    return partition;
  }
}
```

❶ Retrieve the number of partitions for the topic

❷ If the name of the customer is "CUSTOM" return 0

❸ Determine the partition to use in the non-custom order case

To create your own partitioner you implement the `Partitioner` interface which has 3 methods, `partition`, `configure`, and `close`. I'm only showing the `partition` method here as the other two are no-ops in this particular case. The logic is straight forward; if the customer name equates to "CUSTOM", return zero for the partition. Otherwise you determine the partition as usual, but with a small twist. First we subtract one from the number of candidate partitions since the 0 partition is reserved. Then we shift the partiton number by 1 which ensures we always return 1 or greater for the non-custom order case.

> **NOTE** **This example does not represent a typical use case and is presented only for the purpose of demonstrating how to you can provide a custom partitioner. In most cases it's best to go with one of the provided ones.**

You've just seen how to construct a custom partitioner and next we'll wire it up with our producer.

4.1.5 Specifying a custom partitioner

Now that you've written a custom partitioner, you need to tell the producer you want to use it instead of the default partitioner. You specify a different partitioner when configuring the Kafka producer:

```
producerConfigs.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
 CustomOrderPartitioner.class);
```

The bbejeck.chapter_4.sales.SalesProducerClient is configured to use the `CustomOrderPartitioner`, but you can simply comment out the line if you don't want to use it. You should note that since the partitioner config is a producer setting, it must be done on each one you want to use the custom partitioner.

4.1.6 Timestamps

The `ProducerRecord` object contains a timestamp field of type `Long`. If you don't provide a timestamp, the `KafkaProducer` adds one to the record, which is simply the current time of the system the producer is running on. Timestamps are an important concept in Kafka. The broker uses them to determine when to delete records, by taking the oldest timestamp in a segment and comparing it to the current time. If the difference exceeds the configured retention time, the broker removes the segment. Kafka Streams and ksqlDB also rely heavily on timestamps, but I'll defer those discussions until we get to their respective chapters.

There are two possible timestamps that Kafka may use depending on the configuration of the topic.

In Kafka topics have a configuration, `message.timestamp.type` which can either be `CreateTime` or `LogAppendTime`. A configuration of `CreatTime` means the broker stores the record with the timestamp provided by the producer. If you configure your topic with `LogAppendTime`, then the broker overwrites the timestamp in the record with its current wall-clock (i.e, system) time when the broker appends the record in the topic. In practice, the difference between these timestamps should be close.

Another consideration is that you can embed the timestamp of the event in payload of the record value when you are creating it.

This wraps up our discussion on the producer related issues. Next we'll move on to the mirror image of producing records to Kafka, consuming records.

4.2 Consuming records with the KafkaConsumer

So you're back on the job at Vandelay Industries and you now have a new task. Your producer application is up an running happily pushing sales records into a topic. But now you're asked to develop a `KafkaConsumer` application to serve as a model for consuming records from a Kafka topic.

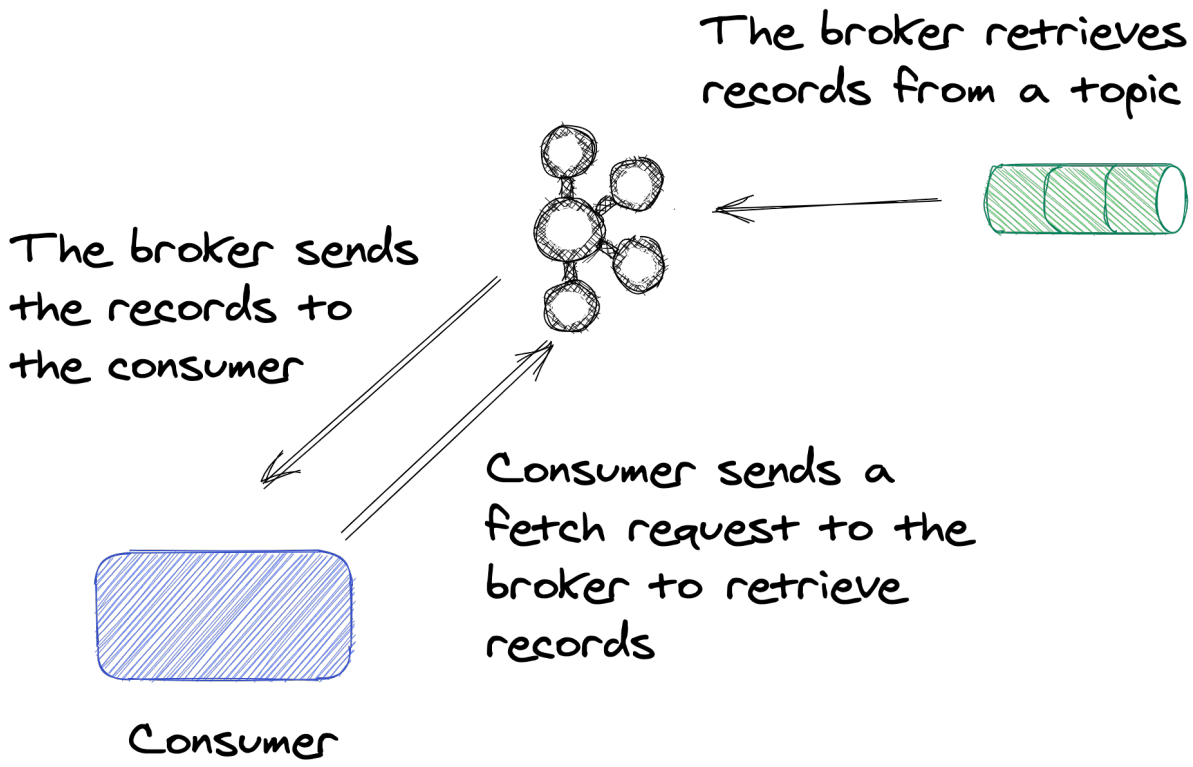


Figure 4.7 Consumers send fetch requests to consume records from a topic, the broker retrieves those records to fulfill the request

The `KafkaConsumer` sends a fetch request to the broker to retrieve records from topics it's subscribed to. The consumer makes what known as a `poll` call to get the records. But each time the consumer polls, it doesn't necessarily result in the broker fetching records. Instead it could be retrieving records cached by a previous call.

> **NOTE** There are producer and consumer clients available in other programming languages, but in this book we'll focus on the clients available in the Apache Kafka distribution, which are written in Java. To see a list of clients available in other languages checkout take a look at this resource docs.confluent.io/platform/current/clients/index.html#ak-clients

Let's get started by looking at the code for creating a `KafkaConsumer` instance:

Listing 4.3 KafkaConsumer code found in bbejeck.chapter_4.sales.SalesConsumerClient

```
// Details left out for clarity
 try (
  final Consumer<String, ProductTransaction> consumer = new KafkaConsumer<>(
    consumerConfigs)) {   ❶
    consumer.subscribe(topicNames);     ❷
    while (keepConsuming) {
        ConsumerRecords<String, ProductTransaction> consumerRecords =
          consumer.poll(Duration.ofSeconds(5));   ❸
        consumerRecords.forEach(record -> {       ❹
            ProductTransaction pt = record.value();
            LOG.info("Sale for {} with product {} for a total sale of {}",
                    record.key(),
                    pt.getProductName(),
                    pt.getQuantity() * pt.getPrice());
        });
    }
}
```

❶ Creating the new consumer instance

❷ Subscribing to topic(s)

❸ Polling for records

❹ Doing some processing with each of the returned records

In this code example, you're creating a `KafkaConsumer`, again using the try-with-resources statement. After subscribing to a topic or topics, you begin processing records returned by the `KafkaConsumer.poll` method. When the `poll` call returns records, you start doing some processing with them. In this example case we're simply logging out the details of the sales transactions.

TIP **Whenever you create either a `KafkaProducer` or `KafkaConsumer` you need to close them when your done to make sure you clean up all of the threads and socket connections. The try-with-resources (docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) in Java ensures that resources created in the `try` portion are closed at the end of the statement. It's a good practice to always use the try-with-resources statement as it's easy to overlook adding a `close` call on either a producer or a consumer.**

You'll notice that just like with the producer, you create a `Map` of configurations and pass them as a parameter to the constructor. Here I'm going to show you some of the more prominent ones.

- `bootstrap.servers` - One or more host:port configurations specifying a broker for the consumer to connect to. Here we have a single value, but this could be a comma separated list.
- `max.poll.interval.ms` - The maximum amount of time a consumer can take between calls to `KafkaConsumer.poll()` otherwise the consumer group coordinator considers

the individual consumer non-active and triggers a rebalance. We'll talk more about the consumer group coordinator and relabances in this section.

- `group.id` - An arbitrary string value used to associate individual consumers as part of the same consumer group. Kafka uses the concept of a consumer group to logically map multiple consumers as one consumer.
- `enable.auto.commit` - A boolean flag that sets whether the consumer will automatically commit offsets. If you set this to false, your application code must manually commit the offsets of records you considered successfully processed.
- `auto.commit.interval.ms` - The time interval at which offsets are automatically committed.
- `auto.offset.reset` - When a consumer starts it will resume consuming from the last committed offset. If offsets aren't available for the consumer then this configuration specifies where to start consuming records, either the earliest available offset or the latest which means the offset of the next record that arrives after the consumer started.
- `key.deserializer.class` - The classname of the deserializer the consumer uses to convert record key bytes into the expected object type for the key.
- `value.deserializer.class` - The classname of the deserializer the consumer uses to convert record value bytes into the expected object type for the value. Here we're using the provided `KafkaAvroDeserializer` for the value which requires the `schema.registry.url` configuration which we have in our configuration.

The code we use in our first consumer application is fairly simple, but that's not the main point. Your business logic, what you do when you consume the records is always going to be different on a case-by-case basis.

It's more important to grasp how the `KafkaConsumer` works and the implications of the different configurations. By having this understanding you'll be in a better position to know how to best write the code for performing the desired operations on the consumed records. So just as we did in the producer example, we're going to take a detour from our narrative and go a little deeper on the implications of these different consumer configurations.

4.2.1 The poll interval

Let's first discuss the roll of `max.poll.interval.ms`. It will be helpful to look at an illustration of what the poll interval configuration in action to get a full understanding:

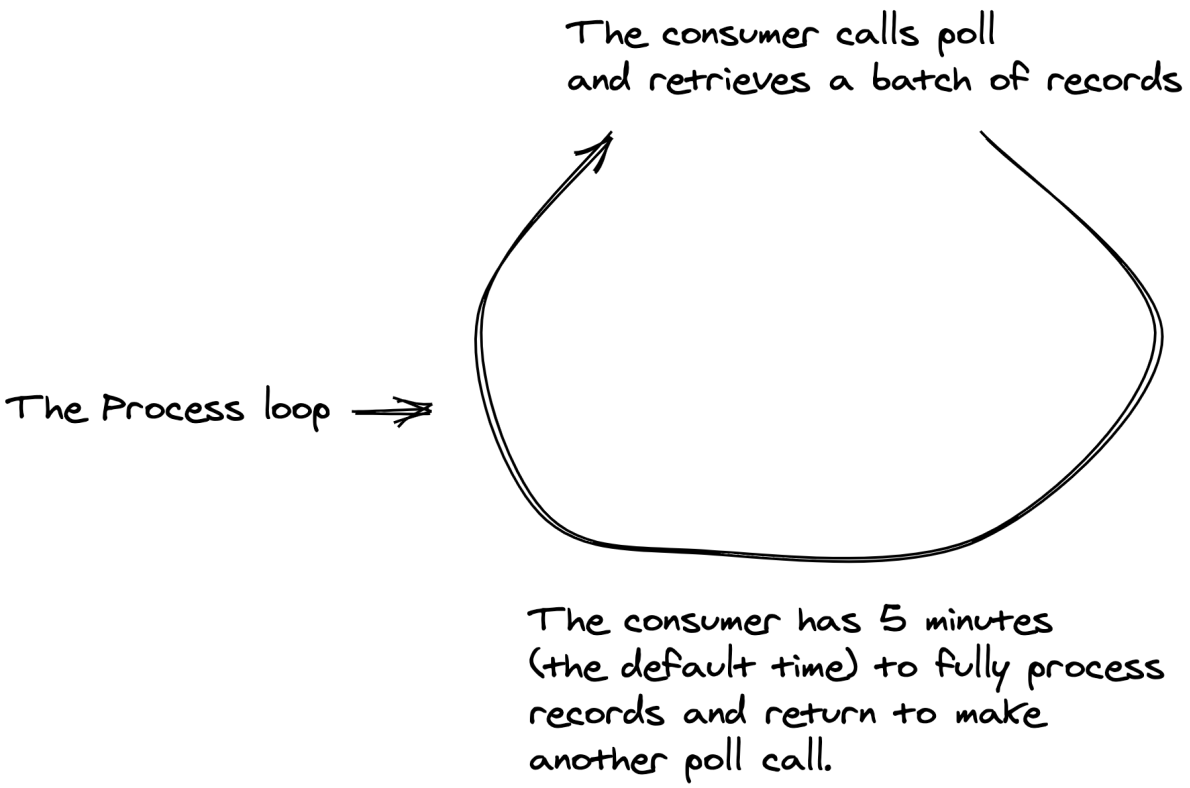


Figure 4.8 The max.poll.interval.ms configuration specifies how long a consumer may take between calls to `KafkaConsumer.poll()` before the consumer is considered inactive and removed from the consumer group

In the illustration here, the consumer processing loop starts with a call to `KafkaConsumer.poll(Duration.ofSeconds(5))`, the time passed to the `poll(Duration)` call is the maximum time the consumer waits for new records, in this case five seconds. When the `poll(Duration)` call returns, if there are any records present, the `for` loop over the `ConsumerRecords` executes your code over each one. Had there been no records returned, the outer `while` loop simply goes back to the top for another `poll(Duration)` call.

Going through this illustration, iterating over all the records and execution for each record must complete before the `max.poll.interval.ms` time elapses. By default this value is five minutes, so if your processing of returned records takes longer, then that individual consumer is considered dead and the group coordinator removes the consumer from the group and triggers a rebalance. I know that I've mentioned a few new terms in group coordinator and rebalancing, we'll cover them in the next section when we cover the `group.id` configuration.

If you find that your processing takes longer than the `max.poll.interval.ms` there are a couple of things you can do. The first approach would be to validate what you're doing when processing the records and look for ways to speed up the processing. If you find there's no changes to make to your code, the next step could be to to reduce the maximum number of records the consumer

retrieves from a `poll` call. You can do this by setting the `max.poll.records` configuration to a setting less than the default of 500. I don't have any recommendations, you'll have to experiment some to come up with a good number.

4.2.2 Group id

The `group.id` configuration takes into a deeper conversation about consumer groups in Kafka. Kafka consumers use a `group.id` configuration which Kafka uses to map all consumers with the same `group.id` into the same consumer group. A consumer group is a way to logically treat all members of the group as one consumer. Here's an illustration to demonstrating how group membership works:

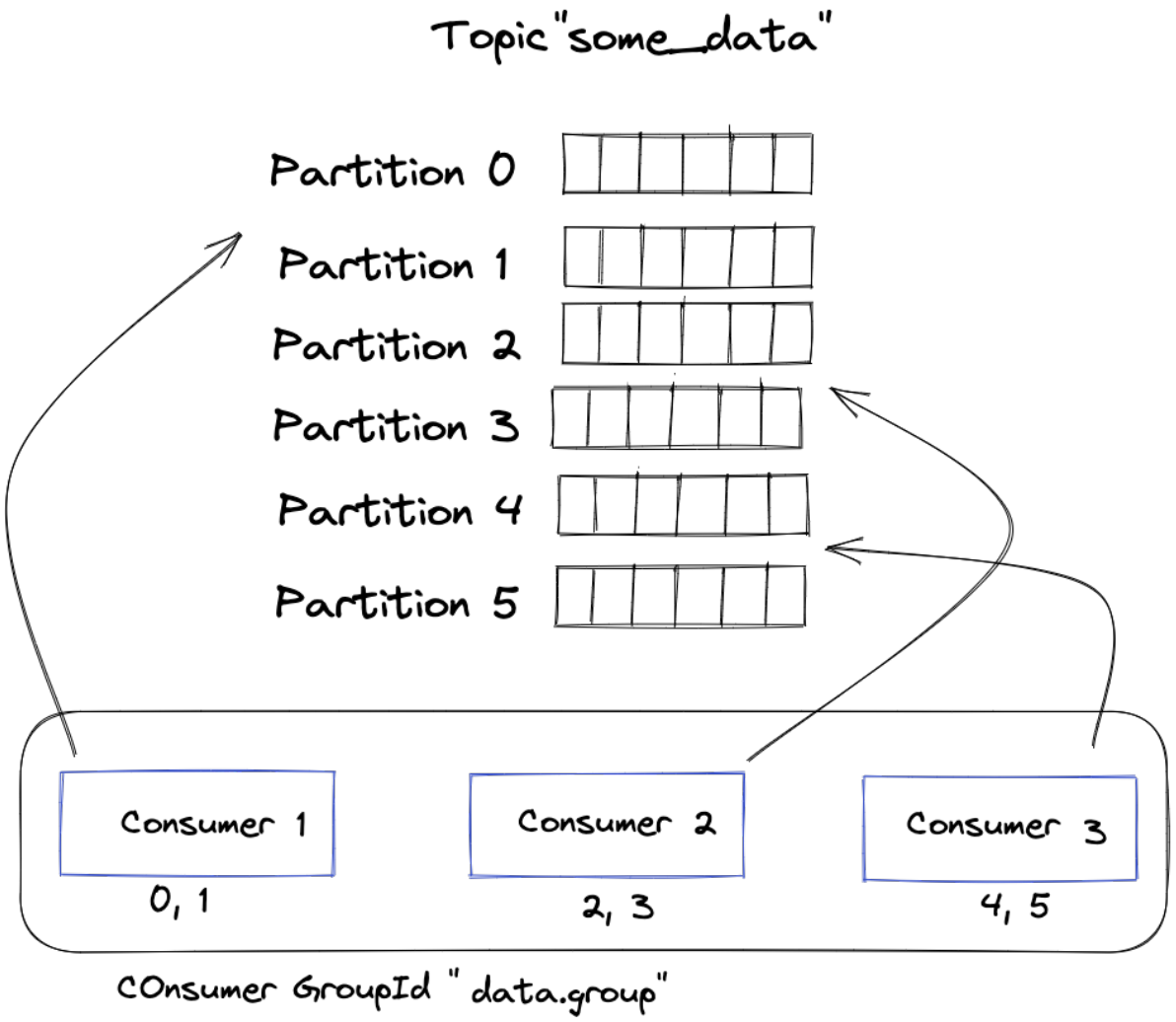


Figure 4.9 Consumer groups allow for assigning topic-partitions across multiple consumers

So going off the image above, there is one topic with six partitions. There are three consumers in the group, so each consumer has an assignment of two partitions. Kafka guarantees that only a single consumer maintains an assignment for a given topic-partition. To have more than one consumer assigned to a single topic-partition would lead to undefined behavior.

Life with distributed systems means that failures aren't to be avoided, but embraced with sound practices to deal with them as they occur. So what happens with our scenario here if one of the consumers in the group fails whether from an exception or missing a required timeout like we described above with the `max.poll.interval.ms` timeout? The answer is the Kafka rebalance protocol, depicted below:

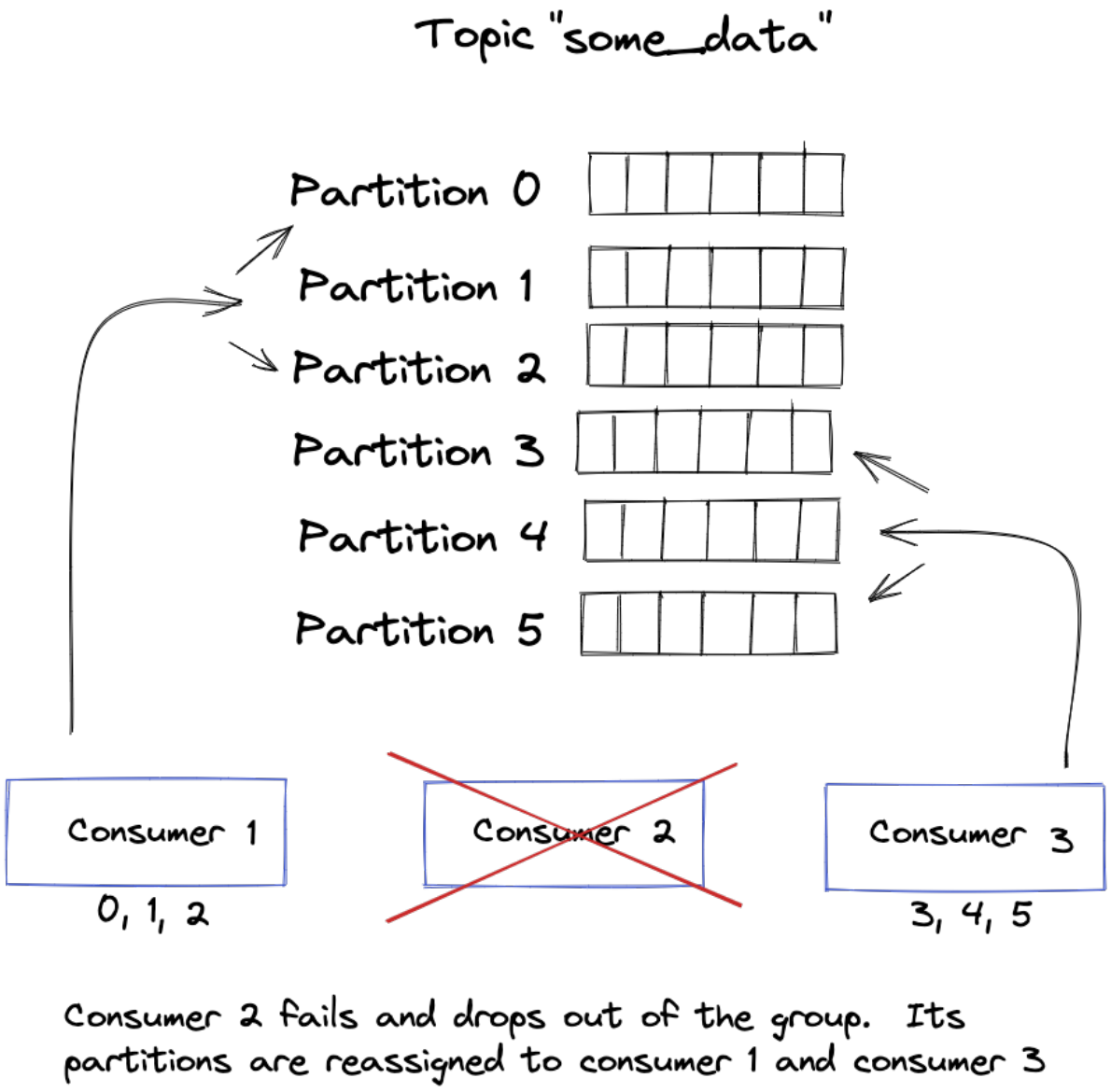


Figure 4.10 The Kafka rebalance protocol re-assigns topic-partitions from failed consumers to still alive ones

What we see here is that consumer-two fails and can longer function. So rebalancing takes the topic-partitions owned by consumer-two and reassigns one topic-partition each to other active consumers in the group. Should consumer-two become active again (or another consumer join the group), then another rebalance occurs and reassigns topic-partitions from the active members and each group member will be responsible for two topic-partitions each again.

NOTE The number of active consumers that you can have is bounded by the number of partitions. From our example here, you can start up to six consumers in the group, but any more beyond six will be idle. Also note that different groups don't affect each other, each one is treated independently.

So far, I've discussed how not making a `poll()` call within the specified timeout will cause a consumer to drop out of the group triggering a rebalance and assigning its topic-partition assignments to other consumers in the group. But if you recall the default setting for `max.poll.interval.ms` is five minutes. Does this mean it takes up to five minutes for potentially dead consumer to get removed from the group and its topic-partitions reassigned? The answer is no and let's look at the poll interval illustration again but we'll update it to reflect session timeouts:

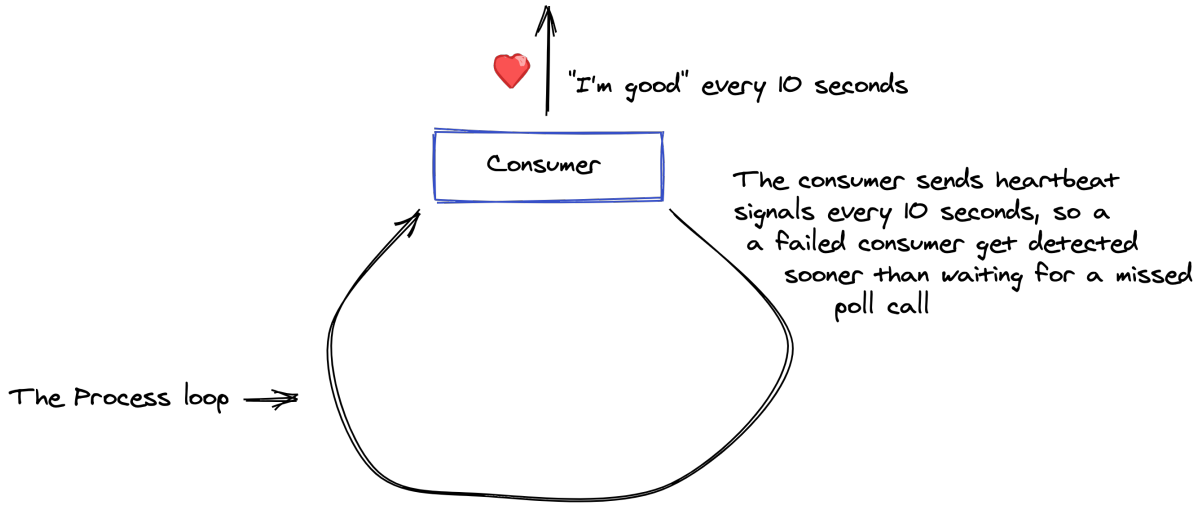


Figure 4.11 In addition to needing to call poll within the timeout, a consumer must send a heartbeat every ten seconds

There is another configuration timeout the `session.timeout.ms` which is set at ten seconds for default value. Each `KafkaConsumer` runs a separate thread for sending heartbeats indicating its still alive. Should a consumer fail to send a heartbeat within ten seconds, it's marked as dead and removed from the group, triggering a rebalance. This two level approach for ensuring consumer liveliness is essential to make sure all consumers are functioning and allows for reassigning their topic-partition assignments to other members of the group to ensure continued processing should one of them fail.

To give you a clear picture of how group membership works, let's discuss a the new terms group coordinator, rebalancing, and the group leader I just spoke about. Let's start with a visual representation of how these parts are tied together:

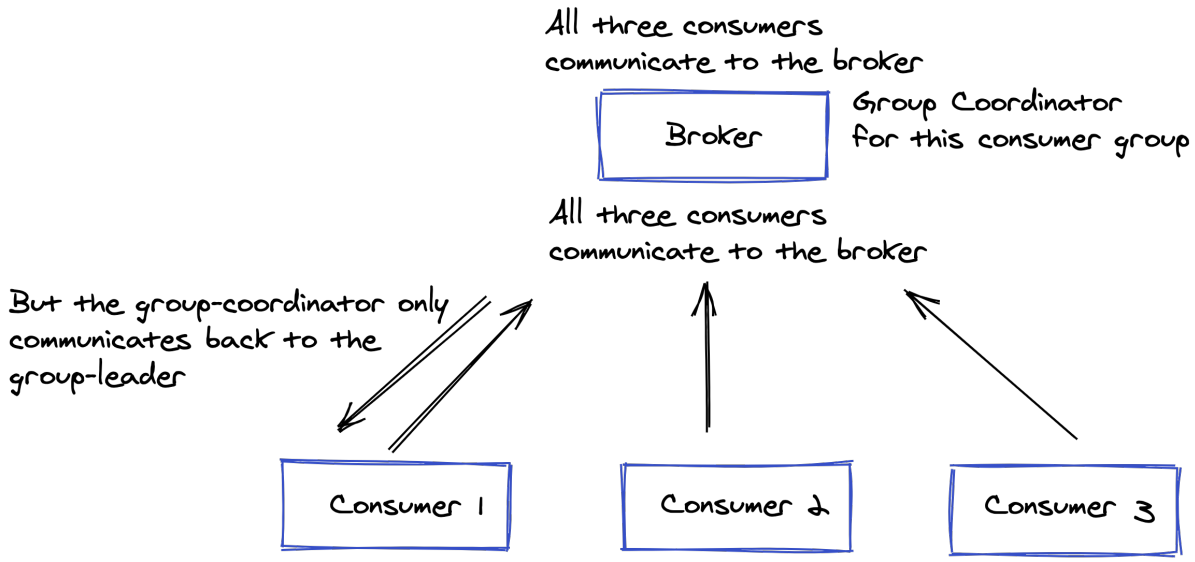


Figure 4.12 Group coordinator is a broker assigned to track a subset of consumer groups and the group leader is a consumer that communicates with the group coordinator

The group coordinator is a broker that handles membership for subset of all available consumer groups. Not one single broker will act as the group coordinator, the responsibility for that is spread around the different brokers. The group coordinator monitors the membership of a consumer group via requests to join a group or when a member is considered dead when it fails to communicate (either a poll or heartbeat) within the given timeouts.

When the group coordinator detects a membership change it triggers a rebalance for the existing members.

A rebalance is the process of having all members of the group rejoin so that group resources (topic-partitions) can be evenly (as possible) distributed to the other members. When a new member joins, then some topic partitions are removed from some or all members of the existing group and are assigned to the new member. When an existing member leaves, the opposite process occurs, its topic-partitions are reassigned to the other active members.

The rebalance process is fairly straight forward, but it comes at a cost of time lost processing waiting for the rebalance process to complete, known as a "stop-the-world" or a eager rebalance. But with the release of Kafka 2.4, there’s new rebalance protocol you can use called cooperative rebalancing.

Let’s take a quick look at both of these protocols, first with the eager rebalancing

EAGER REBALANCING

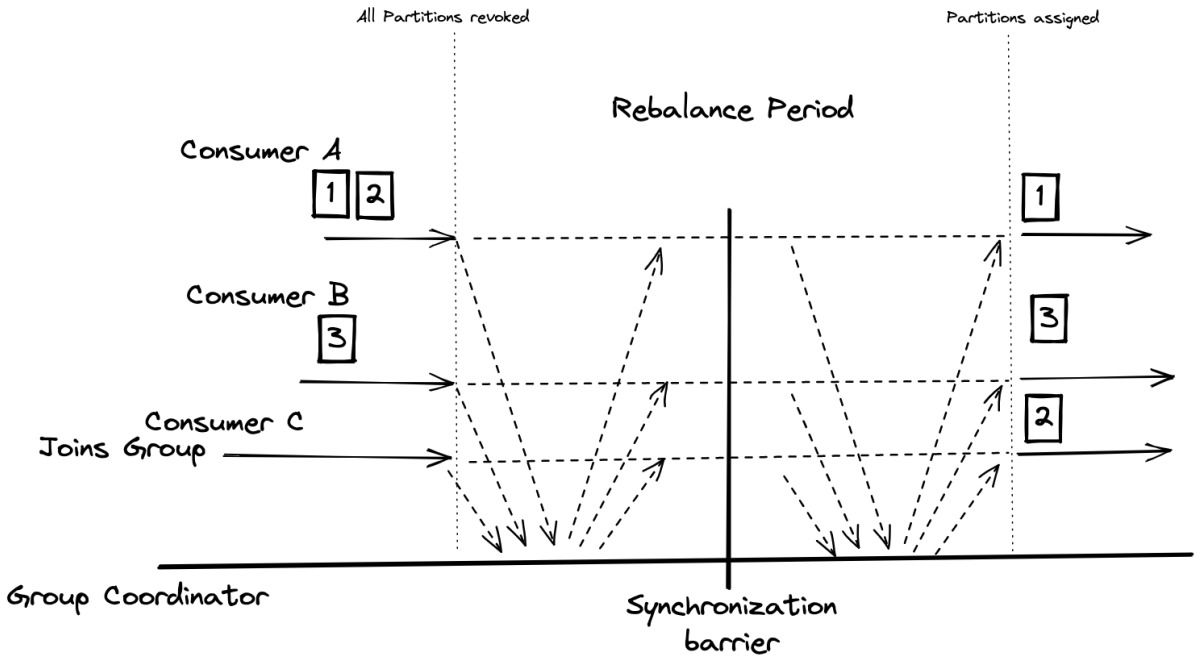


Figure 4.13 Rebalancing with the eager or "stop-the-world" approach processing on all partitions stops until reassigned but most of the partitions end up on with the original consumer

When the group coordinator detects a change in membership it triggers a rebalance. This is true of both rebalance protocols we're going to discuss.

Once the rebalance process initiates, each member of the group first gives up ownership of all its assigned topic-partitions. Then they send a `JoinGroup` request to the controller. Part of the request includes the topic-partitions that consumer is interested in, the ones they just relinquished control of. As a consequence of the consumers giving up their topic partitions is that processing now stops.

The controller collects all of the topic-partition information from the group and sends out the `JoinGroup` response, but the group leader receives all of included topic-partition information.

> **NOTE** **Remember from chapter two in our discussion of the broker all actions are executed in a request/response process.**

The group leader takes this information and creates topic-partition assignments for all members of the group. Then the group leader sends assignment information to the coordinator in a `SyncGroup` request. Note that the other members of the group also send `SyncGroup` requests, but don't include any assignment information. After the group controller receives the assignment information from the leader, all members of the group get their new assignment via the `SyncGroup` response.

Now with their topic-partition assignments, all members of the group begin processing again. Take note again that no processing occurred from the time group members sent the `JoinGroup` request until the `SyncGroup` response arrived with their assignments. This gap in processing is known as a synchronization barrier, and is required as it's important to ensure that each topic-partition only has one consumer owner. If a topic-partition had multiple owners, undefined behavior would result.

> **NOTE** **During this entire process, consumer clients don't communicate with each other. All the consumer group members communicate only with the group coordinator. Additionally only one member of the group, the leader, sets the topic-partition assignments and sends it to the coordinator.**

While the eager rebalance protocol gets the job done of redistributing resources and ensuring only one consumer owns a given topic-partition, it comes at a cost of downtime as each consumer is idle during the period from the initial `JoinGroup` request and the `SyncGroup` response. For smaller applications this cost might be negligible, but for applications with several consumers and a large number of topic-partitions, the cost of down time increases. Fortunately there's another rebalancing approach that aims to remedy this situation.

INCREMENTAL COOPERATIVE REBALANCING

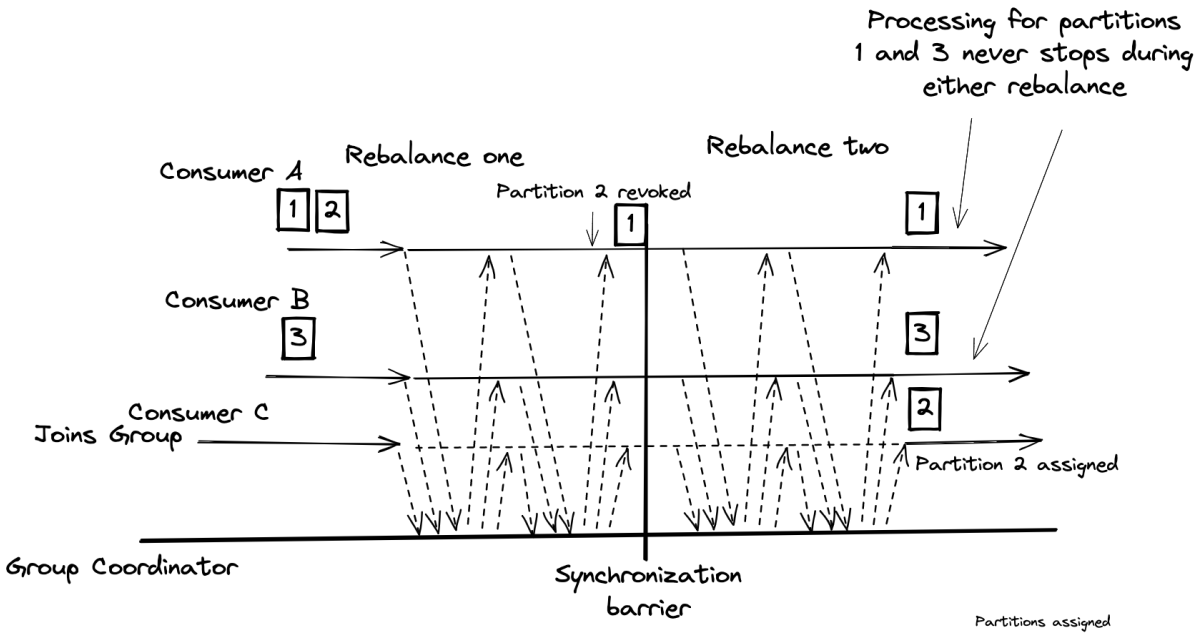


Figure 4.14 Rebalancing with cooperative approach processing continues and only stops for partitions that will be reassigned

Introduced in the 2.4 Kafka release the incremental cooperative rebalance protocol takes the approach that relabances don't need to be so expensive. The incremental cooperative rebalancing

approach takes a different view of rebalancing that we can summarize below:

1. Consumers don't automatically give up ownership of all their topic-partitions
2. The group leader identifies specific topic-partitions requiring new ownership
3. Processing continues for topic-partitions that ***are not changing ownership***

The third bullet point here is the big win (in my opinion) with the cooperative rebalancing approach. Instead of the "stop the world" approach, only those topic-partitions which are moving will experience a pause in processing. In other words, the synchronization barrier is much smaller.

I'm skipping over some of some details, so let's walk through the process of the incremental cooperative rebalancing protocol.

Just like before when the group controller detects a change in group membership, it triggers a rebalance. Each member of the group encodes their current topic-partition subscriptions in a `JoinGroup` request, but each member retains ownership for the time being.

The group coordinator assembles all the subscription information and in the `JoinGroup` response the group leader looks at the assignments and determines which topic-partitions, if any, need to migrate to new ownership. The leader removes any topic-partitions requiring new ownership from the assignments and sends the updated subscriptions to the coordinator via a `SyncGroup` request. Again, each member of the group sends a `SyncGroup` request, but only the leaders` request contains the subscription information.

> **NOTE** **All members of the group receive a `JoinGroup` response, but only the response to the group leader contains the assignment information. Likewise, each member of the group issues a `SyncGroup` group request, but only the leader encodes a new assignment. In the `SyncGroup` response, all members receive their respective, possible updated assignment.**

The members of group take the `SyncGroup` response and potentially calculate a new assignment. Either revoking topic-partitions that are not included or adding ones in the new assignment but not the previous one. Topic-partitions that are included in both the old and new assignment require no action.

Members then trigger a second rebalance, but only topic-partitions changing ownership are included. This second rebalance acts as the synchronization barrier as in the eager approach, but since it only includes topic partitions receiving new owners, it is much smaller. Additionally, topic-partitions that are not moving, continue to process records!

After this discussion of the different rebalance approaches, we should cover some broader information about partition assignment strategies available and how you apply them.

APPLYING PARTITION ASSIGNMENT STRATEGIES

We've already discussed that a broker serves as a group coordinator for some subset of consumer groups. Since two different consumer groups could have different ideas of how to distribute resources (topic-partitions), the responsibility for which approach to use is entirely on the client side.

To choose the partition strategy you want your the `KafkaConsumer` instances in a group to use, you set the the `partition.assignment.strategy` by providing a list of supported partition assignment strategies. All of the available petitioners implement the `ConsumerPartitionAssignor` interface. Here's a list of the available assignors with a brief description of the functionality each one provides.

NOTE **For Kafka Connect and Kafka Streams, which are abstractions built on top of Kafka producers and consumers, use cooperative rebalance protocols and I'd generally recommend to stay with the default settings. This discussion about partitioners is to inform you of what's available for applications directly using a `KafkaConsumer`.**

- RangeAssignor - This is the default setting. The `RangeAssignor` uses an algorithm of sorting the partitions in numerical order and assigning them to consumers by dividing the number of available partitions by number of available consumers. This strategy assigns partition to consumers in lexicographical order.
- RoundRobinAssignor - The `RoundRobinAssignor` takes all available partitions and assigns a partition to each available member of the group in a round-robin manner.
- StickyAssignor - The `StickyAssignor` attempts to assign partitions in a balanced manner as possible. Additionally, the `StickyAssignor` attempts to always preserve existing assignments during a rebalance as much as possible. The `StickyAssignor` follows the eager rebalancing protocol.
- CooperativeStickyAssignor - The `CooperativeStickyAssignor` follows the same assignment algorithm as the `StickyAssignor`. The difference lies in fact that the `CooperativeStickyAssignor` uses the cooperative rebalance protocol.

While it's difficult to provide concrete advice as each use case requires careful analysis of its unique needs, in general for newer applications one should favor using the `CooperativeStickyAssignor` for the reasons outlined in the section on incremental cooperative rebalancing.

TIP **If you are upgrading from a version of Kafka 2.3 or earlier you need to follow a specific upgrade path found in the 2.4 upgrade documentation (kafka.apache.org/documentation/{hash}upgrade_240_notable) to safely use the cooperative rebalance protocol.**

We've concluded our coverage of consumer groups and how the rebalance protocol works. Next

we'll cover a different configuration - static membership, that when a consumer leaves the group, there's no initial rebalance.

4.2.3 Static membership

In the previous section you learned that when a consumer instance shuts down it sends a leave group request to the group controller. Or if it's considered unresponsive by the controller, it gets removed from the consumer group. Either way the end result is the same, the controller triggers a rebalance to re-assign resources (topic-partitions) to the remaining members of the group.

While this protocol is exactly what you want to keep your applications robust, there are some situations where you'd prefer slightly different behavior. For example, let's say you have several consumer applications deployed. Any time you need to update the applications, you might do what's called a rolling upgrade or restart.

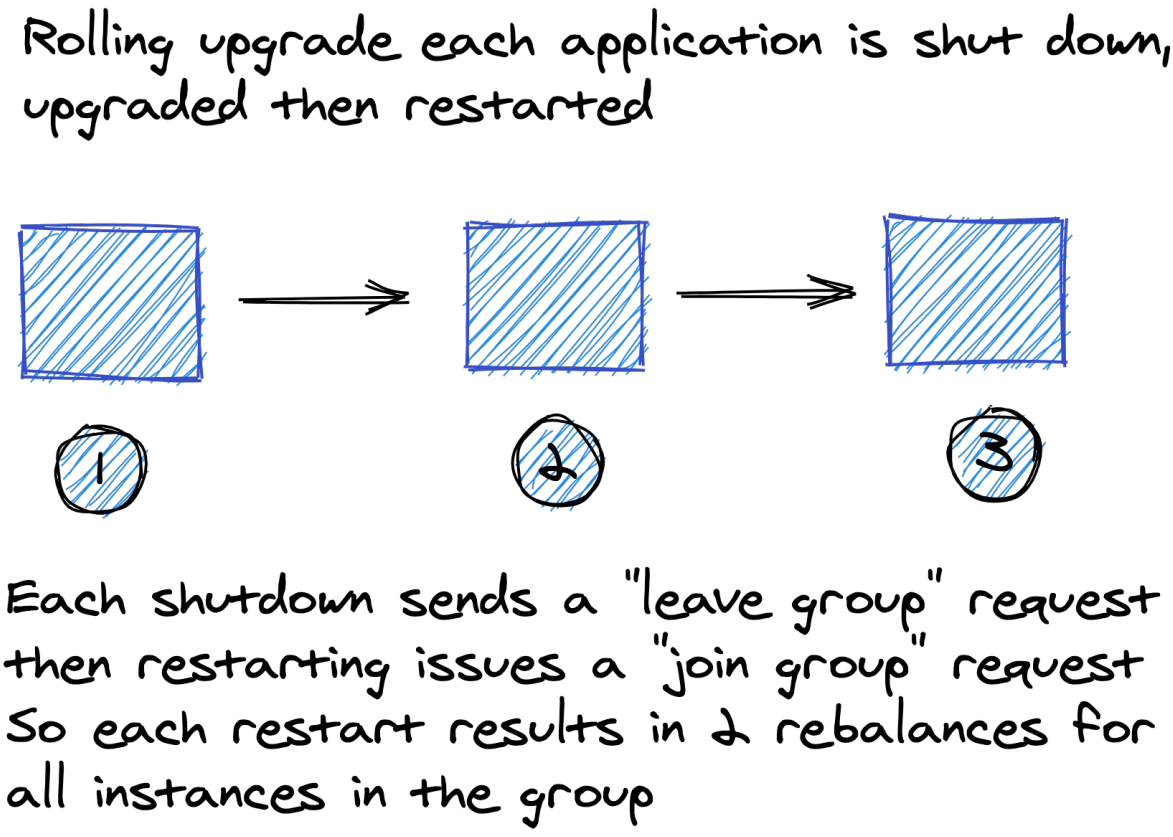


Figure 4.15 Rolling upgrades trigger multiple relabances

You'll stop instance 1, upgrade and restart it, then move on to instance number 2 and so it continues until you've updated every application. By doing a rolling upgrade, you don't lose nearly as much processing time if you shut down every application at the same time. But what happens is this "rolling upgrade", triggers two rebalances for every instance, one when the

application shuts down and another when it starts back up. Or consider a cloud environment where an application node can drop off at any moment only to have it back up an running once its failure is detected.

Even with the improvements brought by cooperative rebalancing, it would be advantageous in these situations to not have a rebalance triggered automatically for these transient actions. The concept of "static membership" was introduced in the 2.3 version of Apache Kafka. We'll use the following illustration to help with our discussion of how static membership works

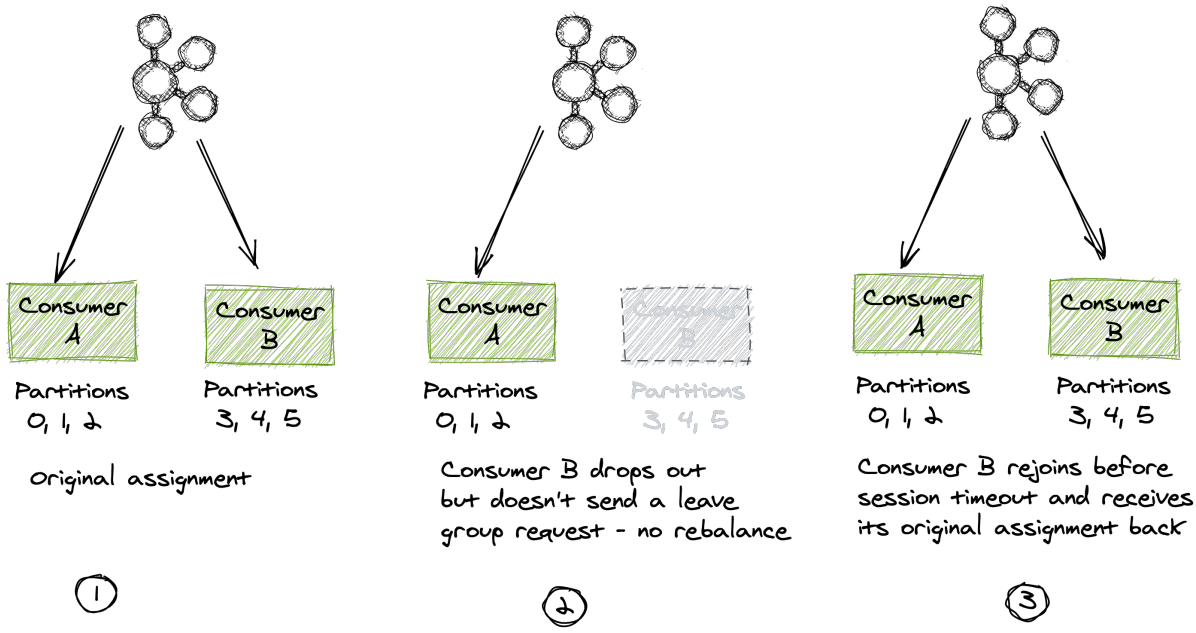


Figure 4.16 Static members don't issue leave group requests when dropping out of a group and a static id allows the controller to remember them

At a high-level with static membership you set a unique id in the consumer configuration, `group.instance.id`. The consumer provides this id to the controller when it joins a group and the controller stores this unique group-id. When a consumer leaves the group, it does not send a leave group request. When it rejoins it presents this unique membership id to the controller. The controller looks it up and can then give back the original assignment to this consumer with no rebalancing involved at all! The trade-off for using static membership is that you'll need to increase the `session.timeout.ms` configuration to a value higher than the default of 10 seconds, as once a session timeout occurs, then the controller kicks the consumer out of the group and triggers a rebalance.

The value you choose should be long enough to account for transient unavailability and not triggering a rebalance but not so long that a true failure gets handled correctly with a rebalance. So if you can sustain ten minutes of partial unavailability then maybe set the session timeout to eight minutes. While static membership can be a good option for those running KafkaConsumer applications in a cloud environment, it's important to take into account the performance

implications before opting to use it. Note that to take advantage of static membership, you must have Kafka brokers and clients on version 2.3.0 or higher.

Next, we'll cover a subject that is very important when using a `KafkaConsumer`, commit the offsets of messages.

4.2.4 Committing offsets

In chapter two, we talked about how the broker assigns a number to incoming records called an offset. The broker increments the offset by one for each incoming record. Offsets are important because they serve to identify the logical position of a record in a topic. A `KafkaConsumer` uses offsets to know where it last consumed a record. For example if a consumer retrieves a batch of records with offsets from 10 to 20, the starting offset of the next batch of records the consumer wants to read starts at offset 21.

To make sure the consumer continues to make progress across restarts for failures, it needs to periodically commit the offset of the last record it has successfully processed. Kafka consumers provide a mechanism for automatic offset commits. You enable automatic offset commits by setting the `enable.auto.commit` configuration to `true`. By default this configuration is turned on, but I've listed it here so we can talk about how automatic commits work. Also, we'll want to discuss the concept of a consumers' position vs. its latest committed offset. There is also a related configuration, `auto.commit.interval.ms` that specifies how much time needs to elapse before the consumer should commit offsets and is based on the system time of the consumer.

But first, lets show how automatic commits work.

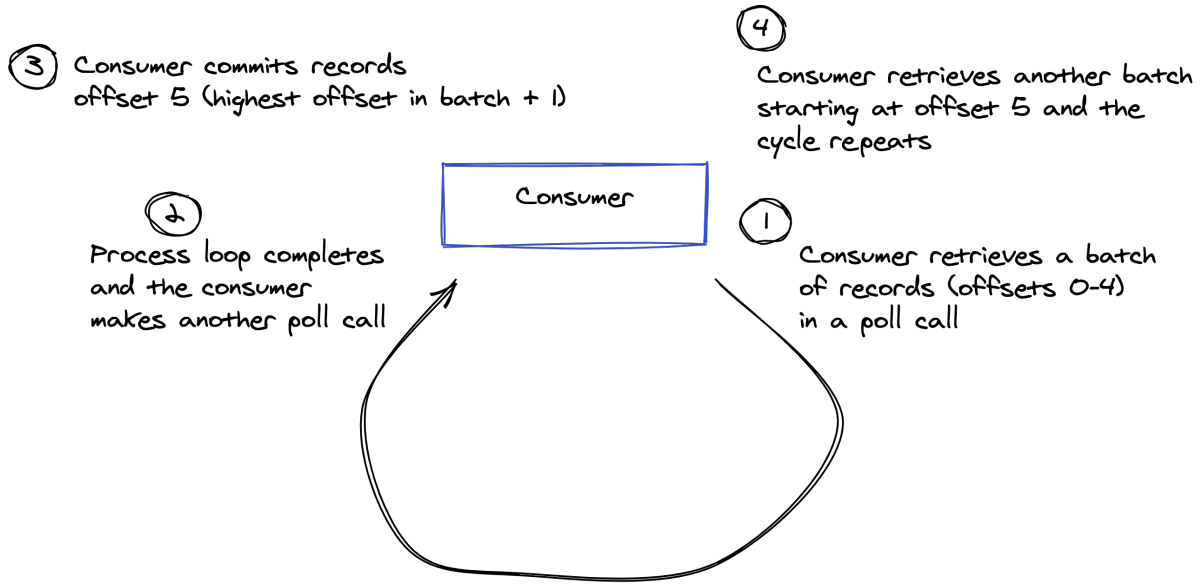


Figure 4.17 With automatic commits enabled when returning to the top of the poll loop the highest offset +1 of the previous batch could be committed if the auto commit interval has passed

Following from the graphic above, the consumer retrieves a batch of records from the `poll(Duration)` call. Next the code takes the `ConsumerRecords` and iterates over them and does some processing of the records. After that the code returns to top of the `poll` loop and attempts to retrieve more records. But before retrieving records, if the consumer has auto-commit enabled and the amount of time elapsed since the last auto-commit check is greater than the `auto.commit.interval.ms` interval, the consumer commits the offsets of the records from the previous batch. By committing the offsets, we are marking these records as consumed, and under normal conditions the consumer won't process these records again. I'll describe what I mean about this statement a little bit later.

What does it mean to commit offsets? Kafka maintains an internal topic named `_offsets` where it stores the committed offsets for consumers. When we say a consumer commits it's not storing the offsets for each record it consumes, it's the highest offset, per partition, plus one that the consumer has consumed so far that's committed.

For example, in the illustration above, let's say the records returned in the batch contained offsets from 0-4. So when the consumer commits, it will be offset 5.

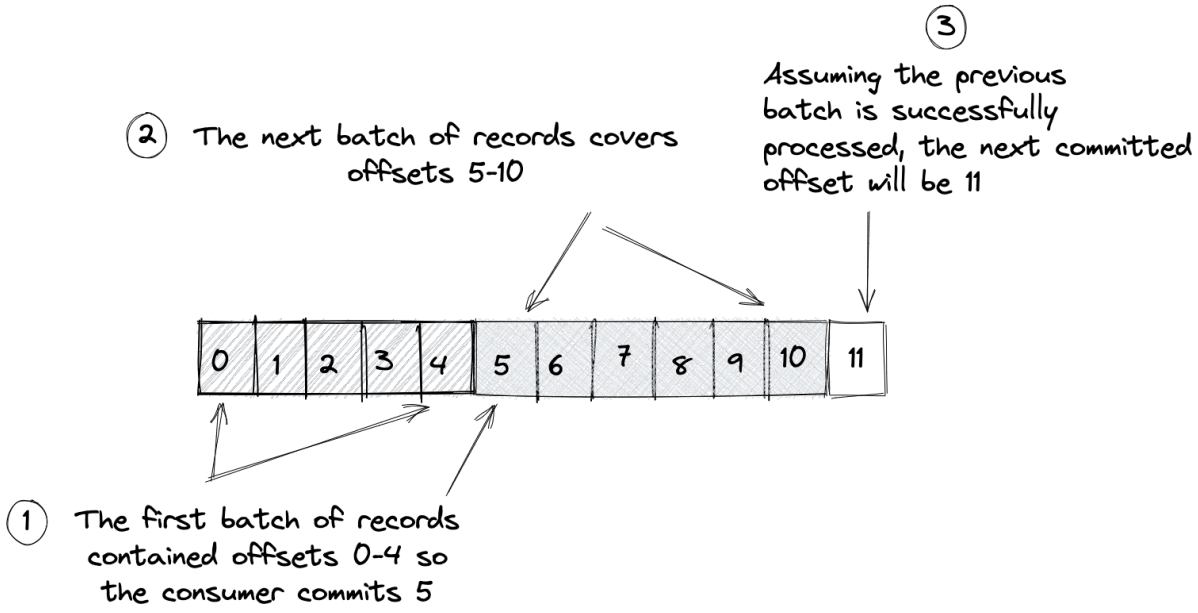


Figure 4.18 A consumers committed position is the largest offset it has consumed so far plus one

So the committed position is offset that has been sucessfully stored, and it indicates the starting record for the next batch it will retrieve. In this illustration it's 5. Should the consumer in this example fail or you restarted the application the consumer would consume records starting at offset 5 again since it wasn't able to commit prior to the failure or restart.

Consuming from the last committed offset means that you are guaranteed to not miss processing a record due to errors or application restarts. But it also means that you may process a record

more than once.

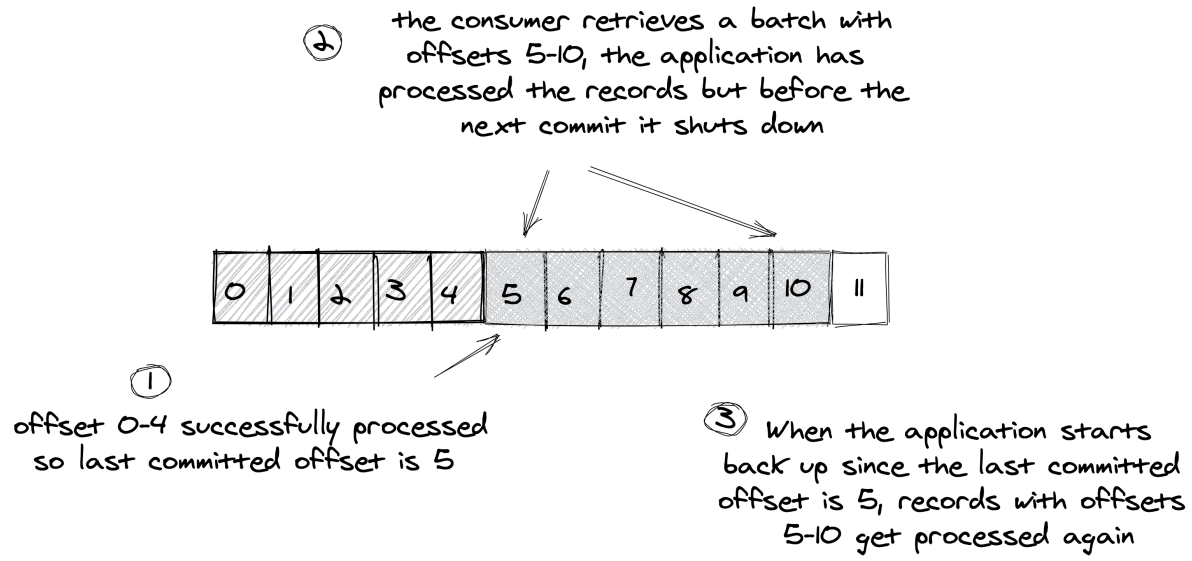


Figure 4.19 Restarting a consumer after processing without a commit means reprocessing some records

If you processed some of the records with offsets larger than the latest one committed, but your consumer failed to commit for whatever reason, this means when you resume processing, you start with records from the committed offset, so you'll reprocess some of the records. This potential for reprocessing is known as at-least-once. We covered at-least-once delivery in the delivery semantics earlier in the chapter.

To avoid reprocessing records you could manually commit offsets immediately after retrieving a batch of records, giving you at-most-once delivery. But you run the risk of losing some records if your consumer should encounter an error after committing and before it's able to process the records. Another option (probably the best), to avoid reprocessing is to use the Kafka transactional API which guarantees exactly-once delivery.

COMMITTING CONSIDERATIONS

When enabling auto-commit with a Kafka consumer, you need to make sure you've fully processed all the retrieved records before the code returns to the top of the poll loop. In practice, this should present no issue assuming you are working with your records synchronously meaning your code waits for the completion of processing of each record. However, if you were to hand off records to another thread for asynchronous processing or set the records aside for later processing, you also run the risk of potentially not processing all consumed records before you commit. Let me explain how this could happen.

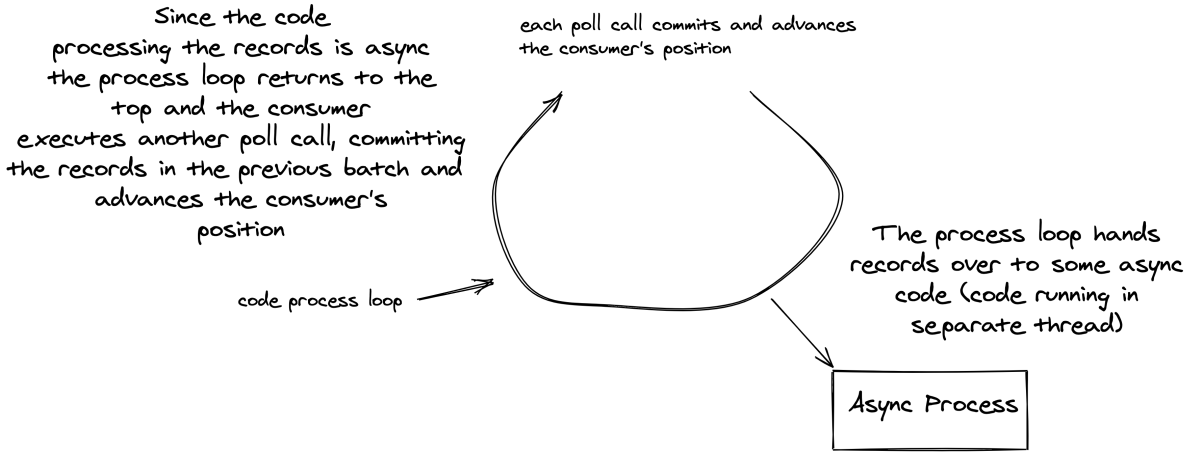


Figure 4.20 Asynchronous processing with auto committing can lead to potentially lost records

When you hand the records off to an asynchronous process, the code in your poll loop won't wait for the successful processing of each record. When your application calls the `poll()` method again, it commits the current position i.e the highest offset + 1 from the for each topic-partition consumed in the previous batch. But your async process may not completed working with all the records up to the highest offset at the time of the commit. If your consumer application experienced a failure or a shutdown for any reason, when it resumes processing, it will start from the last committed offset, which skips over the un-processed records in the last run of your application.

To avoid prematurely-maturely committing records before you consider them fully processed, then you'll want to disable auto-commits by setting `enable.auto.commit` to `false`.

But why would you need to use asynchronous processing requiring manually committing? Let's say when you consume records, you do some processing that takes long time (up to 1 second) to process each record. The topic you consume from has a high volume of traffic, so you don't want to fall behind. So you decide that as soon as you consume a batch of records, you'll hand them off to an async process so the consumer can immediately return to the poll call to retrieve the next batch.

Using an approach like this is called pipeling. But you'll need make sure you're only committing the offsets for records that have been successfully processed, which means turning off auto-committing and coming up with a way to commit only records that your application considers fully processed. The following example code shows one example approach you could take. Note that I'm only showing the key details here and you should consult the source code to see the entire example

Listing 4.4 Consumer code found in bbejeck.chapter_4.pipelining.PipliningConsumerClient

```
// Details left out for clarity
ConsumerRecords<String, ProductTransaction> consumerRecords = consumer.poll(
  Duration.ofSeconds(5));
if (!consumerRecords.isEmpty()) {
    recordProcessor.processRecords(consumerRecords);       ❶
    Map<TopicPartition, OffsetAndMetadata> offsetsAndMetadata =
      recordProcessor.getOffsets();   ❷
    if (offsetsAndMetadata != null) {
        consumer.commitSync(offsetsAndMetadata);   ❸
    }
```

❶ After you've retrieved a batch of records you hand off the batch of records to the async processor.

❷ Checking for offsets of completed records

❸ If the `Map` is not empty, you commit the offsets of the records processed so far.

The key point with this consumer code is that the `RecordProcessor.processRecords()` call returns immediately, so the next call to `RecordProcessor.getOffsets()` returns offsets from a previous batch of records that are fully processed. What I want to emphasize here is how the code hands over new records for processing then collects the offsets of records already fully processed for committing. Let's take a look at the processor code to see this is done:

Listing 4.5 Asynchronous processor code found in bbejeck.chapter_4.piplining.ConcurrentRecordProcessor

```
Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();   ❶
consumerRecords.partitions().forEach(topicPartition -> {    ❷
   List<ConsumerRecord<String,ProductTransaction>> topicPartitionRecords =
  consumerRecords.records(topicPartition);   ❸
   topicPartitionRecords.forEach(this::doProcessRecord);   ❹
   long lastOffset = topicPartitionRecords.get(
  topicPartitionRecords.size() - 1).offset();   ❺
   offsets.put(topicPartition, new OffsetAndMetadata(lastOffset + 1));   ❻
  });
 ....
 offsetQueue.offer(offsets);   ❼
```

❶ Creating the `Map` for collecting the offset for committing

❷ Iterating over the `TopicPartition` objects

❸ Getting records by `TopicPartition` for processing

❹ Doing the actual work on the consumed records

❺ Getting the last offset for all records of a given `TopicPartition`

❻ Storing the offset to commit for the `TopicPartition`

❼ Putting the entire `Map` of offsets in a queue.

The the takeaway with the code here is that by iterating over records by `TopicPartition` it's

easy to create the map entry for the offsets to commit. Once you've iterated over all the records in the list, you only need to get the last offset. You, the observant read might be asking yourself "Why does the code add1 to the last offset?" When committing offsets it's always the offset of the ***next*** record you'll retrieve. For example if the last offset is 5, you want to commit 6. Since you've already consumed 0-5 you're only interested in consuming records from offset 6 forward.

Then you simply use the `TopicPartition` from the top of the loop as the key and the `OffsetAndMetadata` object as the value. When the consumer retrieves the offsets from the queue, it's safe to commit those offsets as the records have been fully processed. The main point to this example is how you can ensure that you only commit records you consider "complete" if you need to asynchronously process records outside of the `Consumer.poll` loop. It's important to note that this approach only uses a **single thread** and consumer for the record processing which means the code still processes the records in order, so it's safe to commit the offsets as they are handed back.

NOTE	**For a fuller example of threading and the `KafkaConsumer` you should consult www.confluent.io/blog/introducing-confluent-parallel-message-processing-client/ and github.com/confluentinc/parallel-consumer.**

WHEN OFFSETS AREN'T FOUND

I mentioned earlier that Kafka stores offsets in an internal topic named `_offsets`. But what happens when a consumer can't find its offsets? Take the case of starting a new consumer against an existing topic. The new `group.id` will not have any commits associated with it. So the question becomes where to start consuming if offsets aren't found for a given consumer? The `KafkaConsumer` provides a configuration, `offset.reset.policy` which allows you to specify a relative position to start consuming in the case there's no offsets available for a consumer.

There are three settings:

1. earliest - reset the offset to the earliest one
2. latest - reset the offset to the latest one
3. none - throw an exception to the consumer

With a setting of `earliest` the implications are that you'll start processing from the head of the topic, meaning you'll see all the records currently available. Using a setting of `latest` means you'll only start receiving records that arrive at the topic once your consumer is online, skipping all the previous records currently in the topic. The setting of `none` means that an exception gets thrown to the consumer and depending if you are using any try/catch blocks your consumer may shut down.

The choice of which setting to use depends entirely on your use case. It may be that once a

consumer starts you only care about reading the latest data or it may be too costly to process all records.

Whew! That was quite a detour, but well worth the effort to learn some of the critical aspects of working with the `KafkaConsumer`.

So far we've covered how to build streaming applications using a `KafkaProducer` and `KafkaConsumer`. What's been discussed is good for those situations where your needs are met with *at-least-once processing*. But there are situations where you need to guarantee that you process records *exactly once*. For this functionality you'll want to consider using the **exactly once** semantics offered by Kafka.

4.3 Exactly once delivery in Kafka

The 0.11 release of Apache Kafka saw the `KafkaProducer` introduce exactly once message delivery. There are two modes for the `KafkaProducer` to deliver exactly once message semantics; the idempotent producer and the transactional producer.

> **NOTE** **Idempotence means you can perform an operation multiple times and the result won't change beyond what it was after the first application of the operation.**

The idempotent producer guarantees that the producer will deliver messages in-order and only once to a topic-partition. The transactional producer allows you to produce messages to multiple topics atomically, meaning all messages across all topics succeed together or none at all. In the following sections, we'll discuss the idempotent and the transactional producer.

4.3.1 Idempotent producer

To use the idempotent producer you only need to set the configuration `enable.idempotence=true`. There are some other configuration factors that come into play:

1. `max.in.flight.requests.per.connection` must not exceed a value of 5 (the default value is 5)
2. `retries` must be greater than 0 (the default value is Integer.MAX_VALUE)
3. `acks` must be set to `all`. If you do not specify a value for the `acks` configuration the producer will update to use the value of `all`, otherwise the producer throws a `ConfigException`.

Listing 4.6 KafkaProducer configured for idempotence

```
// Several details omitted for clarity
Map<String, Object> producerProps = new HashMap<>();
//Standard configs
producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "somehost:9092");
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, ...);
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, ...);

//Configs related to idempotence
producerProps.put(ProducerConfig.ACKS_CONFIG, "all");  ❶
producerProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);  ❷
producerProps.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);  ❸
producerProps.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);  ❹
```

❶ Setting acks to "all"

❷ Enabling idempotence

❸ Setting retries to Integer.MAX_VALUE - this is the default value shown here for completeness

❹ Setting max in flight requests per connection to 5 - this is the default value shown here for completeness

If you recall from our earlier discussion about the `KafkaProducer` we outlined a situation where due to errors and retries record batches within a partition can end up out of order. To avoid that situation, it was suggested to set the `max.infligh.requests.per.connection` to one. Using the idempotent producer removes the need for you to adjust that configuration. We also discussed in the message delivery semantics to avoid possible record duplication, you would need to set retries to zero risking possible data loss.

Using the idempotent producer avoids both of the records-out-of-order and possible-record-duplication-with-retries. If you requirements are for strict ordering within a partition and no duplicated deliver of records then using the idempotent producer is a must.

NOTE **As of the 3.0 release of Apache Kafka the idempotent producer settings are the default so you'll get the benefits of using it out of the box with no additional configuration needed.**

The idempotent producer uses two concepts to achieve its in-order and only-once semantics-unique producer ids and sequence numbers for messages. The idempotent producer gets initiated with a unique producer id (PID). Since each creation of a idempotent producer results in a new PID, idempotence for a producer is only guaranteed during a single producer session. For a given PID a monotonically sequence id (starting at 0) gets assigned to each batch of messages. There is a sequence number for each partition the producer sends records to.

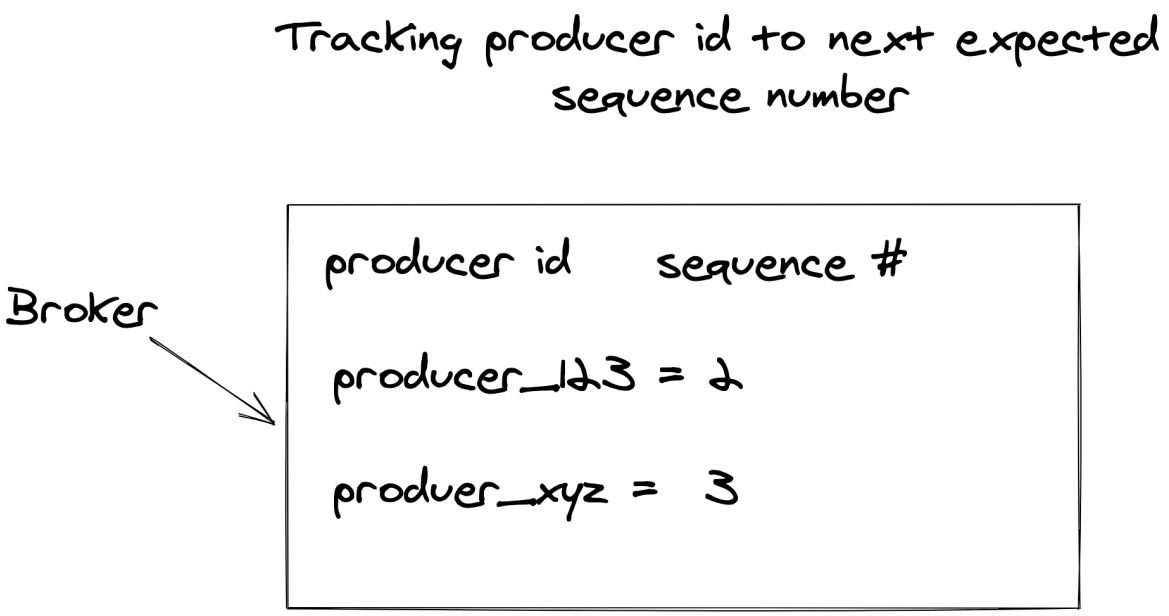


Figure 4.21 The broker keeps track of sequence numbers for each PID and topic-partition it receives

The broker maintains a listing (in-memory) of sequence numbers per topic-partition per PID. If the broker receives a sequence number not ***exactly one greater*** than the sequence number of the last committed record for the given PID and topic-partition, it will reject the produce request.

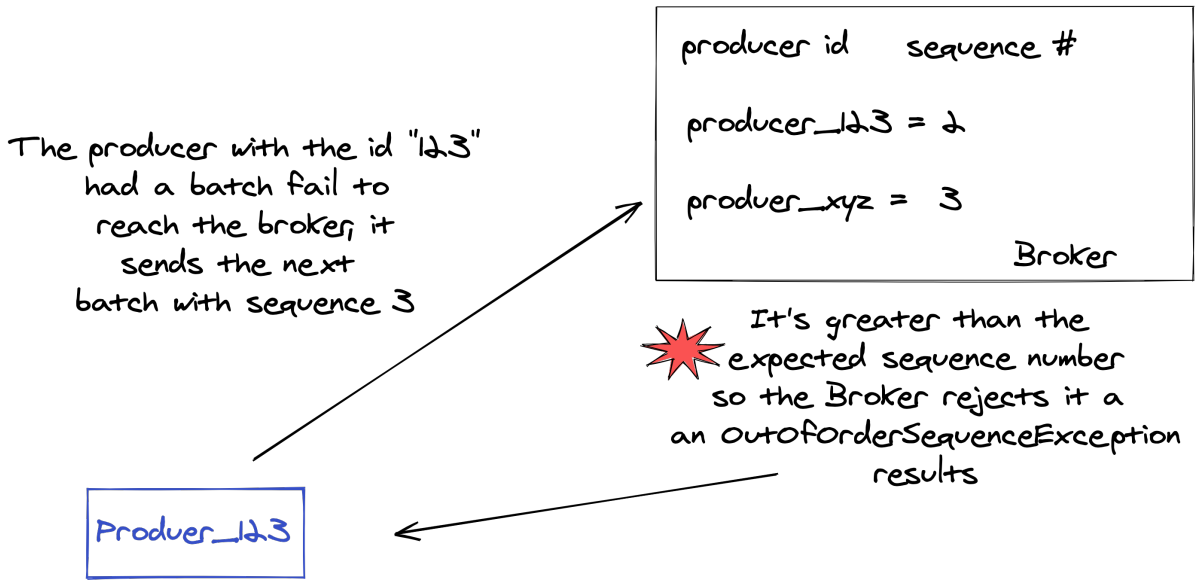


Figure 4.22 The broker rejects produce requests when the message sequence number doesn't match expected one

If the number is less than the expected sequence number, it's a duplication error which the producer ignores. If the number is higher than expected the produce request results in a `OutOfOrderSequenceException`. For the idempotent producer, the

`OutOfOrderSequenceException` is not fatal error and retries will continue. Essentially when there is a retryable error, if there are more than 1 in-flight requests, the broker will reject the subsequent requests and the producer will put them back in order to resend them to the broker.

So if you require strict ordering of records within a partition, then using the idempotent producer is a must. But what do you do if you need to write to multiple topic-partitions atomically? In that case you would opt to use the transactional producer which we'll cover next.

4.3.2 Transactional producer

Using the transactional producer allows you to write to multiple topic-partitions atomically; all of the writes succeed or none of them do. When would you want to use the transactional producer? In any scenario where you can't afford to have duplicate records, like in the financial industry for example.

To use the transaction producer, you need to set the producer configuration `transactional.id` to a unique value for the producer. Kafka brokers use the `transactional.id` to enable transaction recovery across multiple sessions from the same producer instance. Since the id needs to be unique for each producer and applications can have multiple producers, it's a good idea to come up with a strategy where the id for the producers represents the segment of the application its working on.

> **NOTE** **Kafka transaction are a deep subject and could take up an entire chapter on its own. For that reason I'm not going to go into details about the design of transactions. For readers interested in more details here's a link to the original KIP (KIP stands for Kafka Improvement Process) cwiki.apache.org/confluence/display/KAFKA/KIP-98+-+Exactly+Once+Delivery+and+Transactional+Messaging#KIP98ExactlyOnceDeliveryandTransactionalMessaging-Brokerconfigs**

When you enable a producer to use transactions, it is automatically upgraded to an idempotent producer. You can use the idempotent producer without transactions, but you can't do the opposite, using transactions without the idempotent producer. Let's dive into an example. We'll take our previous code and make it transactional

Listing 4.7 KafkaProducer basics for transactions

```
HashMap<String, Object> producerProps = new HashMap<>();

producerProps.put("transactional.id", "set-a-unique-transactional-id");   ❶

Producer<String, String> producer = new KafkaProducer<>(producerProps);
producer.initTransactions();   ❶

try {
    producer.beginTransaction();   ❷
    producer.send(topic, "key", "value");   ❸
    producer.commitTransaction();     ❹
} catch (ProducerFencedException | OutOfOrderSequenceException
  | AuthorizationException e) {    ❺
      producer.close();
} catch (KafkaException e) {       ❻
     producer.abortTransaction();
     // safe to retry at this point   ❼
}
```

❶ Setting a unique id for the producer. Note that it's up to the user to provide this unique id.

❷ Calling initTransactions

❸ The beginning of the transaction, but does not start the clock for transaction timeouts

❹ Sending record(s), in practice probably you'd probably send more than one record but it's shortened here for clarity

❺ Committing the transaction after sending all the records

❻ Handling fatal exceptions, your only choice at this point is to close the producer and re-instantiate the producer instance

❼ Handling a non-fatal exception, you can begin a new transaction with the same producer and try again

After creating a transactional producer instance is to first thing you must here is execute the `initTransactions()` method. The `initTransaction` sends a message to the transaction coordinator (the transaction coordinator is a broker managing transactions for producers) so it can register the `transactional.id` for the producer to manage its transactions. The transaction coordinator is a broker managing transactions for producers.

If the previous transaction has started, but not finished, then this method blocks until its completed. Internally, it also retrieves some metadata including something called an `epoch` which this producer uses in future transactional operations.

Before you start sending records you call `beginTransaction()`, which starts the transaction for the producer. Once the transaction starts,The transaction coordinator will only wait for a period of time defined by the `transaction.timeout.ms` (one minute by default) and it without an update (a commit or abort) it will proactively abort the transaction. But the transaction

coordinator does not start the clock for transaction timeouts until the broker starts sending records. Then after the code completes processing and producing the records, you commit the transaction.

You should notice a subtle difference in error handling between the transactional example from the previous non-transactional one. With the transactional produce you don't have to check of an error occurred either with a `Callback` or checking the returned `Future`. Instead the transactional producer throws them directly for your code to handle.

It's important to note than with any of the exceptions in the first `catch` block are fatal and you must close the producer and to continue working you'll have to create a new instance. But any other exception is considered re-tryable and you just need to abort the current transaction and start over.

Of the fatal exceptions, we've already discussed the `OutOfOrderSequenceException` in the idempotent producer section and the `AuthorizationException` is self explanatory. Be we should quickly discuss the `ProducerFencedException`. Kafka has a strict requirement that there is only one producer instance with a given `transactional.id`. When a new transactional producer starts, it "fences" off any previous producer with the same id must close. However, there is another scenario where you can get a `ProducerFencedException` with out starting a new producer with the same id.

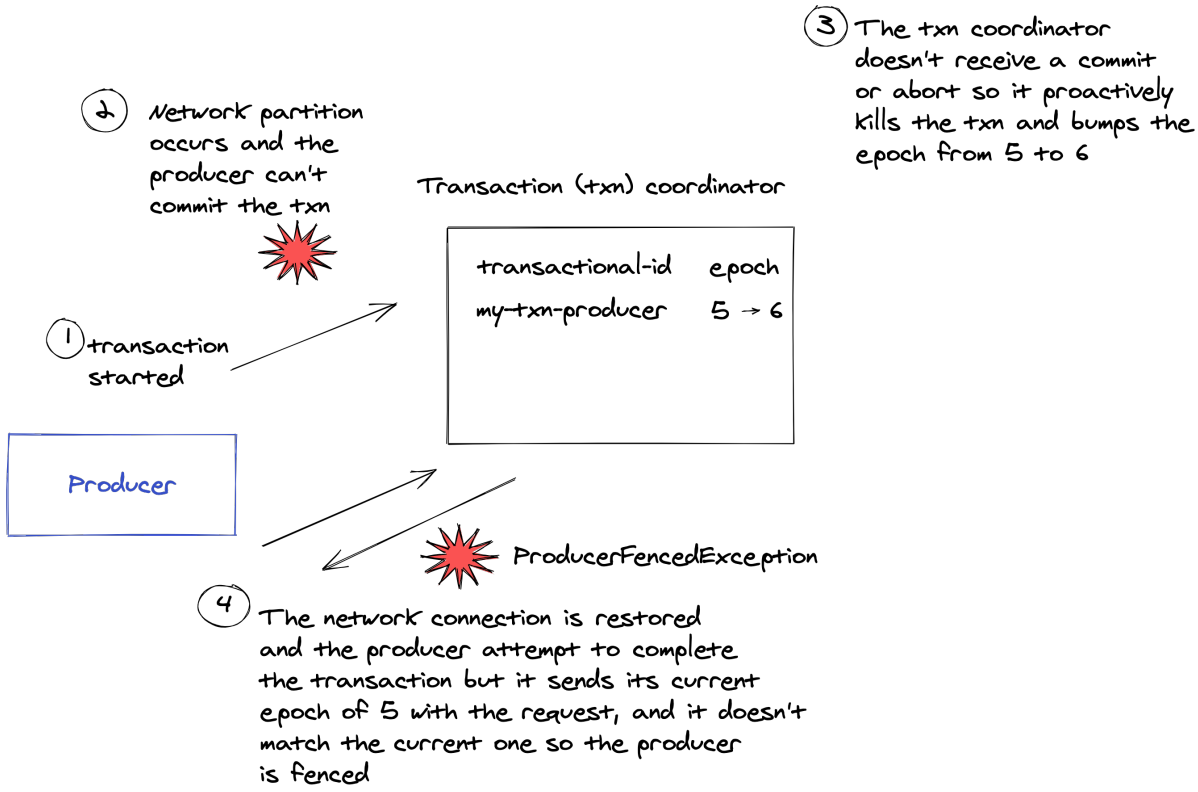


Figure 4.23 Transactions proactively aborted by the Transaction Coordinator cause an increase in the epoch associated with the transaction id

When you execute the `producer.initTransactions()` method, the transaction coordinator increments the producer epoch. The producer epoch is a number the transaction coordinator associates with the transactional id. When the producer makes any transactional request, it provides the epoch along with its transaction id. If the epoch in the request doesn't match the current epoch the transaction coordinator rejects the request and the producer is fenced.

But if the current producer can't communicate with the transaction coordinator for any reason and the timeout expires, as we discussed before, the coordinator proactively aborts the transaction and increments the epoch for that id. When the producer attempts to work again after the break in communication, it finds itself fenced and you must close the producer and restart at that point.

NOTE **There is example code for transactional producers in the form of a test located at src/test/java/bbejeck/chapter_4/TransactionalProducerConsumerTest.java in the source code.**

So far, I've only covered how to produce transactional records, so let's move on consuming them.

4.3.3 Consumers in transactions

Kafka consumers can subscribe to multiple topics at one time, with some of them containing transactional records and others not. But for transactional records, you'll only want to consume ones that have been successfully committed. Fortunately, it's only a matter of a simple configuration. To configure your consumers for transactional records you set `isolation.level` configuration to `read_committed`.

Listing 4.8 KafkaConsumer configuration for transactions

```
// Several details omitted for clarity

HashMap<String, Object> consumerProps = new HashMap<>();
consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, "the-group-id");

consumerProps.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");   ❶

consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, IntegerDeserializer.class);
```

❶ Setting the isolation configuration for the consumer

With this configuration set, your consumer is guaranteed to only retrieve successfully committed transaction records. If you use the `read_uncommitted` setting, then the consumer will retrieve both successful and aborted transactional records. The consumer is guaranteed to retrieve

non-transactional records with either configuration set.

There is difference in highest offset a consumer can retrieve in the `read_committed` mode.

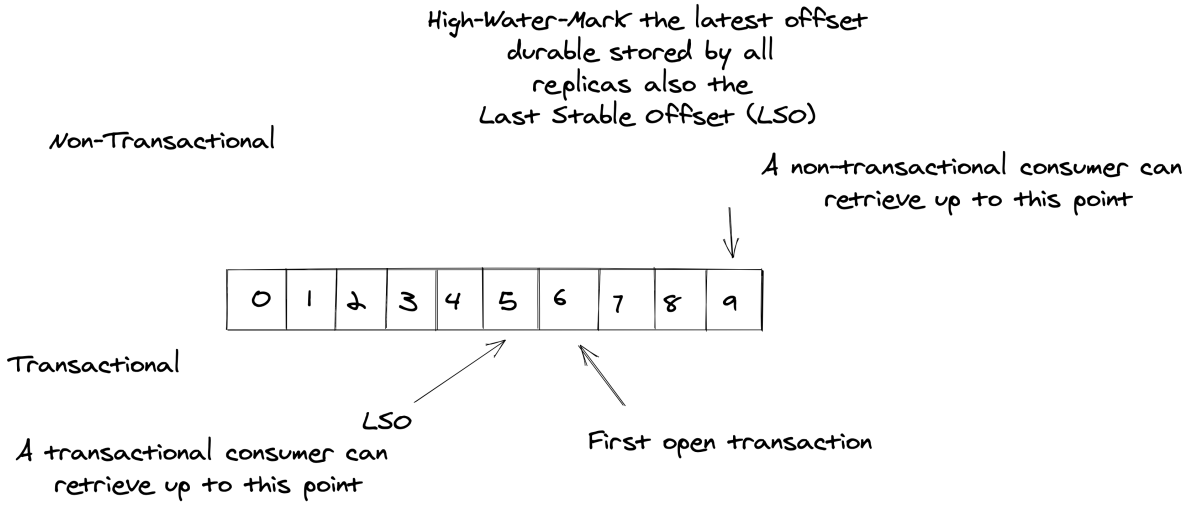


Figure 4.24 High water mark vs. last stable offset in a transactional environment

In Kafka there is a concept of the last stable offset (LSO) which is an offset where all offsets below it have been "decided". There's another concept known as the high water mark. The high water mark is the largest offset successfully written to all replicas. In a non-transactional environment, the LSO is the same as the high water mark as records are considered decided or durable written immediately. But with transactions, an offset can't be considered decided until the transaction is either committed or aborted, so this means the LSO is the offset of the first open transaction minus 1.

This a non-transactional environment, the consumer can retrieve up the the high water mark in a `poll()` call. But with transactions it will only retrieve up to the LSO.

NOTE **The test located src/test/java/bbejeck/chapter_4/TransactionalProducerConsumerTest.java also contains a couple of tests demonstrating consumer behavior with both `read_committed` read `read_uncommitted` configuration.**

So far we've covered how to use a producer and a consumer separately. But there's one more case to consider and that is using a consumer and producer together within a transaction.

4.3.4 Producers and consumers within a transaction

When building applications to work with Kafka it's a fairly common practice to consume records from a topic, perform some type of transformation on the records, then produce those transformed records back to Kafka in a different topic. Records are considered consumed when the consumer commits the offsets. If you recall, committing offsets is simply writing to a topic (_offsets).

So if you are doing a consume - transform - produce cycle, you'd want to make sure that committing offsets is part of the transaction as well. Otherwise you could end up in a situation where you've committed offsets for consumed records, but transaction fails and restarting the application skips the recently processed records as the consumer committed the offsets.

Imagine you have a stock reporting application and you need to provide broker compliance reporting. It's very important that the compliance reports are sent only once so you decide that the best approach is to consume the stock transactions and build the compliance reports within a transaction. This way you are guaranteed that your reports are sent only once.

Listing 4.9 Example of the consume-transform-produce with transactions found in src/test/java/chapter_4/TransactionalConsumeTransformProduceTest.java

```
// Note that details are left out here for clarity

Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();   ❶
producer.beginTransaction();                                        ❷
consumerRecords.partitions().forEach(topicPartition -> {
    consumerRecords.records(topicPartition).forEach(record -> {
        lastOffset.set(record.offset());
        StockTransaction stockTransaction = record.value();
        BrokerSummary brokerSummary = BrokerSummary.newBuilder()    ❸


        producer.send(new ProducerRecord<>(outputTopic, brokerSummary));
    });
    offsets.put(topicPartition,
      new OffsetAndMetadata(lastOffset.get() + 1L));                ❹
});
try {
    producer.sendOffsetsToTransaction(offsets,
      consumer.groupMetadata());                                    ❺
    producer.commitTransaction();                                   ❻
 }
```

❶ Creating the HashMap to hold the offsets to commit

❷ Starting the transaction

❸ Transforming the StockTransaction object into a BrokerSummary

❹ Storing the `TopicPartition` and `OffsetAndMetadata` in the map

❺ Committing the offsets for the consumed records in the transaction

❻ Committing the transaction

From looking at code above, the biggest difference from a non-transactional consume-transform-produce application is that we keep track of the `TopicPartition` objects and the offset of the records. We do this because we need to provide the offsets of the records we just processed to the `KafkaProducer.setOffsetsToTransaction` method. In consume-transform-produce applications with transactions, it's the producer that sends offsets to the consumer group coordinator, ensuring that the offsets are part of the transaction. Should the transaction fail or get aborted, then the offsets are not committed. By having the producer commit the offsets, you don't need any coordination between the producer and consumer in the cases of rolled-back transactions.

So far we've covered using producer and consumer clients for sending and receiving records to and from a Kafka topic. But there's another type of client which uses the `Admin` API and it allows you to perform topic and consumer group related administrative functions programmatically.

4.4 Using the Admin API for programmatic topic management

Kafka provides an administrative client for inspecting topics, broker, ACLs (Access Control Lists) and configuration. While there are several functions you can use the admin client, I'm going to focus on the administrative functions for working with topics and records. The reason I'm doing this is I'm presenting what I feel are the use cases most developers will see in ***development*** of their applications. Most of the time, you'll have a operations team responsible for the management of your Kafka brokers in production. What I'm presenting here are things you can do to facilitate testing a prototyping an application using Kafka.

4.4.1 Working with topics programmatically

To create topics with the admin client is simply a matter of creating the admin client instance and then executing the command to create the topic(s).

Listing 4.10 Creating a topic

```
Map<String, Object> adminProps = new HashMap<>();
adminProps.put("bootstrap.servers", "localhost:9092");

try (Admin adminClient = Admin.create(adminProps)) {   ❶

     final List<NewTopic> topics = new ArrayList<>)();    ❷

    topics.add(new NewTopic("topic-one", 1, 1));   ❸
    topics.add(new NewTopic("topic-two", 1, 1));

    adminClient.createTopics(topics);   ❹
}
```

❶ Creating the Admin instance, note the use of a try with resources block

❷ The list to hold the `NewTopic` objects

❸ Creating the `NewTopic` objects and adding them to the list

❹ Executing the command to create the topics

> **NOTE** **I'm referring to an admin client but the type is the interface `Admin`. There is an abstract class `AdminClient`, but it's use is discouraged over using the `Admin` interface instead. An upcoming release may remove the `AdminClient` class.**

This code can be especially useful when you are prototyping building new applications by ensuring the topics exist before running the code. Let's expand this example some and show how you can list topics and optionally delete one as well.

Listing 4.11 More topic operations

```
Map<String, Object> adminProps = new HashMap<>();
adminProps.put("bootstrap.servers", "localhost:9092");

try (Admin adminClient = Admin.create(adminProps)) {

     Set<String> topicNames = adminClient.listTopics().names.get();   ❶
     System.out.println(topicNames);   ❷
     adminClient.deleteTopics(Collections.singletonList("topic-two"));   ❸
}
```

❶ In this example you're listing all the non-internal topics in the cluster. Note that if you wanted to include the internal topics you would provide a `ListTopicOptions` object where you would call the `ListTopicOptions.listInternal(true)` method.

❷ Printing the current topics found

❸ You delete a topic and list all of the topics again, but you should not see the recently deleted topic in the list.

An additional note for annotation one above, is that the `Admin.listTopics()` returns a `ListTopicResult` object. To get the topic names you use the `ListTopicResult.names()` which returns a `KafkaFuture<Set<String>>` so you use use the `get()` method which blocks until the admin client request completes. Since we're using a broker container running on your local machine, chances are this command completes immediately.

There are several other methods you can execute with the admin client such as deleting records and describing topics. But the way you execute them is very similar, so I wont list them here, but look at the source code (src/test/java/bbejeck/chapter_4/AdminClientTest.java) to see more examples of using the admin client.

TIP	Since we're working on a Kafka broker running in a docker container on your local machine, we can execute all the admin client topic and record operations risk free. But you should exercise caution if you are working in a shared environment to make sure you don't create issues for other developers. Additionally, keep in mind you might not have the opportunity to use the admin client commands in your work environment. And I should stress that you should never attempt to modify topics on the fly in production environments.

That wraps up our coverage of using the admin API. In our next and final section we'll talk about the considerations you take into account for those times when you want to produce multiple event types to a topic.

4.5 Handling multiple event types in a single topic

Let's say you've building an application to track activity on commerce web site. You need to track the click-stream events such as logins and searches and any purchases. Conventional wisdom says that the different events (logins, searches) and purchases could go into separate topics as they are separate events. But there's information you can gain from examining how these related events occurred in sequence.

But you'll need to consume the records from the different topics then try and stitch the records together in proper order. Remember, Kafka guarantees record order within a partition of a topic, but not across partitions of the same topic not to mention partitions of other topics.

Is there another approach you can take? The answer is yes, you can produce those different event types to the same topic. Assuming you providing a consistent key across the event types, you are going receive the different events in-order, on the same topic-partition.

At the end of chapter three (Schema Registry), I covered how you can use multiple event types in a topic, but I deferred on showing an example with producers and consumers. Now we'll go through an example now on how you can produce multiple event types and consume multiple event types safely with Schema Registry.

In chapter three, specifically the `Schema references and multiple events per topic` section I discussed how you can use Schema Registry to support multiple event types in a single topic. I didn't go through an example using a producer or consumer at that point, as I think it fits better in this chapter. So that's what we're going to cover now.

NOTE	Since chapter three covered Schema Registry, I'm not going to do any review in this section. I may mention some terms introduced in chapter three, so you may need to refer back to refresh your memory if needed.

Let's start with the producer side.

4.5.1 Producing multiple event types

We'll use this Protobuf schema in this example:

```
{
syntax = "proto3";

package bbejeck.chapter_4.proto;

import "purchase_event.proto";
import "login_event.proto";
import "search_event.proto";

option java_outer_classname = "EventsProto";

message Events {
      oneof type {
        PurchaseEvent purchase_event = 1;
        LogInEvent login_event = 2;
        SearchEvent search_event = 3;
      }
      string key = 4;
    }
}
```

What happens when you generate the code from the protobuf definition you get a `EventsProto.Events` object that contains a single field `type` that accepts one of the possible three event objects (a Protobuf `oneof` field).

Listing 4.12 Example of creating KafkaProducer using Protobuf with a oneof field

```
// Details left out for clarity
...
producerConfigs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);
producerConfigs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaProtobufSerializer.class);   ❶
...

Producer<String, EventsProto.Events> producer = new KafkaProducer<>(producerConfigs));   ❷
```

❶ Configure the producer to use the Protobuf serializer

❷ Creating the `KafkaProducer` instance

Since Protobuf doesn't allow the `oneof` field as a top level element, the events you produce always have an outer class container. As a result your producer code doesn't look any different for the case when you're sending a single event type. So the generic type for the `KafkaProducer`

and `ProducerRecord` is the class of the Protobuf outer class, `EventsProto.Events` in this case.

In contrast, if you were to use an Avro union for the schema like this example here:

Listing 4.13 Avro schema of a union type

```
[
  "bbejeck.chapter_3.avro.TruckEvent",
  "bbejeck.chapter_3.avro.PlaneEvent",
  "bbejeck.chapter_3.avro.DeliveryEvent"
]
```

Your producer code will change to use a common interface type of all generated Avro classes:

Listing 4.14 KafkaProducer instantiation with Avro union type schema

```
//Some details left out for clarity

producerConfigs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);
producerConfigs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaAvroSerializer.class);   ❶
producerConfigs.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_SCHEMAS,
  false);   ❷
producerConfigs.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VERSION,
  true);   ❸


Producer<String, SpecificRecord> producer = new KafkaProducer<>(
  producerConfigs())   ❹
```

❶ Specifying to use the Kafka Avro serializer

❷ Configuring to producer to not auto register schemas

❸ Setting the use latest schema version to true

❹ Instantiating the producer

Because you don't have an outer class in this case each event in the schema is a concrete class of either a `TruckEvent`, `PlaneEvent`, or a `DeliveryEvent`. To satisfy the generics of the `KafkaProducer` you need to use the `SpecificRecord` interface as every Avro generated class implements it. As we covered in chapter three, it's crucial when using Avro schema references with a union as the top-level entry is to disable auto-registration of schemas (annotation two above) and to enable using the latest schema version (annotation three).

Now let's move to the other side of the equation, consuming multiple event types.

4.5.2 Consuming multiple event types

When consuming from a topic with multiple event types, depending how your approach, you may need to instantiate the `KafkaConsumer` with a generic type of a common base class or interface that all of the records implement.

Let's consider using Protobuf first. Since you will always have an outer wrapper class, that's the class you'll use in the generic type parameter, the value parameter in this example.

Listing 4.15 Configuring the consumer for working with multiple event types in Protobuf

```
//Other configurations details left out for clarity

consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaProtobufDeserializer.class);   ❶
consumerProps.put(
  KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE_TYPE,
    EventsProto.Events.class);   ❷

Consumer<EventsProto.Events> consumer = new KafkaConsumer<>(
  consumerProps);   ❸
```

❶ Using Protobuf deserializer

❷ Setting the Protobuf deserializer to return a specific type

❸ Creating the `KafkaConsumer`

You are setting up your consumer as you've seen before; you're configuring the deserializer to return a specific type, which is the `EventsProto.Events` class in this case. With Protobuf, when you have a `oneof` field, the generated Java code includes methods to help you determine the type of the field with `hasXXX` methods. In our case the `EventsProto.Events` object contains the following 3 methods:

```
hasSearchEvent()
hasPurchaseEvent()
hasLoginEvent()
```

The protobuf generated Java code also contains an enum named `<oneof field name>Case`. In this example, we've named the `oneof` field `type` so it's named `TypeCase` and you access by calling `EventsProto.Events.getTypeCase()`. You can use the enum to determine the underlying object succinctly:

```
//Details left out for clarity
switch (event.getTypeCase()) {
    case LOGIN_EVENT -> {     ❶
        logins.add(event.getLoginEvent());   ❷
    }
    case SEARCH_EVENT -> {
        searches.add(event.getSearchEvent());
    }
    case PURCHASE_EVENT ->  {
        purchases.add(event.getPurchaseEvent());
    }
}
```

❶ Individual case statement base on the enum

❷ Retrieving the event object using `getXXX` methods for each potential type in the `oneof` field

Which approach you use for determining the type is a matter of personal choice.

Next let's see how you would set up your consumer for multiple types with the Avro union schema:

Listing 4.16 Configuring the consumer for working with union schema with Avro

```
//Other configurations details left out for clarity

consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaAvroDeserializer.class);   ❶
consumerProps.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG,
  true);   ❷

Consumer<SpecificRecord> consumer = new KafkaConsumer<>(consumerProps);   ❸
```

❶ Using Avro deserializer

❷ Specifying the deserializer to return a specific Avro type

❸ Creating the `KafkaConsumer`

As you've seen before you specify the `KafkaAvroDeserializer` for the deserializer configuration. We also covered before how Avro is slightly different from Protobuf and JSON Schema in that you tell it to return the specific class type, but you don't provide the class name. So when you have multiple event types in a topic and you are using Avro, the consumer needs to use the `SpecificRecord` interface again in the generics shown in annotation three.

So by using the `SpecificRecord` interface when you start retrieving records from the `Consumer.poll` call you'll need to determine the concrete type to do any work with it.

Listing 4.17 Determining the concrete type of a record returned from a consumer with Avro union schemas

```
// Details left out for clarity

SpecificRecord avroRecord = record.value();
if (avroRecord instanceof PlaneEvent) {
    PlaneEvent planeEvent = (PlaneEvent) avroRecord;
    ....
} else if (avroRecord instanceof TruckEvent) {
    TruckEvent truckEvent = (TruckEvent) avroRecord;
    ....
} else if (avroRecord instanceof DeliveryEvent) {
    DeliveryEvent deliveryEvent = (DeliveryEvent) avroRecord;
   ....
}
```

The approach here is similar to that of what you did with Protobuf but this is at the class level instead of the field level. You could also choose to model your Avro approach to something similar of Protobuf and define record that contains a field representing the union. Here's an example:

Listing 4.18 Avro with embedding the union field in a record

```
{
  "type": "record",
  "namespace": "bbejeck.chapter_4.avro",
  "name": "TransportationEvent",   ❶

  "fields" : [
    {"name": "txn_type", "type": [    ❷
      "bbejeck.chapter_4.avro.TruckEvent",
      "bbejeck.chapter_4.avro.PlaneEvent",
      "bbejeck.chapter_4.avro.DeliveryEvent"
    ]}
  ]
}
```

❶ Outer class definition

❷ Avro union type at the field level

In this case, the generated Java code provides a single method `getTxnType()`, but it has return type of `Object`. As a result you'll need to use the same approach of checking for the instance type as you did above when using a union schema, essentially just pushing the issue of determining the record type from the class level to the field level.

NOTE **Java 16 introduces pattern matching with the `instanceof` keyword that removes the need for casting the object after the `instanceof` check**

4.6 Summary

- Kafka Producers send records in batches to topics located on the Kafka broker and will continue to retry sending failed batches until the `delivery.timeout.ms` configuration expires. You can configure a Kafka Producer to be an idempotent producer meaning it guarantees to send records only once and in-order for a given partition. Kafka producers also have a transactional mode that guarantees exactly once delivery of records across multiple topics. You enable the Kafka transactional API in producers by using the configuration `transactional.id` which must be a unique id for each producer. When using consumers in the transactional API, you want to make sure you set the `isolation.level` to read committed so you only consume committed records from transactional topics.
- Kafka Consumers read records from topics. Multiple consumers with the same group id get topic-partition assignments and work together as one logical consumer. Should one member of the group fail its topic-partition assignment(s) are redistributed to other members of the group via process known as rebalancing. Consumers periodically commit the offsets of consumed records so restarting after a shut-down they pick up where they left of processing.
- Kafka producers and consumers offer three different types of delivery guarantees at least once, at most once, and exactly once. At least once means no records are lost, but you may receive duplicates due to retries. At most once means that you won't receive duplicate records but there could be records lost due to errors. Exactly once delivery means you don't receive duplicates and you won't lose any records due to errors.
- Static membership provides you with stability in environments where consumers frequently drop off, only to come back online within a reasonable amount of time.
- The `CooperativeStickyAssignor` provides the much improved rebalance behavior. The cooperative rebalance protocol is probably the best choice to use in most cases as it significantly reduces the amount of downtime during a rebalance.
- The Admin API provides a way to create and manage topics, partitions and records programmatically.
- When you have different event types but the events are related and processing them in-order is important it's worth considering placing the multiple event types in a single topic.

6 Developing Kafka Streams

This chapter covers

- Introducing the Kafka Streams API
- Building our first Kafka Streams application
- Working with customer data; creating more complex applications
- Splitting, merging and branching streams oh my!

Simply stated, a Kafka Streams application is a graph of processing nodes that transforms event data as it streams through each node. Let's take a look at an illustration of what this means:

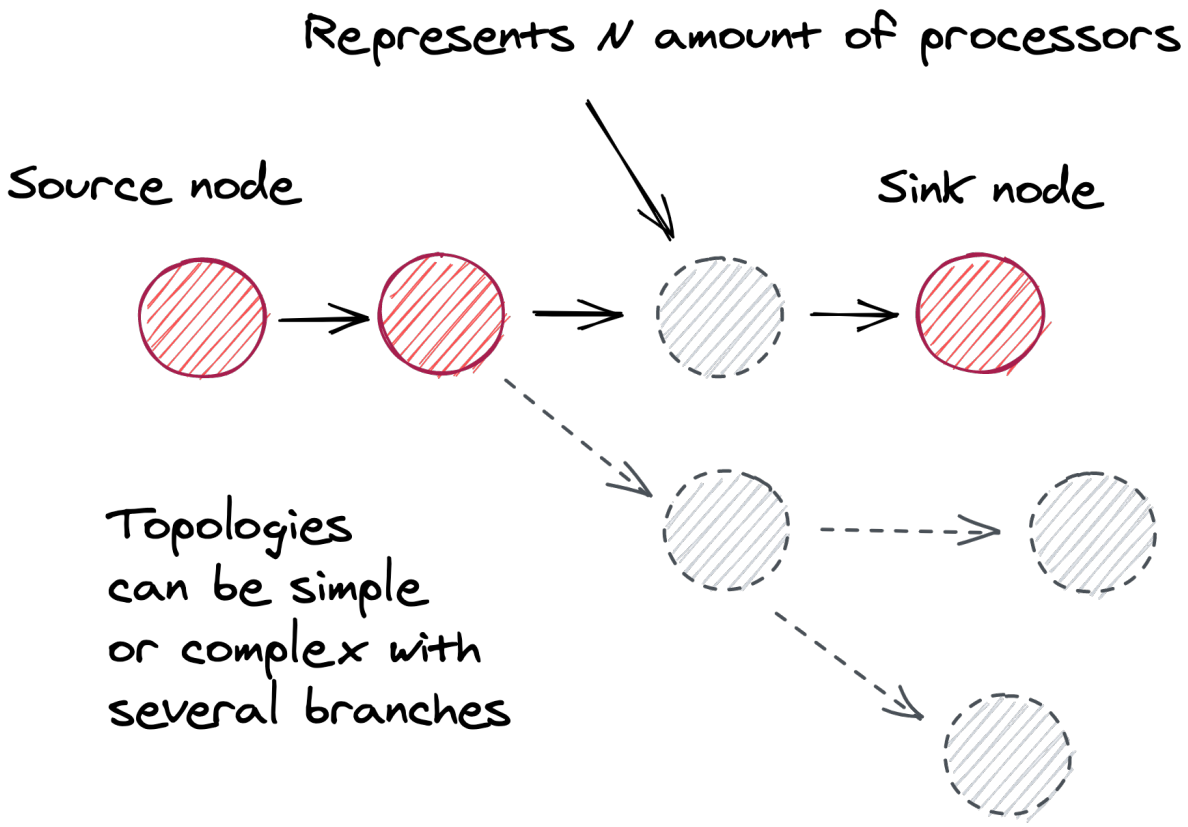


Figure 6.1 Kafka Streams is a graph with a source node, any number of processing nodes and a sink node

This illustration represents the generic structure of most Kafka Streams applications. There is a source node that consumes event records from a Kafka broker. Then there are any number of processing nodes, each performing a distinct task and finally a sink node used to write the transformed records back out to Kafka. In a previous chapter we discussed how to use the Kafka clients for producing and consuming records with Kafka. Much of what you learned in that chapter applies for Kafka Streams, because at it's heart, Kafka Streams is an abstraction over the producers and consumers, leaving you free to focus on your stream processing requirements.

IMPORTANT	**While Kafka Streams is the native stream processing library for Apache Kafka ®, it does not run inside the cluster or brokers, but connects as a client application.**

In this chapter, you'll learn how to build such a graph that makes up a stream processing application with Kafka Streams.

6.1 The Streams DSL

The Kafka Streams DSL is the high-level API that enables you to build Kafka Streams applications quickly. This API is very well thought out, with methods to handle most stream-processing needs out of the box, so you can create a sophisticated stream-processing program without much effort. At the heart of the high-level API is the `KStream` object, which represents the streaming key/value pair records.

Most of the methods in the Kafka Streams DSL return a reference to a `KStream` object, allowing for a fluent interface style of programming. Additionally, a good percentage of the `KStream` methods accept types consisting of single-method interfaces allowing for the use of lambda expressions. Taking these factors into account, you can imagine the simplicity and ease with which you can build a Kafka Streams program.

There's also a lower-level API, the Processor API, which isn't as succinct as the Kafka Streams DSL but allows for more control. We'll cover the Processor API in a later chapter. With that introduction out of the way, let's dive into the requisite Hello World program for Kafka Streams.

6.2 Hello World for Kafka Streams

For the first Kafka Streams example, we'll build something fun that will get off the ground quickly so you can see how Kafka Streams works; a toy application that takes incoming messages and converts them to uppercase characters, effectively yelling at anyone who reads the message. We'll call this the Yelling App.

Before diving into the code, let's take a look at the processing topology you'll assemble for this application:

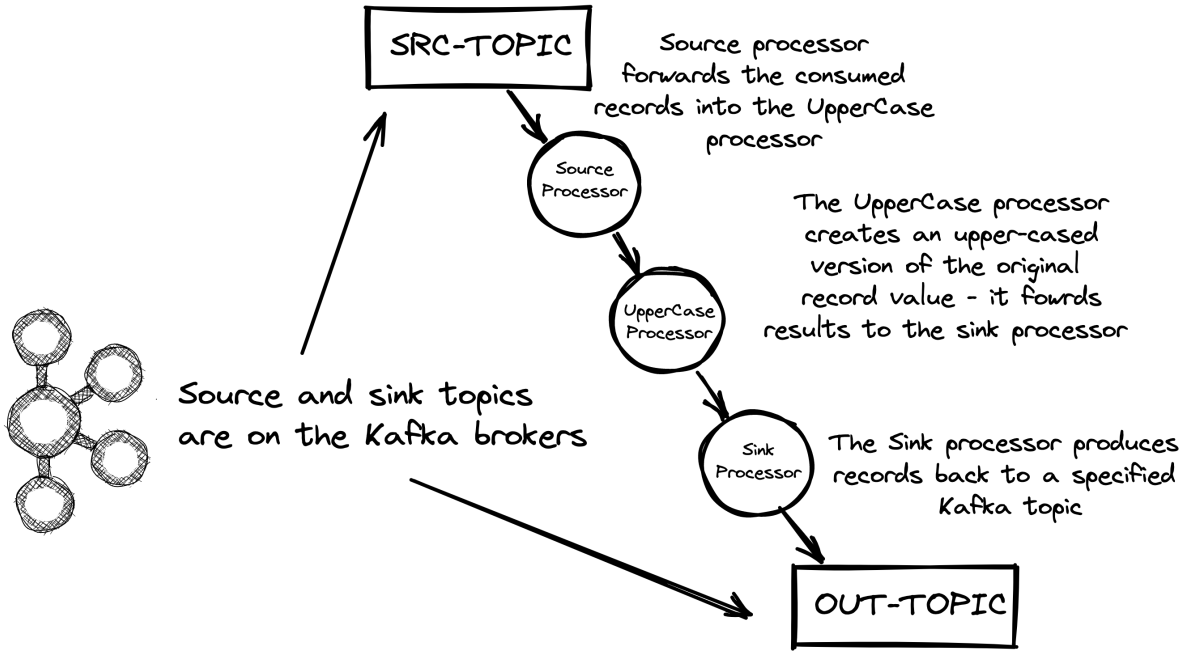


Figure 6.2 Topology of the Yelling App

As you can see, it's a simple processing graph—so simple that it resembles a linked list of nodes more than the typical tree-like structure of a graph. But there's enough here to give you strong clues about what to expect in the code. There will be a source node, a processor node transforming incoming text to uppercase, and a sink processor writing results out to a topic.

This is a trivial example, but the code shown here is representative of what you'll see in other Kafka Streams programs. In most of the examples, you'll see a similar pattern:

1. Define the configuration items.
2. Create `Serde` instances, either custom or predefined, used in deserialization/serializtion of records.
3. Build the processor topology.
4. Create and start the `Kafka Streams`.

When we get into the more advanced examples, the principal difference will be in the complexity of the processor topology. With all this in mind, it's time to build your first application.

6.2.1 Creating the topology for the Yelling App

The first step to creating any Kafka Streams application is to create a source node and that's exactly what you're going to do here. The source node is the root of the topology and fowards thge consumed records into application. Figure 6.3 highlights the source node in the graph.

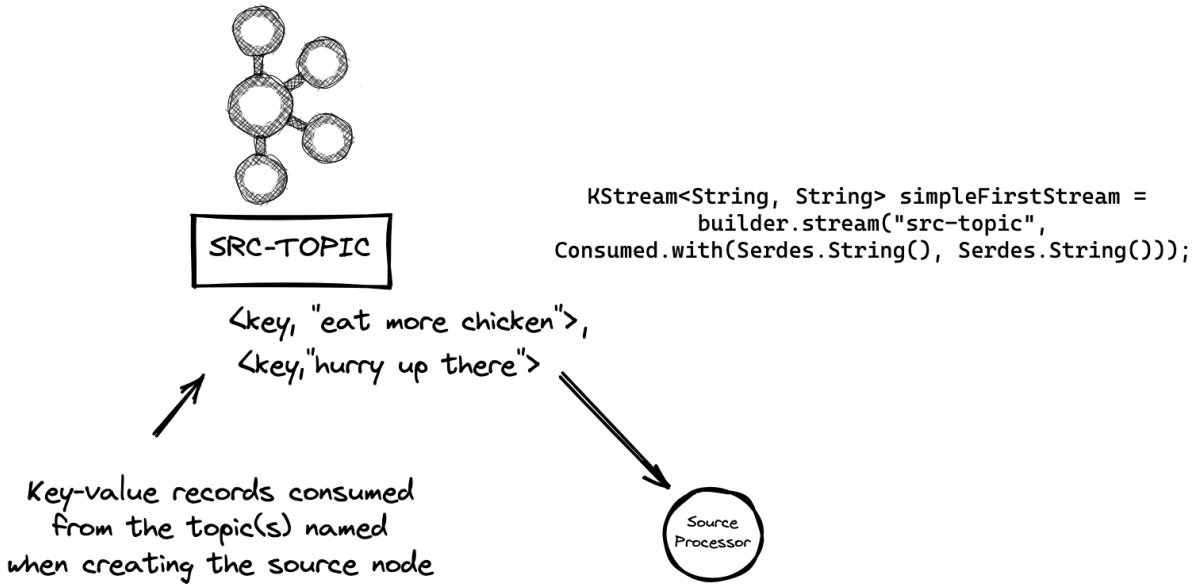


Figure 6.3 Creating the source node of the Yelling App

The following line of code creates the source, or parent, node of the graph.

Listing 6.1 Defining the source for the stream

```
KStream<String, String> simpleFirstStream = builder.stream("src-topic",
Consumed.with(Serdes.String(), Serdes.String()));
```

The `simpleFirstStream` instance is set to consume messages from the `src-topic` topic. In addition to specifying the topic name, you can add a `Consumed` object that Kafka Streams uses to configure optional parameters for a source node. In this example you've provided `Serde` instances, the first for the key and the second one for the value. A `Serde` is a wrapper object that contains a serializer and deserializer for a given type.

If you remember from our dicussion on consumer clients in a previous chapter, the broker stores and forwards records in byte array format. For Kafka Streams to perform any work, it needs to deserialize the bytes into concrete objects. Here both `Serde` objects are for strings, since that's the type of both the key and the value. Kafka Streams will use the `Serde` to deserialize the key and value, separately, into string objects. We'll explain Serdes in more detail soon. You can also use the `Consumed` class to configure a `TimestampExtractor`, the offset reset for the source node, and provide a name. We'll cover the `TimestampExtractor` and providing names in later sections and since we covered offset resets in a previous chapter, I won't cover them again here.

And that is how to create a `KStream` to read from a Kafka topic. But a single topic is not our only choice. Let's take a quick look at some other options. Let's say that there are several topics you'd like to like to "yell at". In that case you can subscribe to all of them at one time by using a `Collection<String>` to specify all the topic names as shown here:

Listing 6.2 Creating the Yelling Application with multiple topics as the source

```
KStream<String, String> simpleFirstStream =
  builder.stream(List.of("topicA", "topicB", "topicC"),
        Consumed.with(Serdes.String(), Serdes.String()))
```

Typically you'd use this approach when you want to apply the same processing to multiple topics at the same time. But what if you have long list of similarly named topics, do you have to write them all out? The answer is no! You can use a regular expresion to subscribe to any topic that matches the pattern:

Listing 6.3 Using a regular expression to subscribe to topics in the Yelling Application

```
KStream<String, String> simpleFirstStream =
   buider.source(Pattern.compile("topic[A-C]"),
      Consumed.with(Serdes.String(), Serdes.String()))
```

Using a regular expression for subscribing to topics is particulary handy when your organization uses a common naming pattern for topics related to their business function. You just have to know the naming pattern and you can subscribe to all of them concisely. Additionally as topics are created or deleted your subscription will automatically update to reflect the changes in the topics.

When subscribing to mulitple topics, there are a few caveats to keep in mind. The keys and values from all subscribed topics must be the same type, for example you can't combine topics where one topic contains `Integer` keys and another has `String` keys. Also, if they all aren't partitioned the same, it's up to you to repartition the data before performing any key based operation like aggregations. We'll cover repartitioning in the next chapter. Finally, there's no ordering guarantees of the incoming records.

You now have a source node for your application, but you need to attach a processing node to make use of the data, as shown in figure 6.4.

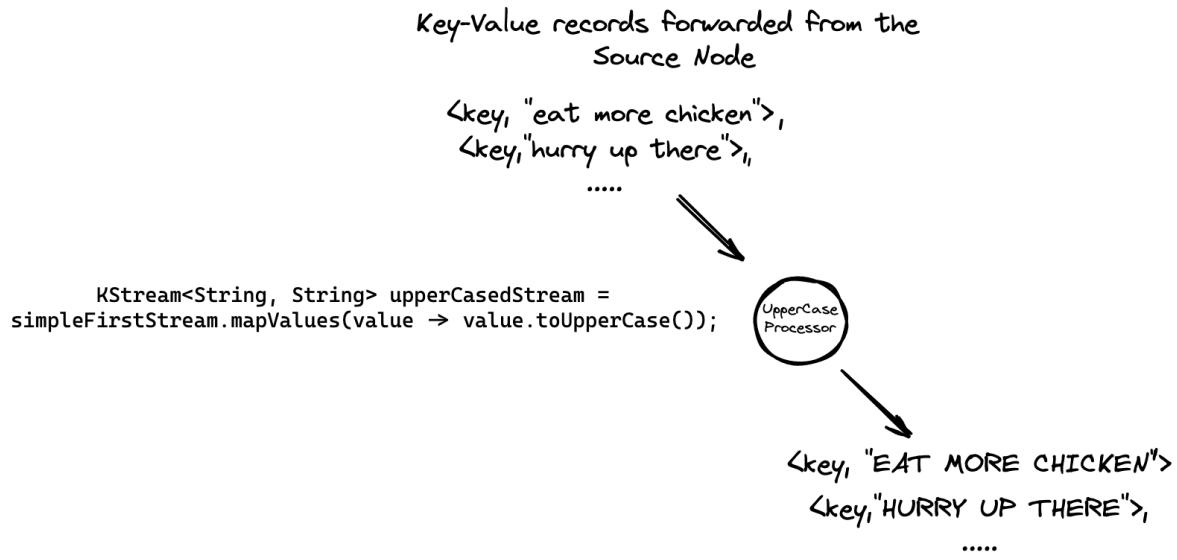


Figure 6.4 Adding the uppercase processor to the Yelling App

Listing 6.4 Mapping incoming text to uppercase

```
KStream<String, String> upperCasedStream =
  simpleFirstStream.mapValues(value -> value.toUpperCase());
```

In the introduction to this chapter I mentioned that a Kafka Streams application is a graph of processing nodes, a directed acyclic graph or DAG to be precise.

You build the graph one processor at a time. With each method call, you establish a parent-child relationship between the nodes of the graph. The parent-child relationship in Kafka Streams establishes the direction for the flow of data, parent nodes forward records to their children. A parent node can have multiple children, but a child node will only have one parent.

So looking at the code example here, by executing `simpleFirstStream.mapValues` , you're creating a new processing node whose inputs are the records consumed in the source node. So the source node is the "parent" and it forwards records to its "child", the processing node returned from the `mapValues` operation.

NOTE **As you tell from the name `mapValues` only affects the value of the key-value pair, but the key of the original record is still forwared along.**

The `mapValues()` method takes an instance of the `ValueMapper<V, V1>` interface. The `ValueMapper` interface defines only one method, `ValueMapper.apply`, making it an ideal candidate for using a lambda expression, which is exactly what you've done here with `value value.toUpperCase()`.

> **NOTE** Many tutorials are available for lambda expressions and method references. Good starting points can be found in Oracle's Java documentation: "Lambda Expressions" (mng.bz/J0Xm) and "Method References" (mng.bz/BaDW).

So far, your Kafka Streams application is consuming records and transforming them to uppercase. The final step is to add a sink processor that writes the results out to a topic. Figure 6.5 shows where you are in the construction of the topology.

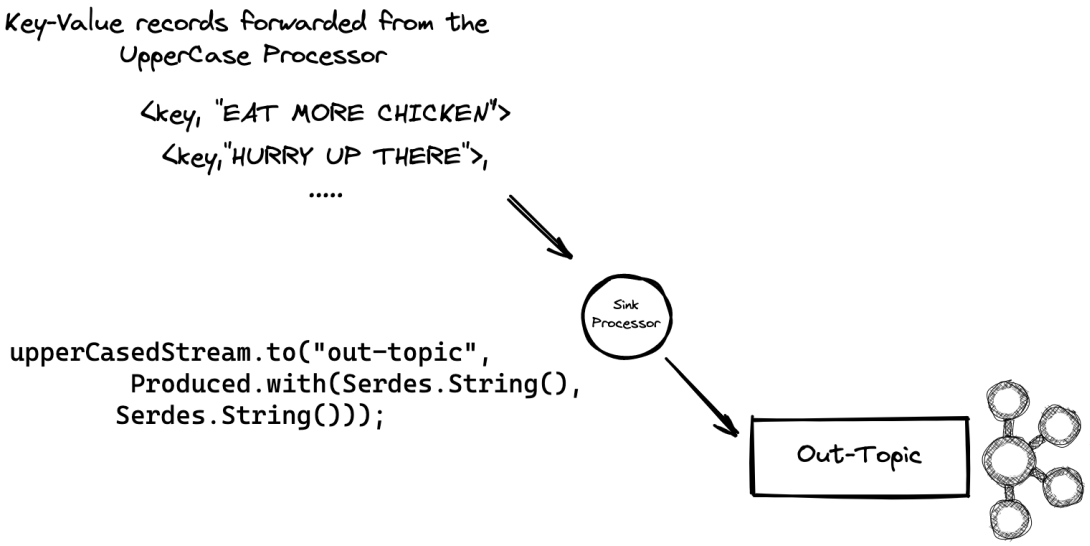


Figure 6.5 Adding a processor for writing the Yelling App results

The following code line adds the last processor in the graph.

Listing 6.5 Creating a sink node

```
upperCasedStream.to("out-topic",
              Produced.with(Serdes.String(), Serdes.String()));
```

The `KStream.to` method creates a processing node that writes the final transformed records to a Kafka topic. It is a child of the `upperCasedStream`, so it receives all of its inputs directly from the results of the `mapValues` operation.

Again, you provide `Serde` instances, this time for serializing records written to a Kafka topic. But in this case, you use a `Produced` instance, which provides optional parameters for creating a sink node in Kafka Streams.

NOTE **You don't always have to provide `Serde` objects to either the `Consumed` or `Produced` objects. If you don't, the application will use the serializer/deserializer listed in the configuration. Additionally, with the `Consumed` and `Produced` classes, you can specify a `Serde` for either the key or value only.**

The preceding example uses three lines to build the topology:

```
KStream<String,String> simpleFirstStream =
builder.stream("src-topic", Consumed.with(Serdes.String(), Serdes.String()));

KStream<String, String> upperCasedStream =
simpleFirstStream.mapValues(value -> value.toUpperCase());
upperCasedStream.to("out-topic", Produced.with(Serdes.String(), Serdes.String()));
```

Each step is on an individual line to demonstrate the different stages of the building process. But all methods in the `KStream` API that don't create terminal nodes (methods with a return type of `void`) return a new `KStream` instance, which allows you to use the fluent interface style of programming. A fluent interface (martinfowler.com/bliki/FluentInterface.html) is an approach where you chain method calls together for more concise and readable code. To demonstrate this idea, here's another way you could construct the Yelling App topology:

```
builder.stream("src-topic", Consumed.with(Serdes.String(), Serdes.String()))
 .mapValues(value -> value.toUpperCase())
 .to("out-topic", Produced.with(Serdes.String(), Serdes.String()));
```

This shortens the program from three lines to one without losing any clarity or purpose. From this point forward, all the examples will be written using the fluent interface style unless doing so causes the clarity of the program to suffer.

You've built your first Kafka Streams topology, but we glossed over the important steps of configuration and `Serde` creation. We'll look at those now.

6.2.2 Kafka Streams configuration

Although Kafka Streams is highly configurable, with several properties you can adjust for your specific needs, the uses the two required configuration settings, `APPLICATION_ID_CONFIG` and `BOOTSTRAP_SERVERS_CONFIG`:

```
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
```

Both are required because there's no practical way to provide default values for these configurations. Attempting to start a Kafka Streams program without these two properties defined will result in a `ConfigException` being thrown.

The `StreamsConfig.APPLICATION_ID_CONFIG` property uniquely identifies your Kafka Streams application. Kafka Streams instances with the same application-id are considered one logical application. We'll discuss this concept later in Kafka Streams internals section. The application-id also serves as a prefix for the embedded client (`KafkaConsumer` and `KafkaProducer`) configurations. You can choose to provide custom configurations for the embedded clients by using one of the various prefix labels found in the `StreamsConfig` class. However, the default client configurations in Kafka Streams have been chosen to provide the best performance, so one should exercise caution when adjusting them.

The `StreamsConfig.BOOTSTRAP_SERVERS_CONFIG` property can be a single `hostname:port` pair or multiple `hostname:port` comma-separated pairs. The `BOOTSTRAP_SERVERS_CONFIG` is what Kafka Streams uses to establish a connection to the Kafka cluster. We'll cover several more configuration items as we explore more examples in the book.

6.2.3 Serde creation

In Kafka Streams, the `Serdes` class provides convenience methods for creating `Serde` instances, as shown here:

```
Serde<String> stringSerde = Serdes.String();
```

This line is where you create the `Serde` instance required for serialization/deserialization using the `Serdes` class. Here, you create a variable to reference the `Serde` for repeated use in the topology. The `Serdes` class provides default implementations for the following types: String, Byte Array, Bytes, Long, Short, Integer, Double, Float, ByteBuffer, UUID, and Void.

Implementations of the `Serde` interface are extremely useful because they contain the serializer and deserializer, which keeps you from having to specify four parameters (key serializer, value serializer, key deserializer, and value deserializer) every time you need to provide a `Serde` in a `KStream` method. In upcoming examples, you'll use Serdes for working with Avro, Protobuf, and JSONSchema as well as create a `Serde` implementation to handle serialization/deserialization of more-complex types.

Let's take a look at the whole program you just put together. You can find the source in src/main/java/bbejeck/chapter_6/KafkaStreamsYellingApp.java (source code can be found on the book's website here: www.manning.com/books/kafka-streams-in-action-second-edition).

Listing 6.6 Hello World: the Yelling App

```
//Details left out for clarity
public class KafkaStreamsYellingApp extends BaseStreamsApplication {

private static final Logger LOG =
  LoggerFactory.getLogger(KafkaStreamsYellingApp.class);

@Override
public Topology topology(Properties streamProperties) {

  Serde<String> stringSerde = Serdes.String();   ❶
  StreamsBuilder builder = new StreamsBuilder();   ❷

  KStream<String, String> simpleFirstStream = builder.stream("src-topic",
            Consumed.with(stringSerde, stringSerde));   ❸
  KStream<String, String> upperCasedStream =
  simpleFirstStream.mapValues(value)-> value.toUpperCase());   ❹

  upperCasedStream.to("out-topic",
      Produced.with(stringSerde, stringSerde));   ❺

  return builder.build(streamProperties);
}

public static void main(String[] args) throws Exception {
    Properties streamProperties = new Properties();
    streamProperties.put(StreamsConfig.APPLICATION_ID_CONFIG,
      "yelling_app_id");
    streamProperties.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
      "localhost:9092");
    KafkaStreamsYellingApp yellingApp = new KafkaStreamsYellingApp();
    Topology topology = yellingApp.topology(streamProperties);

    try(KafkaStreams kafkaStreams =
                    new KafkaStreams(topology, streamProperties)) {
        LOG.info("Hello World Yelling App Started");
        kafkaStreams.start();   ❻

        LOG.info("Shutting down the Yelling APP now");
    }
 }
}
```

❶ Creates the Serdes and store in a variable used to serialize/deserialize keys and values

❷ Creates the StreamsBuilder instance used to construct the processor topology

❸ Creates the actual stream with a source topic to read from (the parent node in the graph)

❹ A processor using a lambda (the first child node in the graph)

❺ Writes the transformed output to another topic (the sink node in the graph)

❻ Kicks off the Kafka Streams threads

You've now constructed your first Kafka Streams application. Let's quickly review the steps involved, as it's a general pattern you'll see in most of your Kafka Streams applications:

1. Create a `Properties` instance for configurations.

2. Create a `Serde` object.
3. Construct a processing topology.
4. Start the Kafka Streams program.

We'll now move on to a more complex example that will allow us to explore more of the Streams DSL API.

6.3 Masking credit card numbers and tracking purchase rewards in a retail sales setting

Imagine you work as a infrastructure engineer for the retail giant, ZMart. ZMart has adopted Kafka as its data processing backbone and is looking to capitalize on the ability to quickly process customer data, intended to help ZMart do business more efficiently.

At this point you're tasked to build a Kafka Streams application to work with purchase records as they come streaming in from transactions in ZMart stores.

Here are the requirements for the streaming program, which will also serve as a good description of what the program will do:

1. All Purchase objects need to have credit card numbers protected, in this case by masking the first 12 digits.
2. You need to extract the items purchased and the ZIP code to determine regional purchase patterns and inventory control. This data will be written out to a topic.
3. You need to capture the customer's ZMart member number and the amount spent and write this information to a topic. Consumers of the topic will use this data to determine rewards.

With these requirements at hand, let's get started building a streaming application that will satisfy ZMart's business requirements.

6.3.1 Building the source node and the masking processor

The first step in building the new application is to create the source node and first processor of the topology. You'll do this by chaining two calls to the `KStream` API together. The child processor of the source node will mask credit card numbers to protect customer privacy.

Listing 6.7 Building the source node and first processor

```
KStream<String, RetailPurchase> retailPurchaseKStream =
     streamsBuilder.stream("transactions",
     Consumed.with(stringSerde, retailPurchaseSerde))
    .mapValues(creditCardMapper);
```

You create the source node with a call to the `StreamBuilder.stream` method using a default `String` serde, a custom serde for `RetailPurchase` objects, and the name of the topic that's the

source of the messages for the stream. In this case, you only specify one topic, but you could have provided a comma-separated list of names or a regular expression to match topic names instead.

In this code example, you provide `Serdes` with a `Consumed` instance, but you could have left that out and only provided the topic name and relied on the default `Serdes` provided via configuration parameters.

The next immediate call is to the `KStream.mapValues` method, taking a `ValueMapper<V, V1>` instance as a parameter. Value mappers take a single parameter of one type (a `RetailPurchase` object, in this case) and map that object to a to a new value, possibly of another type. In this example, `KStream.mapValues` returns an object of the same type (`RetailPurchase`), but with a masked credit card number.

When using the `KStream.mapValues` method, you don't have access to the key for the value computation. If you wanted to use the key to compute the new value, you could use the `ValueMapperWithKey<K, V, VR>` interface, with the expectation that the key remains the same. If you need to generate a new key along with the value, you'd use the `KStream.map` method that takes a `KeyValueMapper<K, V, KeyValue<K1, V1>>` interface.

IMPORTANT **Keep in mind that Kafka Streams functions are expected to operate without side effects, meaning the functions don't modify the original key and or value, but return new objects when making modifications.**

6.3.2 Adding the patterns processor

Now you'll build the second processor, responsible for extracting geographical data from the purchase, which ZMart can use to determine purchase patterns and inventory control in regions of the country. There's also an additional wrinkle with building this part of the topology. The ZMart business analysts have determined they want to see individual records for each item in a purchase and they want to consider purchases made regionally together.

The `RetailPurchase` data model object contains all the items in a customer purchase so you'll need to emit a new record for each one in the transaction. Additionally, you'll need to add the zip-code in the transaction as the key. Finally you'll add a sink node responsible for writing the pattern data to a Kafka topic.

In pattrens processor example you can see the `retailPurchaseKStream` processor using a `flatMap` operator. The `KStream.flatMap` method takes a `ValueMapper` or a `KeyValueMapper` that accepts a single record and returns an `Iterable` (any Java `Collection`) of new records, possibly of a different type. The `flapMap` processor "flattens" the `Iterable` into one or more records forwarded to the topology. Let's take a look at an illustrating how this works:

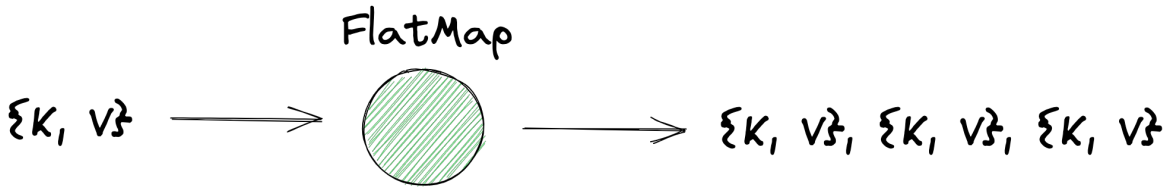


Figure 6.6 FlatMap emits zero or more records from a single input records by flattening a collection returned from a KeyValueMapper or ValueMapper

The process of a flatMap is a common operation from functional programming where one input results creating a collection of items (the map portion of the function) but instead of returning the collection, it "flattens" the collection or grouping into a sequence of records.

In our case here with Kafka Streams, a retail purchase of five items results in five individual `KeyValue` objects with the keys corresponding to the zip-code and the values a `PurchasedItem` object.

Here's the code listing for the `KeyValueMapper`:

Listing 6.8 KeyValueMapper returning a collection of PurchasedItem objects

```
KeyValueMapper<String, RetailPurchase,
 Iterable<KeyValue<String, PurchasedItem>>> retailTransactionToPurchases =
    (key, value) -> {
      String zipcode = value.getZipCode();   ❶
      return value.getPurchasedItemsList().stream()   ❷
                .map(purchasedItem ->
                    KeyValue.pair(zipcode, purchasedItem))
                .collect(Collectors.toList());
}
```

❶ Extracting the zipcode on the purchase for the new key

❷ Using the Java stream API to create a list of KeyValue pairs

The `KeyValueMapper` here takes an individual transaction object and returns a list of KeyValue objects. The key is the zipcode where the transaction took place and the value is an item included in the purchase. Now let's put our new `KeyValueMapper` into this section of the topology we're creating:

Listing 6.9 Patterns processor and a sink node that writes to Kafka

```
KStream<String, Pattern> patternKStream = retailPurchaseKStream
                .flatMap(retailTransactionToPurchases)   ❶
                .mapValues(patternObjectMapper);   ❷

patternKStream.print(Printed.<String,Pattern>toSysOut()
              .withLabel("patterns"));   ❸
patternKStream.to("patterns",
        Produced.with(stringSerde,purchasePatternSerde));   ❹
```

❶ Using flatMap to create new object for each time in a transaction

❷ Mapping each purchase to a pattern object

❸ Printing records to the console

❹ Producing each record from the purchase to a Kafka topic called "patterns"

In this code example you declare a variable to hold the reference of the new `KStream` instance and you'll see why in an upcoming section. The purchase-patterns processor forwards the records it receives to a child node of its own, defined by the method call `KStream.to`, writing to the `patterns` topic. Note the use of a `Produced` object to provide the previously built `Serde`. I've also snuck in a `KStream#print` processor that prints the key-values of the stream to the console, we'll talk more about viewing stream records in an upcoming section.

The `KStream.to` method is a mirror image of the `KStream.source` method. Instead of setting a source for the topology to read from, it defines a sink node that's used to write the data from a `KStream` instance to a Kafka topic. The `KStream.to` method also provides overloads which accept an object allowing for dynamic topic selection and we'll discuss that soon.

6.3.3 Building the rewards processor

The third processor in the topology is the customer rewards accumulator node shown in figure 8, which will let ZMart track purchases made by members of their preferred customer club. The rewards accumulator sends data to a topic consumed by applications at ZMart HQ to determine rewards when customers complete purchases.

Listing 6.10 Third processor and a terminal node that writes to Kafka

```
 KStream<String, RewardAccumulatorProto.RewardAccumulator> rewardsKStream =
       retailPurchaseKStream.mapValues(rewardObjectMapper);
rewardsKStream.to("rewards",
        Produced.with(stringSerde,rewardAccumulatorSerde));
```

You build the rewards accumulator processor using what should be by now a familiar pattern: creating a new `KStream` instance that maps the raw purchase data contained in the retail purchase object to a new object type. You also attach a sink node to the rewards accumulator so the results of the rewards `KStream` can be written to a topic and used for determining customer reward levels.

Now that you've built the application piece by piece, let's look at the entire application (src/main/java/bbejeck/chapter_6/ZMartKafkaStreamsApp.java).

Listing 6.11 ZMart customer purchase `KStream` program

```
public class ZMartKafkaStreamsApp {

// Details left out for clarity


@Override
public Topology topology(Properties streamProperties) {

StreamsBuilder streamsBuilder = new StreamsBuilder();

KStream<String, RetailPurchaseProto.RetailPurchase> retailPurchaseKStream =
        streamsBuilder.stream("transactions",
            Consumed.with(stringSerde, retailPurchaseSerde))
        .mapValues(creditCardMapper);    ❶

KStream<String, PatternProto.Pattern> patternKStream =
   retailPurchaseKStream
        .flatMap(retailTransactionToPurchases)
        .mapValues(patternObjectMapper);    ❷

patternKStream.to("patterns",
      Produced.with(stringSerde,purchasePatternSerde));

KStream<String, RewardAccumulatorProto.RewardAccumulator> rewardsKStream =
      retailPurchaseKStream.mapValues(rewardObjectMapper);    ❸

rewardsKStream.to("rewards",
      Produced.with(stringSerde,rewardAccumulatorSerde));
retailPurchaseKStream.to("purchases",
      Produced.with(stringSerde,retailPurchaseSerde));

return streamsBuilder.build(streamProperties);

}
```

❶ Builds the source and first processor

❷ Builds the PurchasePattern processor

❸ Builds the RewardAccumulator processor

NOTE **I've left out some details in the listing clarity. The code examples in the book aren't necessarily meant to stand on their own. The source code that accompanies this book provides the full examples.**

As you can see, this example is a little more involved than the Yelling App, but it has a similar flow. Specifically, you still performed the following steps:

- Create a `StreamsBuilder` instance.
- Build one or more `Serde` instances.
- Construct the processing topology.
- Assemble all the components and start the Kafka Streams program.

You'll also notice that I haven't shown the logic responsible for creating the various mappings from the original transaction object to new types and that is by design. First of all, the code for a

`KeyValueMapper` or `ValueMapper` is going to be distinct for each use case, so the particular implementations don't matter too much.

But more to the point, if you look over the entire Kafka Streams application you can quickly get a sense of what each part is accomplishing, and for the most part any details of working directly with Kafka are abstracted away. And to me that is the strength of Kafka Streams; with the DSL you get specify ***what*** operations you need to perform on the event stream and Kafka Streams handles the details. Now it's true that no one framework can solve every problem and sometimes you need a more hands-on lower level approach and you'll learn about that in a upcoming chapter when we cover the Processor API.

In this application, I've mentioned using a `Serde`, but I haven't explained why or how you create them. Let's take some time now to discuss the role of the `Serde` in a Kafka Streams application.

6.3.4 Using Serdes to encpsulate serializers and deserializers in Kafka Streams

As you learned in previous chapters, Kafka brokers work with records in byte array format. It's the responsibility of the client to serialize when producing records and deserialize when consuming. It's no different with Kafka Streams as it uses embedded consumers and producers. There is one small difference when configuring a Kafka Streams application for serialization vs. raw producer or consumer clients. Instead of providing a specific deserializer or serializer, you configure Kafka Streams with a `Serde`, which contains both the serializer and deserializer for a specific type.

Some serdes are provided out of the box by the Kafka client dependency, (`String`, `Long`, `Integer`, and so on), but you'll need to create custom serdes for other objects.

In the first example, the Yelling App, you only needed a serializer/deserializer for strings, and an implementation is provided by the `Serdes.String()` factory method. In the ZMart example, however, you need to create custom `Serde` instances, because of the arbitrary object types. We'll look at what's involved in building a `Serde` for the `RetailPurchase` class. We won't cover the other `Serde` instances, because they follow the same pattern, just with different types.

> **NOTE** **I'm including this discussion on Serdes creation for completeness, but in the source code there is a class `SerdeUtil` which provides a `protobufSerde` method which you'll see in the examples and encapsulates the steps described in this section.**

Building a `Serde` requires implementations of the `Deserializer<T>` and `Serializer<T>` interfaces. We covered creating your own serializer and deserializer instances towards the end of

chapter 3 on Schema Registry, so I won't go over those details again here. For reference you can see the full code for the `ProtoSerializer` and `ProtoDeserializer` in the `bbejeck.serializers` package in the source for the book.

Now, to create a `Serde<T>` object, you'll use the `Serdes.serdeFrom` factory method taking steps like the following:

```
Deserializer<RetailPurchaseProto.RetailPurchase> purchaseDeserializer =
      new ProtoDeserializer<>();   ❶
Serializer<RetailPurchaseProto.RetailPurchase> purchaseSerializer =
      new ProtoDeserializer<>();   ❷
Map<String, Class<RetailPurchaseProto.RetailPurchase>> configs
            = new HashMap<>();
   configs.put(false, RetailPurchaseProto.RetailPurchase.class);
         deserializer.configure(configs,isKey);    ❸
Serde<RetailPurchaseProto.RetailPurchase> purchaseSerde =
     Serdes.serdeFrom(purchaseSerializer,purchaseDeserializer);   ❹
```

❶ Creates the Deserializer for the RetailPurchaseProto.RetailPurchase class

❷ Creates the Serializer for the RetailPurchaseProto.RetailPurchase class

❸ Configurations for the deserializer

❹ Creates the Protobuf Serde for RetailPurchaseProto.RetailPurchase objects

As you can see, a `Serde` object is useful because it serves as a container for the serializer and deserializer for a given object. Here you need to create a custom Serde for the Protobuf objects because the streams example does not use Schema Registry, but using it with Kafka Streams is a perfectly valid use case. Let's take a quick pause to go over how you configure your Kafka Streams application when using it with Schema Registry.

6.3.5 Kafka Streams and Schema Registry

In chapter four I discussed the reasons why you'd want to use Schema Registry with a Kafka based application. I'll briefly describe those reasons here. The domain objects in your application represent an implicit contract between the different users of your application. For example imagine one team of developers change a field type from a `java.util.Date` to a `long` and start producing those changes to Kafka, the downstream consumers applications will break due to the unexpected field type change.

So by using a schema and using Schema Resitry to store it, you make it much easier to enforce this contract by enabling better coordination and compatibility checks. Additionally, there are Schema Registry project provides Schema Registry "aware" (de)serializers and Serdes, alleviating the developer from writing the serialization code.

IMPORTANT **Schema Registry provides both a `JSONSerde` and a `JSONSchemaSerde`, but they are not interchangeable! The `JSONSerde` is for Java objects that use JSON for describing the object. The `JSONSchemaSerde` is for objects that use `JSONSchema` as the formal definition of the object.**

So how would the `ZMartKafkaStreamsApp` change to work with Schema Registry? All that is required is to use Schema Registry aware Serde instances. The steps for creating a Schema Regeistry aware Serde are simple:

1. Create an instance of one the provided Serde instances
2. Configure it with the URL for a Schema Registry server.

Here are the concrete steps you'll take:

```
KafkaProtobufSerdePurchase> protobufSerde =
 new KafkaProtobufSerde<>(Purchase.class);   ❶
String url = "https://...";                                  ❷
Map<String, Object> configMap = new HashMap<>();
configMap.put(
  AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  url);                                                      ❸
protobufSerde.configure(configMap, false);     ❹
```

❶ Instantiating the KafkaProtobufSerde providing the class type as a constructor parameter

❷ The URL for the location of a Schema Registry instance

❸ Putting the URL in a HashMap

❹ Calling the KafkaProtobufSerde#configure method

So, with just few lines of code, you've created a Schema Registry aware Serde that you can use in your Kafka Streams application.

IMPORTANT **Since Kafka Streams contains consumer and producer clients, the same rules for schema evolution and compatibility apply**

We've covered a lot of ground so far in developing a Kafka Streams application. We still have much more to cover, but let's pause for a moment and talk about the development process itself and how you can make life easier for yourself while developing a Kafka Streams application.

6.4 Interactive development

You've built the graph to process purchase records from ZMart in a streaming fashion, and you have three processors that write out to individual topics. During development it would certainly be possible to have a console consumer running to view results. But instead of using an external tool, it would be more convenient to have your Kafka Streams application print or log from anywhere you want inside the topology. This visual type of feedback directly from the application is very efficient during the development process. You enable this output by using the `KStream.peek()` or the `KStream.print()` method.

The `KStream.peek()` allows you to perform a stateless action (via the `ForeachAction` interface) on each record flowing throw the `KStream` instance. It's important to note that this operation is not expected to alter the incoming key and value. Instead the `peek` operator is an opportunity to print, log, or collect information at arbitrary points in the topology. Let's take a another look at Yelling application, but now add a way to view the records before and after the application starts "yelling":

Listing 6.12 Printing records flowing through the Yelling application found in bbejeck/chapter_6/KafkaStreamsYellingAppWithPeek

```
// Details left out for clarity

ForeachAction<String, String> sysout =
  (key, value) ->
   System.out.println("key " + key
   + " value " + value);


builder.stream("src-topic",
  Consumed.with(stringSerde, stringSerde))
  .peek(sysout)        ❶
  .mapValues(value -> value.toUpperCase())
  .peek(sysout)        ❷
  .to( "out-topic",
  Produced.with(stringSerde, stringSerde));
```

❶ Printing records to the console as they enter the application

❷ Printing the yelling events

Here we've strategically placed these `peek` operations that will print records to the console, both pre and post the `mapValues` call.

The `KStream.print()` method is purpose built for printing records. Some of the previous code snippets contained examples of using it, but we'll show it again here.

Listing 6.13 Printing records using KStream.print found in bbejeck/chapter_6/KafkaStreamsYellingApp

```
// Details left out for clarity
 ...
 KStream<...> upperCasedStream = simpleFirstStream.mapValues(...);
 upperCasedStream.print(Printed.toSysOut());   (1)
 upperCasedStream.to(...);
```

(1) Printing the upper cased lettters this is an example of a terminal method in Kafka Streams

In this case, you're printing the upper-cased words immediately after transformation. So what's the difference between the two approaches? You should notice with the `KStream.print()` operation, you didn't chain the method calls together like you did using `KStream.peek()` and this is becuase `print` is a terminal method.

Terminal methods in Kafka Streams have a return signature of `void`, hence you can't chain another method call afterward, as it terminates the stream. The terminal methods in `KStream` interface are `print`, `foreach`, `process`, and `to`. Asside from the `print` method we just discussed, you'll use `to` when you write results back to Kafka. The `foreach` method is useful for performing an operation on each record when you don't need to write the results back to Kafka, such as calling a microservice. The `process` method allows for integrating the DSL with Processor API which we'll discuss in an upcoming chapter.

While either printing method is a valid approach, my preference is to use the `peek` method because it makes it easy to slip a print statement into an existing stream. But this is a personal preference so ultimately it's up to you to decide which approach to use.

So far we've covered some of the basic things we can do with a Kafka Streams application, but we've only scratched the surface. Let's continue exploring what we can do with an event stream.

6.5 Choosing which events to process

So far you've seen how to apply operations to events flowing through the Kafka Streams application. But you are processing every event in the stream and in the same manner. What if there are events you don't want to handle? Or what about events with a given attribute that require you to handle them differently?

Fortunately, there are methods available to provide you the flexibility to meet those needs. The `KStream#filter` method drops records from the stream not matching a given predicate. The `KStream#split` allows you split the original stream into branches for different processing based on provided predicate(s) to reroute records. To make these new methods more concrete let's update the requirements to the original ZMart application:

- The ZMart updated their rewards program and now only provides points for purchases over $10. With this change it would be ideal to simply drop any non-qualifying purchases from the rewards stream.
- ZMart has expanded and has bought an electronics chain and a popular coffee house chain. All purchases from these new stores will flow into the streaming application you've set up, but you'll need to separate those purchases out for different treatment while still processing everything else in the application the same.

NOTE From this point forward, all code examples are pared down to the essentials to maximize clarity. Unless there's something new to introduce, you can assume that the configuration and setup code remain the same. These truncated examples aren't meant to stand alone—the full code listing for this example can be found in src/main/java/bbejeck/chapter_6/ZMartKafkaStreamsFilteringBranchingApp.java.

6.5.1 Filtering purchases

The first update is to remove non-qualifying purchases from the rewards stream. To accomplish this, you'll insert a `KStream.filter()` before the `KStream.mapValues` method. The `filter` takes a `Predicate` interface as a parameter, and it has one method defined, `test()`, which takes two parameters—the key and the value—although, at this point, you only need to use the value.

NOTE There is also `KStream.filterNot`, which performs filtering, but in reverse. Only records that don't match the given predicate are processed further in the topology.

By making these changes, the processor topology graph changes as shown in figure 6.12.

Listing 6.14 Adding a filter to drop purchases not meeting rewards criteria

```
KStream<String, RewardAccumulatorProto.RewardAccumulator> rewardsKStream =
   retailPurchaseKStream   ❶
   .mapValues(rewardObjectMapper)   ❷
   .filter((key,potentialReward) ->
           potentialReward.getPurchaseTotal() > 10.00);   ❸
```

❶ The original rewards stream

❷ Mapping the purchase into a `RewardAccumulator` object

❸ The `KStream.filter` method, which takes a `Predicate<K,V>` instance as a parameter

You have now successfully updated the rewards stream to drop purchases that don't qualify for reward points.

6.5.2 Splitting/branching the stream

There are new events flowing into the purchase stream and you need to process them differently. You'll still want to mask any credit card information, but after that the purchases from the acquired coffee and electronics chain need to get pulled out and sent to different topics. Additionally, you need to continue to process the original events in the same manner.

What you need to do is split the original stream into 3 sub-streams or branches; 2 for handling the new events and 1 to continue processing the original events in the topology you've already built. This splitting of streams sounds tricky, but Kafka Streams provides an elegant way to do this as we'll see now. Here's an illustration demonstrating the conceptual idea of what splitting a stream involves:

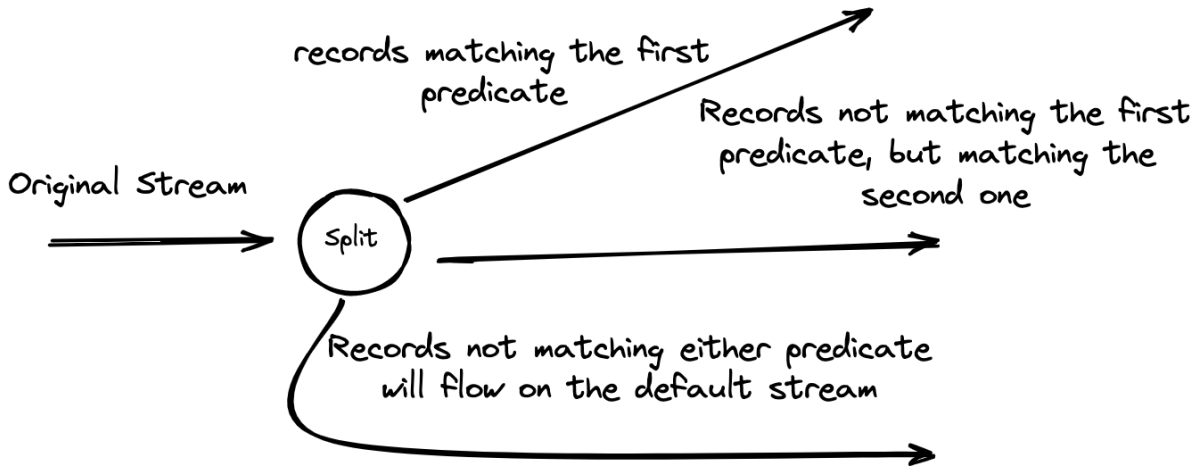


Figure 6.7 Creating branches for the two specific purchase types

The general steps you'll take to split a stream into branches are the following:

1. Use the `KStream.split()` method which returns a `BranchedKStream` object
2. Call `BranchedKStream.branch()` with a with a pair of `Predicate` and `Branched` objects as parameters. The `Predicate` contains a condition when tested against a record returns either true or false. The `Branched` object contains the logic for processing a record. Each execution of this method creates a new branch in the stream.
3. You complete the branching with a call to either `BranchedKStream.defaultBranch()` or `BranchedKStream.noDefaultBranch()`. If you define a default branch any records not matching all the predicates are routed there. With the `noDefaultBranch` option, non-matching records get dropped. When calling either of the branching termination methods a `Map<String, KStream<K, V>` is returned. The `Map` ***may*** contain `KStream` objects for new branch, depending on how you've built the `Branched` objects. We'll cover more options for branching soon.

The `Predicate` acts as a logical gate for it's companion `Branched` object. If the condition returns `true`, then the "gate" opens and the record flows into the processor logic for that branch.

IMPORTANT **When splitting a `KStream` you can't change the types of the keys or values, as each branch has the same types as the parent or original branch.**

In our case here, you'll want to filter out the two purchase types into their own branch. Then create a default branch consisting of everything else. This default branch is really the original purchase stream so it will handle all of the records that don't match either predicate.

Now that we've reviewed the concept let's take a look at the code you'll implement:

Listing 6.15 Splitting the stream found in bbejeck/chapter_6/ZMartKafkaStreamsFilteringBranchingApp

```
//Several details left out for clarity

Predicate<String, Purchase> isCoffee =
  (key, purchase) ->
   purchase.getDepartment().equalsIgnoreCase("coffee");   ❶

Predicate<String, Purchase> isElectronics =
  (key, purchase) ->
  purchase.getDepartment().equalsIgnoreCase("electronics");   ❶

purchaseKStream.split())    ❷
.branch(isCoffee,
 Branched.withConsumer(coffeeStream -> coffeeStream.to("coffee-topic")))   ❸
.branch(isElectronics,
  Branched.withConsumer(electronicStream ->
  electronicStream.to("electronics"))    ❹
.defaultBranch(Branched.withConsumer(retailStream ->
                retailStream.to("purchases"));   ❺
```

❶ Create the predicates for determining branches

❷ Splitting the stream

❸ Writing the coffee purchases out to a topic

❹ Writing the electronic purchases out to a topic

❺ The default branch where non-matching records go

Here in this example you've split the purchase stream into two new streams, one each for the coffee and electronic purchases. Branching provides an elegant way to process records differently within the same stream. While in this initial example each one is a single processor writing records to a topic, these branched streams can be as complex as you need to make them.

> **NOTE** This example sends records to several different topics. Although you can configure Kafka to automatically create topics it's not a good idea to rely on this mechanism. If you use auto-creation, the topics are configured with default values from the server.config properties file, which may or may not be the settings you need. You should always think about what topics you'll need, the number of partitions, the replication factor and create them before running your Kafka Streams application.

In this branching example, you've split out discrete `KStream` objects, which stand alone and don't interact with anything else in the application and that is perfectly an acceptable approach. But now let's consider a situation where you have an event stream you want to tease out into separate components, but you need to combine the new streams with existing ones in the application.

Consider you have IoT sensors and early on you combined two related sensor readings into one topic, but as time went on newer sensors started to send results to distinct topics. The older sensors are fine as is and it would be cost prohibited to go back and make the necessary changes to fit the new infrastructure. So you'll need an application that will split the legacy stream into two streams ***and*** combine or merge them with the newer streams consisting of a single reading type. Another factor is that any proximity readings are reported in feet, but the new ones are in meters, so in addition to extracting the proximity reading into a separate stream, you need to convert the reading values into meters.

Now let's walk through an example of how you'll do splitting and merging starting with the splitting

Listing 6.16 Splitting the stream in a way you have access to new streams

```
//Details left out for clarity

KStream<String, SensorProto.Sensor> legacySensorStream =
    builder.stream("combined-sensors", sensorConsumed);


 Map<String, KStream<String, SensorProto.Sensor>> sensorMap =
        legacySensorStream.split(Named.as("sensor-"))   ❶
        .branch(isTemperatureSensor, Branched.as("temperature"))   ❷
        .branch(isProximitySensor,
            Branched.withFunction(
                ps -> ps.mapValues(feetToMetersMapper), "proximity"))   ❸
        .noDefaultBranch();   ❹
```

❶ Splitting the stream and providing the base name for the map keys

❷ Creating the temperature reading branch and naming the key

❸ Creating the proximity sensor branch with a ValueMapper function

❹ Specifying no default branch, because we know all records fall into only two categories

What's happening overall is each branch call places an entry into a `Map` where the key is the concatenation of name passed into the `KStream.split()` method and the string provided in the `Branched` parameter and the value is a `KStream` instance resulting from each `branch` call.

In the first branching example, the split and subsequent branching calls also returns a `Map`, but in that case it would have been empty. The reason is that when you pass in a `Branched.withConsumer` (a `java.util.Consumer` interface) it's a void method, it returns nothing, hence no entry is placed in the map. But the `Branched.withFunction` (a `java.util.Function` interface) accepts a `KStream<K, V>` object as a parameter and ***returns a KStream<K, V>*** instance so it goes into the map as an entry. At annotation three, the function takes the branched `KStream` object and executes a `MapValues` to convert the proximity sensor reading values from feet to meters, since the sensor records in the updated stream are in meters.

I'd like to point out some subtlety here, the `branch` call at annotation two does not provide a function, but it still ends up in the resulting `Map`, how is that so? When you only provide a `Branched` parameter with name, it's treated the same if you had used a `java.util.Function` that simply returns the provided `KStream` object, also known as an **identity function**. So what's the determining factor to use either `Branched.withConsumer` or `Branched.withFunction`? I can answer that question best by going over the next block of code in our example:

Listing 6.17 Splitting the stream and gaining access to the newly created streams

```
KStream<String, SensorProto.Sensor> temperatureSensorStream =    ❶
      builder.stream("temperature-sensors", sensorConsumed);

KStream<String, SensorProto.Sensor> proximitySensorStream =    ❷
      builder.stream("proximity-sensors", sensorConsumed);

temperatureSensorStream.merge(sensorMap.get("sensor-temperature"))
      .to("temp-reading", Produced.with(stringSerde, sensorSerde));   ❸

proximitySensorStream.merge(sensorMap.get("sensor-proximity"))
     .to("proximity-reading", Produced.with(stringSerde, sensorSerde));   ❹
```

❶ The stream with the new temperature IoT sensors

❷ The stream with the updated proximity IoT sensors

❸ Merging the legacy temperature readings with the new ones

❹ Merging the updated to meters proximity stream with the new proximity stream

To refresh your memory, the requirements for splitting the stream were to extract the different IoT sensor results by type and place them in the same stream as the new updated IoT results and

convert any proximity readings into meters. You accomplish this task by extracting the `KStream` from the map with the corresponding keys created in the branching code in the previous code block.

To accomplish putting the branched legacy stream with the new one, you use a DSL operator `KStream.merge` which is the functional analog of `KStream.split` it merges different `KStream` objects into one. With `KStream.merge` there is no ordering guarantees between records of the different streams, but the relative order of each stream remains. In other words the order of processing between the legacy stream and the updated one is not guaranteed to be in any order but the order inside in each stream is preserved.

So now it should be clear why you use `Branched.withConsumer` or `Branched.withFunction` in the latter case you need to get a handle on the branched `KStream` so you can integrate into the outer application in some manner, while with the former you don't need access to the branched stream.

That wraps up our discussion on branching and merging, so let's move on to cover naming topology nodes in the DSL.

6.5.3 Naming topology nodes

When you build a topology in the DSL, Kafka Streams creates a graph of processor nodes, giving each one a unique name. Kafka Streams generates these node names by taking the name the function of the processor and appending globally incremented number. To view this description of the topology, you'll need to get the `TopologyDescription` object. Then you can view it by printing it to the console.

Listing 6.18 Getting a description of the topology and printing it out

```
TopologyDescription topologyDescription =
  streamsBuilder.build().describe();
System.out.println(topologyDescription.toString());
```

Running the code above yields this output on the console:

Listing 6.19 Full topology description of the KafkaStreamsYellingApplication

```
Topologies:
   Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [src-topic])    ❶
      --> KSTREAM-MAPVALUES-0000000001    ❷
    Processor: KSTREAM-MAPVALUES-0000000001 (stores: [])    ❸
      --> KSTREAM-SINK-0000000002
      <-- KSTREAM-SOURCE-0000000000    ❹
    Sink: KSTREAM-SINK-0000000002 (topic: out-topic)    ❺
      <-- KSTREAM-MAPVALUES-0000000001
```

❶ The source node name

❷ The processor that the source node sends records to

❸ The name of the map values processor

❹ The processor that provided input the the map values processor

❺ The name of the sink node

From looking at the names, you can see the first node ends in a zero, with the second node `KSTREAM-MAPVALUES` ending in a one etc. The `Sub-topology` listing indicates a portion of the topology that is a distinct source node and every processor downstream of the source node is a member of the given `Sub-topology`. If you were to define a second stream with a new source, then that would show up as `Sub-topology: 1`. We'll see more about sub-topologies a bit later in the book when we cover repartitioning.

The arrows `->` pointing to the right show the flow of records in the topology. The arrows pointing left `<-` indicate the lineage of the record flow, where the current processor received records one. Note that a processor could forward records to more than one node and a single node could get input from multiple nodes.

Looking at this topology description, it's easy to get sense of the structure of the Kafka Streams application. However, once you start building more complex applications, the generic names with the numbers become hard to follow. For this reason, Kafka Streams provides a way to name the processing nodes in the DSL.

Almost all the methods in the Streams DSL have an overload that takes a `Named` object where you can specify the name used for the node in the topology. Being able to provide the name is important as you can make it relate to the processing nodes *role* in your application, not just what the processor *does*. Configuration objects like `Consumed` and `Produced` have a `withName` method for giving a name to the operator. Let's revisit the `KafkaStreamsYellingApplication` but this time we'll add a name for each processor:

Listing 6.20 Updated KafkaStreamsYellingApplication with names

```
builder.stream("src-topic",
               Consumed.with(stringSerde, stringSerde)
                       .withName("Application Input"))   ❶
       .mapValues((key, value) -> value.toUpperCase(),
                  Named.as("Convert to Yelling"))   ❷
       .to("out-topic",
            Produced.with(stringSerde, stringSerde)
                    .withName("Application Output"))   ❸
```

❶ Naming the source node

❷ Giving a name to the mapValues processor

❸ Naming the sink node

And the description from the updated topology with names will now look like this:

Listing 6.21 Full topology description with provided names

```
Topologies:
   Sub-topology: 0
    Source: Application-Input (topics: [src-topic])
      --> Convert-to-Yelling
    Processor: Convert-to-Yelling (stores: [])
      --> Application-Output
      <-- Application-Input
    Sink: Application-Output (topic: out-topic)
      <-- Convert-to-Yelling
```

Now you can view the topology description and get a sense of the role for each processor in the overall application, instead of just what the processor itself does. Naming the processor nodes becomes critical for your application when there is state involved, but we'll get to that in a later chapter.

Next we'll take a look at how you can use dynamic routing for your Kafka Streams application.

6.5.4 Dynamic routing of messages

Say you need to differentiate which department of the store the purchase comes from—housewares, say, or shoes. You can use dynamic routing to accomplish this task on a per-record basis. The `KStream.to()` method has an overload that takes a `TopicNameExtractor` which will dynamically determine the correct Kafka topic name to use. Note that the topics need to exist ahead of time, by default Kafka Streams will not create extracted topic names automatically.

So, if we go back to the branching example each object has a `department` field, so instead of creating a branch we will process these events with everything else and use the `TopicNameExtractor` to determine the topic where we route the events to.

The `TopicNameExtractor` has one method `extract` which you implement to provide the logic for determining the topic name. What you've going to do here is check if the department of the purchase matches one of the special conditions for routing the purchase events to a different topic. If it does match, then return the name of the department for the topic name (knowing they've been created ahead of time). Otherwise return the name of topic where the rest of the purchase events are sent to.

Listing 6.22 Implementing the extract method to determine the topic name based on purchase department

```
@Override
public String extract(String key,
                      Purchase value,
                      RecordContext recordContext) {
    String department = value.getDepartment();
    if (department.equals("coffee")
            || department.equals("electronics")) {   ❶
        return department;
    } else {
        return "purchases";                          ❷
    }
}
```

❶ Checking if the department matches one of special cases

❷ The default case for the topic name

NOTE **The `TopicNameExtractor` interface only has one method to implement, I've chosen to use a concrete class because you can then write a test for it.**

Although the code example here is using the value to determine the topic to use, it could very well use the key or a combination of the key and the value. But the third parameter to the `TopicNameExtractor#extract` method is a `RecordContext` object. Simply stated the `RecordContext` is the context associated with a record in Kafka Streams.

The context contains metadata about the record- the original timestamp of the record, the original offset from Kafka, the topic and partition it was received, and the `Headers`. We discussed headers in the chapter on Kafka clients, so I won't go into the details again here. One of the primary use cases for headers is routing information, and Kafka Streams exposes them via the `ProcessorContext`. Here's one possible example for retrieving the topic name via a `Header`

In this example you'll extract the `Headers` from the record context. You first need to check that the `Headers` are not null, then you proceed to drill down to get the specific routing information. From there you return the name of topic to use based on the value stored in the `Header`. Since `Headers` are optional and may not exist or contain the specific "routing" `Header` you've defined a default value in the `TopicNameExtractor` and return it in the case where the output topic name isn't found.

Listing 6.23 Using information in a Header for dynamically determining the topic name to send a record to

```
public String extract(String key,
                      PurchaseProto.Purchase value,
                      RecordContext recordContext) {

       Headers headers = recordContext.headers();   ❶
      if (headers != null) {
       Iterator<Header> routingHeaderIterator =
         headers.headers("routing").iterator();

       if (routingHeaderIterator.hasNext()) {
            Header routing = routingHeaderIterator.next();   ❷

            return new String(routing.value(),
                              StandardCharsets.UTF_8);   ❸
       }
     }
      return defaultTopicName;   ❹
   }
```

❶ Retrieving the headers from the `RecordContext`

❷ Extracting the specific routing `Header`

❸ Returning the name of the topic to use from the `Header` value

❹ If no routing information found, return a default topic name

Now you've learned about using the Kafka Streams DSL API.

6.6 Summary

- Kafka Streams is a graph of processing nodes called a topology. Each node in the topology is responsible for performing some operation on the key-value records flowing through it. A Kafka Streams applciation is minimally composed of a source node that consumes records from a topic and sink node that produces results back to a Kafka topic. You configure a Kafka Streams application minimally with the application-id and the bootstrap servers configuration. Multiple Kafka Streams applications with the same application-id are logically considered one application.
- You can use the `KStream.mapValues` function to map incoming record values to new values, possibly of a different type. You also learned that these mapping changes shouldn't modify the original objects. Another method, `KStream.map`, performs the same action but can be used to map both the key and the value to something new.
- To selectively process records you can use the `KStream.filter` operation where records that don't match a predicate get dropped. A predicate is a statement that accepts an object as a parameter and returns `true` or `false` depending on whether that object matches a given condition. There is also the `KStream.filterNot` method that does the opposite - it only forwards key-value pairs that ***don't match*** the predicate.
- The `KStream.branch` method uses predicates to split records into new streams when a record matches a given predicate. The processor assigns a record to a stream on the first match and drops unmatched records. Branching is an elegant way of splitting a stream up into multiple streams where each stream can operate independantly. To perform the opposite action there is `KStream.merge` which you can use to merge 2 `KStream` objects into one stream.
- You can modify an existing key or create a new one using the `KStream.selectKey` method.
- For viewing records in the topology you can use either `KStream.print` or `KStream.peek` (by providing a `ForeachAction` that does the acutal printing). `KStream.print` is a terminal operation meaning that you can't chain methods after calling it. `KStream.peek` returns a `KStream` instance and this makes it easier to embed before and after `KStream` methods.
- You can view the generated graph of a Kafka Streams application by using the `Topology.describe` method. All graph nodes in Kafka Streams have auto-generated names by default which can make the graph hard to understand when the application grows in complexity. You can avoid this situation by providing names to each `KStream` method so when you print the graph, you have names describing the role of each node.
- You can route records to different topics by passing a `TopicNameExtractor` as a parameter to the `KStream.to` method. The `TopicNameExtractor` can inspect the key, value, or headers to determine the corect topic name to use for producing records back to Kafka. The topics must be created ahead of time.

7 Streams and state

This chapter covers

- Adding stateful operations to Kafka Streams
- Using state stores in Kafka Streams
- Enriching event streams with joins
- Learning how timestamps drive Kafka Streams

In the last chapter, we dove headfirst into the Kafka Streams DSL and built a processing topology to handle streaming requirements from purchase activity. Although you built a nontrivial processing topology, it was one dimensional in that all transformations and operations were stateless. You considered each transaction in isolation, without any regard to other events occurring at the same time or within certain time boundaries, either before or after the transaction. Also, you only dealt with individual streams, ignoring any possibility of gaining additional insight by joining streams together.

In this chapter, you'll extract the maximum amount of information from the Kafka Streams application. To get this level of information, you'll need to use state. State is nothing more than the ability to recall information you've seen before and connect it to current information. You can utilize state in different ways. We'll look at one example when we explore the stateful operations, such as the accumulation of values, provided by the Kafka Streams DSL.

We'll get to another example of using state when we'll discuss the joining of streams. Joining streams is closely related to the joins performed in database operations, such as joining records from the employee and department tables to generate a report on who staffs which departments in a company.

We'll also define what the state needs to look like and what the requirements are for using state

when we discuss state stores in Kafka Streams. Finally, we'll weigh the importance of timestamps and look at how they can help you work with stateful operations, such as ensuring you only work with events occurring within a given time frame or helping you work with data arriving out of order.

7.1 Stateful vs stateless

Before we go on with examples, let's provide a description of the difference between stateless and stateful. In a stateless operation there is no additional information retrieved, what's present is enough to complete the desired action. On the other hand, a stateful operation is more complex because it involves keeping the state of previous event. A basic example of a stateful operation is an aggregation.

For example, consider this function:

Listing 7.1 Stateless function example

```
public boolean numberIsOnePredicate (Widget widget) {

    return widget.number == 1;
}
```

Here all the `Widget` object contains all the information needed to execute the predicate, there's no need to lookup or store data. Now let's take a look at an example of a stateful function

Listing 7.2 Stateful function example

```
public int count(Widget widget) {

  int widgetCount = hashMap.compute(widget.id,
   (key, value) -> (value == null) ? 1 : value + 1)

  return widgetCount;
}
```

Here in the `count` function, we are computing the total of widgets with the same id. To perform the count we first must look up the current number by id, increment it, and then store the new number. If no number is found, we go ahead and provide an initial value, a 1 in this case.

While this is a trivial example of using state, the principals involved are what matter here. We are using a common identifier across different objects, called a key, to store and retrieve some value type to track a given state that we want to observe. Additionally, we use an initializing function to produce a value when one hasn't been calculated yet for a given key.

These are the core steps we're going to explore and use in this chapter, although it will be far more robust than using the humble `HashMap`!

7.2 Adding stateful operations to Kafka Streams

So the next question is why you need to use state when processing an event stream? The answer is any time you need to track information or progress across related events. For example consider a Kafka Streams application tracking the progress of players in an online poker game. Participants play in rounds and their score from each round is transmitted to a server then reset to zero for the start of the next round. The game server the produces the players score to a topic.

A stateless event stream will give you the opportunity to work with the current score from the latest round. But for tracking their total, you'll need to keep the state of all their previous scores.

This scenario leads us to our first example of a stateful operation in Kafka Streams. For this example we're going to use a reduce. A reduce operation takes multiple values and reduces or merges them into a single result. Let's look at an illustration to help understand how this process works:

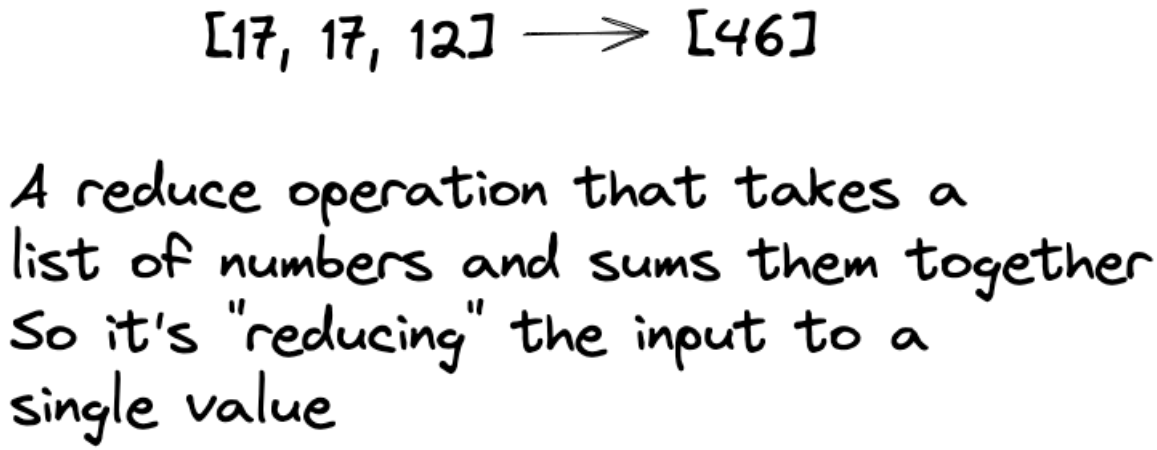


Figure 7.1 A reduce takes several inputs and merges them into a single result of the same type

As you can see in the illustration, the reduce operation takes five numbers and "reduces" them down to a single result by summing the numbers together. So Kafka Streams takes an unbounded stream of scores and continues to sum them per player. At this point we've described the reduce operation itself, but there's some additional information we need to cover regarding how Kafka Streams sets up to perform the reduce.

When describing our online poker game scenario, I mentioned that there are individual players, so it stands to reason that we want to calculate total scores for each ***individual***. But we aren't guaranteed the order of the incoming player scores, so we need the ability to group them. Remember Kafka works with key-value pairs, so we'll assume the incoming records take the form of playerId-score for the key-value pair.

So if the key is the player-id, then all Kafka Streams needs to do is bucket or group the scores by the id and you'll end up with the summed scores per player. It will probably be helpful for us to

view an illustration of the concept:

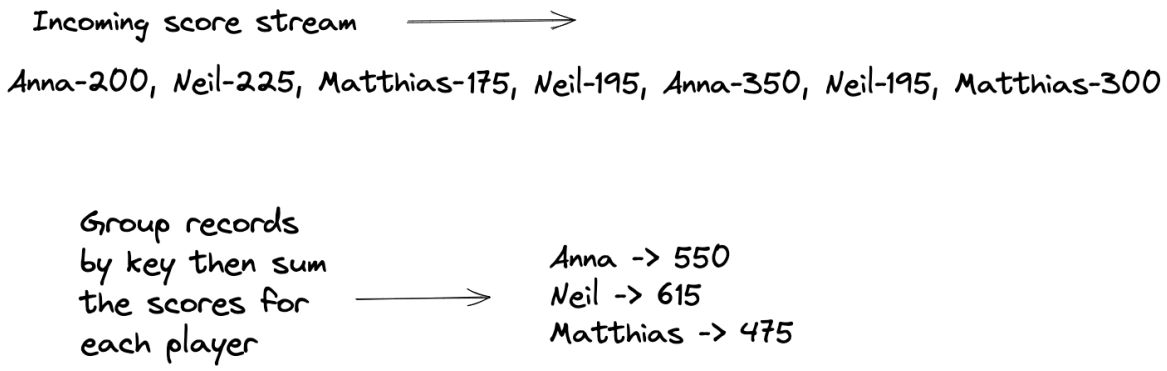


Figure 7.2 Grouping the scores by player-id ensures we only sum the scores for the individual players

So by grouping the scores by player-id, you are guaranteed to only sum the scores for each player. This group-by functionality in Kafka Streams is similar to the SQL group-by when performing aggregation operations on a database table.

> **NOTE** At this point going forward, I'm not going to show the basic setup code needed i.e. creating the `StreamBuilder` instance and serdes for the record types. You've learned in the previous chapter how these components fit into an application, so you can refer back if you need to refresh your memory.

Now let's see the reduce in action with Kafka Streams

Listing 7.3 Performing a reduce in Kafka Streams to show running total of scores in an online poker game

```
KStream<String, Double> pokerScoreStream = builder.stream("poker-game",
        Consumed.with(Serdes.String(), Serdes.Double()));

pokerScoreStream
        .groupByKey()  ❶
        .reduce(Double::sum,    ❷
                Materialized.with(Serdes.String(), Serdes.Double()))
        .toStream()    ❸
        .to("total-scores",
                Produced.with(Serdes.String(), Serdes.Double()));   ❹
```

❶ Grouping by key so that scores are calculated by individual keys

❷ Reducer as a method reference

❸ Converting the KTable to a stream

❹ Writing the results out to a topic

This Kafka Streams application results in key-value pairs like "Neil, 650" and it's a continual stream of summed scores, continually updated.

Looking over the code you can see you first perform a `groupByKey` call. It's important to note that grouping by key is a prerequisite for stateful aggregations in Kafka Streams. So what do you do when there is no key or you need to derive a different one? For the case of using a different key, the `KStream` interface provides a `groupBy` method that accepts a `KeyValueMapper` parameter that you use to select a new key. We'll see an example of selecting a new key in the next example.

7.2.1 Group By details

We should take a quick detour to briefly discuss the return type of the group-by call, which is a `KGroupedStream`. The `KGroupedStream` is an intermediate object and it provides methods `aggregate`, `count`, and `reduce`. In most cases, you won't need to keep a reference to the `KGroupedStream`, you'll simply execute the method you need and its existence is transparent to you.

What are the cases when you'd want to keep a reference to the `KGroupedStream`? Any time you want to perform multiple aggregation operations from the same key grouping is a good example. We'll see one when we cover windowing later on. Now let's get back to the discussion of our first stateful operation.

Immediately after the `groupByKey` call we execute `reduce`, and as I've explained before the `KGroupedStream` object is transparent to us in this case. The `reduce` method has overloads taking anywhere from one to three parameters, in this case we're using the two parameter version which accepts a `Reducer` interface and a `Materialized` configuration object as parameters. For the `Reducer` you're using a method reference to the static method `Double.sum` which sums the previous total score with the newest score from the game.

The `Materialized` object provides the serdes used by the state store for (de)serializing keys and values. Under the covers, Kafka Streams uses local storage to support stateful operations. The stores store key-value pairs as byte arrays, so you need to provide the serdes to serialize records on input and deserialize them on retrieval. We'll get into the details of state stores in an upcoming section.

After `reduce` you call `toStream` because the result of all aggregation operations in Kafka Streams is a `KTable` object (which we haven't covered yet, but we will in the next chapter), and to forward the aggregation results to downstream operators we need to convert it to a `KStream`.

Then we can send the aggregation results to an output topic via a sink node represented by the `to` operator. But stateful processors don't have the same forwarding behavior as stateless ones, so we'll take a minute here to describe that difference.

Kafka Streams provides a caching mechanism for the results of stateful operations. Only when Kafka Streams flushes the cache are stateful results forwarded to downstream nodes in the

topology. There are two scenarios when Kafka Streams will flush the cache. The first when the cache is full, which by default is 10MB, or secondly when Kafka Streams commits, which is every thirty seconds with default settings. An illustration of this will help to cement your understanding of how the caching works in Kafka Streams.

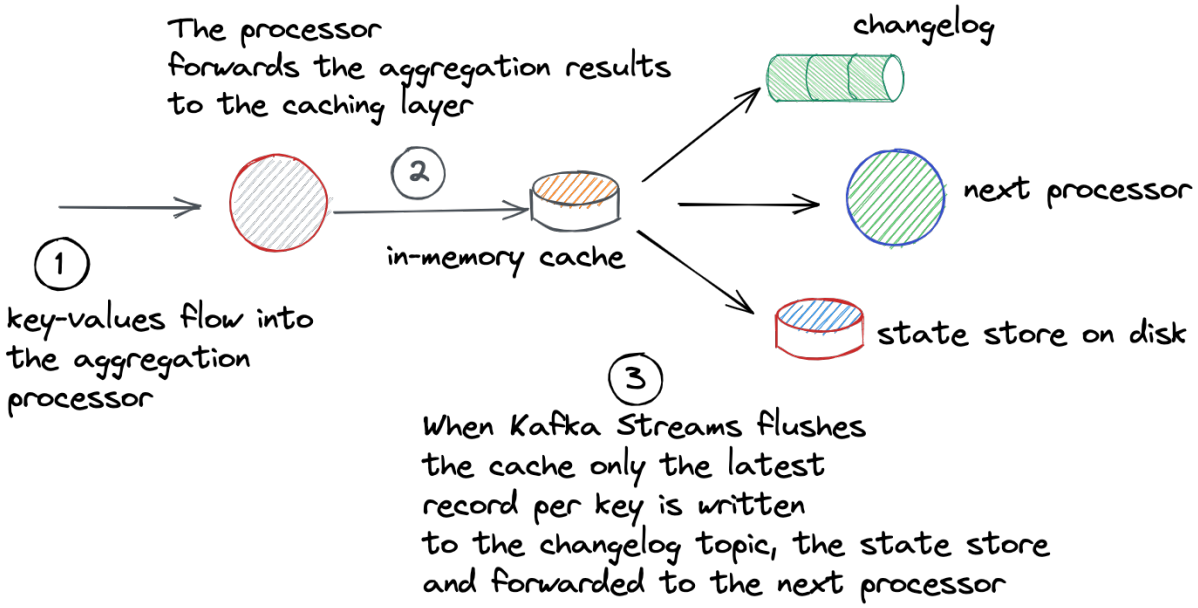


Figure 7.3 Caching intermediate results of an aggregation operation

So from looking at the illustration you can see that the cache sits in front forwarding records and as a result you won't observe several of the intermediate results, but you will always see the latest updates at the time of a cache flush. This also has the effect of limiting writes to the state store and its associated changelog topic. Changelog topics are internal topics created by Kafka Streams for fault tolerance of the state stores. We'll cover changelog topics in an upcoming section.

TIP **If you want to observe every result of a stateful operation you can disable the cache by setting the `StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG` setting to 0.**

7.2.2 Aggregation vs. reducing

At this point you've learned about one stateful operator, but we have another option for stateful operations. If you noticed with `reduce`, since you are merging record values, it's expected that a `reduce` returns the same type as a result. But sometimes you'll want to build a different result type and for that you'll want to use the `aggregate` operation. The concept behind an aggregation is similar, but you have the flexibility to return a type different from the record value. Let's look at an example to answer why you would use `aggregate` over `reduce`.

Imagine you work for ETrade you need to create an application that tracks stock transactions of individual customers, not large institutional traders. You want to keep a running tally of the total volume of shares bought and sold, the dollar volume of sales and purchases, and the highest and lowest price seen at any point.

To provide this type of information, you'll need to create a custom data object. This where the `aggregate` comes into play, because it allows for a different return type from the incoming value. In this case the incoming record type is singular stock transaction object, but the aggregation result will be a different type containing the required information listed in the previous paragraph.

Since we'll need to put this custom object in a state store which requires serialization, we'll create a Protobuf schema so we can generate it and leverage utility methods for creating Protobuf serdes . Since this application has detailed aggregation requirements, we'll implement the `Aggregator<K, V, VR>` interface as a concrete class which will allow us to test it independently.

Let's take a look at part of the aggregator implementation. Since this class contains some logic not directly related to learning Kafka Streams, I'm only going to show partial information on the class, to view full details, consult the source code and look for the `bbejeck.chapter_7.aggregator.StockAggregator` class.

Listing 7.4 Aggregator implementation used for creating stock transaction summaries

```
public class StockAggregator implements Aggregator<String,
                                          Transaction,
                                          Aggregate> {

    @Override
    public Aggregate apply(String key,
                           Transaction transaction,
                           Aggregate aggregate) {   ❶

   Aggregate.Builder currAggregate =
                                aggregate.toBuilder();   ❷

    double transactionDollars =
           transaction.getNumberShares()
         * transaction.getSharePrice();   ❸

    if (transaction.getIsPurchase()) {        ❹
        long currentPurchaseVolume =
             currAggregate.getPurchaseShareVolume();
        currAggregate.setPurchaseShareVolume(
                      currentPurchaseVolume
                     + transaction.getNumberShares());

        double currentPurchaseDollars =
                currAggregate.getPurchaseDollarAmount();

        currAggregate.setPurchaseDollarAmount(
                      currentPurchaseDollars
                      + transactionDollars);
    }
    //Further details left out for clarity
```

❶ Implementation of the apply method the second parameter is the incoming record, third parameter is the current aggregate

❷ Need to use a builder to update Protobuf object

❸ Getting the total dollars of the transaction

❹ If the transaction is a purchase update the purchase related details

I'm not going to go into much detail about the `Aggregator` instance here, since the main point of this section how to build a Kafka Streams aggregation application, the particulars of how you implement the aggregation is going to vary from case to case. But from looking at this code, you can see how we're building up the transactional data for a given stock. Now let's look at how we'll plug this `Aggregator` implementation into a Kafka Streams application to capture the information. The source code for this example can be found in bbejeck.chapter_7.StreamsStockTransactionAggregations

NOTE **There's some details I'm going to leave out of the source code as presented in the book, printing records to the console for example. Going forward our Kafka Streams applications will get more complex and it will be easier to learn the essential part of the lesson if I only show the key details. Rest assured the source code is complete and will be thoroughly tested to ensure that the examples compile and run.**

Listing 7.5 Kafka Streams aggregation

```
KStream<String, Transaction> transactionKStream =
   builder.stream("stock-transactions",
                  Consumed.with(stringSerde, txnSerde));   ❶


transactionKStream.groupBy((key, value) -> value.getSymbol(),   ❷
      Grouped.with(Serdes.String(), txnSerde))
  .aggregate(() -> initialAggregate,   ❸
           new StockAggregator(),
           Materialized.with(stringSerde, aggregateSerde))
  .toStream()   ❹
  .peek((key, value) -> LOG.info("Aggregation result {}", value))
  .to("stock-aggregations", Produced.with(stringSerde, aggregateSerde));   ❺
```

❶ Creating the `KStream` instance

❷ Grouping by key and providing a function to select the key

❸ Calling the aggregate function

❹ Converting the resulting aggregation `KTable` to a `KStream`

❺ Writing the aggregation results out to a topic

In annotation one, this application starts out in familiar territory, that is creating the `KStream` instance by subscribing it to a topic and providing the serdes for deserialization. I want to call

your attention to annotation two, as this is something new.

You've seen a group-by call in the reduce example, but in this example the inbound records are keyed by the client-id and we need to group records by the stock ticker or symbol. So to accomplish the key change, you use `GroupBy` which takes a `KeyValueMapper`, which is a lambda function in our code example. In this case the lambda returns the ticker symbol in the record to enable to proper grouping.

Since the topology changes the key, Kafka Streams needs to repartition the data. I'll discuss repartitioning in more detail in the next section, but for now it's enough to know that Kafka Streams takes care of it for you.

Listing 7.6 Kafka Streams aggregation

```
transactionKStream.groupBy((key, value) -> value.getSymbol(),
      Grouped.with(Serdes.String(), txnSerde))
  .aggregate(() -> initialAggregate,   ❶
             new StockAggregator(),
             Materialized.with(stringSerde, aggregateSerde))
  .toStream()   ❷
  .peek((key, value) -> LOG.info("Aggregation result {}", value))
  .to("stock-aggregations", Produced.with(stringSerde, aggregateSerde));   ❸
```

❶ Calling the aggregate function

❷ Converting the resulting aggregation `KTable` to a `KStream`

❸ Writing the aggregation results out to a topic

At annotation three is where we get to the crux of our example, applying the aggregation operation. Aggregations are little different from the reduce operation in that you need to supply an initial value.

Providing an initial value is required, because you need an existing value to apply for the first aggregation as the result could possibly be a new type. With the reduce, if there's no existing value, it simply uses the first one it encounters.

Since there's no way for Kafka Streams to know what the aggregation will create, you need to give it an initial value to seed it. In our case here it's an instantiated `StockAggregateProto.Aggregate` object, with all the fields uninitialized.

The second parameter you provide is the `Aggregator` implementation, which contains your logic to build up the aggregation as it is applied to each record it encounters. The third parameter, which is optional, is a `Materialized` object which you're using here to supply the serdes required by the state store.

The final parts of the application are used to covert the `KTable` resulting from the aggregation to a `KStream` so that you can forward the aggregation results to a topic. Here you're also using a

peek operate before the sink processor to view results without consuming from a topic. Using a peek operator this way is typically for development or debugging purposes only.

> **NOTE** **Remember when running the examples that Kafka Streams uses caching so you won't immediately observe results until the cache gets flushed either because it's full or Kafka Streams executes a commit.**

So at this point you've learned about the primary tools for stateful operations in the Kafka Streams DSL, reduce and aggregation. There's another stateful operation that deserves mention here and that is the count operation. The count operation is a convenience method for a incrementing counter aggregation. You'd use the count when you simply need a running tally of a total, say the number of times a user has logged into your site or the total number of readings from an IoT sensor. We won't show an example here, but you can see one in action in the source code at bbejeck/chapter_7/StreamsCountingApplication.

In this previous example here where we built stock transaction aggregates, I mentioned that changing the key for an aggregation requires a repartitioning of the data, let's discuss this in a little more detail in the next section.

7.2.3 Repartitioning the data

In the aggregation example we saw how changing the key required a repartition. Let's have a more detailed conversation on why Kafka Streams repartition and how it works. Let's talk about the why first.

We learned in a previous chapter that the key of a Kafka record determines the partition. When you modify or change the key, there's a strong probability it belongs on another partition. So, if you've changed the key and you have a processor that depends on it, an aggregation for example, Kafka Streams will repartition the data so the records with the new key end up on the correct partition. Let's look at an illustration demonstrating this process in action:

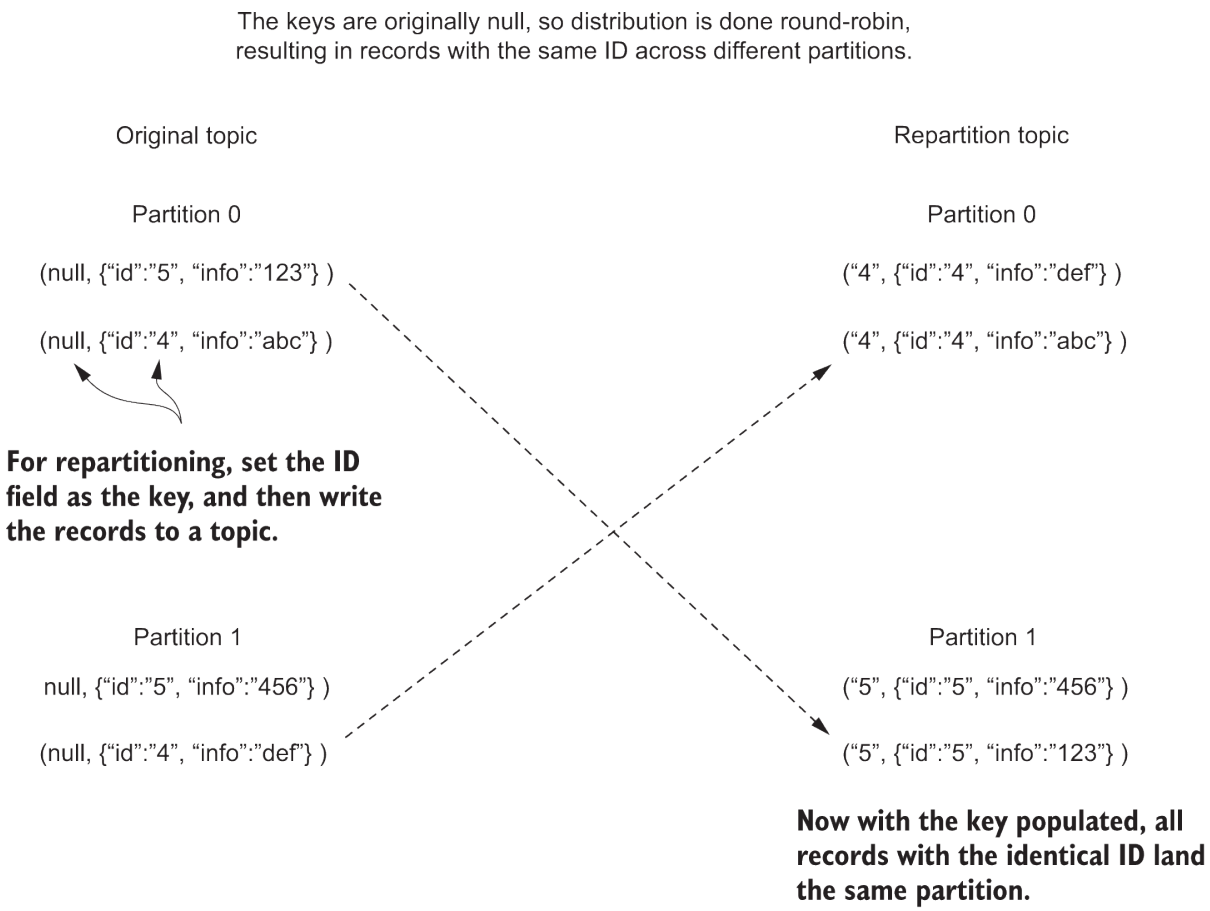


Figure 7.4 Repartitioning: changing the original key to move records to a different partition

As you can see here, repartitioning is nothing more than producing records out to a topic and then immediately consuming them again. When the Kafka Streams embedded producer writes the records to the broker, it uses the updated key to select the new partition. Under the covers, Kafka Streams inserts a new sink node for producing the records and a new source node for consuming them, here’s an illustration showing the before and after state where Kafka Streams updated the topology:

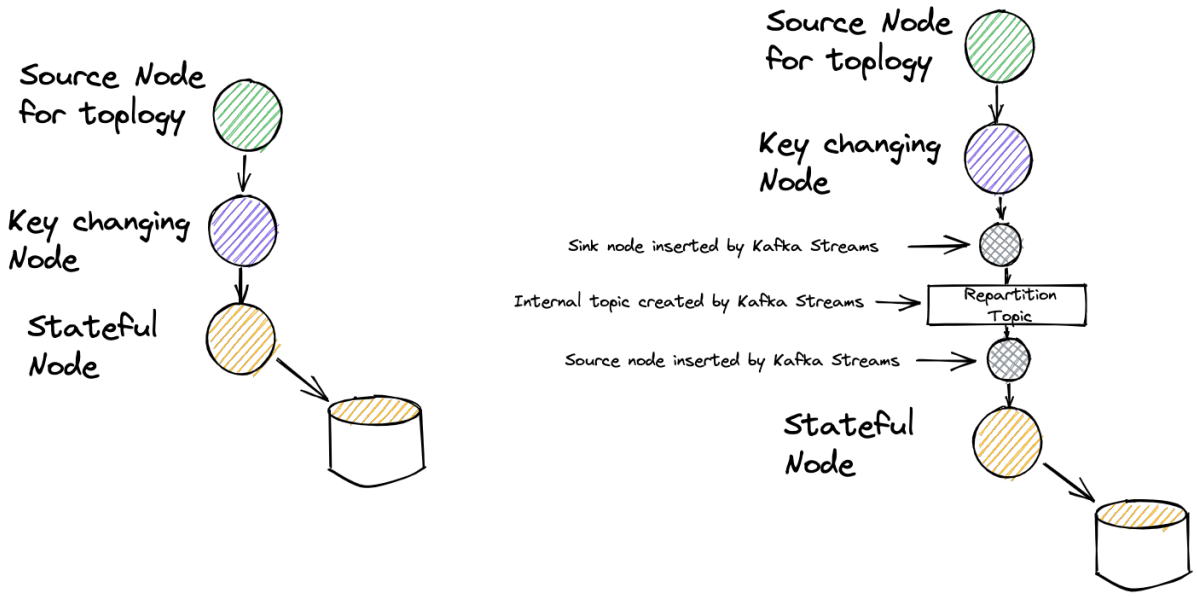


Figure 7.5 Updated topology where Kafka Streams adds a sink and source node for repartitioning of the data

The newly added source node creates a new sub-topology in the overall topology for the application. A sub-topology is a portion of a topology that share a common source node. Here's an updated version of the repartitioned topology demonstrating the sub-topology structures:

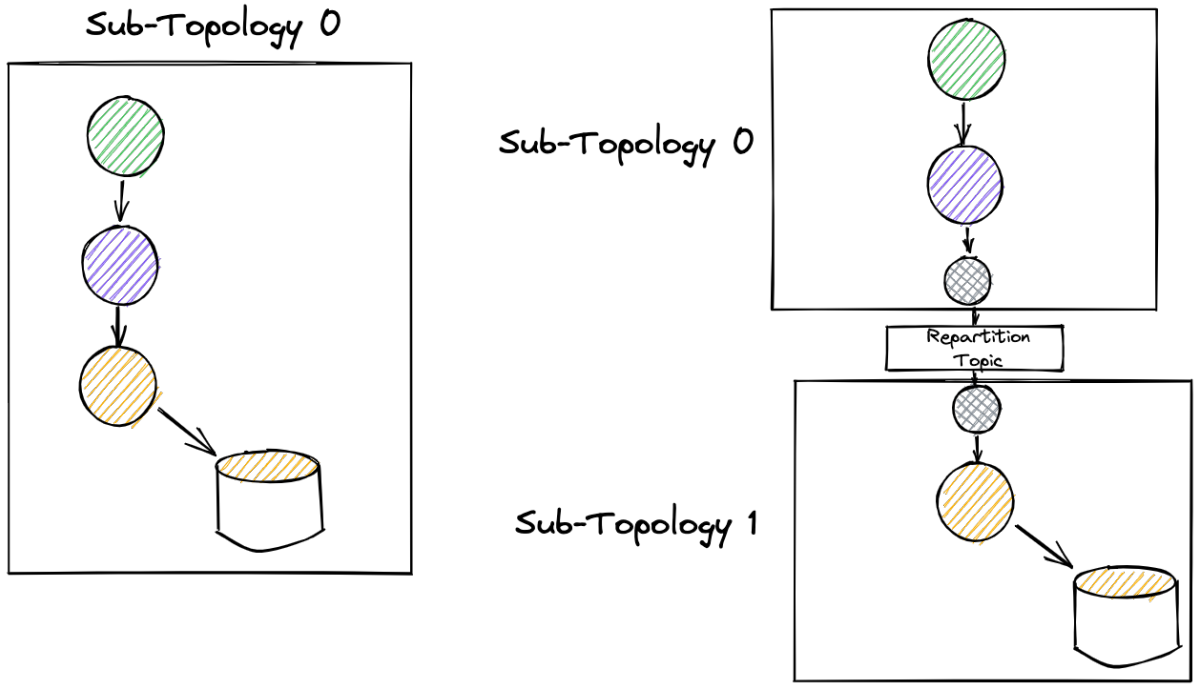


Figure 7.6 Adding a sink and source node for repartitioning creates a new sub-topology

So any processors that come after the new source node are part of the new sub-topology.

What is the determining factor that causes Kafka Streams to repartition? If you have a key

changing operation ***and*** there's a downstream operation that relies on the key, such as a `groupByKey`, aggregation, or join (we'll get to joins soon). Otherwise, if there are no downstream operations dependent on the key, Kafka Streams will leave the topology as is. Let's look a couple of examples to help clarify this point:

Listing 7.7 Examples of when repartitioning is needed

```
myStream.groupBy(...).reduce(...)...   ❶

myStream.map(...).groupByKey().reduce(...)...   ❷

filteredStream = myStream.selectKey(...).filter(...);   ❸
....
filteredStreaam.groupByKey().aggregate(...)...   ❸
```

❶ Using `groupBy` followed by a `reduce`

❷ Executing a `map` followed by a `groupByKey`

❸ Using a `selectKey` to choose a new key and the resulting `KStream` later calls `groupByKey`

What these code examples demonstrate is when you execute an operation where you ***could*** change the key, Kafka Streams sets an internal boolean flag, `repartitionRequired`, to `true`. Since Kafka Streams can't possibly know if you changed the key or not, when it finds an operation dependent on the key and the internal flag evaluates to `true`, it will automatically repartition the data.

On the other hand, even if you change the key, but don't do an aggregation or join, the topology remains the same:

Listing 7.8 Examples of when repartitioning is not needed

```
myStream.map(...).peek(...).to(...);   ❶

myStream.selectKey(...).filter(...).to(...);   ❷
```

❶ Using a map but no downstream operation depends on the key

❷ Using a selectKey but also no downstream operations rely on the key

In these examples, even if you updated the key, it doesn't affect the results of the downstream operators. For example filtering a record solely depends on if the predicate evaluates to `true` or not. Additionally, since these `KStream` instances write out to a topic, the records with updated keys will end up on the correct partition.

So the bottom line is to only use key-changing operations (`map`, `flatMap`, `transform`) when you actually need to change the key. Otherwise it's best to use processors that only work on values i.e. `mapValues`, `flatMapValues` etc. this way Kafka Streams won't needlessly repartition the

data. There are overloads to XXXValues methods that provide **access** to the key when updating a value, but changing the key in this case will lead to undefined behavior.

> **NOTE** **The same is true when grouping records before an aggregation. Only use `groupBy` when you need to change the key, otherwise favor `groupByKey`.**

It's not that you should avoid repartitioning, but since it adds processing overhead it is a good idea to only do it when required.

Before we wrap up coverage of repartitioning we should talk about an important additional subject; inadvertently creating redundant repartition nodes and ways to prevent it. Let's say you have an application with two input streams. You need to perform an aggregation on the first stream as well as join it with the second stream. Your code would look something like this:

Listing 7.9 Changing the key then aggregate and join

```
// Several details omitted for clarity

KStream<String, String> originalStreamOne = builder.stream(...);

KStream<String, String> inputStreamOne = originalStreamOne.selectKey(...);   ❶

KStream<String, String> inputStreamTwo = builder.stream(...);   ❷

inputStreamOne.groupByKey().count().toStream().to(...);   ❸

KStream<String, String> joinedStream =          ❹
  inputStreamTwo.join(inputStreamOne,
                (v1, v2)-> v1+":"+v2,
                JoinWindows.ofTimeDifferenceWithNoGrace(...),
                StreamJoined.with(...);

....
```

❶ Changing the key of the original stream setting the "needsRepartition" flag

❷ The second stream

❸ Performing a group-by-key triggering a repartition

❹ Performing a join between inputStreamOne and inputStreamTwo triggering another repartition

This code example here is simple enough. You take the `originalStreamOne` and need you to change the key since you'll need to do an aggregation and a join with it. So you use a `selectKey` operation, which sets the `repartitionRequired` flag for the returned `KStream`. Then you perform a `count()` and then a `join` with `inputStreamOne`. What is not obvious here is that Kafka Streams will automatically create two repartition topics, one for the `groupByKey` operator and the other for the `join`, but in reality you only need one repartition.

It will help to fully understand what's going on here by looking at the topology for this example.

Notice there are two reparititions, but you only need the first one where the key is changed.

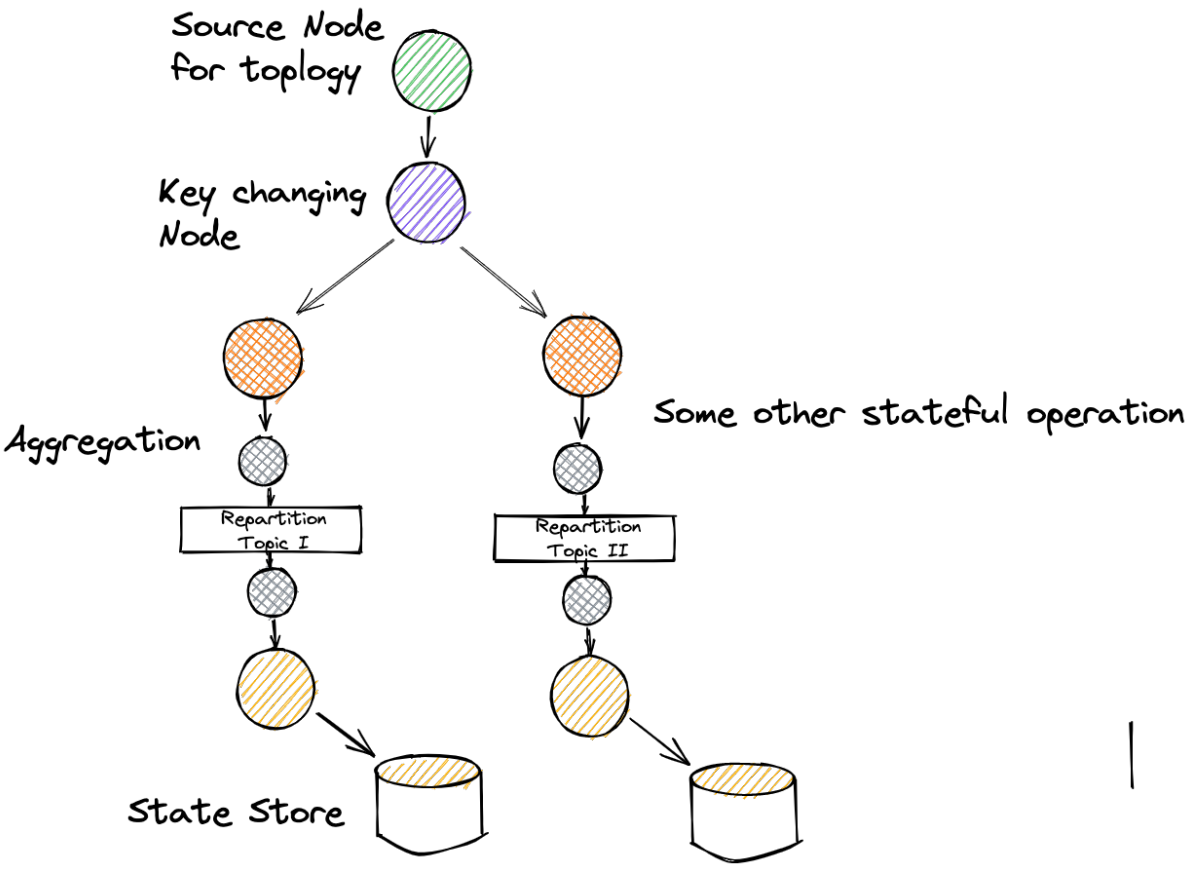


Figure 7.7 Redundant repartition nodes due to a key changing operation occurring previously in the topology

When you used the key-changing operation on `originalStreamOne` the resulting `KStream`, `inputStreamOne`, now carries the `repartitionRequired = true` setting. So any `KStream` resulting from `inputStreamOne` that uses a processor involving the key will trigger a repartition.

What can you do to prevent this from happening? There are two choices here; manually repartition earlier which sets the repartition flag to `false`, so any subsequent streams won't trigger a repartition. The other option is let Kafka Streams handle it for you by enabling optimizations. Let's talk about using the manual approach first.

> **NOTE** **While repartition topics do take up disk space, Kafka Streams actively purges records from them, so you don't need to be concerned with the size on disk, but avoiding redundant repartitions is still a good idea.**

7.2.4 Proactive Repartitioning

For the times when you might need to repartition the data yourself, the KStream API provides the `repartition` method. Here's how you use it to manually repartition after a key change:

Listing 7.10 Changing the key, repartitioning then performing an aggregation and a join

```
//Details left out of this example for clarity

KStream<String, String> originalStreamOne = builder.stream(...);
KStream<String, String> inputStreamOne = originalStreamOne.selectKey(...);   ❶

KStream<String, String> inputStreamTwo = builder.stream(...);

KStream<String, String> repartitioned =
  inputStreamOne.repartition(Repartitioned     ❷
                .with(stringSerde, stringSerde)
                .withName("proactive-repartition"));

repartitioned.groupByKey().count().toStream().to(...);   ❸

KStream<String, String> joinedStream = inputStreamTwo.join(...)   ❹

.....
```

❶ Changing the key setting the "needs repartition" flag

❷ Calling the repartition method and providing key-value serdes and a name for the repartition topic

❸ Performing an aggregation on the repartitioned stream

❹ Performing a join with the repartitioned stream.

The code here has only one change, adding `repartition` operation before performing the `groupByKey`. What happens as a result is Kafka Streams creates a new sink-source node combination that results in a new subtopology. Let's take a look at the topology now and you'll see the difference compared to the previous one:

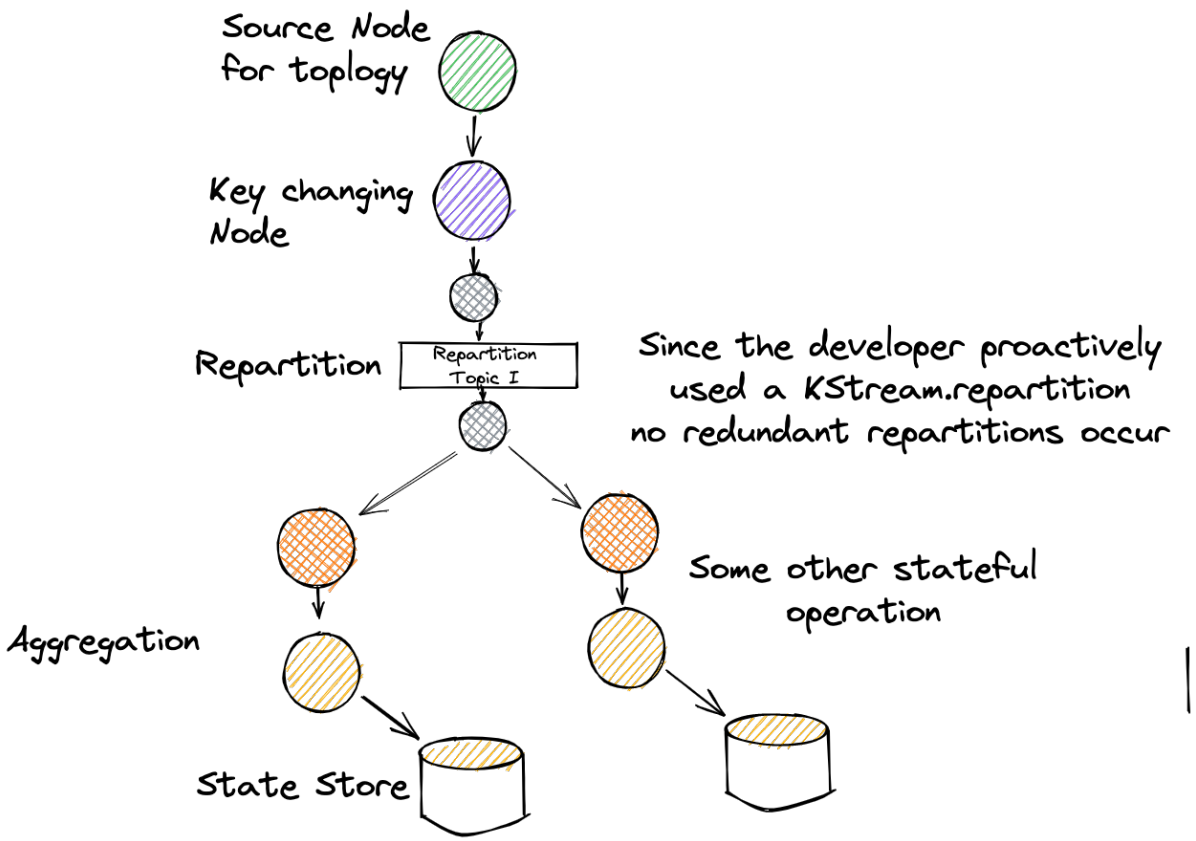


Figure 7.8 Now only one repartition node due to a proactive repartition also allows for more stateful operations without repartitioning

This new sub-topology ensures that the new keys end up on the correct partition, and equally as important, the returned `KStream` object has the `needsRepartition` flag set to `false`. As a result, all downstream stateful operations that are descendants of this `KStream` object don't trigger any further repartitions (unless the one of them changes the key again).

The `KStream.repartition` method accepts one parameter, the `Repartitioned` configuration object. `Repartitioned` allows you to specify:

1. The serdes for the key and value
2. The base name for the topic
3. The number of partitions to use for the topic
4. A `StreamPartitioner` instance should you need customize the distribution of records to partitions

Let's pause on our current discussion and review some of these options. Since I've already covered serdes and the `StreamPartitioner` in the previous chapter, I'm going to leave them out here.

Providing a base-name for the repartition topic is always a good idea. I'm using the term base-name because Kafka Streams takes the name you provide and adds a prefix of "<application-id>-" which comes from the value you supplied in the configs and a suffix of

"-repartition".

So given an application-id of "streams-financial" and a name of "stock-aggregation" results in a repartition topic named "streams-financial-stock-aggregation-repartition". The reason it's a good idea to always provide a name is two fold. First having a meaningful topic name is always helpful to understand its role when you list the topics on your Kafka cluster.

Secondly, and probably more important, is the name you provide remains fixed even if you change you topology upstream of the repartition. Remember, when you don't provide names for processors, Kafka Streams generates names for them, and part of the name includes a zero padded number generated by a global counter.

So if you add or remove operators upstream of your repartition operation and you ***haven't*** explicitly named it, its name will change due to changes in global counter. This name shift can be problematic when re-deploying an existing application. I'll talk more about importance of naming stateful components of a Kafka Streams application in an upcoming section.

NOTE	**Although there are four parameters for the `Repartitioned` object, you don't have to supply all of them. You can use any combination of the parameters that suit your needs.**

Specifying the number of partitions for the repartition topic is particularly useful in two cases: co-partitioning for joins and increasing the number of tasks to enable higher throughput. Let's discuss the co-partitioning requirement first. When performing joins, both sides must have the same number of partitions (we'll discuss why this is so in the upcoming joins section). So by using the `repartition` operation, you can change the number partitions to enable a join, without needing to change the original source topic, keeping the changes internal to the application.

7.2.5 Repartitioning to increase the number of tasks

If you recall from the previous chapter, the number of partitions drive the number of tasks which ultimately determines the amount of active threads a application can have. So one way to increase the processing power is to increase the number of partitions, since that leads to more tasks and ultimate more threads that can process records. Keep in mind that tasks are evenly assigned to all applications with the same id, so this approach to increase throughput is particularly useful in an environment where you can elastically expand the number of running instances.

While you could increase the number of partitions for the source topic, this action might not always be possible. The source topic(s) of a Kafka Streams application are typically "public" meaning other developers and applications use that topic and in most organizations, changes to shared infrastructure resources can be difficult to get done.

Let's look at an example of performing a repartition to increase the number of tasks (example found in bbejeck.chapter_7.RepartitionForThroughput)

Listing 7.11 Increasing the number of partitions for a higher task count

```
KStream<String, String> repartitioned =

initialStream.repartition(Repartitioned
            .with(stringSerde, stringSerde)
            .withName("multiple-aggregation")
            .withNumberOfPartitions(10));   ❶
```

❶ Increasing the number of partitions

Now this application will have 10 tasks which means there can up to 10 stream threads processing records driven by the increase in the number of partitions.

You need to keep in mind however, that adding partitions for increased throughput will work best when there is a fairly even distribution of keys. For example, if seventy percent of you key space lands on one partition, increasing the number of partitions will only move those keys to a new partition. But since the overall **distribution** of the keys is relatively unchanged, you won't see any gains in throughput, since one partition, hence one task, is shouldering most of the processing burden.

So far we've covered how you can proactively repartition when you've changed the key. But this requires you to know when to repartition and always remember to do so, but is there a better approach, maybe have Kafka Streams take care of this for you automatically? Well there is a way, by enabling optimizations.

7.2.6 Using Kafka Streams Optimizations

While you're busy creating a topology with various methods, Kafka Streams builds a graph or internal representation of it under the covers. You can also consider the graph to be a "logical representation" of your Kafka Streams application. In your code, when you execute `StreamBuilder#build` method, Kafka Streams traverses the graph and builds the final or physical representation of the application.

At a high level, it works like this: as you apply each method, Kafka Streams adds a "node" to the graph as depicted in the following illustration:

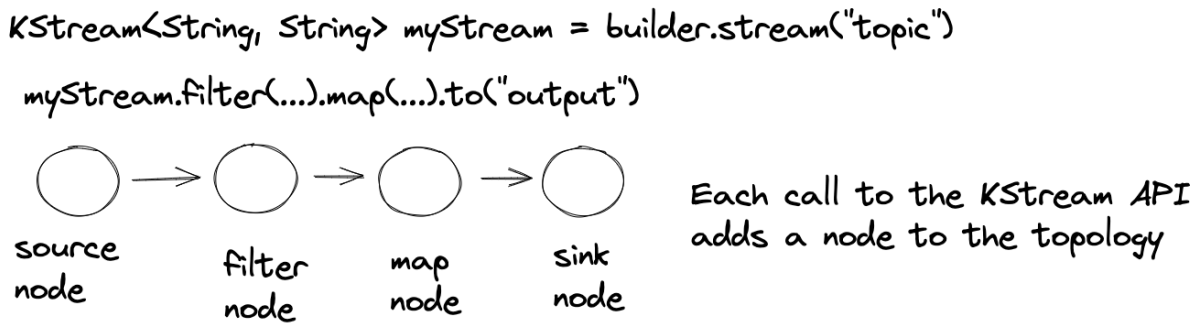


Figure 7.9 Each call to a KStream method adds a node to the graph

When you make an additional method call, the previous "node" becomes the parent of the current one. This process continues until you finish building your application.

Along the way, Kafka Streams will record metadata about the graph it's building, specifically it records if it has encountered a repartition node. Then when use the `StreamsBuilder#build` method to create the final topology, Kafka Streams will examine the graph for redundant repartition nodes, and if found, it will re-write your topology to have only one! This is opt-in behavior for Kafka Streams, so to get this feature working, you'll need to enable optimizations by doing the following:

Listing 7.12 Enabling optimizations in Kafka Streams

```
streamProperties.put(StreamsConfig.TOPOLOGY_OPTIMIZATION_CONFIG,
                     StreamsConfig.OPTIMIZE);     ❶
builder.build(streamProperties);    ❷
```

❶ Enabling optimizations via configuration

❷ Passing properties to the StreamBuilder

So to enable optimizations you need to first set the proper configuration because by default it is turned off. The second step is to pass the properties object to the `StreamBuilder#build` method. Then Kafka Streams will optimize your repartition nodes when building the topology.

NOTE **If you have more than one key-changing operation with a stateful one further downstream the optimizing will not remove that repartition. It only takes away redundant repartitions for single key-changing processor.**

But when you enable optimizations Kafka Streams automatically updates the topology by removing the three repartition nodes preceding the aggregation and inserts a new single repartition node immediately after the key-changing operation which results in a topology that looks like the illustration in the "Proactive Repartitioning" section.

So with a configuration setting and passing the properties to the `StreamBuilder` you can

automatically remove any unnecessary repartitions! The decision on which one to use really comes down to personal preference, but by enabling optimizations it guards against you overlooking where you may need it.

Now we've covered repartitioning, let's move on to our next stateful operation, joins

7.3 Stream-Stream Joins

Sometimes you may need to combine records from different event streams to "complete the picture" of what your application is tasked with completing. Say we have stream of purchases with the customer ID as the key and a stream of user clicks and we want to join them so we can make connection between pages visted and purchases. To do this in Kafka Streams you use a join operation. Many of you readers are already familiar with the concept of a join from SQL and the relational database world and the concept is the same in Kafka Streams.

Let's look at an illustration to demonstrate the concept of joins in Kafka Streams

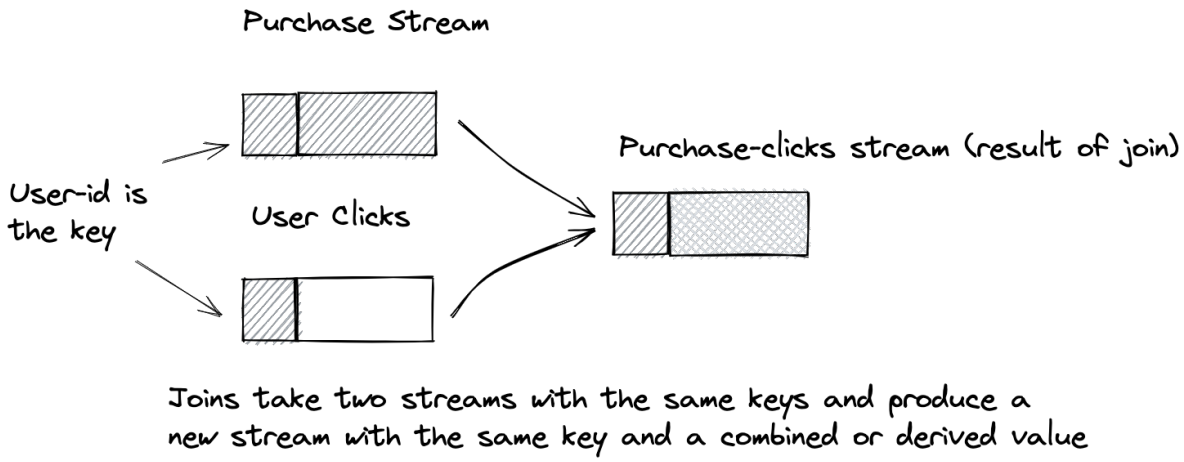


Figure 7.10 Two Streams with the same keys but different values

From looking at the graphic above, there are two event streams that use the same values for the key, a customer id for example, but the values are different. In one stream the values are purchases and the other stream the values are links to pages the user clicked visiting the site.

> **IMPORTANT** **Since joins depend on identical keys from different topics residing on the same partition, the same rules apply when it comes to using a key-changing operation. If a `KStream` instance is flagged with `repartitionRequired`, Kafka Streams will partition it before the join operation. So all the information in the repartitioning section of this chapter applies to joins as well.**

In this section, you'll take different events from two streams with the same key, and combine them to form a new event. The best way to learn about joining streams is to look at a concrete

example, so we'll return to the world of retail. Consider a big box retailer that sells just about anything you can imagine. In an never ending effort to lure more customers in the store, the retailer partners with a national coffee house and it embeds a cafe in each store.

To encourage customers to come into the store, the retailer has started a special promotion where if you are a member of the customer-club and you buy a coffee drink from the embedded cafe and a purchase in the store (in either order), they'll automatically earn loyalty points at the completion of your second purchase. The customers can redeem those points for items from either store. It's been determined that purchases must made within 30 minutes of each other to qualify for the promotion.

Since the main store and the cafe run on separate computing infrastructure, the purchase records are in two event streams, but that's not an issue as they both use the customer id from the club membership for the key, so it's a simply a case of using a stream-stream join to complete the task.

7.3.1 Implementing a stream-stream join

The next step is to perform the actual join. So let's show the code for the join, and since there are a couple of components that make up the join, I'll explain them in a section following the code example. The source code for this example can be found in src/main/java/bbejeck/chapter_7/KafkaStreamsJoinsApp.java).

Listing 7.13 Using the `join()` method to combine two streams into one new stream based on keys of both streams

```
// Details left out for clarity

KStream<String, CoffeePurchase>
                    coffeePurchaseKStream = builder.stream(...)   ❶

KStream<String, RetailPurchase>
                    retailPurchaseKStream = builder.stream(...)   ❶

ValueJoiner<CoffeePurchase,
            RetailPurchase,
            Promotion> purchaseJoiner =
                                        new PromotionJoiner();   ❷

JoinWindows thirtyMinuteWindow =
     JoinWindows.ofTimeDifferenceWithNoGrace(Duration.minutes(30));   ❸

KStream<String, Promotion> joinedKStream =
    coffeePurchaseKStream.join(retailPurchaseKStream,   ❹
                               purchaseJoiner,
                               thirtyMinuteWindow,
                                StreamJoined.with(stringSerde,   ❺
                                                  coffeeSerde,
                                                  storeSerde)
                                      .withName("purchase-join")
                                      .withStoreName("join-stores"));
```

❶ The streams you will join

❷ ValueJoiner instance which produces the joined result object

❸ JoinWindow specifying the max time difference between records to participate in join

❹ Constructs the join

❺ StreamJoined configuration object

You supply four parameters to the `KStream.join` method:

- `retailPurchaseKStream` — The stream of purchases from to join with.
- `purchaseJoiner` — An implementation of the `ValueJoiner<V1, V2, R>` interface. `ValueJoiner` accepts two values (not necessarily of the same type). The `ValueJoiner.apply` method performs the implementation-specific logic and returns a (possibly new) object of type R. In this example, `purchaseJoiner` will add some relevant information from both `Purchase` objects, and it will return a `PromotionProto` object.
- `thirtyMinuteWindow` — A `JoinWindows` instance. The `JoinWindows.ofTimeDifferenceWithNoGrace` method specifies a maximum time difference between the two values to be included in the join. Specifically the timestamp on the secondary stream, `retailPurchaseKStream` can only be a maximum of 30 minutes before or after the timestamp of a record from the `coffeePurchaseKStream` with the same key.
- A `StreamJoined` instance — Provides optional parameters for performing joins. In this case, it's the key and the value `Serde` for the calling stream, and the value `Serde` for the secondary stream. You only have one key `Serde` because, when joining records, keys must be of the same type. The `withName` method provides the name for the node in the topology and the base name for a repartition topic (if required). The `withStoreName` is the base name for the state stores used for the join. I'll cover join state stores usage in an upcoming section.

> **NOTE** **`Serde` objects are required for joins because join participants are materialized in windowed state stores. You provide only one `Serde` for the key, because both sides of the join must have a key of the same type.**

Joins in Kafka Streams are one of the most powerful operations you can perform and it's also one the more complex ones to understand. Let's take a minute to dive into the internals of how joins work.

7.3.2 Join internals

Under the covers, the KStream DSL API does a lot of heavy lifting to make joins operational. But it will be helpful for you to understand how joins are done under the covers. For each side of the join, Kafka Streams creates a join processor with its own state store. Here's an illustration showing how this looks conceptually:

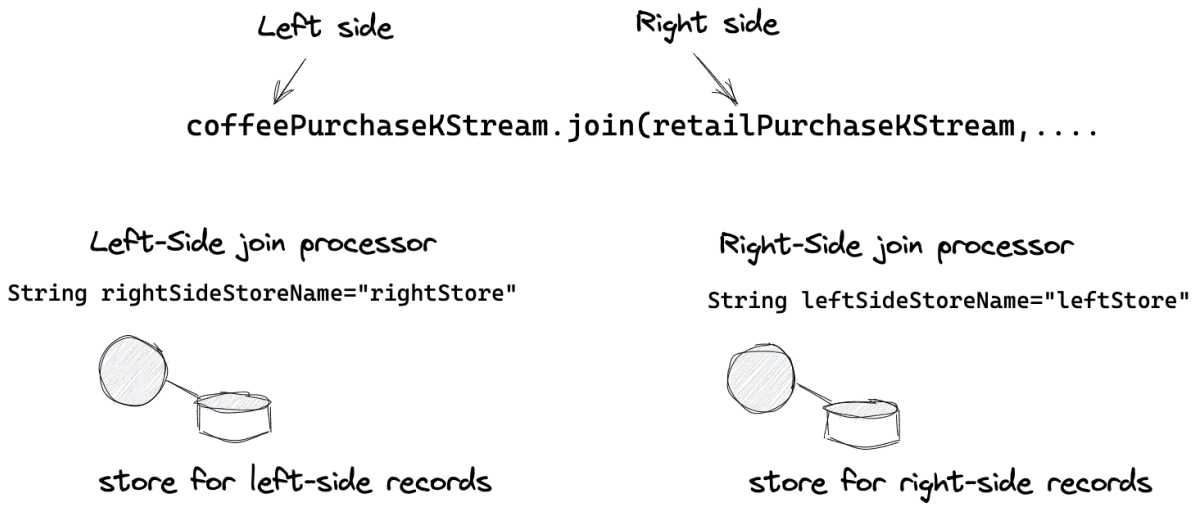


Figure 7.11 In a Stream-Stream join both sides of the join have a processor and state store

When building the processor for the join for each side, Kafka Streams includes the name of the state store for the reciprocal side of the join - the left side gets the name of the right side store and the right side processor contains the left store name. Why does each side contain the name of opposite side store? The answer gets at the heart of how joins work in Kafka Streams. Let's look at another illustration to demonstrate:

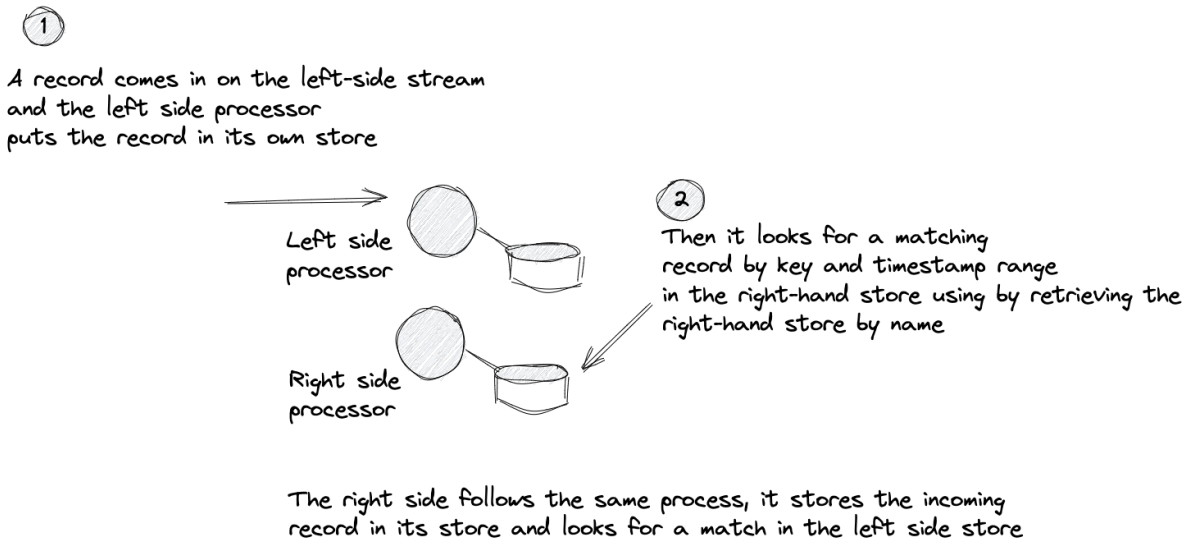


Figure 7.12 Join processors look in the other side's state store for matches when a new record arrives

When a new record comes in (we're using the left-side processor for the `coffeePurchaseKStream`) the processor puts the record in its own store, but then looks for a

match by retrieving the right-side store (for the `retailPurchaseKStream`) by name. The processor retrieves records with the same key and ***within the time range specifed by the JoinWindows***.

Now, the final part to consider is if a match is found. Let's look at one more illustration to help us see what's going on:

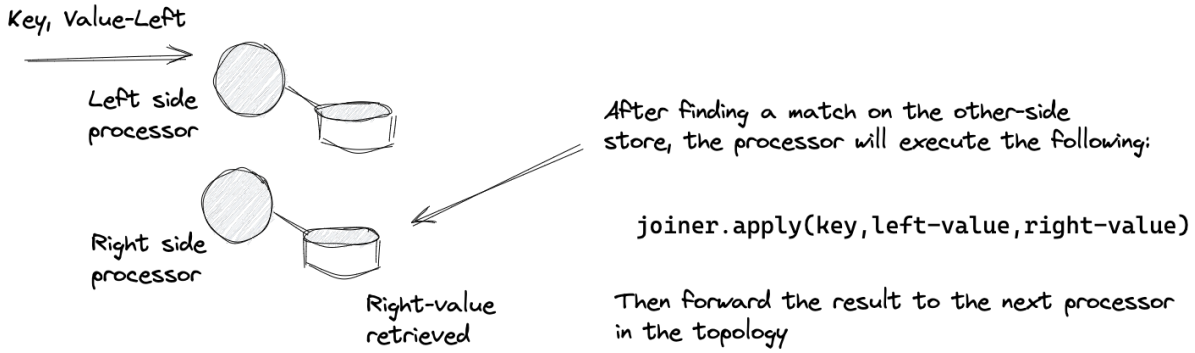


Figure 7.13 When matching record(s) is found the processor executes the joiner's apply method with the key, its own record value and the value from the other side

So now, after an incoming record finds a match by looking in the store from the other join side, the join processor (the coffeePurchaseKStream in our illustration) takes the key and the value from its incoming record, the value for each record it has retrieved from the store and executes the `ValueJoiner.apply` method which creates the join record specified by the implementation you've provided. From there the join processor forwards the key and join result to any down stream processors.

Now that we've discussed how joins operate internally let's discuss in more detail some of the parameters to the join

7.3.3 ValueJoiner

To create the joined result, you need to create an instance of a `ValueJoiner<V1, V2, R>`. The `ValueJoiner` takes two objects, which may or may not be of the same type, and it returns a single object, possibly of a third type. In this case, `ValueJoiner` takes a `CoffeePurchase` and a `RetailPurchase` and returns a `Promotion` object. Let's take a look at the code (found in src/main/java/bbejeck/chapter_7/joiner/PromotionJoiner.java).

Listing 7.14 `ValueJoiner` implementation

```
public class PromotionJoiner
     implements ValueJoiner<CoffeePurchase,
                            RetailPurchase,
                            Promotion> {

    @Override
    public Promotion apply(
            CoffeePurchase coffeePurchase,
            RetailPurchase retailPurchase) {

    double coffeeSpend = coffeePurchase.getPrice();   ❶
    double storeSpend = retailPurchase.getPurchasedItemsList()   ❷
            .stream()
            .mapToDouble(pi -> pi.getPrice() * pi.getQuantity()).sum();
    double promotionPoints = coffeeSpend + storeSpend;    ❸
    if (storeSpend > 50.00) {
        promotionPoints += 50.00;
    }
    return Promotion.newBuilder()    ❹
            .setCustomerId(retailPurchase.getCustomerId())
            .setDrink(coffeePurchase.getDrink())
            .setItemsPurchased(retailPurchase.getPurchasedItemsCount())
            .setPoints(promotionPoints).build();
}
```

❶ Extracting how much was spent on coffee

❷ Summing the total of purchased items

❸ Calculating the promotion points

❹ Build and return the new Promotion object

To create the `Promotion` object, you extract the amount spent from both sides of the join and perform a calculation resulting in the total amount of points to reward the customer. I'd like to point out that the `ValueJoiner` interface only has one method, `apply`, so you could use a lambda to represent the joiner. But in this case you create a concrete implementation, because you can write a separate unit test for the `ValueJoiner`. We'll come back this approach in the chapter on testing.

NOTE **Kafka Streams also provides a `ValueJoinerWithKey` interface which provides access to the key for calculating the value of the join result. However the key is considered read-only and making changes to it in the joiner implementation will lead undefined behavior.**

7.3.4 Join Windows

The `JoinWindows` configuration object plays a critical role in the join process; it specifies the difference between the timestamps of records from both streams to produce a join result.

Let's refer to the following illustration as an aid to understand the `JoinWindows` role.

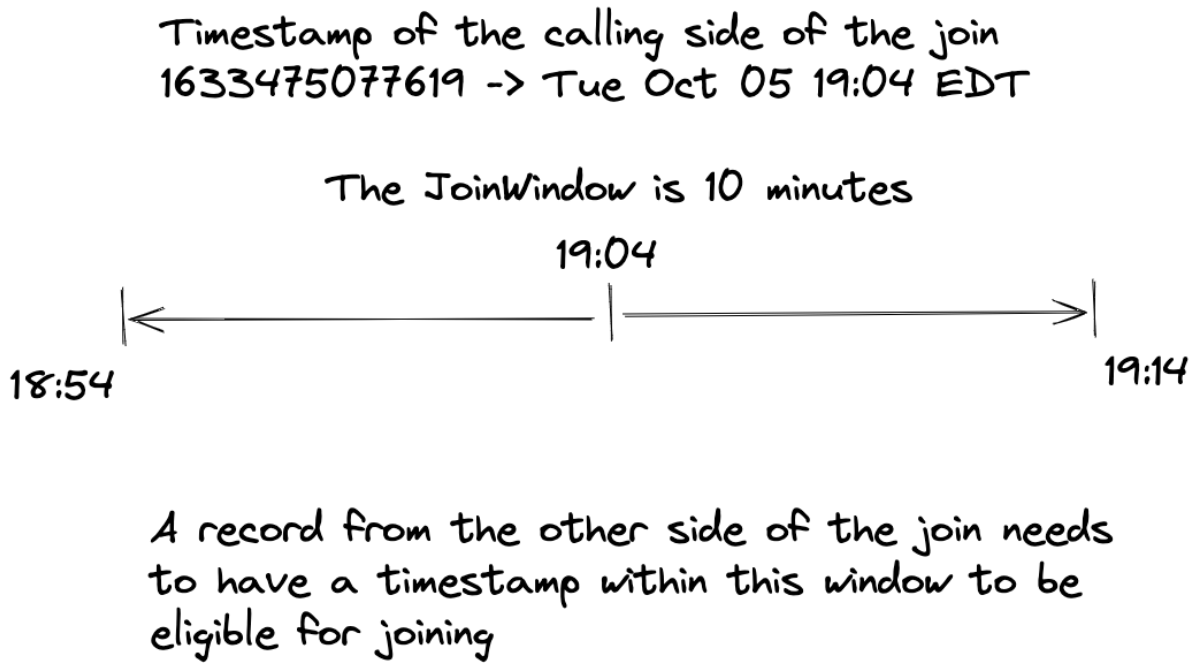


Figure 7.14 The JoinWindows configuration specifies the max difference (before or after) from the timestamp of the calling side the secondary side can have to create a join result.

More precisely the `JoinWindows` setting is the maximum difference, either before or after, the secondary (other) side's timestamp can from the primary side timestamp to create a join result. Looking at the example in listing XXX, the join window there is set for thirty minutes. So let's say a record from the `coffeeStream` has a timestamp of 12:00 PM, for a corresponding record in the `storeStream` to complete the join, it will need a timestamp from 11:30 AM to 12:30 PM.

There are two additional `JoinWindows()` methods are available `after` and `before`, which you can use to specify the timing and possibly the order of events for the join.

Let's say you're fine with the opening window of the join at thirty minutes but you want the closing window to be shorter, say five minutes. You'd use the `JoinWindows.after` method (still using the example in listing XXX) like so

Listing 7.15 Using the JoinWindows.after method to alter the closing side of the join window

```
coffeeStream.join(storeStream,...,
    thirtyMinuteWindow.after(Duration.ofMinutes(5))....
```

Here the opening window stays the same, the `storeStream` record can have a timestamp of at least 11:30 AM , but the closing window is shorter, the latest it can be is now 12:05 PM.

The `JoinWindows.before` method works in a similar manner, just in the opposite direction. Let's say now you want to shorten the opening window, so you'll now use this code:

Listing 7.16 The JoinWindows.before method changes the opening side of the join window

```
coffeeStream.join(storeStream,...,
    thirtyMinuteWindow.before(Duration.ofMinutes(5))....
```

Now you've changed things so the timestamp of the `storeStream` record can be at most 5 minutes *before* the timestamp of a `coffeeStream` record. So the acceptable timestamps for a join (`storeStream` records) now start at 11:55 AM but end at 12:30 PM. You can also use `JoinWindows.before` and `JoinWindows.after` to specify the order of arrival of records to perform a join.

For example to set up a join when a store purchase ***only happens within 30 minutes after a cafe purchase*** you would use `JoinWindows.of(Duration.ofMinutes(0).after(Duration.ofMinutes(30)` and to only consider store purchases ***before*** you would use `JoinWindows.of(Duration.ofMinutes(0).before(Duration.ofMinutes(30))`.

IMPORTANT **In order to perform a join in Kafka Streams, you need to ensure that all join participants are co-partitioned, meaning that they have the same number of partitions and are keyed by the same type. Co-partitioning also requires all Kafka producers to use the same partitioning class when writing to Kafka Streams source topics. Likewise, you need to use the same `StreamPartitioner` for any operations writing Kafka Streams sink topics via the `KStream.to()` method. If you stick with the default partitioning strategies, you won't need to worry about partitioning strategies.**

As you can see the `JoinWindows` class gives you plenty of options to control joining two streams. It's important to remember that it's the timestamps on the records driving the join behavior. The timestamps can be either the ones set by Kafka (broker or producer) or they can be embedded in the record payload itself. To use a timestamp embedded in the record you'll need to provide a custom `TimestampExtractor` and I'll cover that as well as timestamp semantics in the next chapter.

7.3.5 StreamJoined

The final paramter to discuss is the `StreamJoined` configuration object. With `StreamJoined` you can provide the serdes for the key and the values involved in the join. Providing the serdes for the join records is always a good idea, because you may have different types than what has been configured at the application level. You can also name the join processor and the state stores used for storing record lookups to complete the join. The importance of naming state stores is covered in the upcoming 7.4.5 section.

Before we move on from joins let's talk about some of the other join options available.

7.3.6 Other join options

The join in listing for the current example is an *inner join*. With an inner join, if either record isn't present, the join doesn't occur, and you don't emit a `Promotion` object. There are other options that don't require both records. These are useful if you need information even when the desired record for joining isn't available.

7.3.7 Outer joins

Outer joins always output a record, but the result may not include both sides of the join. You'd use an outer join when you wanted to see a result regardless of a successful join or not. If you wanted to use an outer join for the join example, you'd do so like this:

```
coffeePurchaseKStream.outerJoin(retailPurchaseKStream,..)
```

An outer join sends a result that contains records from either side or both. For example the join result could be `left+right`, `left+null`, or `null+right`, depending on what's present. The following illustration demonstrates the three possible outcomes of the outer join.

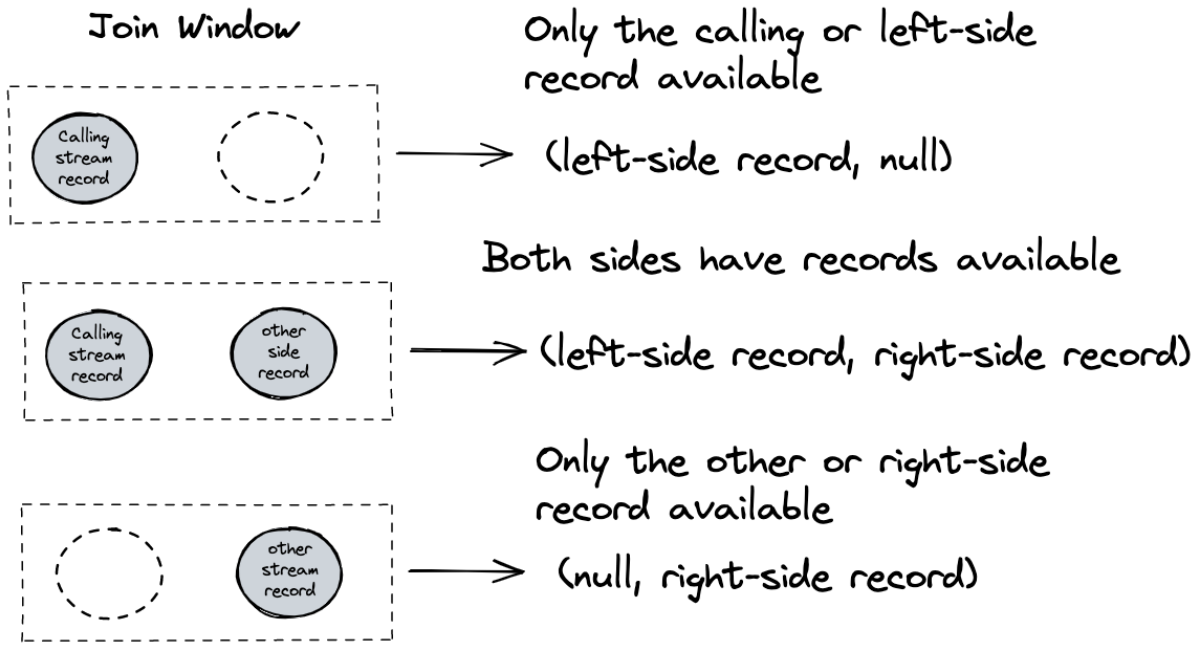


Figure 7.15 Three outcomes are possible with outer joins: only the calling stream's event, both events, and only the other stream's event.

7.3.8 Left-outer join

A left-outer join also always produces a result. But the difference from the outer-join is the left or calling side of the join is always present in the result, `left+right` or `left+null` for example. You'd use a left-outer join when you consider the left or calling side stream records essential for your business logic. If you wanted to use a left-outer join in listing 7.13, you'd do so like this:

```
coffeePurchaseKStream.leftJoin(retailPurchaseKStream..)
```

Figure 7.17 shows the outcomes of the left-outer join.

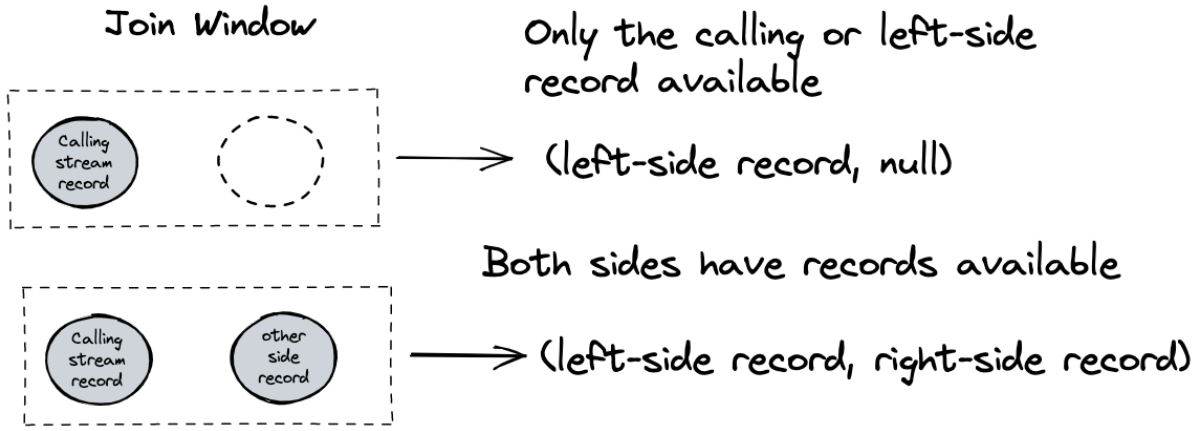


Figure 7.16 Two outcomes are possible with the left-outer join left and right side or left and null.

At this point you've learned the different join types, so what are the cases when you need to use them? Let's start with the current join example. Since you are determining a promotional reward based on the purchase of two items, each in their own stream an inner-join makes sense. If there is no corresponding purchase on the other side, then you don't have an actionable result, so to emit nothing is desired.

For cases where one side of the join is critical and the other is useful, but not essential then a left-side join is a good choice where you'd use the critical stream on the left or calling side. I'll cover an example when we get to stream-table joins in an upcoming section.

Finally, for a case where you have two streams where both sides enhance each other, but each one is important on its own, then an outer join fits the bill. Consider IoT, where you have two related sensor streams. Combining the sensor information provides you with a more complete picture but you want information from either side if it's available.

In the next section, let's go into the details of the workhorse of stateful operations, the state store.

7.4 State stores in Kafka Streams

So far, we've discussed the stateful operations in the Kafka Streams DSL API, but glossed over the underlying storage mechanism those operations use. In this section, we'll look at the essentials of using state stores in Kafka Streams and the key factors related to using state in streaming applications in general. This will enable you to make practical choices when using state in your Kafka Streams applications.

Before I go into any specifics, let's cover some general information. At a high-level, the state stores in Kafka Streams are key-value stores and they fall into two categories, persistent and in-memory. Both types are durable due to the fact that Kafka Streams uses changelog topics to back the stores. I'll talk more about changelog topics soon.

Persistent stores store their records in local disk, so they maintain their contents over restarts. The in-memory stores place records well, in memory, so they need to be restored after a restart. Any store that needs restoring will use the changelog topic to accomplish this task. But to understand how a state store leverages a changelog topic for restoration, let's take a look at how Kafka Streams implements them.

In the DSL, when you apply a stateful operation to the topology, Kafka Streams creates a state store for the processor (persistent are the default type). Along with the store, Kafka Streams also creates a changelog topic backing the store at the same time. As records are written the store, they are also written to the changelog. Here's an illustration depicting this process:

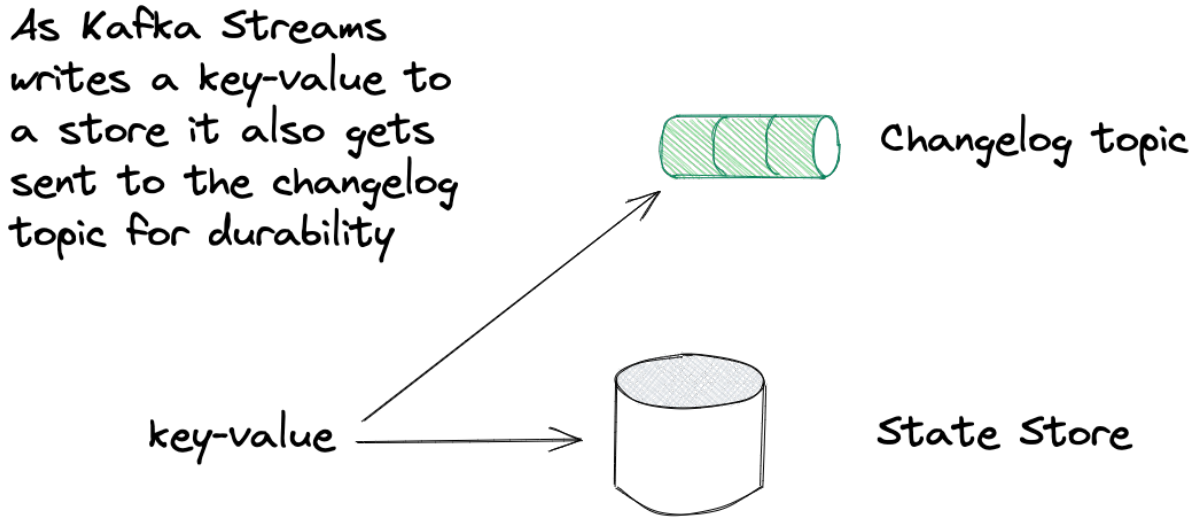


Figure 7.17 As the key-value records get written to the store they also get written to the changelog topic for data durability

So as Kafka Streams places record into a state store, it also sends it to a Kafka topic that backs the state store. Now if you remember from earlier in the chapter, I mentioned that with an aggregation you don't see every update as Kafka Streams uses a cache to initially hold the

results. It's only on when Kafka Streams flushes the cache, either at a commit or when it's full, that records from the aggregation go to downstream processors. It's at this point that Kafka Streams will produce records to the changelog topic.

NOTE	If you've disabled the cache then every record gets sent to the state store so this also means every record goes to the changelog topic as well.

7.4.1 Changelog topics restoring state stores

So how does the Kafka Stream leverage the changelog topic? Let's first consider the case of a in-memory state store. Since an in-memory store doesn't maintain it's contents across restarts, when starting up, any in-memory stores will rebuild their contents from head record of the changelog topic. So even though the in-memory store loses all its contents on application shut-down, it picks up where it left off when restarted.

For persistent stores, usually it's only after all local state is lost, or if data corruption is detected that it will need to do a ***full*** restore. For persistent stores, Kafka Streams maintains a checkpoint file for persistent stores and it will use the offset in the file as a starting point to restore from instead of restoring from scratch. If the offset is no longer valid, then Kafka Streams will remove the checkpoint file and restore from the beginning of the topic.

This difference in restoration patterns brings an interesting twist to the discussion of the trade-offs of using either persistent or in-memory stores. While an in-memory store should yield faster look-ups as it doesn't need to go to disk for retrieval, under "happy path" conditions the topology with persistent stores will generally resume to processing faster as it will not have as many records to restore.

IMPORTANT	An exception to using a checkpoint file for restoration is when you run Kafka Streams in EOS mode (either `exactly_once` or `exactly_once_v2` is enabled) as state stores are fully restored on startup to ensure the only records in the stores are ones that were included in successful transactions.

Another situation to consider is the make up of running Kafka Streams applications. If you recall from our discussion on task assignments, you can change the number of running applications dynamically, either by expansion or contraction. Kafka Streams will automatically assign tasks from existing applications to new members, or add tasks to those still running from an application that has dropped out of the group. A task that is responsible for a stateful operation will have a state store as part of its assignment (I'll talk about state stores and tasks next).

Let's consider the case of a Kafka Streams application that loses one of its members, remember you can run Kafka Streams applications on different machines and those with the same application id are considered all part of one logical application. Kafka Streams will issue a

rebalance and the tasks from the defunct application get reassigned. For any reassigned stateful operations, since Kafka Streams creates a new **empty** store for the newly assigned task, they'll need to restore from the beginning of the changelog topic before they resume processing.

Here's an illustration demonstrating this situation:

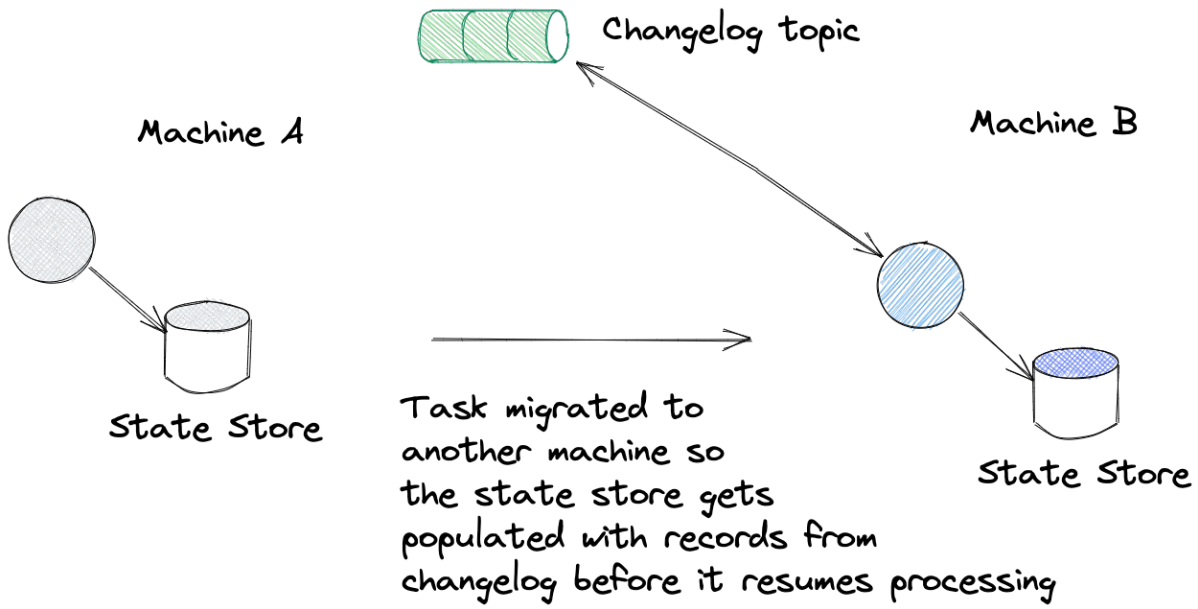


Figure 7.18 When a stateful task gets moved to a new machine Kafka Streams rebuilds the state store from the beginning of the changelog topic

So by using changelog topics you can be assured your applications will have a high degree of data durability even in the face of application loss, but there's delayed processing until the store is fully online. Fortunately, Kafka Streams offers a remedy for this situation, the standby task.

7.4.2 Standby Tasks

To enable fast failover from an application instance dropping out of the group Kafka Streams provides the standby task. A standby task "shadows" an active task by consuming from the changelog topic into a state store local to the standby. Then should the active task drop out of the group, the standby becomes the new active task. But since it's been consuming from the changelog topic, the new active task will come online with minimum latency.

> **IMPORTANT** **To enable standby tasks you need to set the `num.standby.replicas` configuration to a value greater than 0 and you need to deploy N+1 number of Kafka Streams instances (with N being equal to the number of desired replicas). Ideally you'll deploy those Kafka Streams instances on separate machines as well.**

While the concept is straight forward, let's review the standby process by walking through the

following illustration:

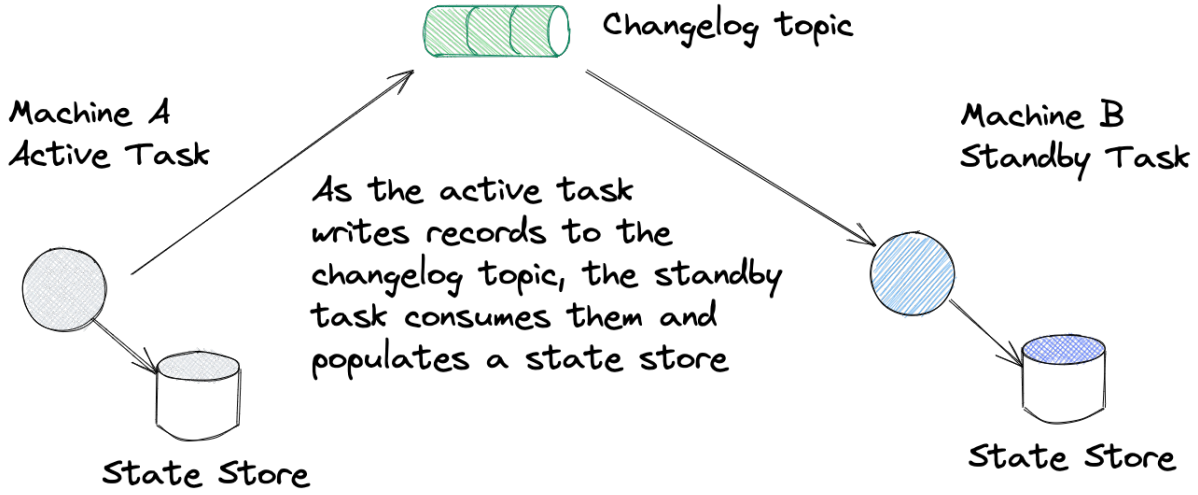


Figure 7.19 A standby task shadows the active task and consumes from the changelog topic keeping a local state store in-sync with store of the active task

So following along with the illustration a standby task consumes records from the changelog topic and puts them in its own local state store. To be clear, a standby task does not process any records, its only job is to keep the state store in sync with the state store of the active task. Just like any standard producer and consumer application, there's no coordination between the active and standby tasks.

With this process since the standby stays fully caught up to the active task or at minimum it will be only a handful of records behind, so when Kafka Streams reassigns the task, the standby becomes the active task and processing resumes with minimal latency as its already caught up. As with anything there is a trade-off to consider with standby tasks. By using standby's you end up duplicating data, but with benefit of near immediate fail-over, depending on your use case it's definitely worth consideration.

NOTE	Significant work went into improving the scaling out performance of Kafka Streams with Kafka KIP-441 (cwiki.apache.org/confluence/display/KAFKA/KIP-441%3A+Smooth+Scaling+Out+for+Kafka+Streams). When you enable standby tasks and the standby instance becomes the active one, if at a later time Kafka Streams determines a more favorable assignment is possible, then that stateful task may get migrated to another instance.

So far we've covered how state stores enable stateful operations and how the stores are robust due to changelog topic and using standby tasks to enable quick failover. But we still have some

more ground to cover. First we'll go over state store assignment, from there you'll learn how to configure state stores including how to specify a store type including an in-memory store and finally how to configure changelog topics if needed.

7.4.3 Assigning state stores in Kafka Streams

In the previous chapter we discussed the role of tasks in Kafka Streams. Here I want to reiterate that tasks operate in a shared nothing architecture and only operate in a single thread. While a Kafka Streams application can have multiple threads and each thread can have multiple tasks, there is nothing shared between them. I emphasize this "shared nothing" architecture again, because this means that when a task is stateful, only the owning task will access its state store, there are no locking or concurrency issues.

Going back to the [Stock-Aggregation] example, let's say the source topic has two partitions, meaning it has two tasks. Let's look at an updated illustration of tasks assignment with state stores for that example:

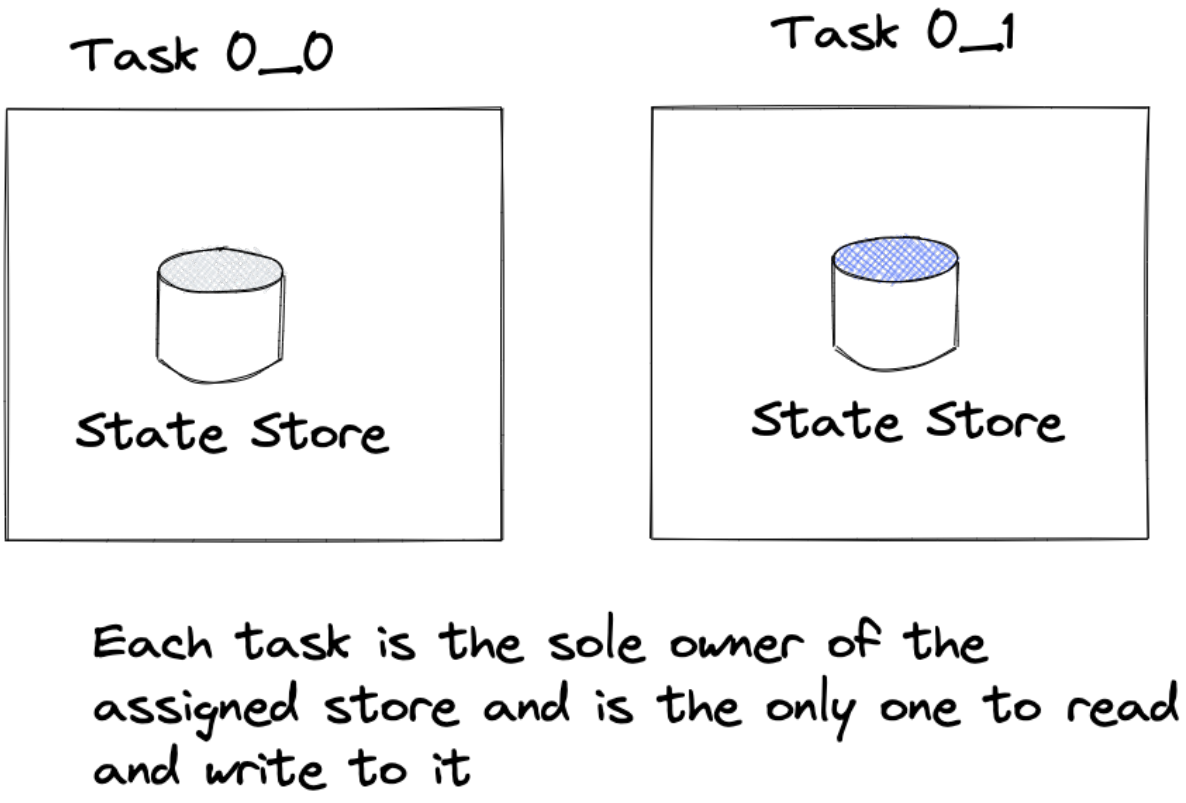


Figure 7.20 Stateful tasks have a state store assigned to it

By looking at this illustration you can see that the task associated with the state store is the only task that will ever access it. Now let's talk about how Kafka Streams places state stores in the file system.

7.4.4 State store location on the file system

When you have a stateful application, when Kafka Streams first starts up, it creates a root directory for all state stores from the `StreamsConfig.STATE_DIR_CONFIG` configuration. If not set, the `STATE_DIR_CONFIG` defaults to the temporary directory for the JVM followed by the system dependent separator and then "kafka-streams".

> **IMPORTANT** **The value of the `STATE_DIR_CONFIG` configuration must be unique for each Kafka Streams instance that shares the same file system**

For example on my MacOS the default root directory for state stores is `/var/folders/lk/d_9__qr558zd6ghbqwty0zc80000gn/T/kafka-streams`.

> **TIP** **To view the system dependent temporary directory on you machine you can start a Java shell from a terminal window by running the `jshell` command. Then type in `System.getProperty("java.io.tmpdir")`, hit the `return` key and it will display on the screen.**

Next Kafka Streams appends the application-id, which you have to provide in the configurations, to the path. Again on my laptop the path is `/var/folders/lk/d_9__qr558zd6ghbqwty0zc80000gn/T/kafka-streams/test-application/`

From here the directory structure branches out to unique directories for each task. Kafka Streams creates a directory for each stateful task using the subtopology-id and partition (separated by an underscore) for the directory name. For example a stateful task from the first subtopology and assigned to partition zero would use `0_0` for the directory name.

The next directory is named for the implementation of the store which is `rocksdb`. So at this point the path would look like `/var/folders/lk/d_9__qr558zd6ghbqwty0zc80000gn/T/kafka-streams/test-application/0_0/rocksdb`. It is under this directory there is the final directory from the processor (unless provided by a `Materialized` object and I'll cover that soon). To understand how the final directory gets its name, let's look at snippet of a stateful Kafka Streams application and the generated topology names.

Listing 7.17 Simple Kafka Streams stateful application

```
builder.stream("input")
       .groupByKey()
       .count()
       .toStream()
       .to("output")
```

This application has topology named accordingly: .Topology names

```
Topologies:
   Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [input])
      --> KSTREAM-AGGREGATE-0000000002
    Processor: KSTREAM-AGGREGATE-0000000002   ❶
     (stores: [KSTREAM-AGGREGATE-STATE-STORE-0000000001])   ❷
      --> KTABLE-TOSTREAM-0000000003
      <-- KSTREAM-SOURCE-0000000000
    Processor: KTABLE-TOSTREAM-0000000003 (stores: [])
      --> KSTREAM-SINK-0000000004
      <-- KSTREAM-AGGREGATE-0000000002
    Sink: KSTREAM-SINK-0000000004 (topic: output)
      <-- KTABLE-TOSTREAM-0000000003
```

❶ The name of the aggregate processor

❷ The name of the store assigned to processor

From the topology here Kafka Streams generates the name `KSTREAM-AGGREGATE-0000000002` for the `count()` method and notice it's associated with the store named `KSTREAM-AGGREGATE-STATE-STORE-0000000001`. So Kafka Streams takes the base name of the stateful processor and appends a `STATE-STORE` and the number generated from the global counter. Now lets take a look at the full path you would find this state store: `/var/folders/lk/d_9__qr558zd6ghbqwty0zc80000gn/T/kafka-streams/test-application/0`

So it's the final directory `KSTREAM-AGGREGATE-STATE-STORE-0000000001` in the path that contains the RocksDB files for that store. Now if you were to check the topics on the broker after starting the Kafka Streams application you'd see this name in the list `test-application-KSTREAM-AGGREGATE-STATE-STORE-0000000001-changelog`. This topic is the changelog for the state store and notice how Kafka Streams uses a naming convention of <application-id>-<state store name>-changelog for the topic.

7.4.5 Naming Stateful operations

This naming raises an interesting question, what happens if we add an operation before the `count()`? Let's say you want to add a filter to exclude certain records from the counting. You'd simply update the topology like so:

Listing 7.18 Updated Topology with a filter

```
builder.stream("input")
       .filter((key, value) -> !key.equals("bad"))
       .groupByKey()
       .count()
       .toStream()
       .to("output")
```

Remember, Kafka Streams uses a global counter for naming the processor nodes, so since you've added an operation, every processor downstream of it will have a new name since the number will be greater by 1. Here's what the new topology will look like:

Listing 7.19 Updated Topology names

```
Topologies:
   Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [input])
      --> KSTREAM-FILTER-0000000001
    Processor: KSTREAM-FILTER-0000000001 (stores: [])
      --> KSTREAM-AGGREGATE-0000000003
      <-- KSTREAM-SOURCE-0000000000
    Processor: KSTREAM-AGGREGATE-0000000003   ❶
     (stores: [KSTREAM-AGGREGATE-STATE-STORE-0000000002])   ❷
      --> KTABLE-TOSTREAM-0000000004
      <-- KSTREAM-FILTER-0000000001
    Processor: KTABLE-TOSTREAM-0000000004 (stores: [])
      --> KSTREAM-SINK-0000000005
      <-- KSTREAM-AGGREGATE-0000000003
    Sink: KSTREAM-SINK-0000000005 (topic: output)
      <-- KTABLE-TOSTREAM-0000000004
```

❶ The new name for the aggregation operation

❷ The new name for the state store

Notice how the state store name has changed which means there is a new directory named `KSTREAM-AGGREGATE-STATE-STORE-0000000002` and the corresponding changelog topic is now named `test-application-KSTREAM-AGGREGATE-STATE-STORE-0000000002-changelog`.

NOTE **Any changes before a stateful operation could result in the generated name shift, i.e. removing operators will have the same shifting effect.**

What does this mean to you? When you redeploy this Kafka Streams application the directory will only contain some basic RocksDB file, but not your original contents they are in the previous state store directory. Normally an empty state store directory does not present an issue, as Kafka Streams will restore it from the changelog topic. Except in this case the changelog topic is also new, so it's empty as well. So while your data is still safe in Kafka, the Kafka Streams application will start over with empty state store due to the name changes.

While it's possible to reset the offsets and process data again, a better approach is to avoid name shifting situation all together by providing a name for the state store instead of relying on the generated one. In the previous chapter I covered naming processor nodes for providing a better understanding of what the topology does. But in this case it goes beyond better understanding of its role in the topology, which is important, but also makes your application robust in the face of a changing topology.

Going back to the simple `count()` example in this section, you'll update the application by passing `Materialized` object to `count()` operation:

Listing 7.20 Naming the state store using a Materialized object

```
builder.stream("input")
       .groupByKey()
       .count(Materialized.as("counting-store"))   ❶
       .toStream()
       .to("output")
```

❶ Explicitly naming the state store

By providing the name of the state store, Kafka Streams will name the directory on disk `counting-store` and the changelog topic becomes `test-application-counting-store-changelog`, and both of these names are "frozen" and will not change regardless of any updates you make to the topology. It's important to note that the names of state stores within a topology must be unique, otherwise you'll get a `TopologyException`.

NOTE **Only stateful operations are affected by name shifting. But since stateless operations don't keep any state, changes in processor names from topology updates will have no impact.**

The bottom line is to ***always*** name state stores and repartition topics using the appropriate configuration object. By naming the stateful parts of your applications, you can ensure that topology updates don't break the compatibility. Here's a table summarizing which configuration object to use and the operation(s) it applies to:

Table 7.1 Kafka Streams configuration objects for naming state stores and repartition topics

Configuration Object	What's Named	Where Used
Materialized	State Store, Changelog topic	Aggregations
Repartitioned	Repartition topic	Repartition (manual by user)
Grouped	Repartition topic	GroupBy (automatic repartitioning)
StreamJoined	State Store, Changelog topic, Repartition topic	Joins (automatic repartitioning)

Naming state stores provides the added benefit of being able to query them while your Kafka Streams application is running, providing live, materialized views of the streams. I'll cover interactive queries in the next chapter.

So far you've learned how Kafka Streams uses state stores in support of stateful operations. You also learned that the default is for Kafka Streams to use persistent stores and there are in-memory store implementations available. In the next section I'm going to cover how you can specify a different store type as well as configuration options for the changelog topics.

7.4.6 Specifying a store type

All the examples so far in this chapter use persistent state stores, but I've stated that you can use in-memory stores as well. So the question is how do you go about using an in-memory store? So far you've used the `Materialized` configuration object to specify `Serdes` and the name for a store, but you can use it to provide a custom `StateStore` instance to use. Kafka Streams makes it easy to provide an in-memory version of the available store types (so far I've only covered "vanilla" key-value stores, but I'll get to sessioned, windowed, timestamped stores in the next chapter).

The best way to learn how to use a different store type is to change one of our existing examples. Let's revisit the first stateful example used to keep track of scores in an online poker game:

Listing 7.21 Performing a reduce in Kafka Streams to show running total of scores in an online poker game updated to use in-memory stores

```
KStream<String, Double> pokerScoreStream = builder.stream("poker-game",
        Consumed.with(Serdes.String(), Serdes.Double()));

pokerScoreStream
        .groupByKey()
        .reduce(Double::sum,
                Materialized.<String, Double>as(
            Stores.inMemoryKeyValueStore("memory-poker-score-store"))   ❶
                  .withKeySerde(Serdes.String())    ❷
                  .withValueSerde(Serdes.Double()))   ❸
        .toStream()
        .to("total-scores",
                Produced.with(Serdes.String(), Serdes.Double()));
```

❶ Passing a `StoreSupplier` to specify an in-memory store

❷ Specifying the `Serdes` for the key

❸ Specifying the `Serdes` for the value

So by using the overloaded `Materialized.as` method, you provide a `StoreSupplier` using one of the factory methods available from the `Stores` class. Notice that you still pass the serde instances needed for the store. And that's all it takes to switch the store type from persistent to in-memory.

> **NOTE** **Switching in a different store type is fairly straight forward so I'll only have the one example here. But the source code will contain a few additional examples.**

So why would you want to use an in-memory store? Well, an in-memory store will give you faster access since it doesn't need to go to disk to retrieve values. So a topology using in-memory stores should have higher throughput than one using persistent ones. But there are trade-offs you should consider.

First, an in-memory store has limited storage space, and once it reaches it's memory limit it will evict entries to make space. The second consideration is when you stop and restart a Kafka Streams application, under "happy-path" conditions, the one with persistent stores will start processing faster due to the fact that it will have all its state already, but the in-memory stores will always need to restore from the changelog topic.

Kafka Streams provides a factory class `Stores` that provides methods for creating either `StoreSuppliers` or `StoreBuilders`. The choice of which one to use depends on the Kafka Streams API. When using the DSL you'll use `StoreSuppliers` with a `Materialized` object. In the Processor API, you'll use a `StoreBuilder` and directly add it to the topology. I'll cover the Processor API in chapter 9.

TIP **To see all the different store types you can create view the JavaDoc for the `Stores` class javadoc.io/doc/org.apache.kafka/kafka-streams/latest/org/apache/kafka/streams/state/Stores.html**

Now that you've learned how to specify a different store type, let's move on to one more topic to cover with state stores, how you can configure the changelog topic.

7.4.7 Configuring changelog topics

There's nothing special about changelog topics, so you can use any configuration parameters available for topics. But for the most part the default settings should suffice, so you should only consider changing the configurations when it's absolutely necessary.

NOTE **State store changelogs are compacted topics, which we discussed in chapter 2. As you may recall, the delete semantics require a null value for a key, so if you want to remove a record from a state store permanently, you'll need to do a put(key, null) operation.**

Let's revisit the example from above where you provided a custom name for the state store. Let's say the data processed by this application also has a large key space. The changelogs in Kafka Streams are ***compacted*** topics. Compacted topics use a different approach to cleaning up older records.

Instead of deleting log segments by size or time, log segments are *compacted* by keeping only the latest record for each key—older records with the same key are deleted. But since the key space is large compaction may not be enough, as the size of the log segment will keep growing. In that case, the solution is simple. You can specify a cleanup policy of `delete` and `compact` .

Listing 7.22 Setting a cleanup policy and using Materialized to set the new configuration

```
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("cleanup.policy", "compact,delete");

builder.stream("input")
       .groupByKey()
       .count(Materialized.as("counting-store")
              .withLoggingEnabled(changeLogConfigs))   ❶
       .toStream()
       .to("output")
```

❶ Using the withLoggingEnabled method to set a configuration

So here you can adjust the configurations for this specific changelog topic. Earlier I mentioned that to disable the caching that Kafka Streams uses for stateful operations, you'd set the `StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG` setting to zero. But since it's in the configuration, it is globally applied to all stateful operations. If you only wanted to disable the cache for a specific one you could disable it by calling `Materialized.withCachingDisabled()` method when passing in the `Materialzied` object.

WARNING **The `Materialized` object also provides a method to disable logging. Doing so will cause the state store to not have a changelog topic, hence it is subject to getting in a state where it can't restore its previous contents. It is recommended to only use this method if absolutely necessary. In my time working with Kafka Streams, I can't say I've encountered a good reason for using this method.**

7.5 Summary

- Stream processing needs state. Stateless processing is acceptable in a lot of cases, but to make more complex decisions you'll need to use stateful operations.
- Kafka Streams provides stateful operations reduce, aggregation, and joins. The state store is created automatically for you and by default they use persistent stores.
- You can choose to use in-memory stores for any stateful operation by passing a `StoreSupplier` from the `Stores` factory class to the `Materialized` configuration object.
- To perform stateful operations your records need to have valid keys-if your records don't have a key or you'd like to group or join records by a different key you can change it and Kafka Streams will automatically repartition the data for you.
- It's important to always provide a name for state stores and repartition topics-this keeps your application resilient from breaking when you make topology changes.

8 Advanced stateful concepts

This chapter covers

- Changelog streams, the KTable and the GlobalKTable
- Aggregating records with a KTable
- Joining a KTable with KStream or another KTable
- Windowing to capture aggregations in specific period of time
- Using suppression for final windowed results
- Understanding the importance of timestamps in Kafka Streams

In this chapter, we're going to continue working with state in a Kafka Streams application. You'll learn about the `KTable` which is considered an update or changelog stream. As a matter of fact you've already used a `KTable` as any aggregation operations in Kafka Streams result in a `KTable`. The `KTable` is an important abstraction for working with records that have the same key. Unlike the `KStream` where records with the same key are still considered independent event, in the `KTable` a record is an update to the previous record that has the same key.

To make a comparison to a relational database, the event stream (a `KStream`) could be considered a series of inserts where the primary key is an auto-incriminating number. As a result each insert of a new record has no relationship to previous ones. But with a `KTable` the key in the key-value pair is the primary key, so each time a record arrives with the same key, it's consider an update to the previous one.

From there you'll learn about aggregation operations with a `KTable`. Aggregations work a little differently because you don't want to group by primary key, you'll only ever have on record that way, instead you'll need to consider how you want to group the records to calculate the aggregate.

Since you can use the `KTable` as lookup table, a join between a stream and table is a power combination, where you can enrich the event stream records by performing a lookup in the table for additional details. You can also join two tables together, even using a foreign key. You'll also learn about a unique construct called the `GlobalKTable` which, unlike the `KTable` which sharded by partitions, contains all records from it's underlying source across all application instances.

After covering the table abstractions we'll get into how to "bucket" your aggregations into specific time periods using windowing. For example, how many purchases have there been over the past hour, updated every ten minutes? Windowing allows you to place data in discrete blocks of time, as opposed to having an unbounded collection. You'll also learn how to produce a single final result from a windowed operation when the window closes. There's also the ability to expose the underlying state stores to queries from outside the application allowing for real-time updates on the information in the event stream.

Our final topic for the chapter is how timestamps drive the behavior in Kafka Streams and especially their impact on windowing and stateful operations.

8.1 KTable The Update Stream

To fully understand the concept of an update stream, it will be useful to compare with an event stream to see the differences between the two. Let's use a concrete example of tracking stock price updates.

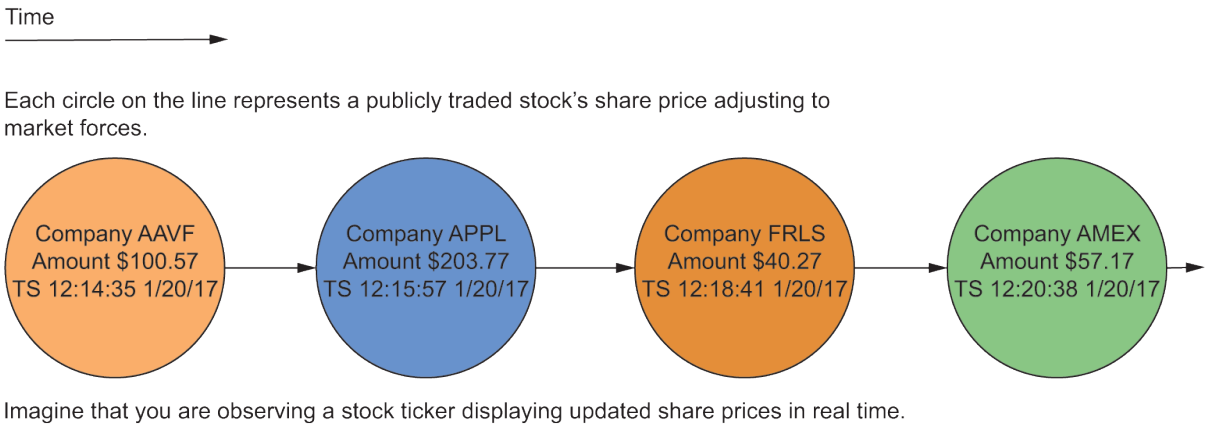
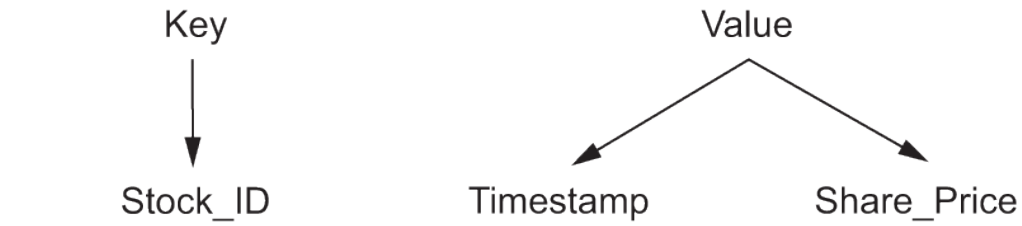


Figure 8.1 A diagram for an unbounded stream of stock quotes

You can see that each stock price quote is a discrete event, and they aren't related to each other. Even if the same company accounts for many price quotes, you're only looking at them one at a time. This view of events is how the `KStream` works—it's a stream of records.

Now, let's see how this concept ties into database tables. Each record is an insert into the table, but the primary key is a number increment for each insert, depicted simple stock quote table in figure 8.2.



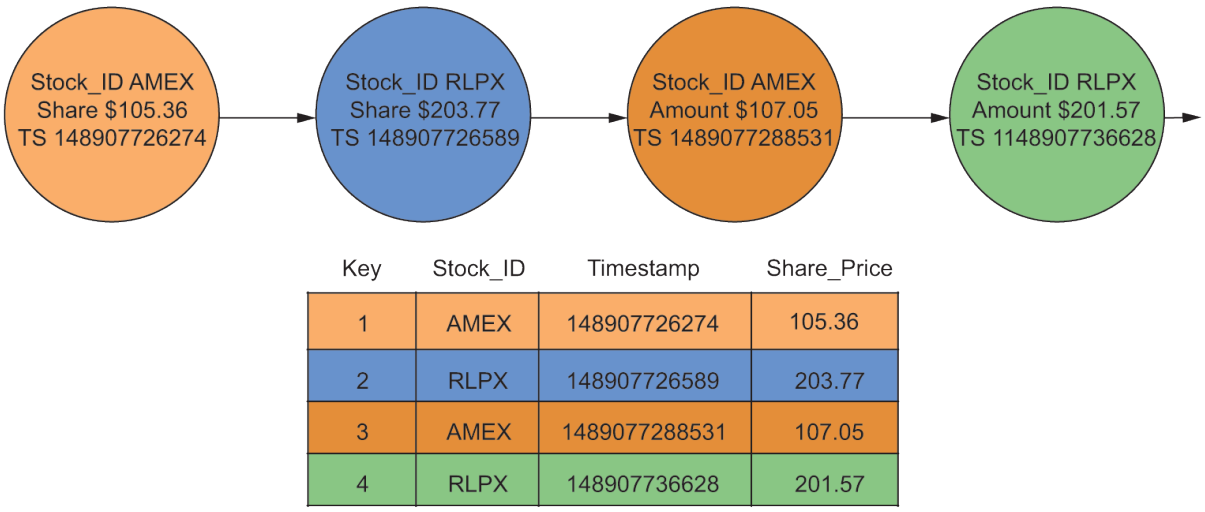
ABVF	32225544289	105.36
APPL	32225544254	333.66

The rows from table above can be recast as key/value pairs. For example, the first row in the table can be converted to this key/value pair:

{key:{stockid:1235588}, value:{ts:3225544289, price:105.36}}

Figure 8.2 A simple database table represents stock prices for companies. There's a key column, and the other columns contain values. You can consider this a key/value pair if you lump the other columns into a "value" container.

Next, let's take another look at the record stream. Because each record stands on its own, the stream represents inserts into a table. Figure 5.3 combines the two concepts to illustrate this point.



Key	Stock_ID	Timestamp	Share_Price
1	AMEX	148907726274	105.36
2	RLPX	148907726589	203.77
3	AMEX	1489077288531	107.05
4	RLPX	148907736628	201.57

This shows the relationship between events and inserts into a database. Even though it's stock prices for two companies, it counts as four events because you consider each item on the stream as a singular event.

As a result, each event is an insert, and you increment the key by one for each insert into the table.

With that in mind, each event is a new, independent record or insert into a database table.

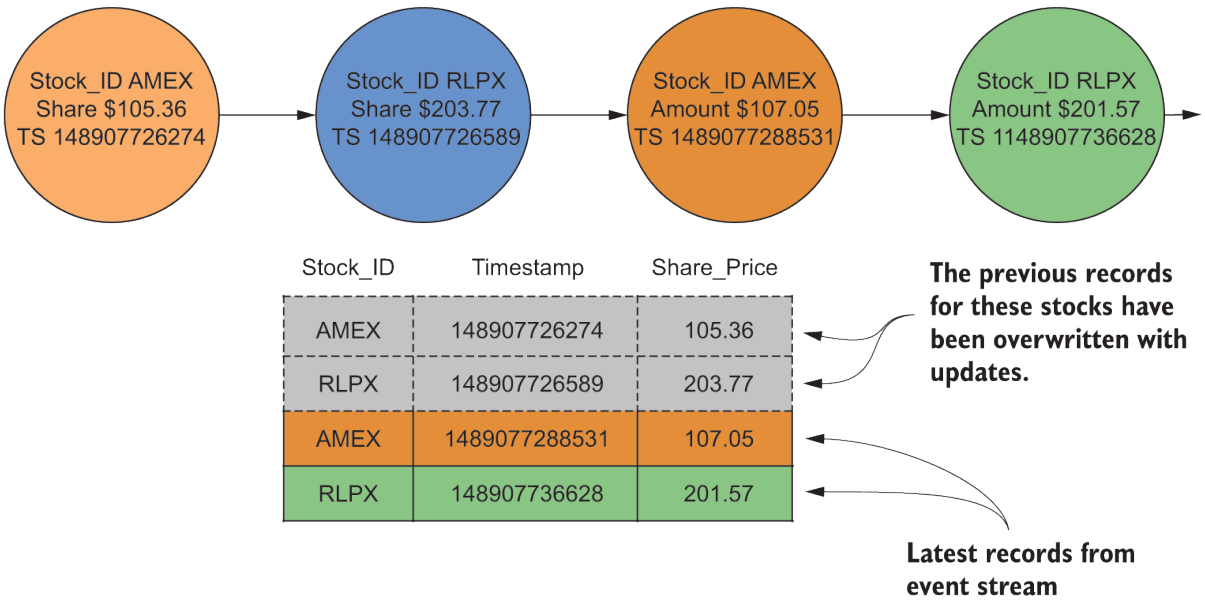
Figure 8.3 A stream of individual events compares to inserts into a database table. You could similarly imagine streaming each row from the table.

What's important here is that you can view a stream of events in the same light as inserts into a table, which can help give you a deeper understanding of using streams for working with events. The next step is to consider the case where events in the stream *are* related to one another.

8.1.1 Updates to records or the changelog

Let's say you want to track customer purchase behavior, so you take the same stream of customer transactions, but now track activity over time. If you add a key of customer ID, the purchase events can be related to each other, and you'll have an update stream as opposed to an event stream.

If you consider the stream of events as a log, you can consider this stream of updates as a changelog. Figure 8.4 demonstrates this concept.



If you use the stock ID as a primary key, subsequent events with the same key are updates in a changelog. In this case, you only have two records, one per company. Although more records can arrive for the same companies, the records won't accumulate.

Figure 8.4 In a changelog, each incoming record overwrites the previous one with the same key. With a record stream, you'd have a total of four events, but in the case of updates or a changelog, you have only two.

Here, you can see the relationship between a stream of updates and a database table. Both a log and a changelog represent incoming records appended to the end of a file. In a log, you see all the records; but in a changelog, you only keep the latest record for any given key.

> **NOTE** **With both a log and a changelog, records are appended to the end of the file as they come in. The distinction between the two is that in a log, you want to see all records, but in a changelog, you only want the latest record for each key.**

To trim a log while maintaining the latest records per key, you can use log compaction, which we discussed in chapter 2. You can see the impact of compacting a log in figure 8.5. Because you only care about the latest values, you can remove older key/value pairs.[6]

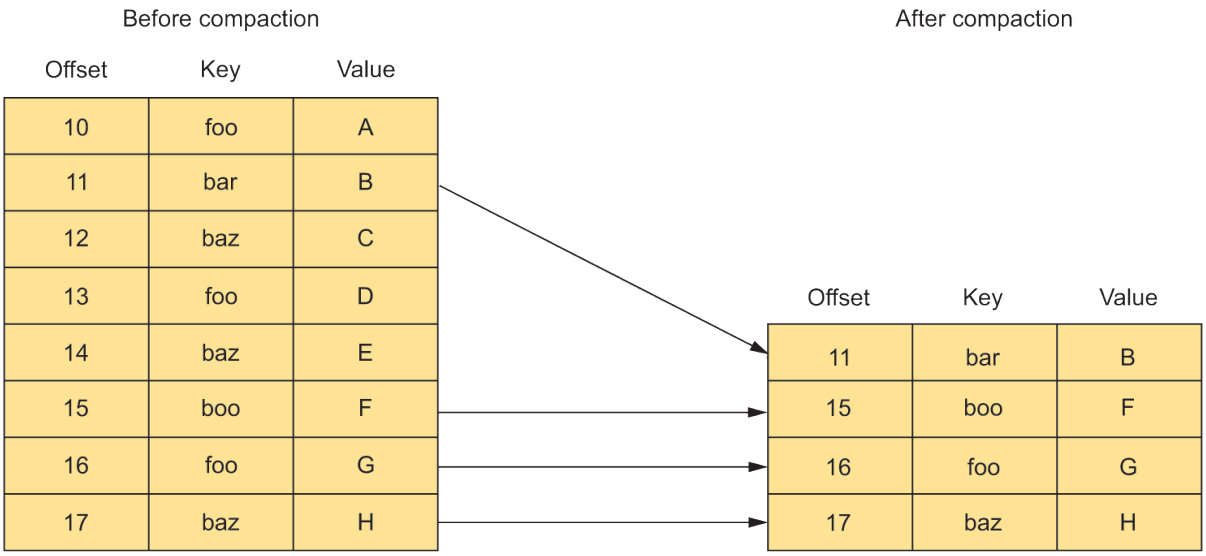


Figure 8.5 On the left is a log before compaction—you'll notice duplicate keys with different values, which are updates. On the right is the log after compaction—you keep the latest value for each key, but the log is smaller in size.

You're already familiar with event streams from working with `KStreams`. For a changelog or stream of updates, we'll use an abstraction known as the `KTable`. Now that we've established the relationship between streams and tables, the next step is to compare an event stream to an update stream.

8.1.2 Event streams vs. update streams

We'll use the `KStream` and the `KTable` to drive our comparison of event streams versus update streams. We'll do this by running a simple stock ticker application that writes the current share price for three (fictitious!) companies. It will produce three iterations of stock quotes for a total of nine records. A `KStream` and a `KTable` will read the records and write them to the console via the `print()` method.

> **NOTE** **The `KTable` does not have methods like `print()` or `peek()` in its API, so to do any printing of records you'll need to convert the `KTable` from an update stream to an event stream by using the `toStream()` method first.**

Figure 8.6 shows the results of running the application. As you can see, the `KStream` printed all nine records. We'd expect the `KStream` to behave this way because it views each record individually. In contrast, the `KTable` printed only three records, because the `KTable` views records as updates to previous ones.

A simple stock ticker for three fictitious companies with a data generator producing three updates for the stocks. The KStream printed all records as they were received. The KTable only printed the last batch of records because they were the latest updates for the given stock symbol.

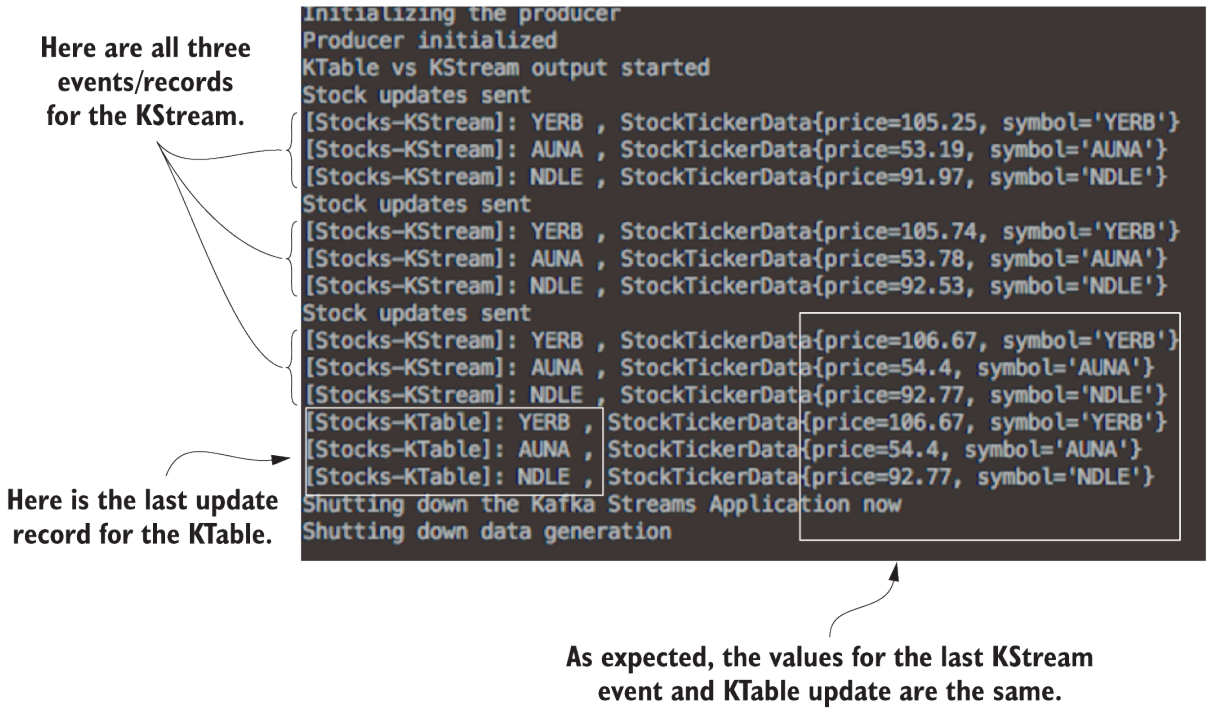


Figure 8.6 `KTable` versus `KStream` printing messages with the same keys

From the `KTable`'s point of view, it didn't receive nine individual records. The `KTable` received three original records and two rounds of updates, and it only printed the last round of updates. Notice that the `KTable` records are the same as the last three records published by the `KStream`. We'll discuss the mechanisms of how the `KTable` emits only the updates in the next section.

Here's the program for printing stock ticker results to the console (found in src/main/java/bbejeck/chapter_8/KStreamVsKTableExample.java; source code can be found on the book's website here: manning.com/books/kafka-streams-in-action-second-edition).

Listing 8.1 `KTable` and `KStream` printing to the console

```
KTable<String, StockTickerData> stockTickerTable =
builder.table(STOCK_TICKER_TABLE_TOPIC); ❶
KStream<String, StockTickerData> stockTickerStream =
builder.stream(STOCK_TICKER_STREAM_TOPIC); ❷

stockTickerTable.toStream()
  .print(Printed.<String, StockTickerData>toSysOut()
  .withLabel("Stocks-KTable")); ❸

stockTickerStream
  .print(Printed.<String, StockTickerData>toSysOut()
  .withLabel("Stocks-KStream")); ❹
```

❶ Creates the KTable instance

❷ Creates the KStream instance

❸ KTable prints results to the console

❹ KStream prints results to the console

SIDEBAR **Using default serdes**

In creating the `KTable` and `KStream`, you didn't specify any serdes to use. The same is true with both calls to the `print()` method. You were able to do this because you registered a default serdes in the configuration. like so:

```
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
   Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
   StreamsSerdes.StockTickerSerde().getClass().getName());
```

If you used different types, you'd need to provide serdes in the overloaded methods for reading or writing records.

The takeaway here is that records in a stream with the same keys are updates, not new records in themselves. A stream of updates is the main concept behind the `KTable`, which is the backbone of stateful operations in Kafka Streams.

8.2 KTables are stateful

In the previous example when you created the table with the `StreamsBuilder.table` statement Kafka Streams also creates a `StateStore` for tracking the state and by default it's a persistent store. Since state stores only work with byte arrays for the keys and values you'll need to provide the `Serde` instances so the store can (de)serialize the keys and values. Just as you can provide specific serdes to an event stream with `Consumed` configuration object, you can do the same when creating a `KTable`:

```
builder.table(STOCK_TICKER_TABLE_TOPIC,
              Consumed.with(Serdes.String(),
                            StockTradeSerde()));
```

Now the serdes you've provided with the `Consumed` object get passed along to the state store. There's an additional overloaded version of `StreamsBuilder.table` that accepts a `Materialized` instance as well. This allows you to customize the type of store and provide a name to make it available for querying. We'll discuss interactive queries later in this chapter.

It's also possible to create a `KTable` directly by using the `KStream.toTable` method. Using this method changes the interpretation of the records from events to updates. You can also use the `KTable.toStream` method to convert the update stream into an event stream. We'll talk more about this conversion from update stream to event stream when we discuss the `KTable` API.

The main point here is you are creating a `KTable` directly from a topic, which results in creating a state store.

So far I've talked about how the `KTable` handles inserts and updates, but what about when you need to delete a record? To remove a record from a `KTable` you send send a key-value pair with the value set to `null` and this will act as a tombstone marker ultimately getting removed from the state store and the changelog topic backing the store, in other words it's deleted from the table.

Just like the `KStream`, the `KTable` is spread out over tasks determined by the number of partitions in the underlying source topic, meaning that the records for the table are potentially distributed over separate application instances. We'll see a little later in this chapter an abstraction where all the records are available in a single table.

8.3 The KTable API

The `KTable` API offers similar methods to what you'd see with the `KStream - filter`, `filterNot`, `mapValues`, and `transformValues` (I won't talk about `transformValues` here, but we'll cover it in Processor API chapter later in the book). Executing these methods also follow the fluent pattern, they return a new `KTable` instance.

While the functionality of these methods are very similar as the same methods in the `KStream` API, there are some differences in how they operate. The differences come into play due to the fact that key-value pairs where the value is `null` has delete semantics.

So the delete semantics have the following effects on how the `KTable` operates:

1. If the incoming value is null the processor is not evaluated at all and the key-value with the `null` is forwarded to the new table as a tombstone marker.
2. In the case of the `filter` and `filterNot` methods records that get dropped a tombstone record is forwarded to the new table as a tombstone marker as well.

As an example to follow along with see the `KTableFilterExample` in the `bbejeck.chapter_8` package. It runs a simple `KTable.filter` example where some of the incoming values are `null` as well as filtering out some of the non-null values. But since we've discussed filtering previously, I won't review the example here and I'll leave up to you as an exercise to do on your own.

Since I've already covered stateless operations in a previous chapter and we've discussed the different semantics of the `KTable`, we'll move on at this point to discuss aggregations and joins.

8.4 KTable Aggregations

Aggregations in the `KTable` operate a little differently than the ones we've seen in the `KStream`, so let's dive in with an example to illustrate. Imagine you build an application to track stocks. You're only interested in the latest price for any given symbol, so using a `KTable` makes sense as that is its default behavior. Additionally, you'd like tp keep track of how different market segments are performing. For example, you'd group the stocks of Google, Apple, and Confluent into the tech market segment. So you'll need to perform an aggregation and group different stocks together by the market segment they belong to. Here's what your `KTable` aggregation would look like:

Listing 8.2 Aggregates with A KTable

```
KTable<String, StockAlertProto.StockAlert> stockTable =
                builder.table("stock-alert",
                 Consumed.with(stringSerde, stockAlertSerde));   ❶

stockTable.groupBy((key, value) ->
                    KeyValue.pair(value.getMarketSegment(), value),
                Grouped.with(stringSerde, stockAlertSerde))   ❷
        .aggregate(segmentInitializer,   ❸
                adderAggregator,                  ❹
                subtractorAggregator,             ❺
                Materialized.with(stringSerde, segmentSerde))
        .toStream()

        .to("stock-alert-aggregate",
                Produced.with(stringSerde, segmentSerde));
```

❶ Creating the original `KTable`

❷ Grouping by the market segment also providing Serdes for the repartition via a `Grouped`

❸ Creating the aggregate

❹ Providing the adder `Aggregator`

❺ Providing the subtractor `Aggregator`

Annotation one is where you create the `KTable` and is what you'd expect to see but annotation two you're performing a `groupBy` and updating the key to be the market segment which will force a repartition of the data. Now this makes sense, since the original key is the stock symbol you're not guaranteed that all stocks from a given market segment reside on the same partition.

But this requirement somewhat hides the fact with a `KTable` aggregation you'll ***always*** need to perform a group-by operation. Why is this so? Remember that with a `KTable`, the incoming key is considered a ***primary key***, and just like in a relational database, grouping by the primary-key always results in a single record - hence not much is provided for an aggregation. So you'll need to group records by another field because the combination of the primary-key and the grouped field(s) will yield results suitable for an aggregation. And similar to the `KStream` API, calling the

`KTable.groupBy` method returns an intermediate table - `KGroupedTable` which you'll use to execute the `aggregate` method.

The second difference occurs with annotations four and five. With the `KTable` aggregations, just like with the `KStream` the first parameter you provide is an `Initializer` instance, to provide the default value for the first aggregation. However you then supply ***two aggregators*** one that adds the new value into the aggregation and the other one ***subtracts*** values from the aggregation for the previous entry with the same key. Let's look at an illustration to help make this process clear:

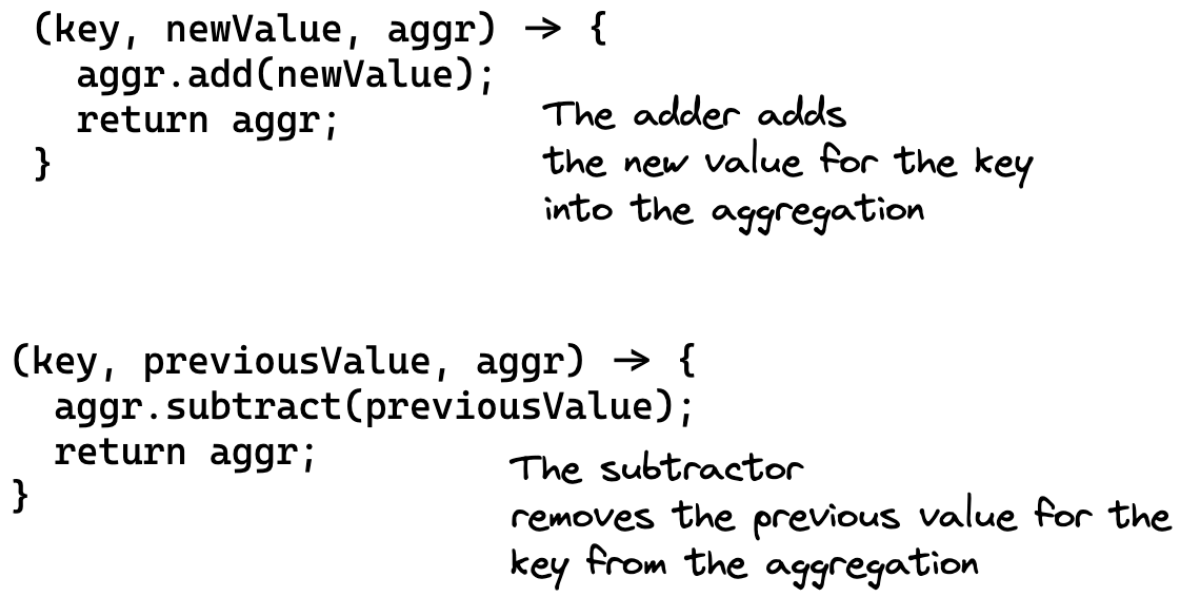


Figure 8.7 KTable Aggregations use an Adder aggregator and a Subtractor aggregator

Here's another way to think about it - if you were to perform the same thing on a relational table, summing the values in the rows created by a grouping, you'd only every get the latest, single value per row created by the grouping. For example the SQL equivalent of this `KTable` aggregation could look something like this:

Listing 8.3 SQL of KTable aggregation

```
SELECT market_segment,
       sum(share_volume) as total_shares,
       sum(share_price * share_volume) as dollar_volume
       FROM stock_alerts
       GROUP BY market_segment;
```

From the SQL perspective, when a new record arrives, the first step is to update the alerts table, then run the aggregation query to get the updated information. This is exactly the process taken by the `KTable`, the new incoming record updates the table for the stock_alerts and it's forwarded to the aggregation. Since you can only have one entry per stock symbol in the roll-up, you add the new record into the aggregation, then remove the previous value for the given symbol.

Consider this example, a record comes in for the ticker symbol CFLT so the `KTable` is updated with new entry. Then the aggregate updates with new entry for CFLT, but since there's already a value for it in the aggregation you must remove it then recalculate the aggregation with the new value.

Now that we've covered how the `KTable` aggregation works, let's take a look at the `Aggregator` instances. But since we've covered them in a previous chapter, let's just take a look at the logic of the adder and subtractor. Even though this is just one example the basic principals will be true for just about any `KTable` aggregation.

Let's start with the adder:

Listing 8.4 KTable adder Aggregator

```
//Some details omitted for clarity

final Aggregator<String,
                 StockAlertProto.StockAlert,
                 SegmentAggregateProto.SegmentAggregate> adderAggregator =
                       (key, newStockAlert, currentAgg) -> {

            long currentShareVolume =
                    newStockAlert.getShareVolume();   ❶
            double currentDollarVolume =
                    newStockAlert.getShareVolume() * newStockAlert.getSharePrice();   ❷

            aggBuilder.setShareVolume(currentAgg.getShareVolume() + currentShareVolume);   ❸
            aggBuilder.setDollarVolume(currentAgg.getDollarVolume() + currentDollarVolume);   ❹
}
```

❶ Extracting the share volume from the current `StockAlert`

❷ Calculating the dollar volume for the current `StockAlert`

❸ Setting the total share volume by adding share volume from the latest `StockAlert` to the current aggregate

❹ Setting the total dollar volume by adding calculated dollar volume to current aggregate

Here the logic is very simple: take the share volume from the latest `StockAlert` and add it to the current aggregate, then do the same with the dollar volume (after calculating it by multiplying the share volume by the share price).

NOTE **Protobuf objects are immutable so when updating values we need to create new instances using a builder that is generated for each unique object.**

Now for the subtractor, you guessed it, you'll simply do the reverse and **subtract** the same values/calculations for the previous record with the same stock ticker symbol in the given market segment. Since the signature is the same I'll only show the calculations:

Listing 8.5 KTable subtractor Aggregator

```
//Some details omitted
long prevShareVolume = prevStockAlert.getShareVolume();
double prevDollarVolume =
      prevStockAlert.getShareVolume() * prevStockAlert.getSharePrice();

aggBuilder.setShareVolume(currentAgg.getShareVolume() - prevShareVolume);   ❶
aggBuilder.setDollarVolume(currentAgg.getDollarVolume() - prevDollarVolume);  ❷
```

❶ Subtracting the share volume from the previous `StockAlert`

❷ Subtracting the dollar volume from the previous `StockAlert`

The logic is straight forward, you're subtracting the values from the `StockAlert` that has been replaced in the aggregate. I've added some logging to the example to demonstrate what is going on and it will be a good idea to look over a portion of that now to nail down what's going on:

Listing 8.6 Logging statements demonstrating the adding and subtracting process of the KTable aggregate

```
Adder                                                                                and
  ❶  : -> key textiles stock alert symbol: "PXLW" share_price: 2.52 share_volume: 4     aggregat
Adder                                                                  ❷              e
     market_segment: "textiles" : <- updated aggregate dollar_volume: 10.08 share_volume: 4
Subtractor: -> key textiles stock alert symbol: "PXLW" share_price: 2.52 share_volume: 4
market_segment: "textiles"
                  and aggregate dollar_volume: 54.57 share_volume: 18  ❸
Subtractor: <- updated aggregate dollar_volume: 44.49 share_volume: 14    ❹

Adder     : -> key textiles stock alert symbol: "PXLW" share_price: 3.39 share_volume: 6
          market_segment: "textiles"                                    ❺
                       and aggregate dollar_volume: 44.49 share_volume: 14
Adder     : <- updated aggregate dollar_volume: 64.83 share_volume: 20   ❻
```

❶ First entry and the aggregate is empty

❷ The aggregate now updates with the values from the PXLW stock statistics

❸ As a result of a new entry for PXLW the aggregation runs the subtractor

❹ Returning the updated aggregation minus previous values for PXLW

❺ The incoming entry for the new PXLW stock alert

❻ Returning the updated aggregation with new values added

By looking at this output excerpt you should be able to clearly see how the `KTable` aggregate works, it keeps only the latest value for each unique combination of the original `KTable` key and the key used to execute the grouping, which is exactly what you'd expect, since you're performing an aggregation over a table with only one entry per primary key.

It's worth noting here that `KTable` API also provides `reduce` and `count` methods which you'll take similar steps. You first perform a `groupBy`, and for the `reduce` provide an adder and

subtractor `Reducer` implementation. I won't cover them here, but there will be examples of both `reduce` and `count` in the source code for the book.

This wraps up our coverage of the `KTable` API but before we move on to more advanced stateful subjects, I'd like to go over another table abstraction offered by Kafka Streams, the `GlobalKTable`.

8.5 GlobalKTable

I alluded to the `GlobalKTable` earlier in the chapter when we discussed that the `KTable` is partitioned, hence its distributed out among Kafka Streams application instances (with the same application id of course). What makes the `GlobalKTable` unique is the fact that it's not partitioned, it fully consumes the underlying source topic. This means there is a full copy of all records in the table for all application instances.

Let's look at an illustration to help make this clear:

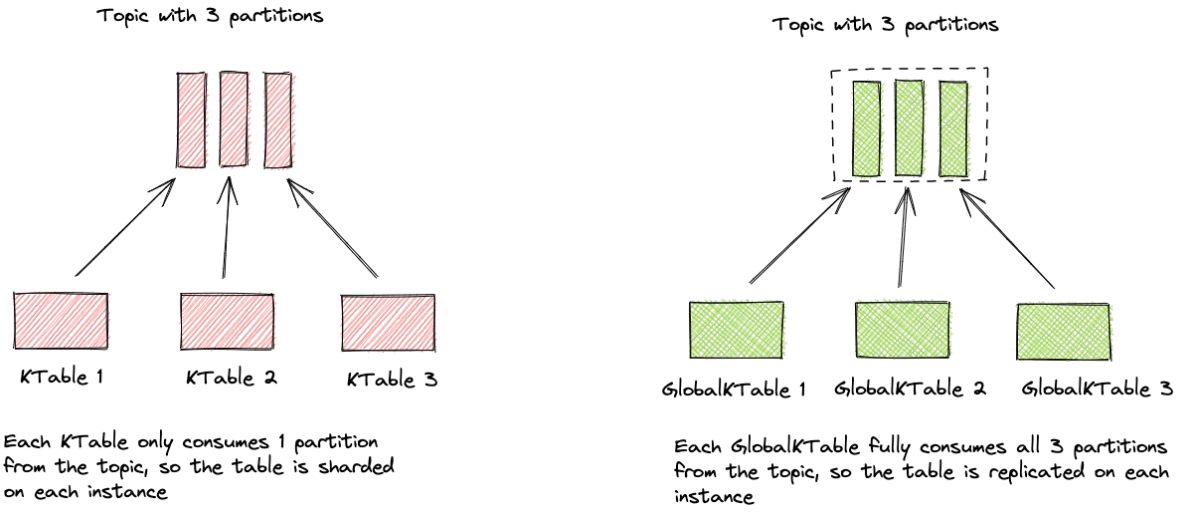


Figure 8.8 GlobalKTable contains all records in a topic on each application instance

As you can see the source topic for the `KTable` has three partitions and with three application instances, each `KTable` is responsible for one partition of data. But the `GlobalKTable` has the **full copy** of its three-partition source topic on each instance. Kafka Streams materializes the `GlobalKTable` on local disk in a `KeyValueStore`, but there is no changelog topic crated for this store as the source topic serves as the backup for recovery as well.

Here's how you'd create one in your application:

Listing 8.7 Creating a GlobalKTable

```
StreamsBuilder builder = new StreamsBuilder();
GlobalKTable<String, String> globalTable =
  builder.globalTable("topic",
                      Consumed.with(Serdes.String(),
                                    Serdes.String()));
```

The interesting thing to note about the `GlobalKTable` is that it doesn't offer an API. So I'm sure you're asking yourself "why would I ever want to use one?". The answer to that question will come in our next section when we discuss joins with the `KTable`.

8.6 KTable Joins

In the previous chapter you learned about performing joins with two `KStream` objects, but you can also perform `KStream-KTable`, `KStream-GlobalKTable`, and `KTable-KTable` joins. Why would you want to join a stream and a table? Stream-table joins represent an excellent opportunity to create an ***enriched*** event with additional information. For the stream-table and table-table joins, both sides need to be co-partitioned - meaning the underlying source topics must have the same number of partitions. If that is not the case then you'll need to do a `repartition` operation to achieve the co-partitioning. Since the `GlobalKTable` has a full copy of the records there isn't a co-partitioning requirement for stream-global table joins.

For example, let's say you have an event stream of user activity on a website, a clickstream, but you also maintain a table of current users logged into the system. The clickstream event object only contains a user-id and the link to the visited page but you'd like more information. Well you can join the clickstream against the user table and you have much more useful information about the usage patterns of your site - in real time. Here's an example to work through:

Listing 8.8 Stream-Table join to enrich the event stream

```
KStream<String, ClickEventProto.ClickEvent> clickEventKStream =
                builder.stream("click-events",
                        Consumed.with(stringSerde, clickEventSerde));

KTable<String, UserProto.User> userTable =
        builder.table("users",
                Consumed.with(stringSerde, userSerde));

clickEventKStream.join(userTable, clickEventJoiner)
        .peek(printKV("stream-table-join"))
        .to("stream-table-join",
                Produced.with(stringSerde, stringSerde));
```

Looking at the code in this example, you first create the click-event stream then a table of logged in users. In this case we'll assume the stream has the user-id for the key and the user tables' primary key is the user-id as well, so we can easily perform a join between them as is. From there you call the `join` method of the stream passing in the table as a parameter.

8.7 Stream-Table join details

At this point I'd like to cover a few of differences with the stream-table joins from the stream-stream join. First of all stream table joins aren't reciprocal - the stream is always on the left or calling side and the table is always on the right side. Secondly, there is no window that the timestamps of the records need to fit into for a join to occur, which dove tails into the third difference; only updates on the stream produce a join result.

In other words, it's only newly arriving records on the stream that trigger a join, new records to the table update the value for the key in table, but don't result in a join result. To capture the join result you provide a `ValueJoiner` object that accepts the value from both sides and produces a new value which can be the same type of either side or a new type altogether. With stream-table joins you can perform an inner (equi) join or a left-outer join (demonstrated here).

8.8 Table-Table join details

Next, let's talk about table-table joins. Joins between the two tables are pretty much the same that you've seen so far with join functionality. Joins between two tables is similar to stream-stream joins, except there is now windowing, but updates to either side will trigger a join result. You provide a `ValueJoiner` instance that calculates the join results and can return an arbitrary type. Also, the constraint that the source topic for both sides have the same number partitions applies here as well.

But there's something extra offered for table-table joins. Let's say you have two `KTables` you'd like to join; users and purchase transactions, but the primary key for the users is user-id and the primary key for transactions is a transaction-id, although the transaction object contains the user-id. Usually a situation like this would require some sort of workaround, but not now, as the `KTable` API offers a foreign-key join, so you can easily join the two tables. To use the foreign-key join you use the signature of the `KTable.join` method that looks like this:

Listing 8.9 KTable Foreign Key join

```
userTable.join(transactionTable,     ❶
               foreignKeyExtractor,  ❷
               joiner);              ❸
```

❶ Other table or right side of the join

❷ The foreign key extractor function

❸ The ValueJoiner parameter

Setting up the foreign key join is done like any other table-table join except that you provide an additional parameter a `java.util.Function` object, that extracts the key used to complete the join. Specifically, the function extracts the key from the left-side value to correspond with the

key of the right side table. If the function returns `null` then no join occurs.

Inner and left-outer joins support joining by a foreign key. As with primary-key table joins, an update on either side will trigger a potential join result. The inner workings of the foreign-key join in Kafka Streams is involved and I won't go into those details, but if you are interested in more details then I suggest reading KIP-213 cwiki.apache.org/confluence/display/KAFKA/KIP-213+Support+non-key+joining+in+KTable.

> **NOTE** **There isn't a corresponding explicit foreign-key joins available in the `KStream` API and that is intentional. The `KStream` API offers methods `map` and `selectKey` where you can easily change the key of a stream to facilitate a join.**

8.9 Stream-GlobaTable join details

The final table join for us to discuss is the stream-global table join. There a few differences with the stream-global table we should cover. First it's the only join in Kafka Streams that does not require co-partitioning. Remember the `GlobalKTable` is not sharded like the `KTable` is, a partition per task, but instead contains all the data of its source topic. So even if the partitions of the stream and the global-table don't match, if the key is present in the global table, a join result will occur.

The semantics of a global table join are different as well. Kafka Streams process incoming `KTable` records along with every other incoming records by timestamps on the records, so with a stream-table join the records are aligned by timestamps. But with a `GlobalKTable`, updates are simply applied when records are available, it's done separately from the other components of the Kafka Streams application.

Having said that, there are some key advantages of using a `GlobalKTable`. In addition to having all records on each instance, stream-global tables support foreign key joins, the key of the stream does not have to match the key of the global table. Let's look at a quick example:

Listing 8.10 KStream GlobalTable Join example

```
userStream.join(detailsGlobalTable,    ❶
                keySelector,    ❷
                valueJoiner);    ❸
```

❶ The `GlobalTable` to join against

❷ A key selector to perform the join

❸ The `ValueJoiner` instance to compute the result

So with the `KStream-GlobalKTable` join the second parameter is a `KeyValueMapper` that takes

the key and value of the stream and creates the key used to join against the global table (in this way it is similar to the `KTable` foreign-key join). It's worth noting that the result of the join will have the key of the stream regardless of the `GlobalTable` key or what the supplied function returns.

Of course every decision involves some sort of trade-off. Using a `GlobalKTable` means using more local disk space and a greater load on the broker since the data is not sharded, but the entire topic is consumed. The stream-global table join is not reciprocal, the `KStream` is always on the calling or left-side of the join. Additionally, only updates on the stream produce a join result, a new record for the `GlobalKTable` only updates the internal state of the table. Finally, either inner or left-outer joins are available.

So what's best to use when joining with a `KStream` a `KTable` or `GlobalKTable`? That's a tough question to answer as there are no hard guidelines to follow. But a good rule of thumb would be to use a `GlobalKTable` for cases where you have fairly static lookup data you want to join with a stream. If the data in your table is large strongly consider using a `KTable` since it will end up being sharded across multiple instances.

At this point, we've covered the different joins available on both the `KTable` and `GlobalKTable`. There's more to cover with tables, specifically viewing the contents of the tables with interactive queries and suppressing output from a table (`KTable` only) to achieve a single final result. We'll cover interactive queries a little later in the chapter. But we'll get to suppression in our next section when we discuss windowing.

8.10 Windowing

So far you've learned about aggregations on both the `KStream` and `KTable`. While they both produce an aggregation, how they are calculated is bit different. Since the `KStream` is an event stream where all records are unrelated, the aggregations will continue to grow over time. But with either case the results produced are cumulative over time. Maybe not as much for `KTable` aggregations, but that's definitely the case for `KStream` aggregations.

There's good chance that you'll want to see results within given time intervals. For example, what's the average reading of a IoT temperature sensor every 15 minutes? To capture aggregations in slices of time, you'll want to use windowing. In Kafka Streams windowing an aggregation means that you'll get results in distinct blocks of time as defined by the size of the window. There are four window types available:

1. Hopping - Windows with a fixed size by time and the advance time is less than the window size resulting in overlapping windows. As a result, results may be included in more than one window. You'd use a hopping window when a result from the previous window is useful for comparison such as fraud detection.
2. Tumbling - A special case of a hopping window where the advance time is the same as

the window size, so each window contains unique results. A good use case for tumbling windows is inventory tracking because you only want the unique amount of items sold per window.

3. Session - A different type of window where its size is not based on time but on behavior instead. Session windows define an inactivity-gap and as long as new events arrive within the defined gap, the window grows in size. But once reaching the inactivity gap, new events will go into a new window. Session windows are great for tracking behavior becuase the windows are determined by activity.
4. Sliding - Sliding windows are fixed time windows, but like the session window, they can continue to grow in size because they are based on behavior as well. Sliding windows specify the maximum difference between timestamps of incoming records for inclusion in the window.

What we'll do next is present some examples and illustrations demonstrating how to add windowing to aggregations and more detail on how they work.

The first example will show how to implement a hopping window on a simple count application. Although the examples will be simple to make learning easier to absorb, the simplicity of the aggregation doesn't matter. You can apply windowing to any aggregation.

Listing 8.11 Setting up hopping windows for an aggregation

```
//Some details omitted for clarity
  countStream.groupByKey()
   .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1))   ❶
            .advanceBy(Duration.ofSeconds(10)))       ❷
   .count(Materialized.as("hopping-window-counting-store"))
   .toStream()            ❸
   .map((windowedKey, value) -> KeyValue.pair(windowedKey.key(), value))   ❹
   .to("counting-output", Produced.with(stringSerde, longSerde));
```

❶ Setting up the window (hopping) with a size of one minute

❷ Establishing the advance by time

❸ Convert the KTable to a KStream

❹ Mapping the Windowed key back to the original inner key

In this example you're using a hopping window with a size of one minute and an advance of 10 seconds. Let's review the code here to understand what your specifying. The `windowedBy` call at annotation one sets up the windowing and specifies the size of the window. You no doubt noticed the method name `ofSizeWithNoGrace`, so what does the `WithNoGrace` mean (other than dribbling your dinner down the front of your shirt!)? Grace is a concept in Kafka Streams that allows you to define how you want to handle out-of-order records, but I'd like to defer that conversation until we've finished discussing the hopping window.

At annotation two, you use the `advanceBy` call which determines the interval that the aggregation will occur. Since the `advanceBy` is less than the window size, it is a hopping window. At annotation three we convert the `KTable` to a `KStream` as we need to convert from

the update stream to an event stream so we can perform some operations on each result.

At annotation four, you use a `map` processor to create a new `KeyValue` object, specifically creating a new key. This new key is actually the original key for the pair when it entered the aggregation. When you perform a windowed aggregation on a `KStream` the key going into the `KTable` gets "upgraded" to a `Windowed` key. The `Windowed` class contains the original key from the record in the aggregation and a reference to the specific `Window` the record belongs to.

Processing the key-values coming from a windowed aggregation presents you with a choice; keep the key as is, or use the `map` processor to set the key the original one. Most of the time it will make sense for you to revert to the original key, but there could be times where you want to keep the `Windowed` one, there really isn't any hard rules here. In the source code there's an example of using both approaches.

Getting back to how a hopping window operates, here's an illustration depicting the action:

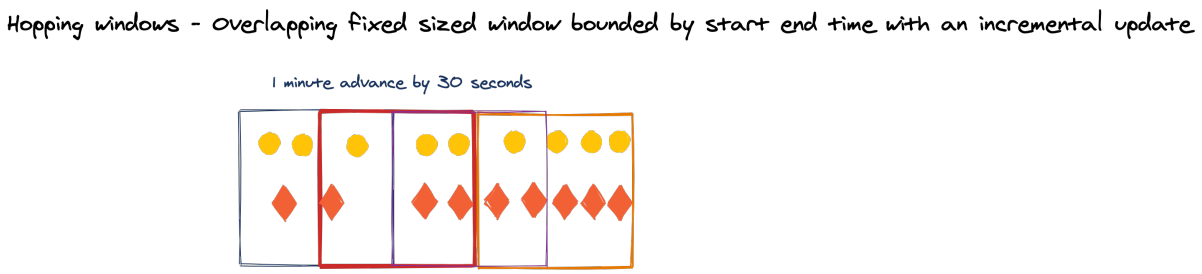


Figure 8.9 Hopping windows hop to the right by the given advance time which

From looking at the illustration, the first record into the aggregation opens the window with the time of its timestamp. Every ten seconds, again based on record timestamps, the aggregation performs its calculation, a simple count in this case. So a hopping window has a fixed size where it collects records, but it doesn't wait the entire time of the window size to perform the aggregation; it does so at intervals within the window corresponding to the advance time. Since the aggregation occurs within the window time it may contain some records from the previous evaluation.

You now have learned about the hopping window, but so far we've assumed that records allways arrive in order. Suppose some of your records don't arrive in order and you'd still like to include them (up to a point) in your count, what whould you do to handle that? Now's a good time to circle back to the concept of out-of-order and grace I mentioned previously.

8.11 Out order records and grace

It will be easier to grasp the concept of grace if we first describe what an out-of-order record is. You've learned in a previous chapter the `KafkaProducer` will set the timestamp on a record. In this case timestamps on the records in Kafka Streams should always increase. But in some cases you may want to use a timestamp embedded in the value, and in that scenario you can't be guaranteed that those timestamps always increase. The following illustration demonstrates this concept:

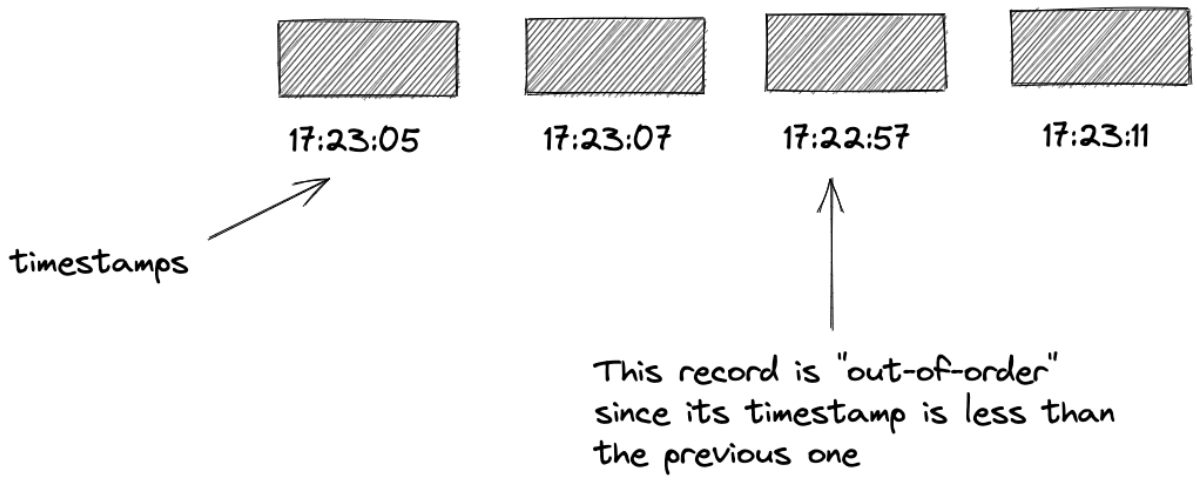


Figure 8.10 Out of order records didn't arrive in the correct sequence

So an out-of-order record is simply one where the timestamp is less than the previous one.

Now moving on to grace, it is the amount of time after a window is considered closed that you're willing to allow an out-of-order record into the aggregation. Here's an illustration demonstrating the concept:

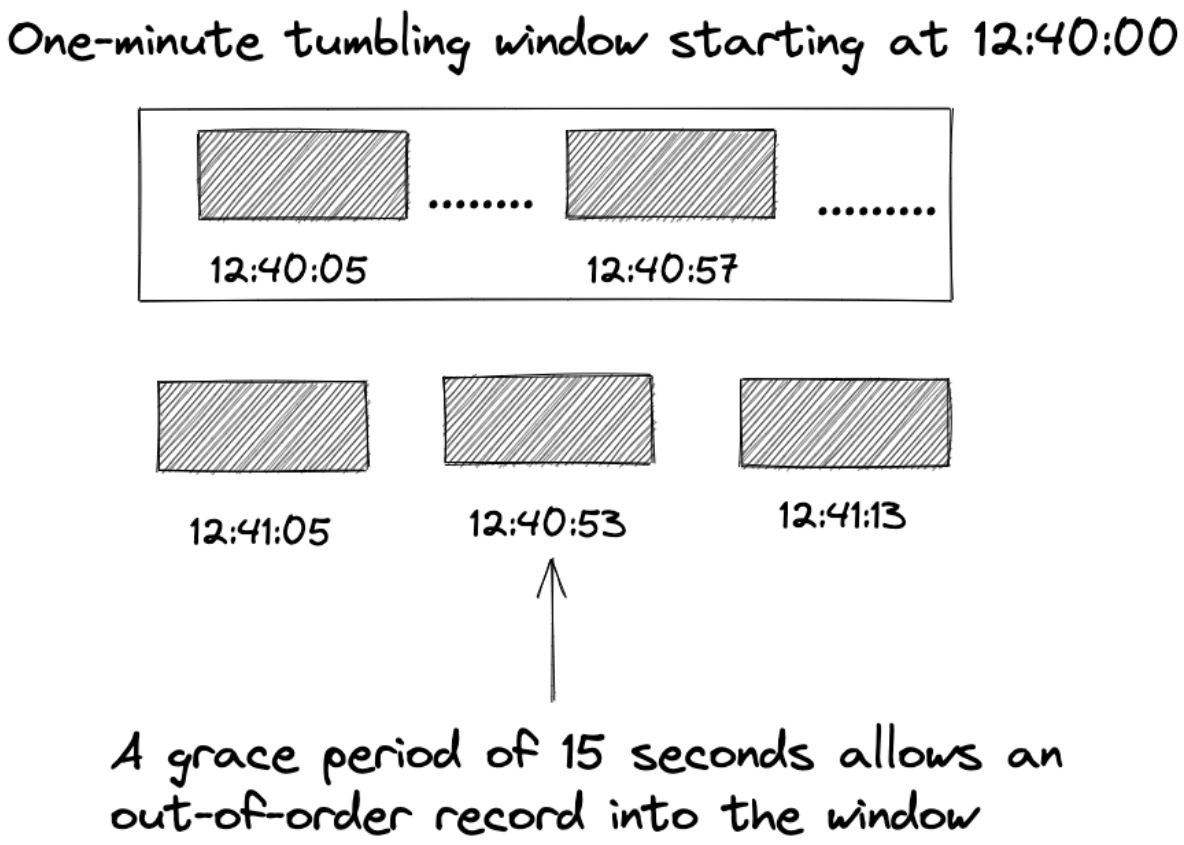


Figure 8.11 Grace is the amount of time you'll allow out-of-order records into a window after its configured close time

So from looking at the illustration, grace allows records into an aggregation that ***would have been included were they on-time***, and allows for a more accurate calculation. Once the grace period has expired, any out-of-order records are considered late and dropped. In the case of our example above, since we're not providing a grace period, when the window closes Kafka Streams drops any out-of-order records.

We'll revisit windowing and grace periods when we discuss timestamps later in the chapter, but for now it's enough to understand that timestamps on the records drive the windowing behavior and grace is a way to ensure you're getting the most accurate calculations by including records that arrive out of order.

8.12 Tumbling windows

Now let's get back to our discussion of window types and move on to the tumbling window. A tumbling window is actually a special case of a hopping window where the advance of the window is the same as its size. Since it advances the size of the window the calculation of the window contains no duplicate results. Let's take a look of an illustration showing the tumbling window in action:

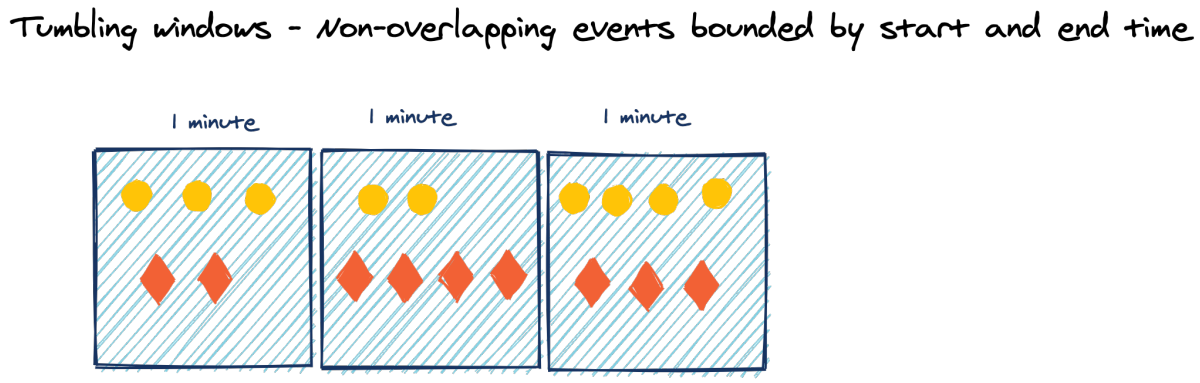


Figure 8.12 Tumbling windows move to the right by an advance equal to the size of the window

So you can see here how a tumbling window gets its name - thinking of the window as square when it's time to advance it "tumbles" into an entirely new space, and as a consequence it's guaranteed to not have any overlapping results. Specifying to use a tumbling window is easy, you simple leave off the `advanceBy` method and the size you set automatically becomes the advance time. Here's the code for setting up a tumbling window aggregation:

Listing 8.12 Setting up tumbling windows for an aggregation

```
//Some details omitted for clarity
countStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeAndGrace(Duration.ofMinutes(1)   ❶
                           ,Duration.ofSeconds(30)))   ❷
    .count(Materialized.as("Tumbling-window-counting-store"))
```

❶ Setting up the tumbling window of one minute

❷ Using 30 seconds for the grace period

From looking at annotation one using a tumbling window is simply a matter not setting the advance time. Also I'd like to point out that in this example you are using a grace period of thirty seconds, so there's a second `Duration` parameter passed into the `TimeWindows.ofSizeAndGrace` method. Note that I'm only showing the required code for tumbling windows with a grace period, but the source code contains a full runnable example.

The choice of using a tumbling or a hopping window depends entirely on your use case. A hopping window gives you finer grained results with potentially overlapping results, but the tumbling window result are a little more course-grained but will not contain any overlapping records. One thing to keep in mind is that a hopping window is re-evaluated more frequently, so how often you want to observe the windowed results is one potential determinant.

8.13 Session windows

Next up in our tour of window type is the session window. The session window differs from hopping and tumbling in that it doesn't have fixed size. Instead you specify an inactivity gap; if there's no new records within the gap time, Kafka Streams closes the window. Any subsequent records coming in after the inactivity gap result in creating a new session. Otherwise it will continue to grow in size Looking at a visual pictorial of a session window is in order to fully understand how it works:

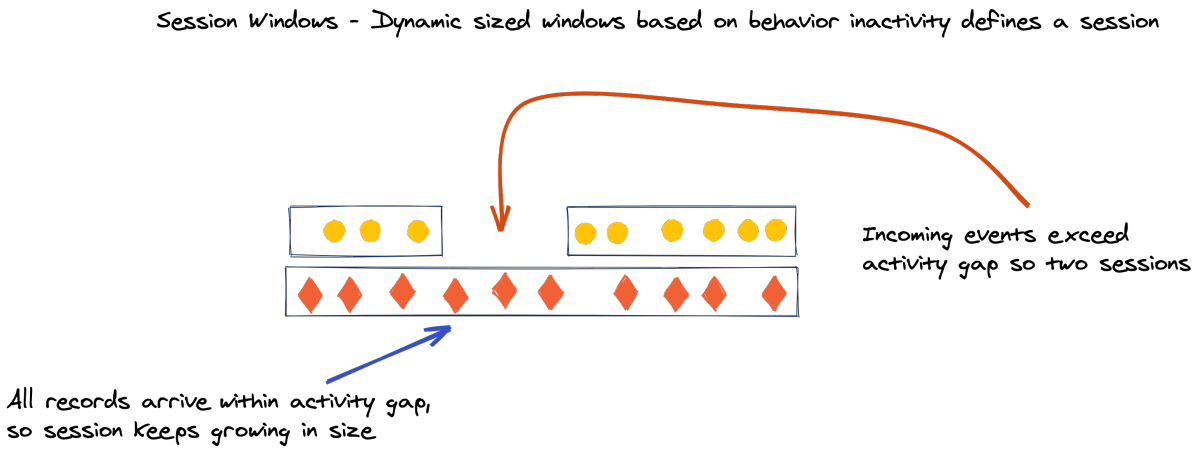


Figure 8.13 Session windows continue to grow unless no new records arrive before the inactivity gap expires

By looking at the illustration you can see that a session window is driven by behavior, unlike the hopping or tumbling window which are governed by time. Let's take a look at an example of using a session window. As before I'm only going to show the essential part here, the source code contains the full, runnable example.

Listing 8.13 Setting up the session window

```
//Some details omitted

countStream.groupByKey()
 .windowedBy(SessionWindows.ofInactivityGapAndGrace(Duration.ofMinutes(1),   ❶
                     Duration.ofSeconds(30)))    ❷
 .count(Materialized.as("Session-window-counting-store"))
```

❶ Using a session window for the aggregation

❷ Specifying a grace period

So to use sessions with your aggregation is to use a `SessionWindows` factory method. In this case you specify an inactivity period of one minute and you include a grace period as well. The grace period for session window works in the similar manner, it provides a time for Kafka Streams to include out-of-order records arriving after the inactivity period passes. As with the other window implementations, there's also a method you can use to specify no grace period.

The choice to use a session window vs. hopping/tumbling is more clear cut, it's best suited where you are tracking behavior. For example think of a user on a web application, as long as they are active on the site you'll want to get calculate the aggregation and it's impossible to know how long that could be.

8.14 Sliding windows

We're now on to the last window type to cover, `SlidingWindows`. The sliding window is a fixed-size window, but instead of specifying the size, it's the difference between timestamps that determine if a record is added to the window. So it's a combination of time-based window, but To fully understand how a `SlidingWindow` operates, take a look at the following illustration:

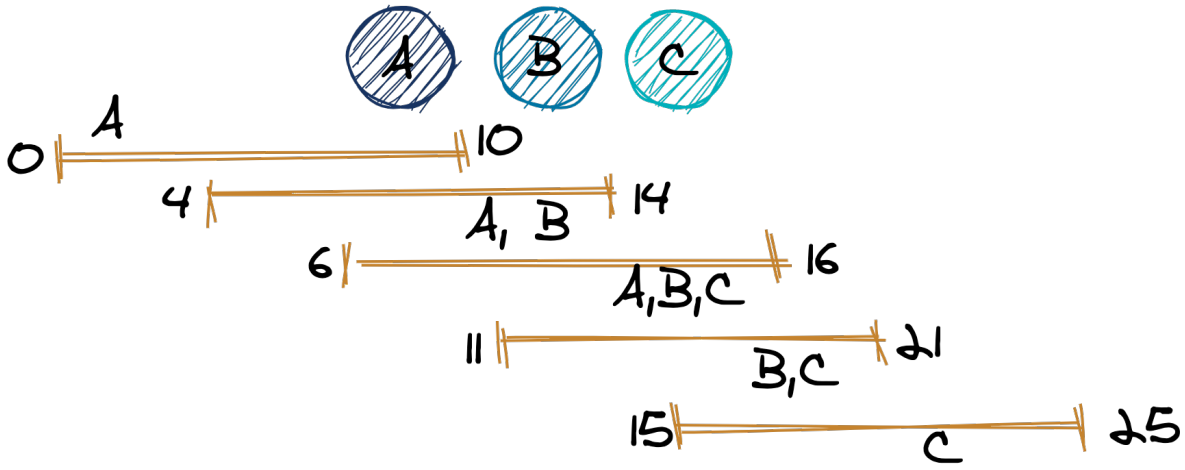


Figure 8.14 Sliding windows are fixed-size and slide along the time-axis

As you can see from the diagram, two records are in the same window if the difference in the value of their timestamps falls within the window size. So as the `SlidingWindow` slides along records may end up in several overlapping calculations, but each window will contain a unique set of records. Another way to look at the `SlidingWindow` is that as it moves along the time-axis, records come into the window and others fall out on continual basis.

Here's how you'd set up a sliding window for an aggregation:

Listing 8.14 Setting up the sliding window

```
//Some details omitted

 countStream.groupByKey()
    .windowedBy(SlidingWindows.ofTimeDifferenceWithNoGrace(
            Duration.ofSeconds(30)))   ❶
    .count(Materialized.as("Sliding-window-counting-store"))
```

❶ Specifying a sliding window with time difference of 30 seconds

As with the all the windowed options we've seen so far, it's simply a matter of providing a factory method for the desired windowing functionality. In this case, you've set the time difference to thirty seconds with no grace period. Just like the other windowing options, you could specify a grace period as well with the `SlidingWindows.ofTimeDifferenceWithGrace` method.

The determining factor to go with a sliding window over a hopping or tumbling window is a matter of how fine-grained of a calculation is desired. When you need to generate a continual running average or sum is a great use-case for the `SlidingWindow`.

You could achieve similar behavior with a `HoppingWindow` by using a small advance interval. But this approach will result in poor performance because the hopping windows will create redundant windows and performing aggregation operations over them is inefficient. Compared to the `SlidingWindow` that only creates windows containing distinct items so the calculations are more efficient.

This wraps up our coverage of the different windowing types that Kafka Streams provides, but before we move on to another section we will cover one more feature that is available for all windows. As records flow into the windowed aggregation processor, Kafka Streams continually updates the aggregation with the new records. Kafka Streams updates the `KTable` with the new aggregation, and it forwards the previous aggregation results to downstream operators.

Remember that Kafka Streams uses caching for stateful operations, so every update doesn't flow downstream. It's only on cache flush or a commit that the updates make it downstream. But depending on the size of your window, this means that you'll get partial results of your windowing operations until the window closes. In many cases receiving a constant flow of fresh calculations is desired.

But in some cases, you may want to have a single, **final** result forwarded downstream from a windowed aggregation. For example, consider a case where your application is tracking IoT sensor readings with a count of temperature readings that exceed a given threshold over the past 30 minutes. If you find a temperature breach, you'll want to send an alert. But with regular updates, you'll have to provide extra logic to determine if the result is an intermediate or final one. Kafka Streams provides an elegant solution to this situation, the ability to suppress intermediate results.

8.15 Suppression

For stateless operations the behavior of always forwarding a result is expected in the nature of a stream processing system. But sometimes for a windowed operation it's desirable for a final result when the window closes. For example, take the case of the tumbling window example above, instead of incremental results, you want a single final count.

NOTE **Final results are only available for windowed operations. With an event streaming application like Kafka Streams, the number of incoming records is infinite, so there's never a point we can consider a record final. But since a windowed aggregation represents a discrete point in time, the available record when the window closes can be considered final.**

So far you've learned about the different windowing operations available, but they all yield intermediate results, now let's suppose you only want the final result from the window. For that you'd use the `KTable.suppress` operation.

The `KTable.suppress` method takes a `Suppressed` configuration object which allows you to configure the suppression in two ways:

1. Strict - results are buffered by time and the buffering is strictly enforced by never emitting a result early until the time bound is met
2. Eager - results are buffered by size (number of bytes) or by number of records and when these conditions are met, results are emitted downstream. This will reduce the number of downstream results, but doesn't guarantee a final one.

So you have two choices - the strict approach which guarantees a final result or the eager one which could produce a final result, but also has the likelihood of emitting a few intermediate results as well. The trade-off to make can be thought of this way - with strict buffering, the size of the buffer doesn't have any bounds, so the possibility of getting an `OutOfMemory` (OOM) exists, but with eager buffering you'll never hit an OOM exception, but you could end up with multiple results. While the possibility of incurring an OOM may sound extreme, if you have feel the buffer won't get that large or you have a sufficiently large heap available then using the strict configuration should be OK.

NOTE **The possibility of an OOM is not as harsh as it seems at first glance. All Java applications that use a data-structures in-memory, List, Set or Map have the potential for causing an OOM if you continually add to them. To use them effectively requires a balance of knowledge between the incoming data and the amount of heap you have available.**

Let's take a look now at how at an example of using suppression.

Listing 8.15 Setting up suppression on a KStream aggregation

```
countStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))   ❶
    .count(Materialized.as("Tumbling-window-suppressed-counting-store"))
    .suppress(untilWindowCloses(unbounded()))   ❷
```

❶ Creating a one-minute tumbling window

❷ Suppressing results until the window closes with an unbounded configuration

So setting up suppression is as easy as adding one line of code, which you can see at annotation two. In this case you're suppressing all output until the window closes along with an unbounded buffering of records. For testing scenarios this is an acceptable configuration, but if running with such a configuration in a production setting gives you pause, let's quick show two alternative settings.

First you can configure the final result with a maximum number records or bytes, then if the constraint is violated, you can have a graceful shutdown:

Listing 8.16 Setting up suppression for final result controlling the potential shutdown

```
.suppress(untilWindowCloses(maxRecords(10_000)    ❶
                            .shutDownWhenFull())  ❷
```

❶ Setting max records to 10K

❷ Specifying to shutdown if limit is reached

Here you're specifying to go with an unbounded window, but you'd rather have a graceful shutdown should the buffer start to grow beyond what you feel is a reasonable amount. So in this case you specify the maximum number of records is 10K and should the buffering exceed that number, the application will shut down gracefully.

Note that we technically could have used a `shutDownWhenFull` with our original suppression example, but the default limit is `LONG.MAX_VALUE`, so in practice most likely that you'd get an OOM exception before reaching that size constraint. With this change you're favoring shutting down before emitting a possible non-final result.

On the other hand, if you'd rather trade-off a possible non-final result over shutting down you could use a configuration like this:

Listing 8.17 Using suppression emulating a final result with a possible early result instead of shutting down

```
.suppress(untilTimeLimit(Duration.ofMinutes(1),    ❶
                         maxRecords(1000)          ❷
                        .emitEarlyWhenFull()))     ❸
```

❶ Setting time limit of one hour before sending result downstream

❷ Specifying to buffer a maximum of 1K records

❸ Take the action of emitting a record if the maximum number of buffered records is reached

With this example, you've set the time limit to match the size of the window (plus any grace

period) so you're reasonable sure to get a final result, but you've set the maximum size of the buffer, and if the number of records reaches that size, the processor will forward a record regardless if the time limit is reached or not. One thing to bear in mind is if you want to set the time limit to correspond to the window closing, you need to include the grace period, if any, as well in the time limit.

This wraps up our discussion on suppression of aggregations in Kafka Steams. Even though the examples in the suppression section only demonstrated using the `KStream` and windowed aggregations, you could apply the same principal to non-windowed `KTable` aggregations by using the time-limit API of suppression.

Now let's move on to the last section of this chapter, timestamps in Kafka Streams.

8.16 Timestamps in Kafka Streams

Earlier in the book, we discussed timestamps in Kafka records. In this section, we'll discuss the use of timestamps in Kafka Streams. Timestamps play a role in key areas of Kafka Streams functionality:

- Joining streams
- Updating a changelog (`KTable` API)
- Deciding when the `Processor.punctuate()` method is triggered (Processor API)
- Window behavior

With stream processing in general, you can group timestamps into three categories, as shown in figure 8.10:

- *Event time* — A timestamp set when the event occurred, usually embedded in the object used to represent the event. For our purposes, we'll consider the timestamp set when the `ProducerRecord` is created as the event time as well.
- *Ingestion time* — A timestamp set when the data first enters the data processing pipeline. You can consider the timestamp set by the Kafka broker (assuming a configuration setting of `LogAppendTime`) to be ingestion time.
- *Processing time* — A timestamp set when the data or event record first starts to flow through a processing pipeline.

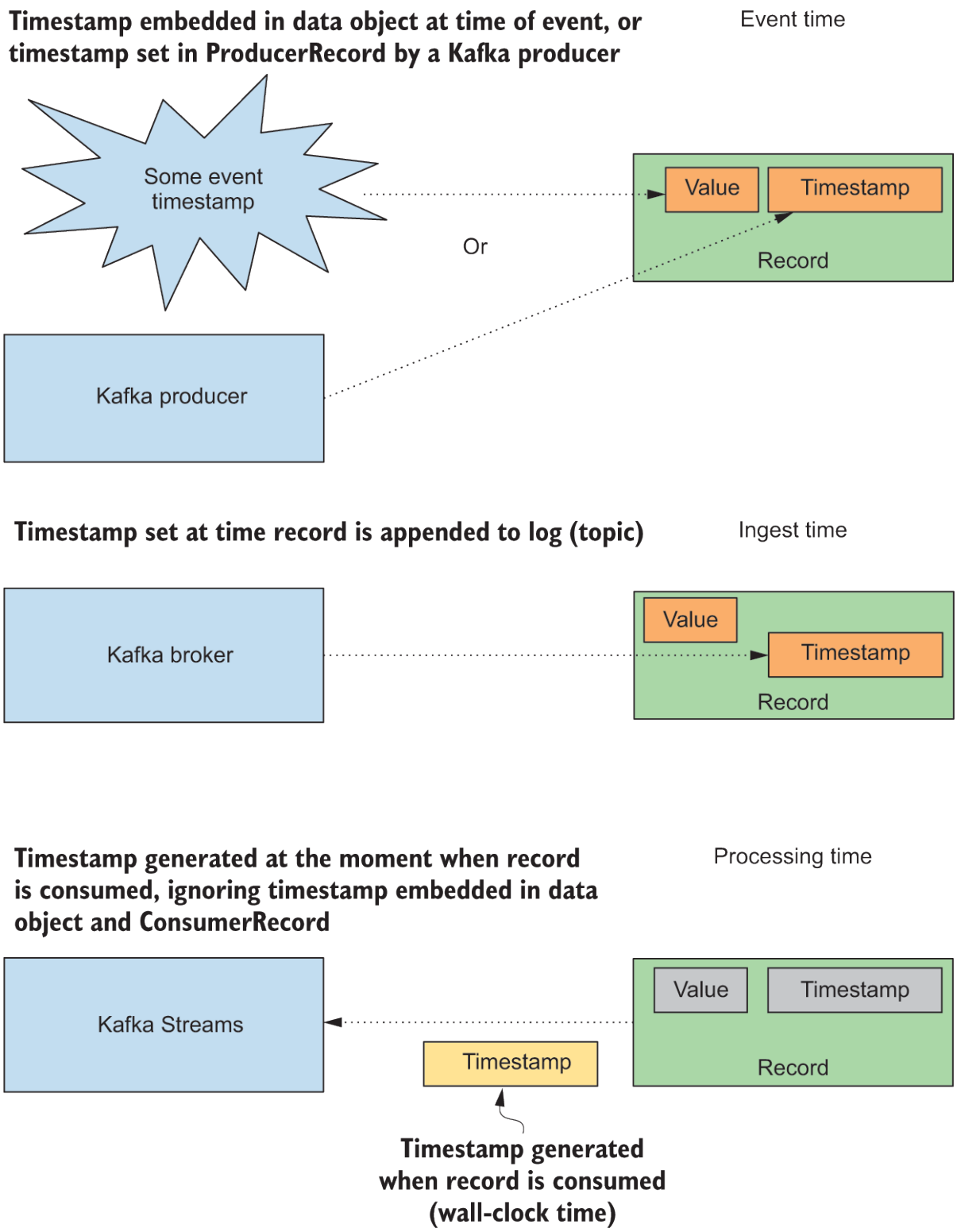


Figure 8.15 There are three categories of timestamps in Kafka Streams: event time, ingestion time, and processing time.

You'll see in this section how the Kafka Streams by using a `TimestampExtractor`, gives you the ability to chose which timestamp semantics you want to support.

> **NOTE** So far, we've had an implicit assumption that clients and brokers are located in the same time zone, but that might not always be the case. When using timestamps, it's safest to normalize the times using the UTC time zone, eliminating any confusion over which brokers and clients are using which time zones.

In most cases using event-time semantics, the timestamp placed in the metadata by the `ProducerRecord` is sufficient. But there may be cases when you have different needs. Consider these examples:

- You're sending messages to Kafka with events that have timestamps recorded in the message objects. There's some lag time in when these event objects are made available to the Kafka producer, so you want to consider only the embedded timestamp.
- You want to consider the time when your Kafka Streams application processes records as opposed to using the timestamps of the records.

8.17 The TimestampExtractor

To enable different processing semantics, Kafka Stream provides a `TimestampExtractor` interface with one abstract and four concrete implementations. If you need to work with timestamps embedded in the record values, you'll need to create a custom `TimestampExtractor` implementation. Let's briefly look at the included implementations and implement a custom `TimestampExtractor`.

Almost all of the provided `TimestampExtractor` implementations work with timestamps set by the producer or broker in the message metadata, thus providing either event-time processing semantics (timestamp set by the producer) or log-append-time processing semantics (timestamp set by the broker). Figure 4.19 demonstrates pulling the timestamp from the `ConsumerRecord` object.

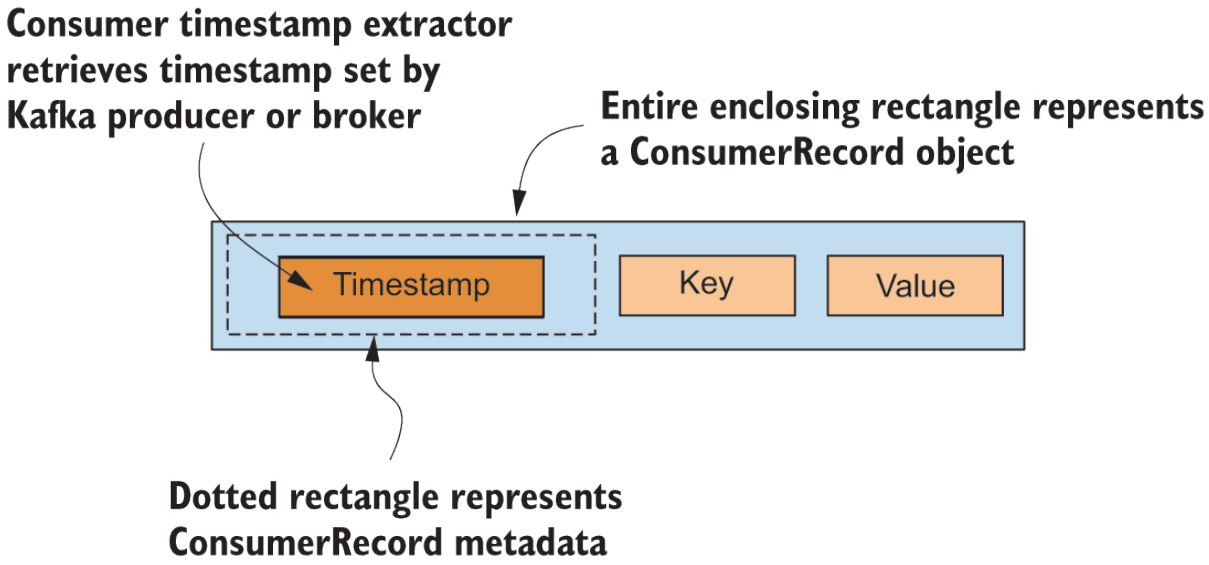


Figure 8.16 Timestamps in the `ConsumerRecord` object: either the producer or broker set this timestamp, depending on your configuration.

Although you're assuming the default configuration setting of `CreateTime` for the timestamp, bear in mind that if you were to use `LogAppendTime`, this would return the timestamp value for when the Kafka broker appended the record to the log. `ExtractRecordMetadataTimestamp` is an abstract class that provides the core functionality for extracting the metadata timestamp from the `ConsumerRecord`. Most of the concrete implementations extend this class. Implementors override the abstract method, `ExtractRecordMetadataTimestamp.onInvalidTimestamp`, to handle invalid timestamps (when the timestamp is less than 0).

Here's a list of classes that extend the `ExtractRecordMetadataTimestamp` class:

- `FailOnInvalidTimestamp` — Throws an exception in the case of an invalid timestamp.
- `LogAndSkipOnInvalidTimestamp` — Returns the invalid timestamp and logs a warning message that the record will be discarded due to the invalid timestamp.
- `UsePreviousTimeOnInvalidTimestamp` — In the case of an invalid timestamp, the last valid extracted timestamp is returned.

We've covered the event-time timestamp extractors, but there's one more provided timestamp extractor to cover.

8.18 WallclockTimestampExtractor

`WallclockTimestampExtractor` provides process-time semantics and doesn't extract any timestamps. Instead, it returns the time in milliseconds by calling the `System.currentTimeMillis()` method. You'd use the `WallclockTimestampExtractor` when you need processing time semantics.

That's it for the provided timestamp extractors. Next, we'll look at how you can create a custom

version.

8.19 Custom TimestampExtractor

To work with timestamps (or calculate one) in the value object from the `ConsumerRecord`, you'll need a custom extractor that implements the `TimestampExtractor` interface. For example, let's say you are working with IoT sensors and part of the information is the exact time of the sensor reading. It's important for your calculations to have the precise timestamp, so you'll want to use the one embedded in the record sent to Kafka and not the one set by the producer.

The figure here depicts using the timestamp embedded in the value object versus one set by Kafka (either producer or broker).

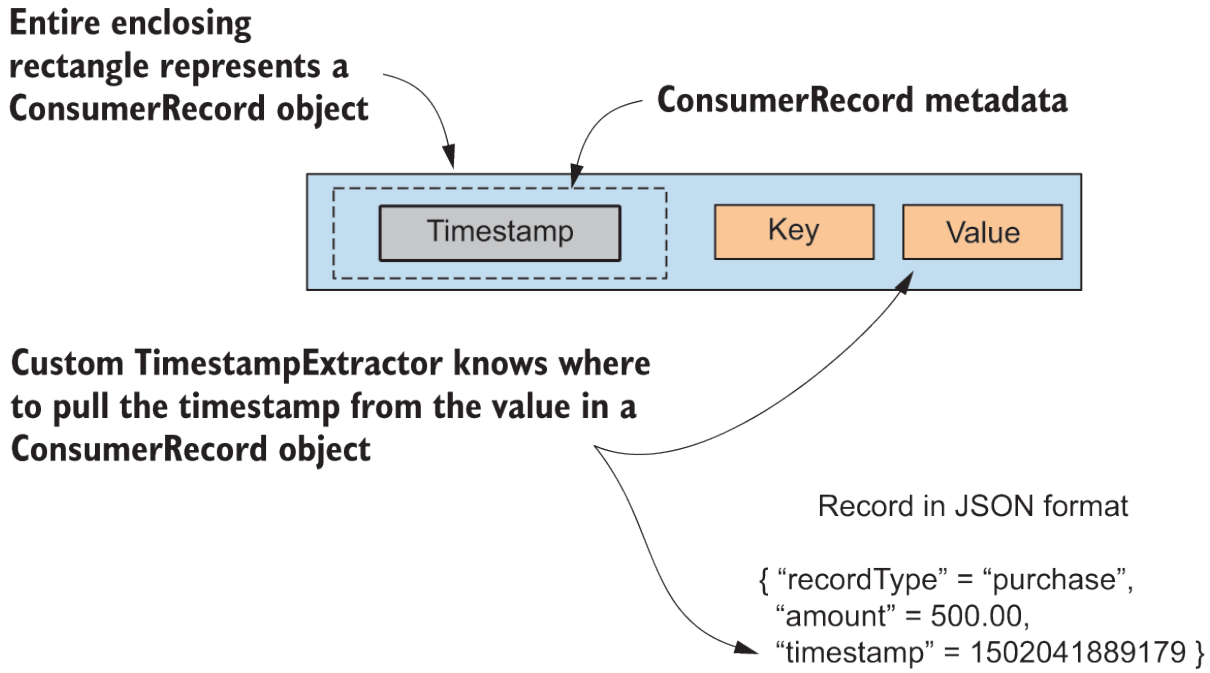


Figure 8.17 A custom `TimestampExtractor` provides a timestamp based on the value contained in the `ConsumerRecord`. This timestamp could be an existing value or one calculated from properties contained in the value object.

Here's an example of a `TimestampExtractor` implementation (found in src/main/java/bbejeck/chapter_4/timestamp_extractor/TransactionTimestampExtractor.java), also used in the join example from listing 4.12 in the section "Implementing the Join" (although not shown in the text, because it's a configuration parameter).

Listing 8.18 Custom `TimestampExtractor`

```
public class TransactionTimestampExtractor implements TimestampExtractor {

    @Override
    public long extract(ConsumerRecord<Object, Object> record,
         long previousTimestamp) {
        Purchase purchaseTransaction = (Purchase) record.value();   ❶
        return purchaseTransaction.getPurchaseDate().getTime();  ❷
    }
}
```

❶ Retrieves the Purchase object from the key/value pair sent to Kafka

❷ Returns the timestamp recorded at the point of sale

In the join example, you used a custom `TimestampExtractor` because you wanted to use the timestamps of the actual purchase time. This approach allows you to join the records even if there are delays in delivery or out-of-order arrivals.

8.20 Specifying a TimestampExtractor

Now that we've discussed how timestamp extractors work, let's tell the application which one to use. You have two choices for specifying timestamp extractors.

The first option is to set a global timestamp extractor, specified in the properties when setting up your Kafka Streams application. If no property is set, the default setting is `FailOnInvalidTimestamp.class`. For example, the following code would configure the `TransactionTimestampExtractor` via properties when setting up the application:

```
props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
         TransactionTimestampExtractor.class);
```

The second option is to provide a `TimestampExtractor` instance via a `Consumed` object:

```
Consumed.with(Serdes.String(), purchaseSerde)
         .withTimestampExtractor(new TransactionTimestampExtractor()))
```

The advantage of doing this is that you have one `TimestampExtractor` per input source, whereas the other option provides a `TimestampExtractor` instance used application wide.

8.21 Streamtime

Before we end this chapter, we should discuss how Kafka Streams keeps track of time while processing, that is by using streamtime. Streamtime is not another category of timestamp, it is the current time in a Kafka Streams processor. As Kafka Streams selects the next record to process by timestamp and as processing continues the values will increase. Streamtime is the largest timestamp seen by a processor and represents the current time for it. Since a Kafka Streams application is broken down into tasks and a task is responsible for records from a given partition, the value of streamtime is not global in a Kafka Streams application, it's only unique at the task level.

Streamtime only moves forward never backwards. Out of order records are always processed, with the exception of windowed operations depending on the grace period, but its timestamp does not affect streamtime. Here's an illustration showing how streamtime works in a Kafka Streams application.

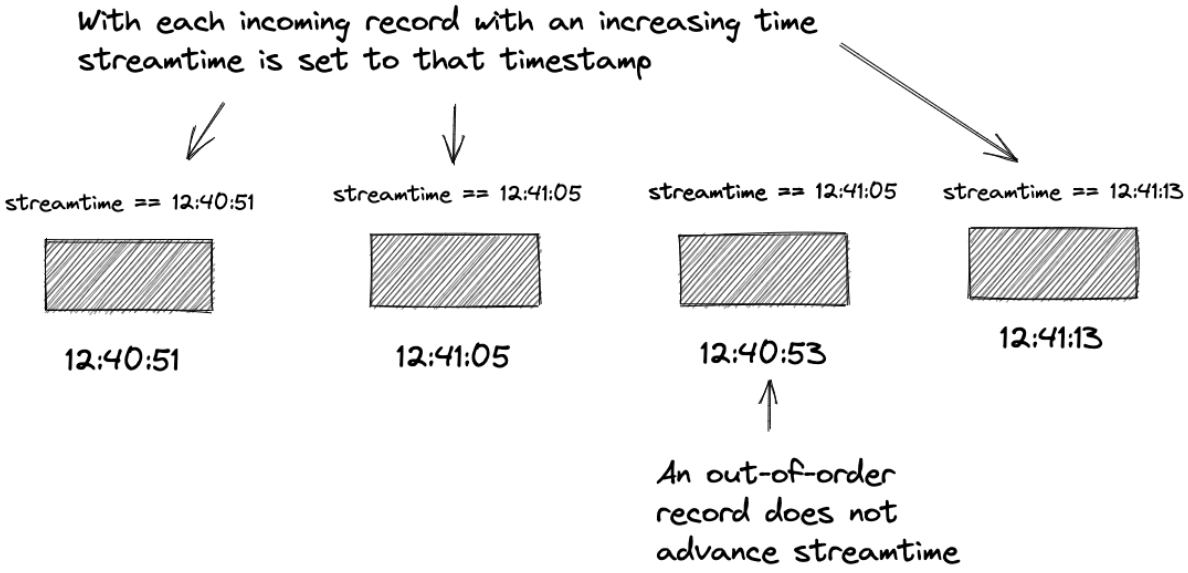


Figure 8.18 Streamtime represents the highest timestamp seen so far and is the current time of the application

So as the illustration shows, the current time of the application moves forward as records go through the topology and out of order records still go through the application but do not change streamtime.

Streamtime is vital for the correctness of windowed operations as a window only advances and closes as streamtime moves forward. If the source topics for your application are bursty or have a sporadic sustain volume of records, you might encounter a situation where you don't observe windowed results. This apparent lack of processing is due to the fact that there hasn't been enough incoming records to move streamtime forward to force window calculations.

This effect that timestamps have on operations in Kafka Streams is important to keep in mind when testing applications, as manually adjusting the value of timestamps can help you drive useful tests to validate behavior. We'll talk more about using timestamps for testing in the chapter on testing.

Streamtime also comes into play when you have punctuations which we'll cover in the next chapter when we discuss the Processor API.

8.22 Summary

- The `KTable` is an update stream and models a database table where the primary key is the key from the key-value pair in the stream. Records with the same key are considered updates to previous ones with the same key. Aggregations with the `KTable` are analogous to running a `Select…. Group By` SQL query against a relational database table.
- Performing joins with a `KStream` against a `KTable` is a great way to enrich an event stream. The `KStream` contains the event data and the `KTable` contains the facts or dimension data.
- It's possible to perform joins between two `KTable`s and you can also do a foreign key join between two `KTable`s
- The `GlobalKTable` contains all records of the underlying topic as it's not sharded so each application instance contains all the records making it suitable for acting as a reference table. Joins with the `GlobalKTable` don't require co-partitioning with the `KStream`, you can supply a function that calculates the correct key for the join.
- Windowing is a way to calculate aggregations for a given period of time. Like all other operations in Kafka Streams, new incoming records mean an update is released downstream, but windowed operations can use suppression to only have a single final result when the window closes.
- There are four types of windows hopping, tumbling, sliding, and session. Hopping and tumbling windows are fixed in size by time. Sliding windows are fixed in size by time, but record behavior drives record inclusion in a window. Session windows are completely driven by record behavior, and the window can continue to grow as long as incoming records are within the inactivity gap.
- Timestamps drive the behavior in a Kafka Streams application and this most obvious in windowed operations as the timestamps of the records drive the opening and closing of these operations. Streamtime is the highest timestamp viewed by a Kafka Streams application during it's processing.
- Kafka Streams provides different `TimestampExtractor` instances so you can use different timestamp semantics event-time, log-append-time, or processing time in your application.

B Schema compatibility workshop

In this appendix, you'll take a guided walk through of updating schemas in different compatiblity modes. You'll change the schemas compatibiltiy mode, make changes, test those changes and finally run updated producers and consumers to see the different compatibilty modes in action. I've already made all the changes, you just need to read along and run the provided commands. There are three sub-projects sr-backward, sr-forward, and sr-full. Each sub-project contains producers, consumers and schemas updated and configured for the representative compatiblity mode.

> **NOTE** **There is a lot of overlap with code and the build.gradle files between the sub-projects. This is intentional as I wanted each module isolated. The focus of using these modules is for learning about evolving schemas in Schema Registry and the related changes you need to make to Kafka Producers and Consumers, not how to set up the ideal Gradle project!**

In this section I'll only go into how Schema Registry ensures compatibility between clients. For schema compatibility rules of the serilaiztion frameworks themselves, you'll want to look at each one specifially. Avro schema resolution rules are available here https://avro.apache.org/docs/current/spec.html#Schema+Resolution. Protobuf provides backward compatibility rules in the langage specification found here https://developers.google.com/protocol-buffers/docs/proto3.

Let's go over the different compatibility modes now. For each compatibility mode you'll see the changes made to the scheama and you'll run the a few steps to needed to sucessfuly migrate a schema.

I'd like to point out that for the sake of clarity each schema migration for the different compatibility modes has it's own gradle sub-module in the source code for the book. I did this as each Avro schema file has changes resulting in different Java class structures when you build the code. Instead of having you rename files, I opted for a structure where each migration type can

stand on its own. In a typical development evrionment you will not follow this practice. You'll modify the schema file, generate the new Java code and update the producers and consumers in the same project.

All of the schema migration examples will modify the original `avenger.avsc` schema file. Here's the original schema file for reference so it's easier to see the changes made for each schema migration.

Listing B.1 The original Avenger Avro schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "real_name", "type": "string"},
    {"name": "movies", "type":
                     {"type": "array", "items": "string"},
      "default": []
    }
  ]
}
```

NOTE **For working through schema evolution and the comaptibility types, I've created three sub-modules in the source code, `sr-backward`, `sr-forward`, and `sr-full`. These sub-modules are self contained and intentionally contain duplicated code and setup. The modules have updated schemas, producers and consumers for each type of compatibility mode. I did this to make the learning process easier, as you can look at the changes and run new examples without stepping on the previous ones.**

B.1 Backward compatibility

Backward compatibility is the default migration setting. With backward compatibility you update the consumer code first to support the new schema. The updated consumers can read records serialized with the new schema or the immediate previous schema.

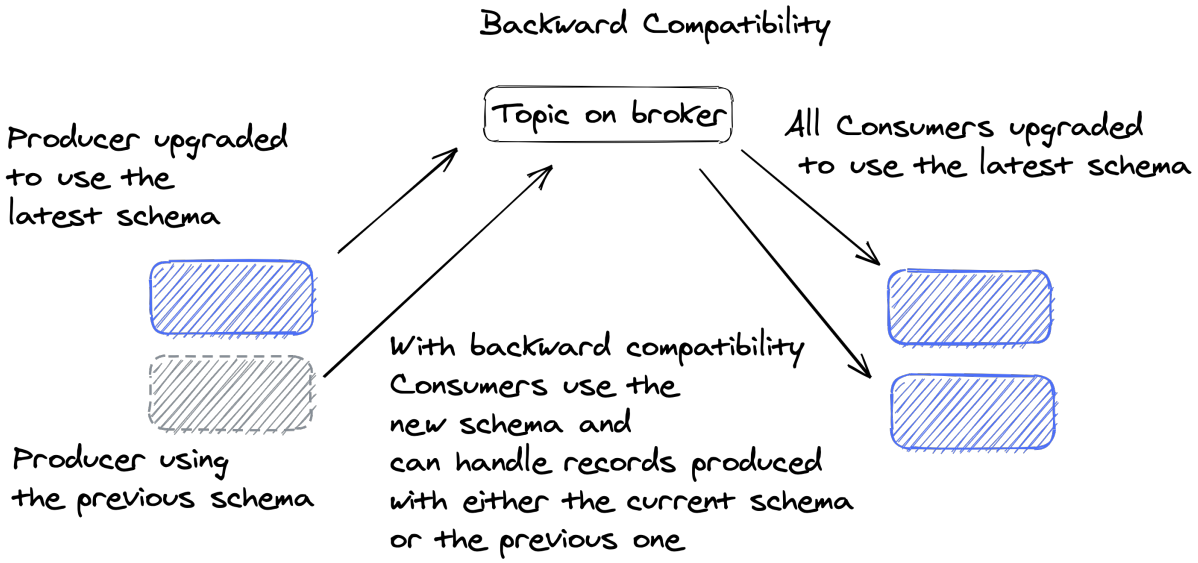


Figure B.1 Backward compatitbility updates consumers first to use the new schema then they can handle records from producers using either the new schema or the previous one

As shown in this illustration the consumer, can work with both the previous and the new schemas. The allowed changes with backwards compatibility are deleting fields or adding optional fields. An field is considered optional when the schema provides a default value. If the serialized bytes don't contain the optional field, then the deserializer uses the specifed default value when deserializing the bytes back into an object.

Before we get started, lets run the producer with the original schema. That way after the next step you'll have records using both the old schema and the new one and you'll be able to see backwards compatibility in action. Make sure you've started docker with `docker-compose up -d`, then run the following commands:

Listing B.2 Producing records with the original schema

```
./gradlew streams:registerSchemasTask   ❶
./gradlew streams:runAvroProducer   ❷
```

❶ Making sure you've registered the original avengers.avsc schema

❷ Run a producer with the original schema

Now you'll have records with the original schema in the topic. When you complete the next step, having these records available will make it clear how backwards compatibility works as the consumer will be able to accept records using the old and the updated schema.

So let's update the original schema and delete the `real_name` field and add a `powers` field with default value.

NOTE The schema file and code can be found in `sr-backward` sub-module in the `src/main/avro` directory in the source code.

Listing B.3 Backwards compatible updated schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "powers", "type":                    ❶
                    {"type": "array", "items": "string"},
      "default": []                                ❷
    },
    {"name": "movies", "type":
                    {"type": "array", "items": "string"},
      "default": []
    }
    ]
}
```

❶ The `powers` field replaced the deleted `real_name` field

❷ Provding a default value of an empty powers list for backwards compatibility.

Now that you have updated the schema, you'll want to test that the schema is compatible before uploading it to Schema Registry. Fortunately testing a new schema is a simple process. You'll use the `testSchemasTask` in the `sr-backward` module from the gradle plugin for testing compatibility. So let's test the compatibility first by running this command from the root of the project:

Listing B.4 Testing a new schema is backwards compatible

```
./gradlew :sr-backward:testSchemasTask
```

IMPORTANT For you to run the example successfully, you need to run the command exacty as its displayed here including the leading `:` character.

The result of running the `testSchemasTask` should be `BUILD SUCESSFUL` which means that the new schema is backwards compatible with the existing one. The `testSchemasTask` makes a call to Schema Registry to compare the proposed new schema against the current one to ensure it's comaptible. Now that we know the new schema is valid, let's ahead and register oit with the following command:

Listing B.5 Registering the new schema

```
./gradlew :sr-backward:registerSchemasTask
```

Again the result of running the register command print a `BUILD SUCESSFUL` on the console. Before we move on to the next step let's run a REST API command to view the latest schema for the `avro-avengers-value`:

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/versions/latest" | jq '.'
```

Running this command should yield results resembling the following:

```
{
  "name" : "avro-avengers-value",
  "version" : 2,
  "schema" : "{\"type\": \"record\", \"namespace\": \"bbejeck.chapter_3\", \"name\":
\"AvengerAvro\"..."
}
```

From the results, you can see the increase in the version from 1 to 2 as you've loaded a new schema. With this changes in place, you'll need to update your clients, starting with the consumer. With compatibility of `BACWARDS` you want to update the consumer first to handle any records produced using the new schema.

For example you originally expected to work with the `real_name` field, but you deleted it in the schema, so you want to remove references to it in the new schema. You also added the `powers` field, so you'll want to be able to work with that field. That also implies you've generated new model objects.

Earlier in the chapter when you ran the `clean, build` command it generated the correct objects for all of our modules. So you should not have to do that now.

Take note that since we are in `BACKWARDS` compatiblity mode, if your updated consumer were to getrecords in the previous format, then it won't blow-up. The updated ignores the `real_name` field, and the `powers` field uses the default value.

After you have updated the consumer, then you'll want to update your producer applications to use the new schema. The `AvroProducer` in the `sr-backward` submodule has had the updates applied already. Now run the following command to producer records using the new schema.

Listing B.6 Producing records with the new schema

```
./gradlew :sr-backward:runAvroProducer
```

You'll see some text scroll by followed by the familiar `BUILD SUCESSFUL` text. If you remember, just a few minutes ago you ran the produce command from the original sub-module adding records in the previous schema. So now that you've run the producer using the new schema, you have a mix of old and new schema records in the topic. But our consumer example should be able to handle both types, since we are in the `BACKWARDS` compatibility mode.

Now when you run the consumer you should be able to see the records produced with the

previous schema as well as the records produced with the new schema. Run the following command to execute the updated consumer:

Listing B.7 Consuming records against new schema

```
./gradlew :sr-backward:runAvroConsumer
```

In the console you should see the first results printing `with powers []`. The empty value indicates those are the older records using the default value, since the original records did not have a `powers` field on the object.

NOTE **For this first compatibility example, your consumer read all the records in the topic. This happened because we used a new `group.id` for the consumer in the `sr-backwards` module and we've configured it to read from the earliest available offset, if none were found. For the rest of the compatiblity examples and modules, we'll use the same `group.id` and the consumer will only read newly produced records. I'll go into full details on the `group.id` configuration and offset behavior in chapter four.**

B.2 Forward compatibility

Forward compatibility is a mirror image of backward compatibility regarding field changes. With forward compatibility you can add fields and delete **optional** fields. Let's go ahead and update the schema again, creating `avenger_v3.avsc` which you can find in the `sr-forward/src/main/avro` directory.

Listing B.8 Foward compatible Avenger schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "powers", "type": {
               "type": "array", "items": "string"},
             "default": []
    },
    {"name": "nemeses","type": {                    (1)
              "type": "array","items": "string"
      }
    }
  ]
}
```

(1) Added a new field, `nemeses`

In this new version of the schema, you've removed the `movies` field which defaults to an empty list and added a new field `nemeses`. In forward compatibility you would upgrade the producer

client code first.

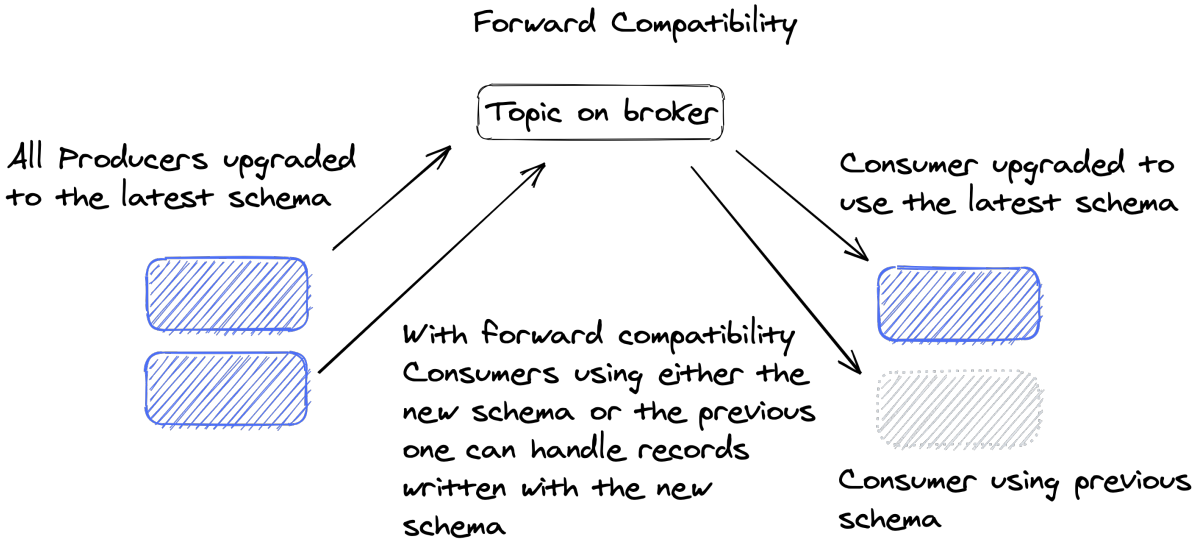


Figure B.2 Forward compatitbility updates producers first to use the new schema and consumers can handle the records either the new schema or the previous one

By upgrading the producer code first, you're making sure the new fields are properly populated and only records in the new format are available. Consumers you haven't upgraded can still work with the new schema as it will simply ignore the new fields and the deleted fields have default values.

Now you need to change the compatibility mode from `BACKWARD` to `FORWARD`. In the `sr-forward` sub-module the configuration for the Schema Registry plugin has this section setting the compatibility:

Listing B.9 Comatibility in build.gradle for sr-forward sub-module

```
config {
     subject('avro-avengers-value', 'FORWARD')
 }
```

Now with the configuration set, to change the compatibility mode, run this command:

Listing B.10 Changing the compatbility mode to FORWARD

```
./gradlew :sr-forward:configSubjectsTask
```

As we've seen before, the result of this command produces a `BUILD SUCCESSFUl` result on the console. If you want to confirm the compatibility mode for your subject, you can use this REST API command:

Listing B.11 REST API to view configured compatibility mode for a subject

```
curl -s "http://localhost:8081/config/avro-avengers-value" | jq '.'
```

The `jq` at the end of the curl command formats the returned JSON and you should see something like:

Listing B.12 Formatted configuration response

```
{
   compatibility: FORWARD
}
```

Now that you have configured the `avro-avengers-value` subject with forward compatibility, go ahead and test the new schema by running the following command:

Listing B.13 Testing a new schema is foward compatible

```
./gradlew :sr-forward:testSchemasTask
```

This command should print a `BUILD SUCESSFUL` on the console, then you can register the new schema:

Listing B.14 Register the new forward compatible schema

```
./gradlew :sr-forward:registerSchemasTask
```

Then run a producer already updated to send records in the new format with this command:

Listing B.15 Run producer updated for records in the new schema format

```
./gradlew :sr-forward:runAvroProducer
```

Now that you've run the producer with an updated schema let's first run the consumer that **is not** updated:

Listing B.16 Run a consumer not yet updated for the new schema changes

```
./gradlew :sr-backward:runAvroConsumer
```

The results of the command show how that with forward compatibility even if the consumer is **not updated** it can still handle records written using the new schema. Now we need to produce some records again for the ***updated*** consumer:

Listing B.17 Run producer again

```
./gradlew :sr-forward:runAvroProducer
```

Now run the consumer that is updated for the new schema:

Listing B.18 Run consumer updated for new schema

```
./gradlew :sr-forward:runAvroConsumer
```

In both cases, the consumer runs sucessfully, but the details in the console are different due to having upgraded the consumer to handle the new schema.

At this point you've seen two compatibility types, backward and forward. As the compatibility name implies, you must consider record changes in one direction. In backward compatibility, you updated the consumers first as records could arrive in either the new or old fomrat. In forward compatiblity, you updated the producers first to ensure the records from that point in time are only in the new format. The last compatibility strategy to explore is the `FULL` compatibility mode.

B.3 Full compatibility

In full compatiblity mode, you free to add or remove fields, but there is one catch. **Any changes** you make must be to **optional** fields only.

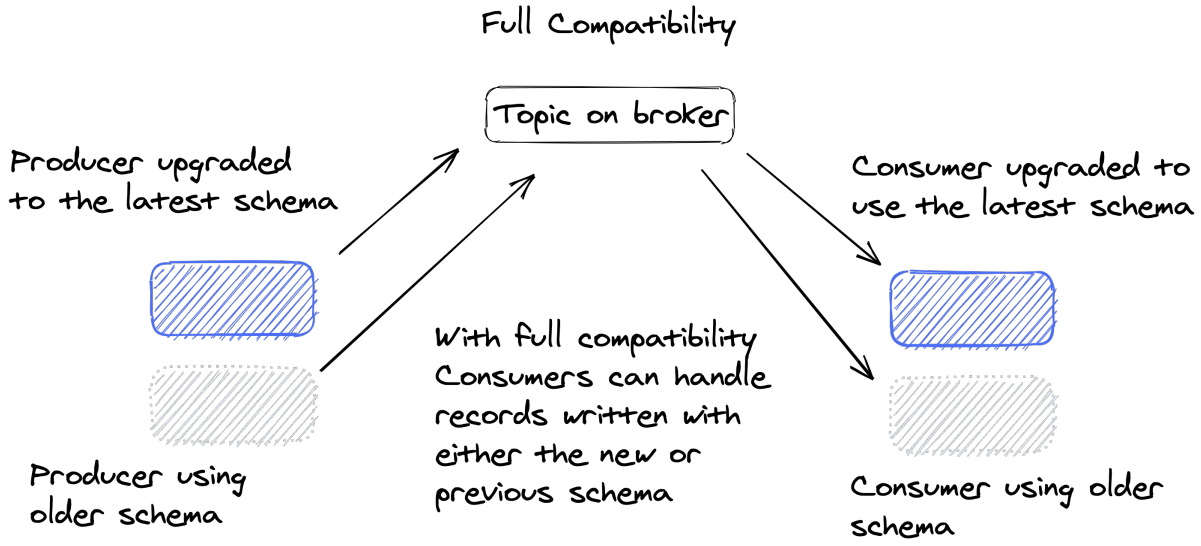


Figure B.3 Full compatitbility allows for producers to send with the previous or new schema and consumers can handle the records either the new schema or the previous one

Since the fields involved in the updated schema are optional, these changes are considered compatibable for existing producer and consumer clients. This means that the upgrade order in this case is up to you. Consumers will continue to work with records produced with the new or old schema.

Let's take a look at an schema to work with `FULL` compatibility:

Listing B.19 Full compatibility schema avengers_v4.avsc

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "yearPublished", "type": "int", "default": 1960 },   ❶
    { "name": "realName", "type": "string", "default": "unknown" },   ❷
    { "name": "partners", "type": {
                           "type": "array","items": "string"},    ❸
                "default": []
    },
    {"name": "nemeses", "type": {
                         "type": "array", "items": "string"},
              "default": []
    }
  ]
}
```

❶ Added new optional field yearPublished

❷ Added back optional field realName

❸ Added new field partners

Before you update the schema, let's produce a set of records one more time so that we can have a batch of records in the format prior to our next schema change. I'll explain why we are doing this in an upcoming section.

Listing B.20 Run producer again to create a batch of records in the format before we migrate the schema

```
./gradlew :sr-forward:runAvroProducer
```

This will give us a batch of records to read with an updated consumer. But first let's change the compatibility, this time to `FULL`:

Listing B.21 Change the compatibility to FULL

```
./gradlew :sr-full:configSubjectsTask
```

And to keep consistent with our process, let's test the compatibility of the schema before we migrate it:

Listing B.22 Test schema for full compatibility

```
./gradlew :sr-full:testSchemasTask
```

With the migrated schema compatibility tested, let's go ahead and register it

Listing B.23 Register the FULL compatibility schema

```
./gradlew :sr-full:registerSchemasTask
```

With the a new version of the schema registered, let's have some fun with the order of records we produce and consume. Since all of the updates to the schema involve optional fields the order in which we update the producers and consumers doesn't matter.

A few minutes ago, I had you create a batch of records in the previous schema format. I did that to demonstrate that we can use an updated consumer in `FULL` compatiblity mode to read older records. Remember before with `FORWARD` compatibility it was essential to ensure the updated consumers would only see records in the new format.

Now let's run an updated consumer to read records using the previous schema. But let's watch what happens now:

Now run the updated consumer

Listing B.24 Consuming with the updated consumer

```
./gradlew :sr-full:runAvroConsumer
```

And it runs just fine! Now let's flip the order of operations and run the updated producer:

Listing B.25 Producing records with the new schema

```
./gradlew :sr-full:runAvroProducer
```

And now you can run the consumer that we haven't updated yet for the new record format:

Listing B.26 Consuming new records with a consumer not updated

```
./gradlew :sr-forward:runAvroConsumer
```

As you can see from playing with the different versions of producers and consumers with `FULL` compatibility, when you update the producer and consumer is up to you, the order doesn't matter.

Notes

1. https://www.merriam-webster.com/dictionary/event

2. Jun Rao, "How to Choose the Number of Topics/Partitions in a Kafka Cluster?" http://mng.bz/4C03.

3. Michael Noll, "https://www.confluent.io/blog/kafka-streams-tables-part-2-topics-partitions-and-storage-fundamentals/"

4. Kafka documentation, "Log Compaction," http://kafka.apache.org/documentation/#compaction.

5. Kafka documentation, "Replication," http://kafka.apache.org/documentation/#replication.

Notes

1. www.merriam-webster.com/dictionary/event

2. Jun Rao, "How to Choose the Number of Topics/Partitions in a Kafka Cluster?" mng.bz/4C03.

3. Michael Noll, "https://www.confluent.io/blog/kafka-streams-tables-part-2-topics-partitions-and-storage-fundamentals/"

4. Kafka documentation, "Log Compaction," kafka.apache.org/documentation/#compaction.

5. Kafka documentation, "Replication," kafka.apache.org/documentation/#replication.

6. This section derived information from Jay Kreps's "Introducing Kafka Streams: Stream Processing Made Simple" (mng.bz/49HO) and "The Log: What Every Software Engineer Should Know About Real-time Data's Unifying Abstraction" (mng.bz/eE3w).

Index Terms