Kafka Streams IN ACTION

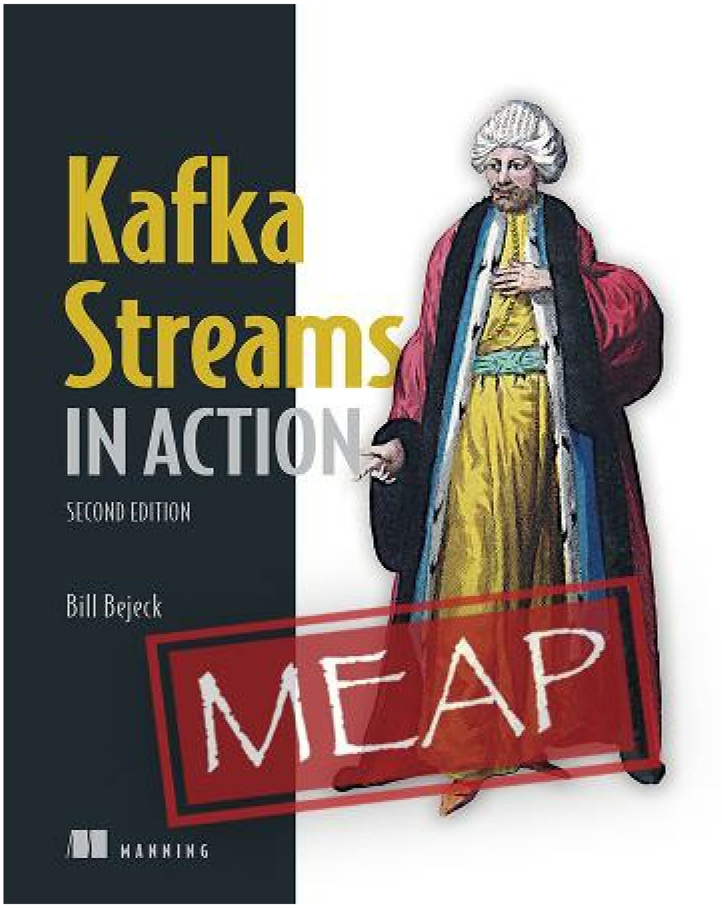
SECOND EDITION

Bill Bejeck

MEAP

MANNING

Kafka Streams in Action, Second Edition MEAP V11



MEAP VERSION 11

MANNING PUBLICATIONS

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

In this book

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:

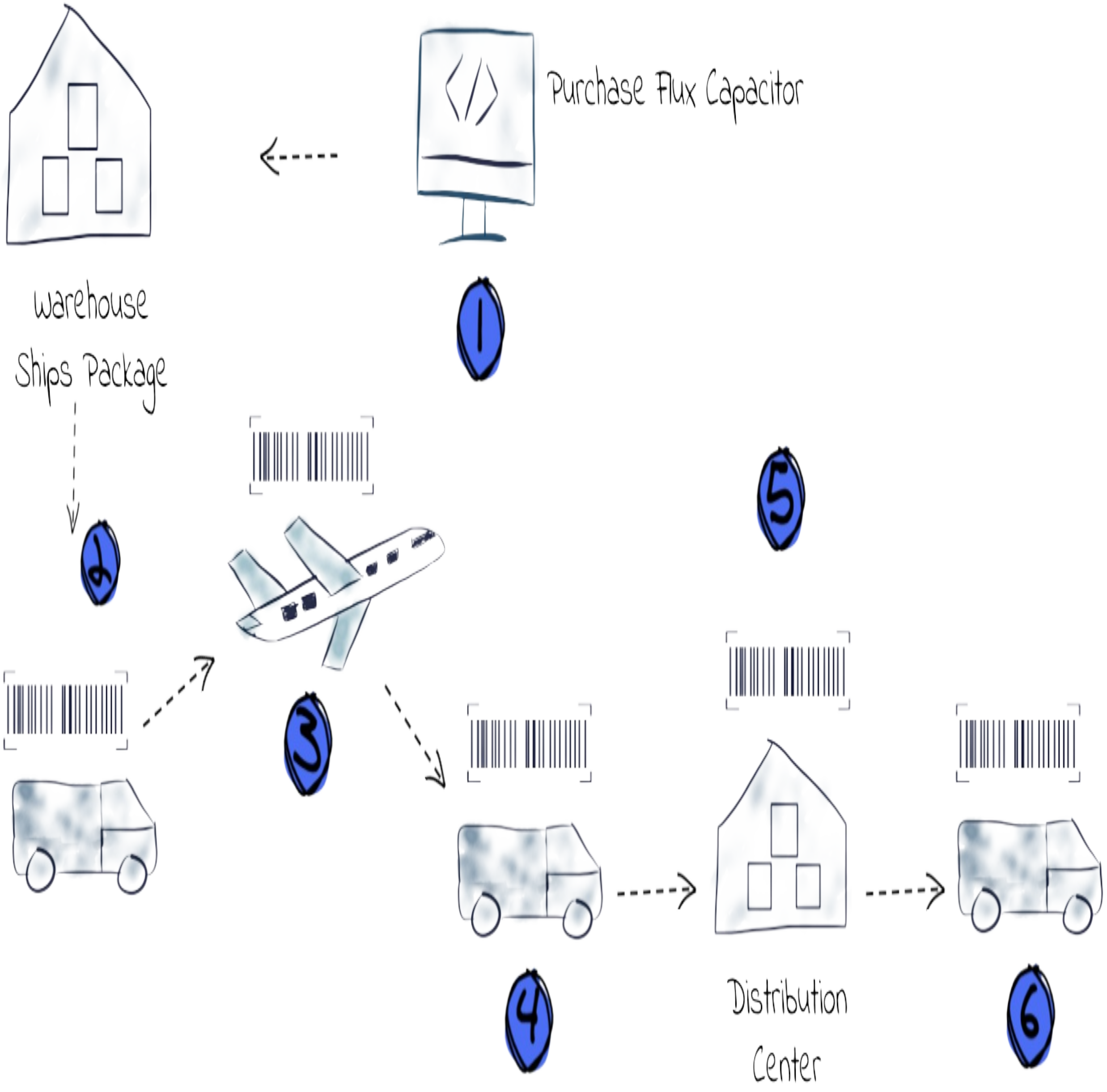
1. Key - an identifier for the event
2. Value - the event itself
3. timestamp - when the event occurred

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.

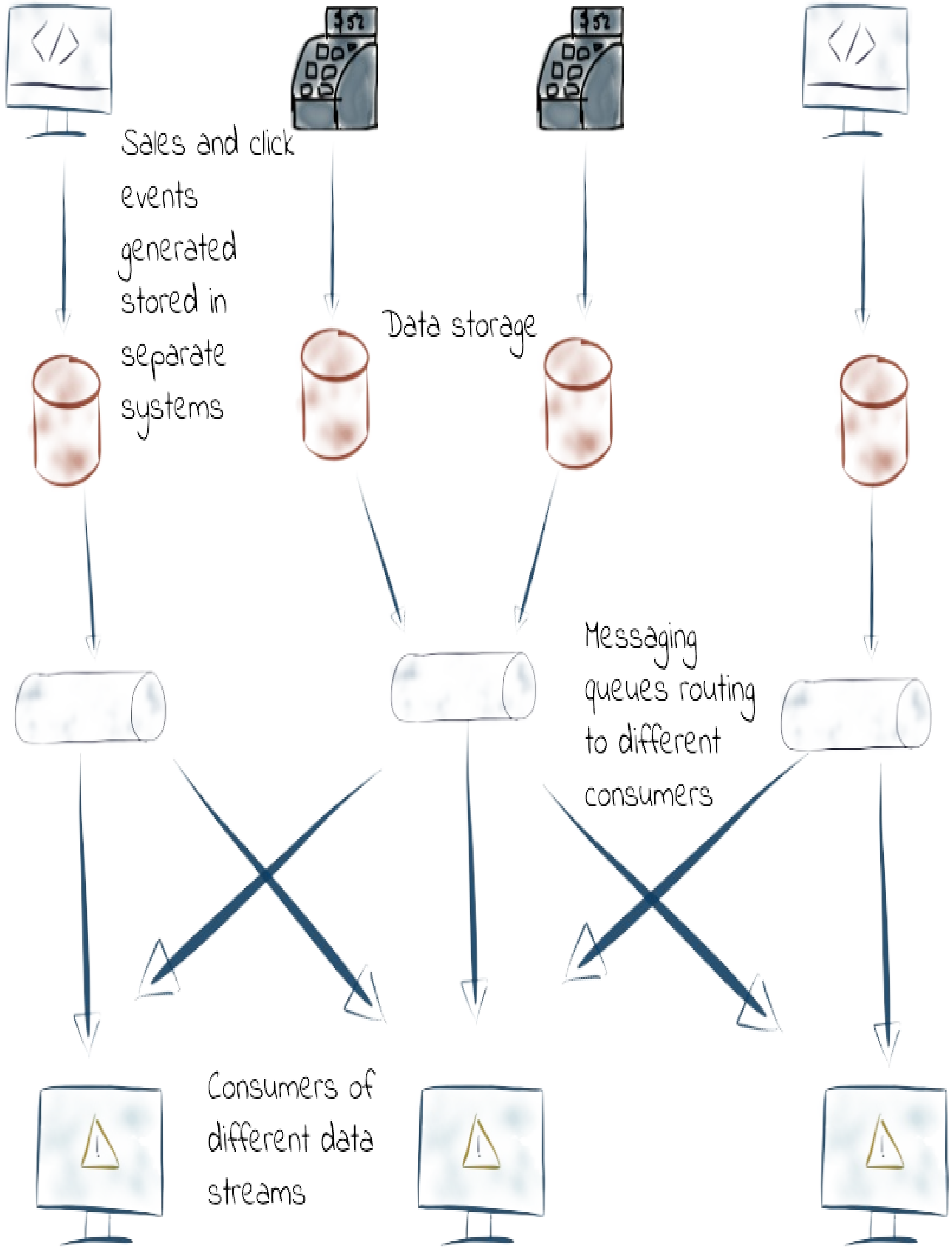
1.2 Introducing the Apache Kafka® event streaming platform

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:

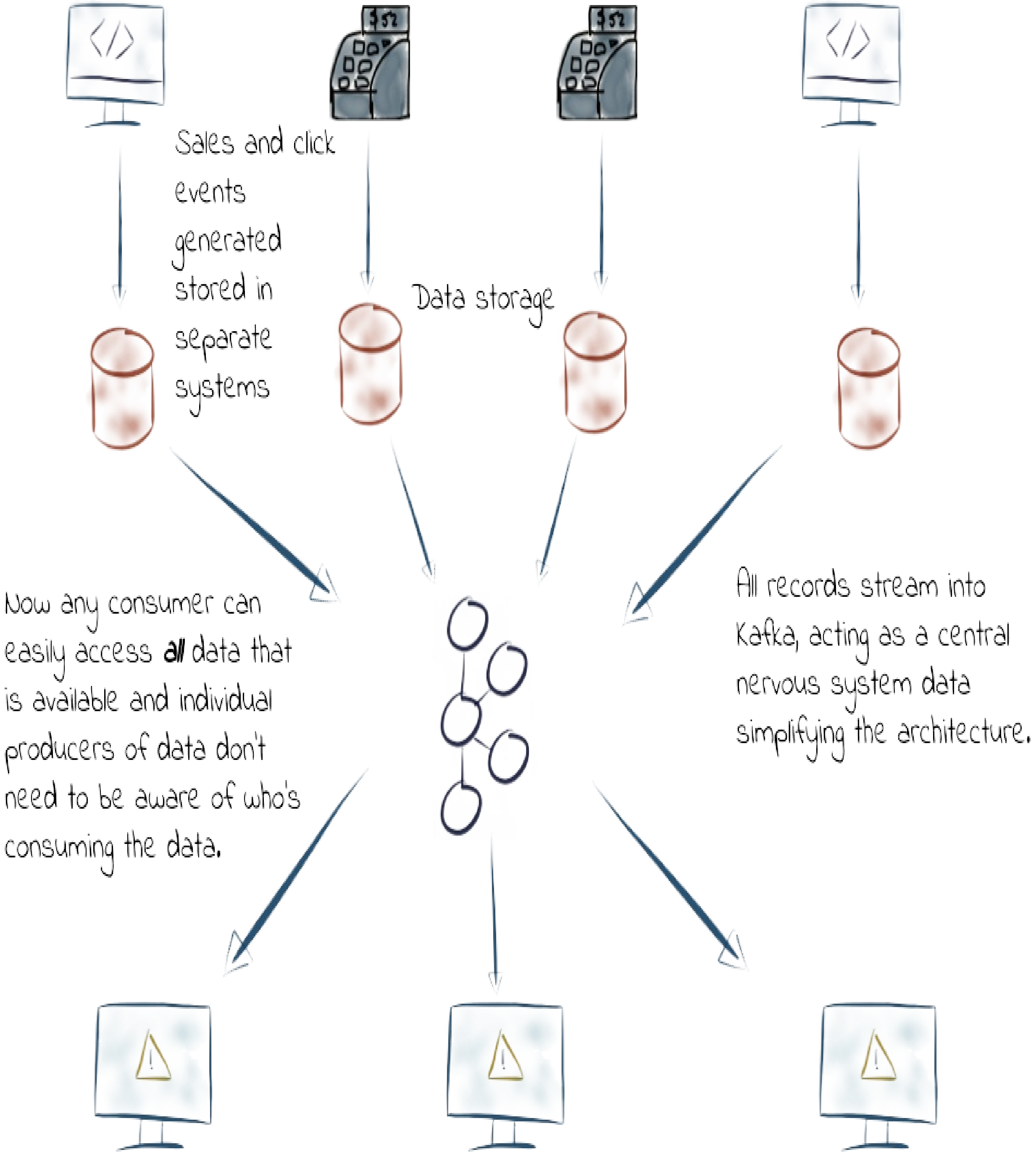
Figure 1.2. Initial event-streaming architecture leads to complexity as the different departments and data streams sources need to be aware of the other sources of events


Sales and click events generated stored in separate systems
Data storage
Messaging queues routing to different consumers
Consumers of different data streams

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


Sales and click events generated stored in separate systems
Data storage
All records stream into Kafka, acting as a central nervous system data simplifying the architecture.
Now any consumer can easily access **all** data that is available and individual producers of data don't need to be aware of who's consuming the data.
$52
$52

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.

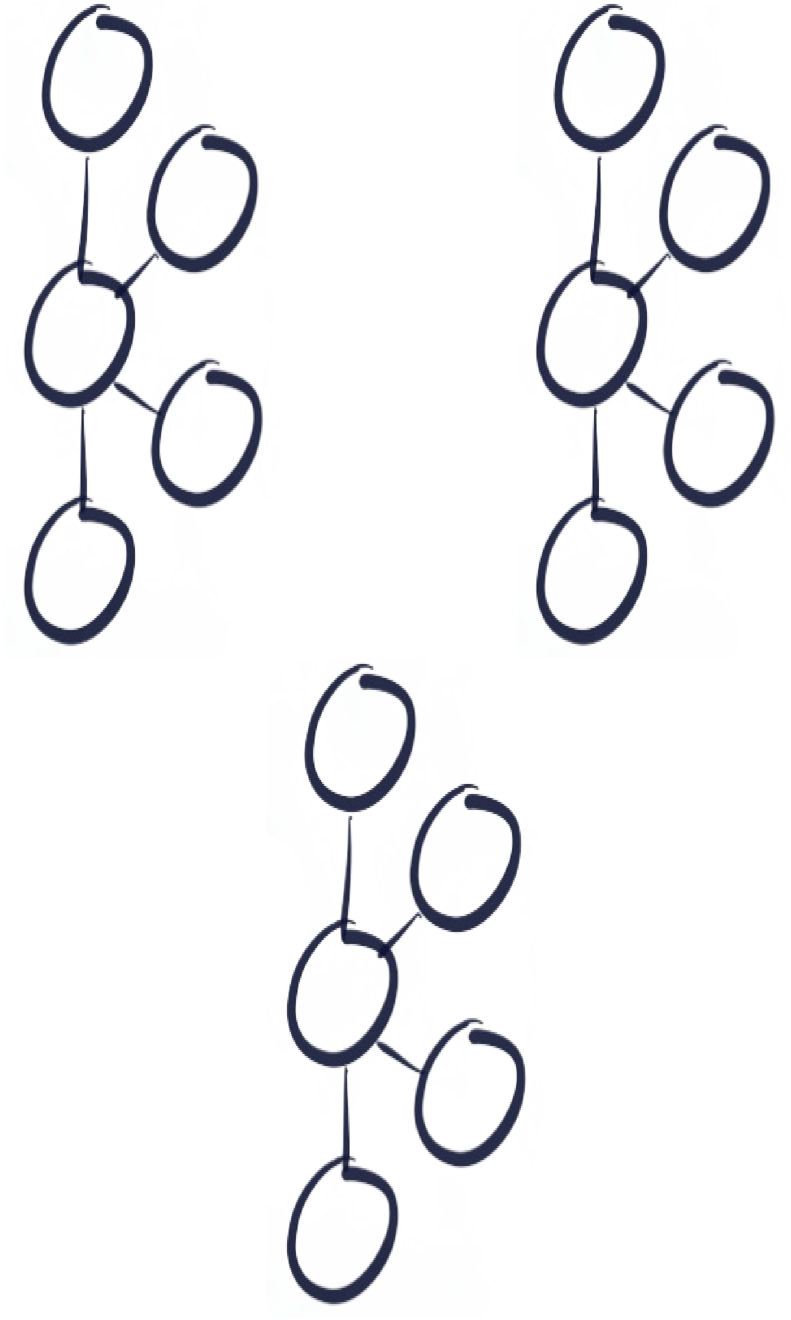
At a high level, Kafka is a distributed system of servers and clients. The servers are called brokers, and the clients are record producers sending records to the brokers, and the consumer clients read records for the processing of events.

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.

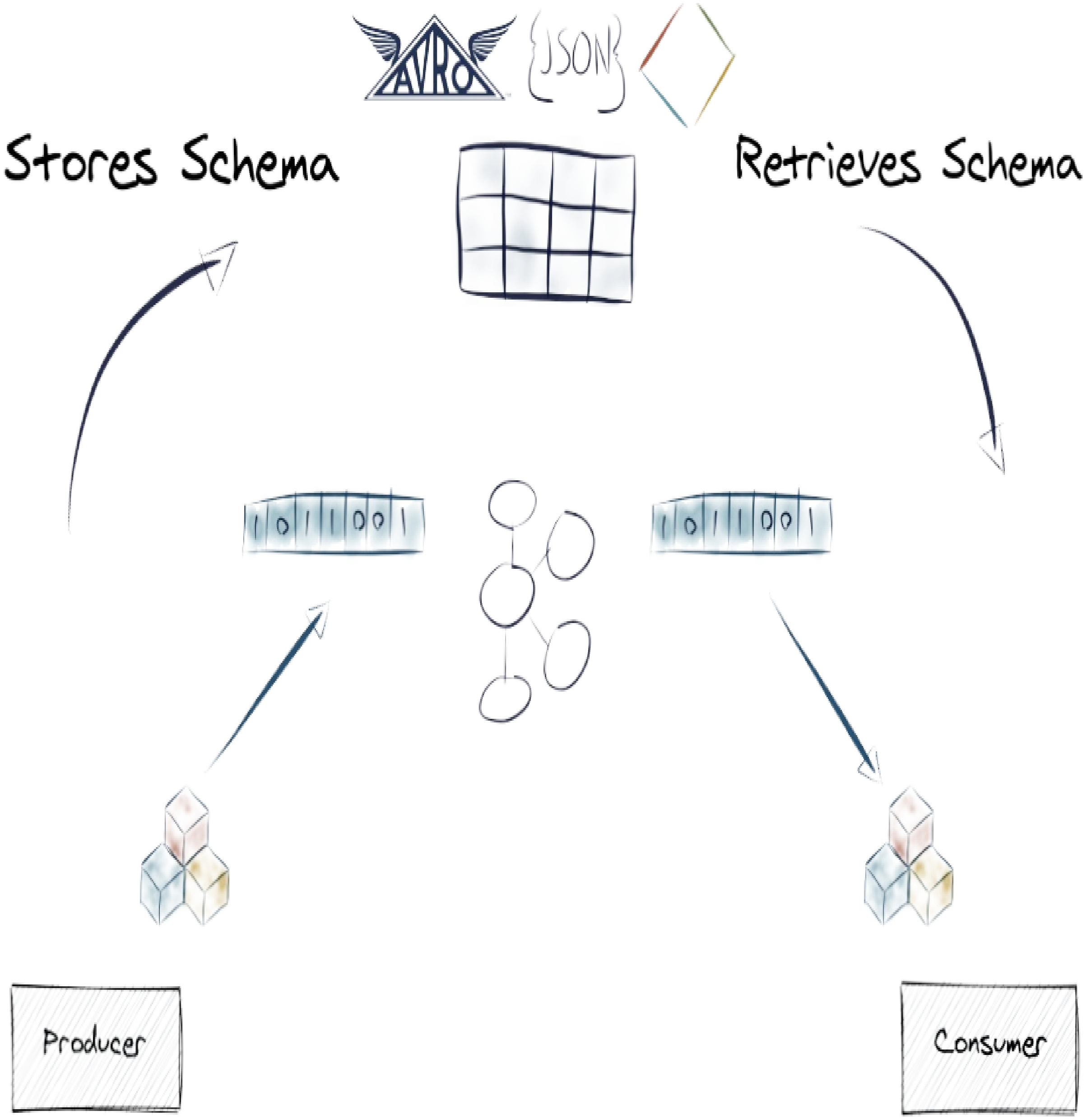
Figure 1.4. You deploy brokers in a cluster, and brokers replicate data for durable storage



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

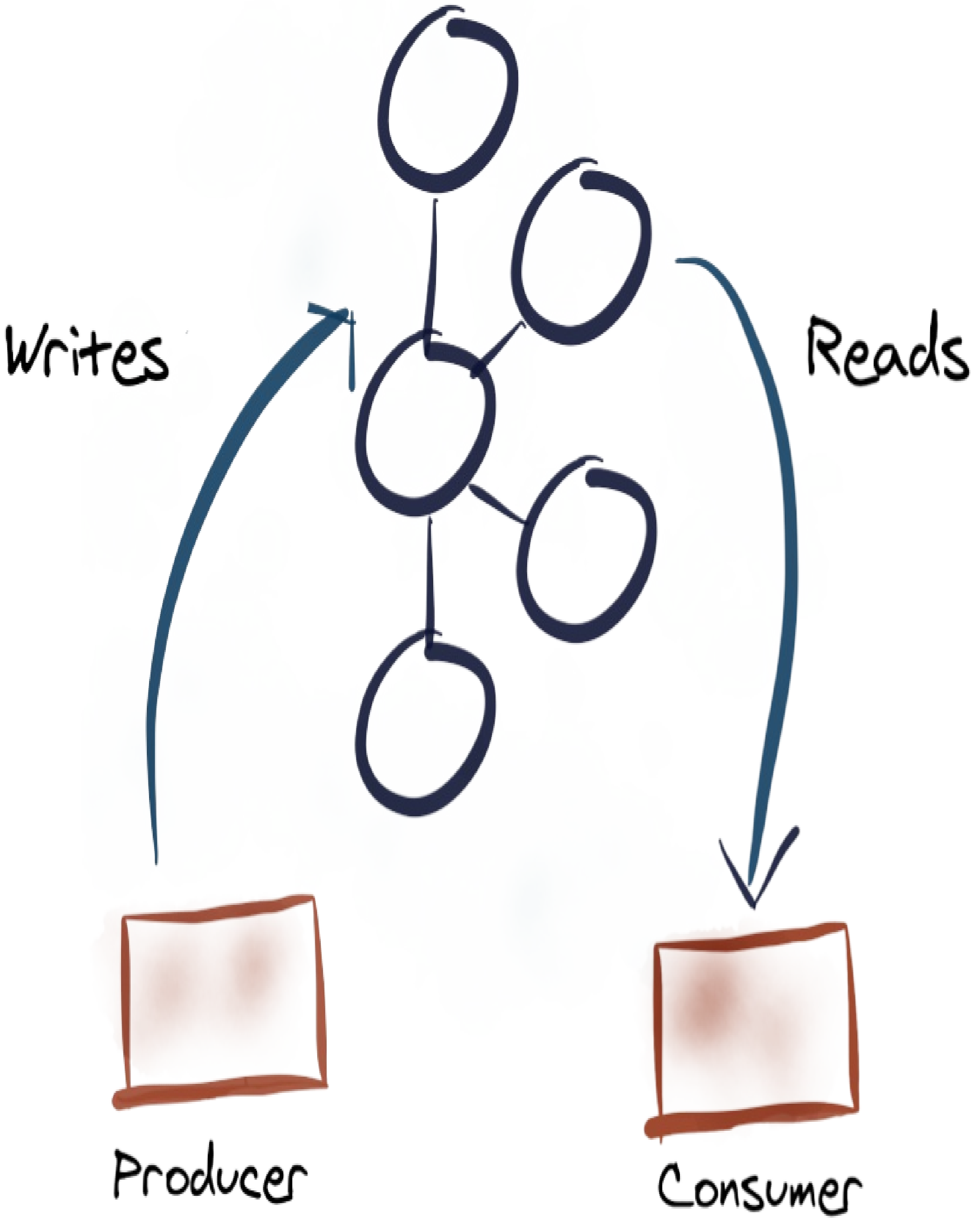
Figure 1.5. Schema registry enforces data modeling across the platform



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

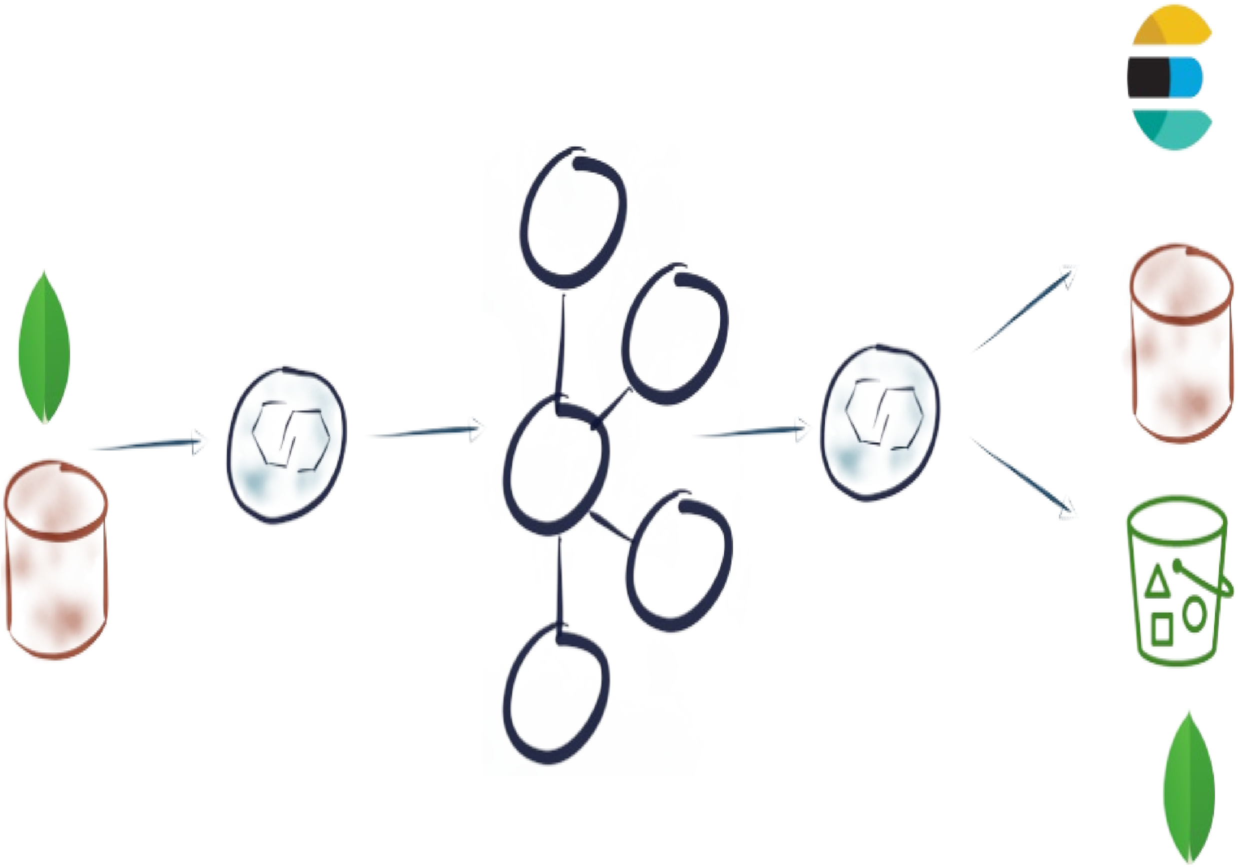
Figure 1.6. producers write records into Kafka, and consumers read records



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

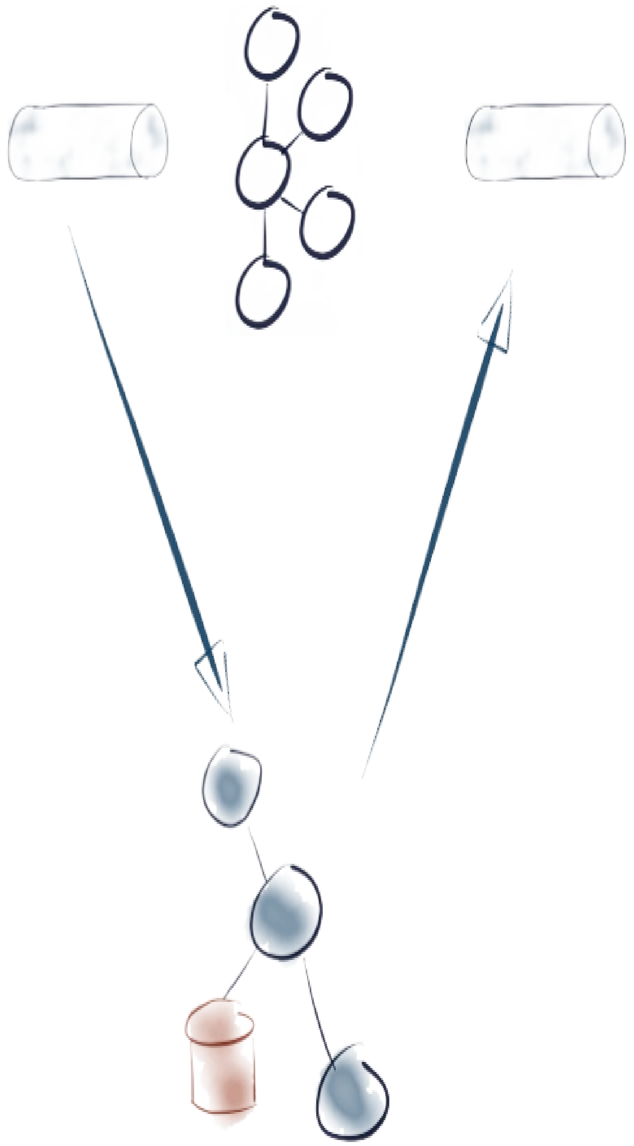
Figure 1.7. Kafka Connect bridges the gap between external systems and Apache Kafka



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



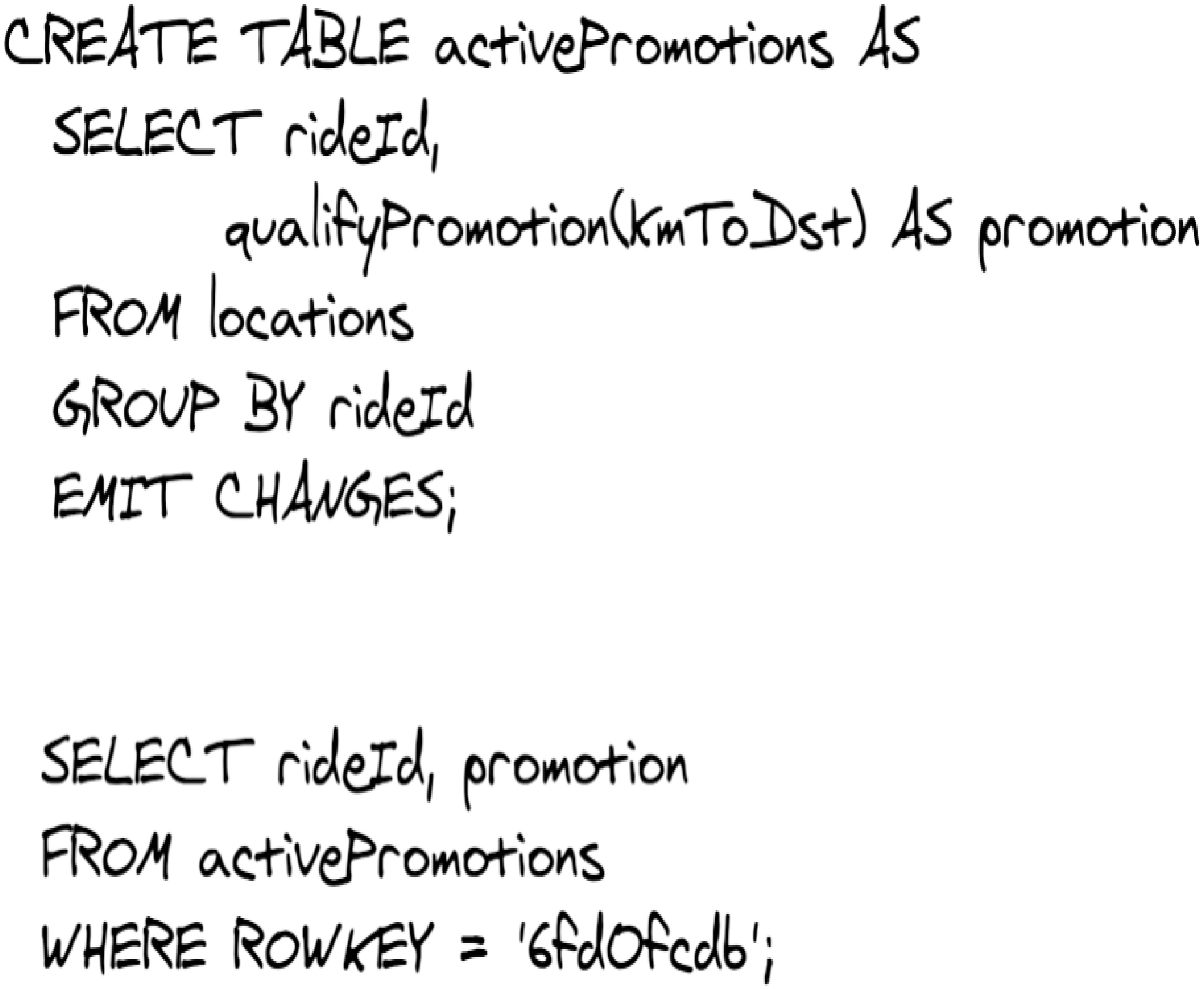
Kafka Streams is the native stream processing library for Kafka. Kafka Streams is written in the Java programming language and is used by client applications at the perimeter of a Kafka cluster; it is **not* run inside a Kafka broker. It provides support for performing operations on event data, including 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.

Figure 1.9. ksqlDB provides streaming database capabilities



```
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';
```

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.

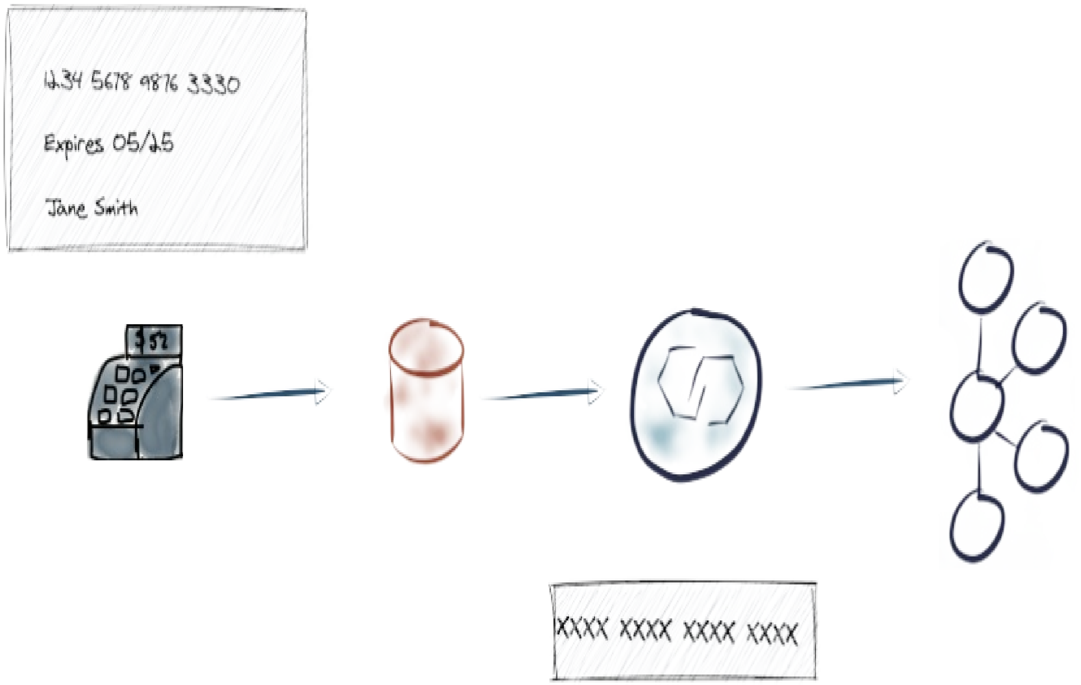
Walking through the store after grabbing everything she needs, Jane walks by the footwear section and notices some new designer shoes that would go great with her new suit. She realizes that’s not what she came in for, but what the heck life is short (ZMart thrives on impulse purchases!), so Jane gets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

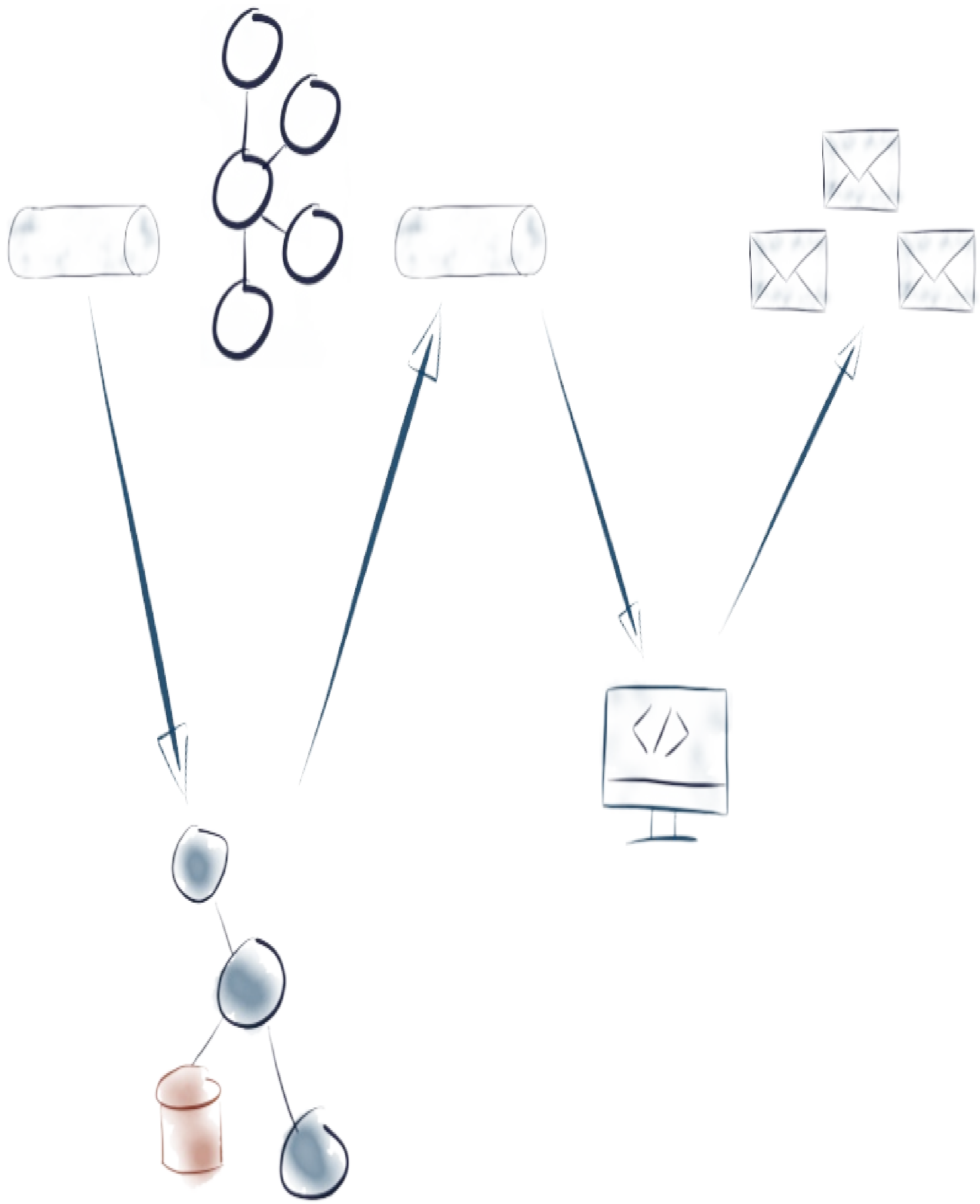
So now ZMart’s sales data streams into Kafka. In this case, ZMart uses Kafka Connect to create a source connector to capture the sales as they occur and send them into Kafka. The sale transaction brings us to the first requirement, the protection of customer data. In this case, ZMart uses an SMT or Simple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

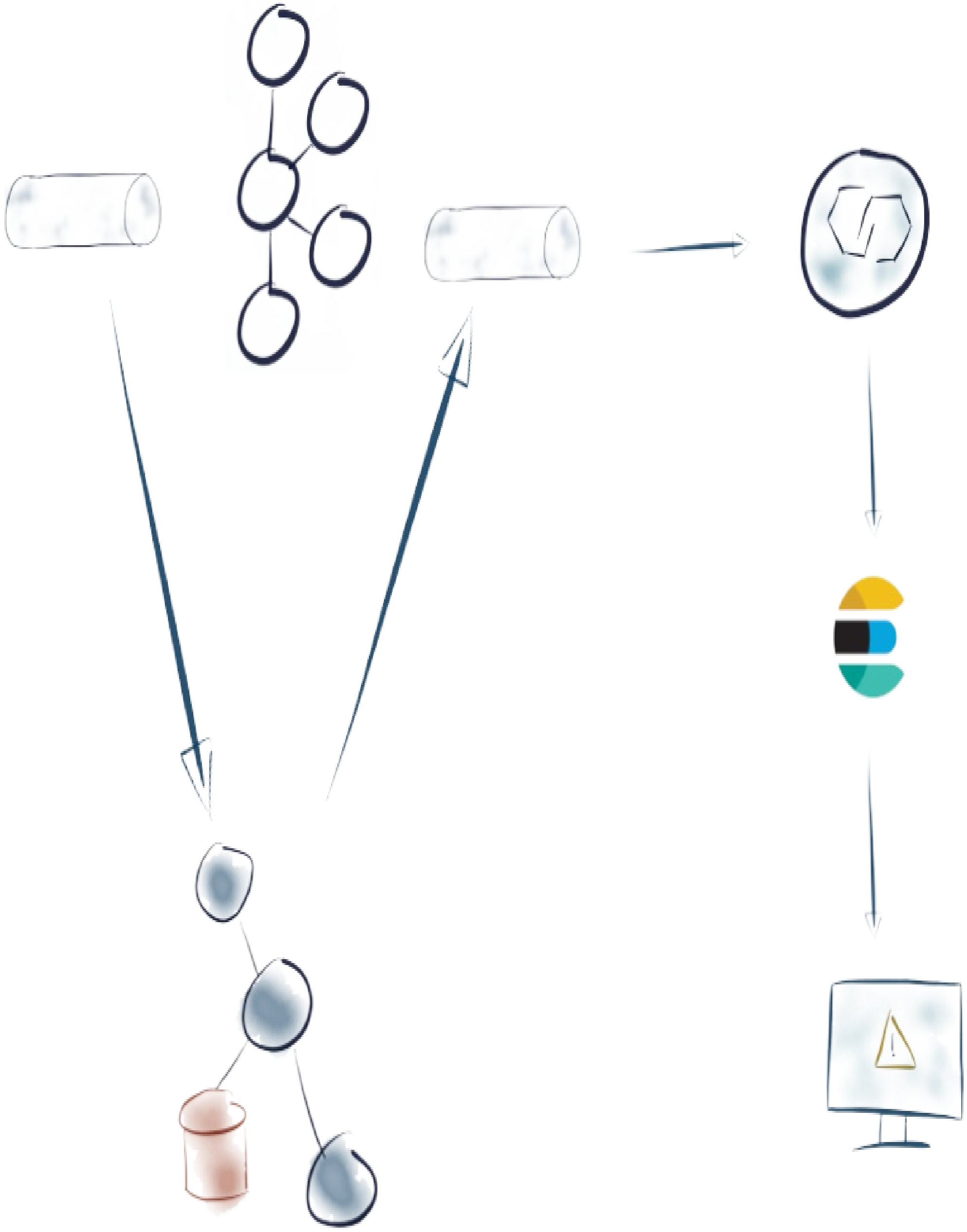
Figure 1.11. Marketing department application for processing customer points and sending out earned emails



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.

[1] https://www.merriam-webster.com/dictionary/event

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

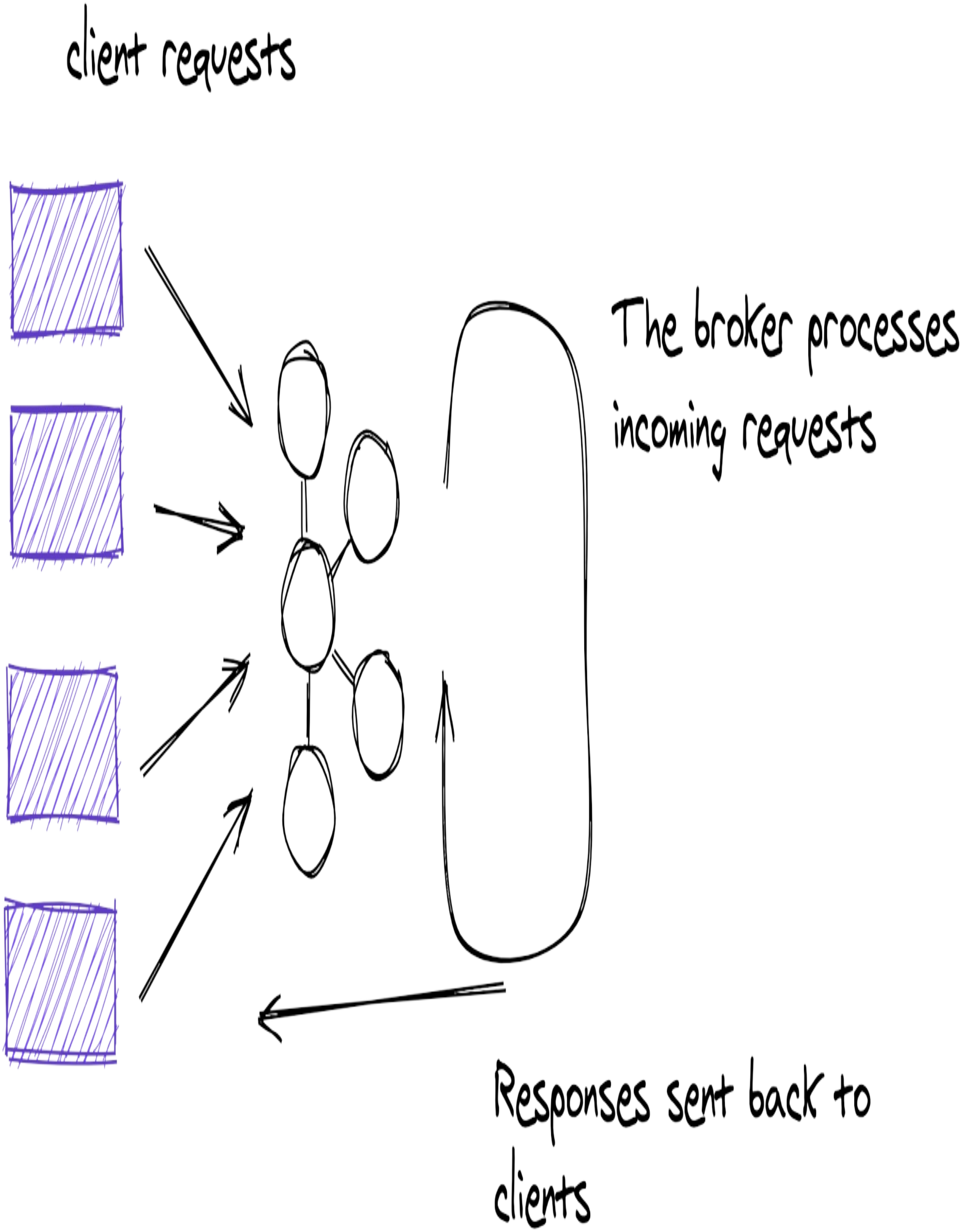
In chapter one, I provided an overall view of the Kafka event streaming platform and the different components that make up the platform. In this chapter, we will focus on the heart of the system, the Kafka broker. The Kafka broker is the server in the Kafka architecture and serves as the storage layer.

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


client requests
The broker processes incoming requests
Responses sent back to clients

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.

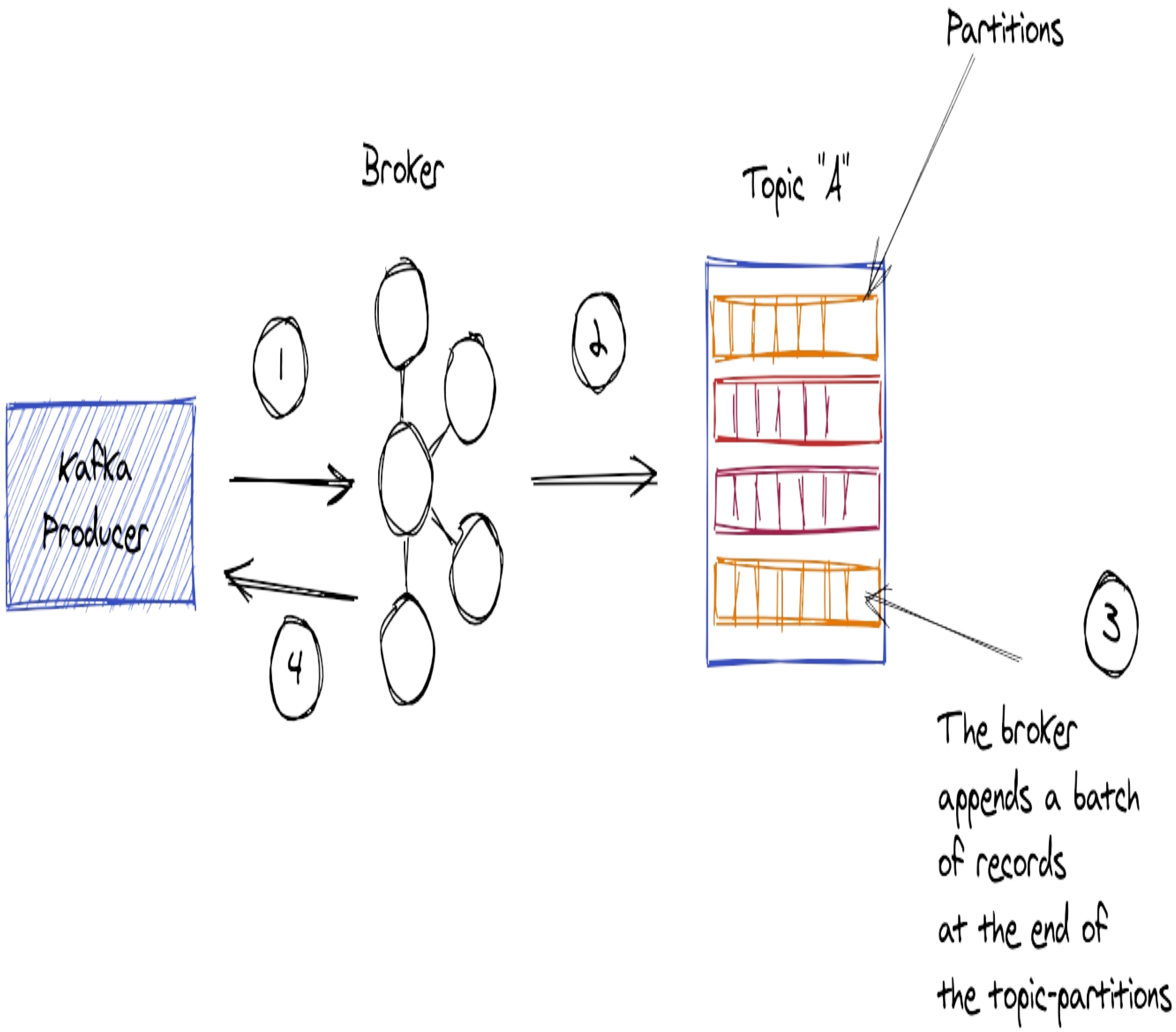
So, let’s get started with some walkthroughs of how a broker handles client requests, starting with producing.

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


Partitions
Broker
Topic "A"
1
2
Kafka Producer
3
4
The broker appends a batch of records at the end of the topic-partitions

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.

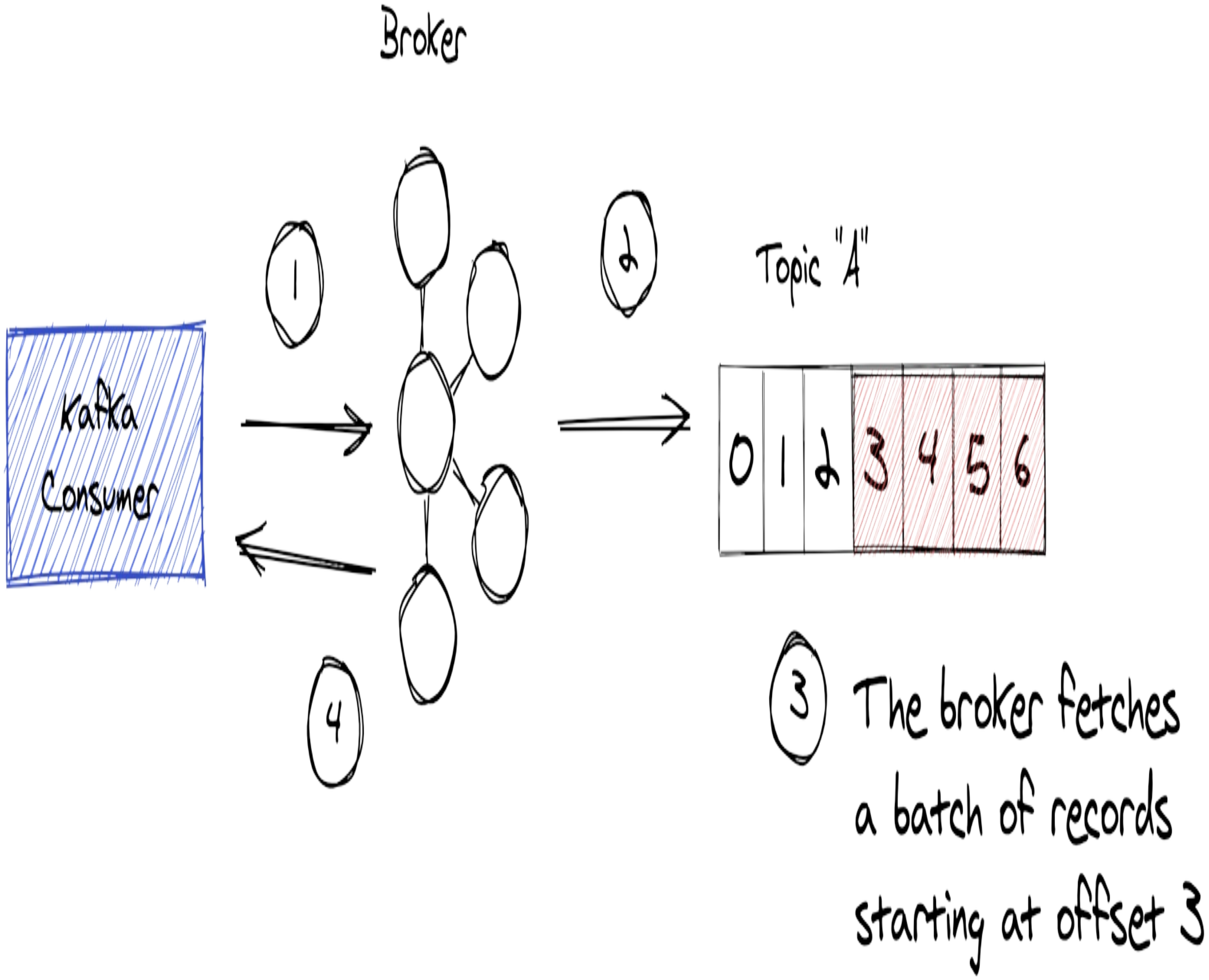
Now that we've walked through an example produce request, let's walk through another request type, fetch, which is the logical opposite of producing records; consuming records.

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.

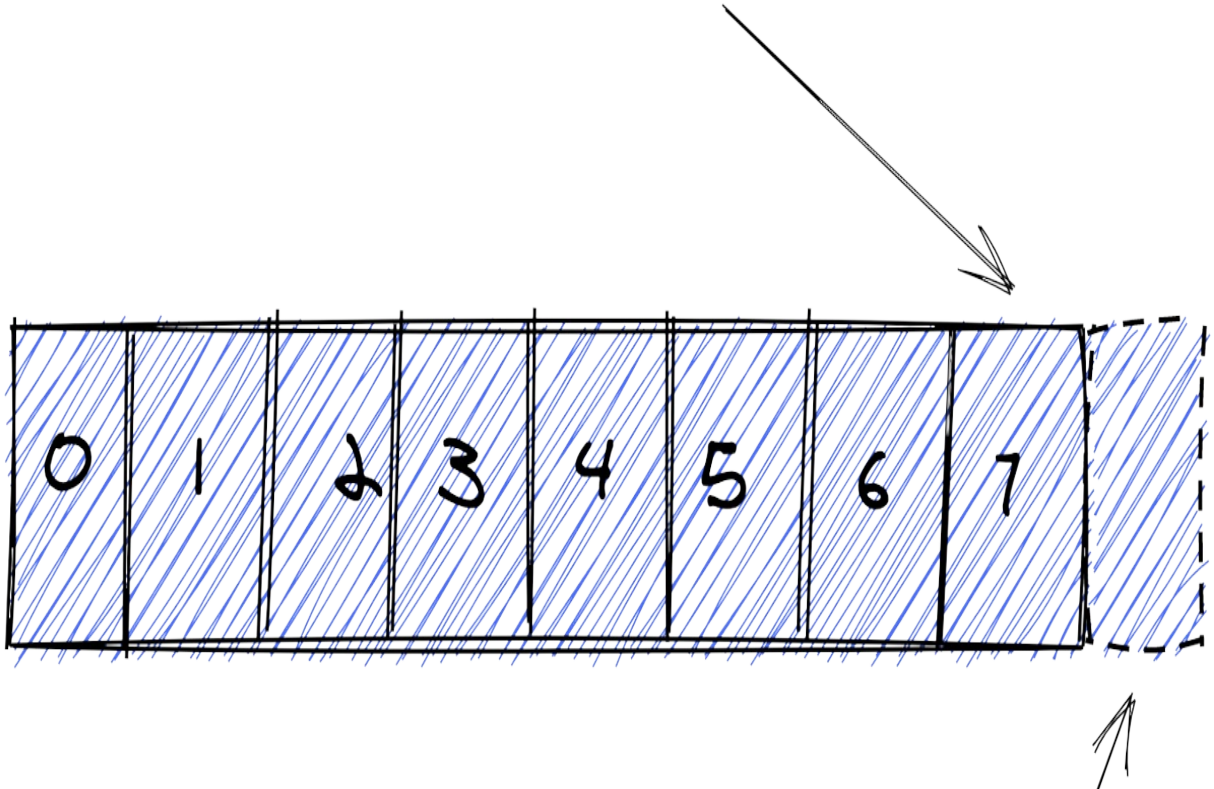
Now that we’ve covered how topic partitions work let’s revisit that records are always appended at the end of the file. I’m sure you noticed the files in the directory example with an extension of `.log` (we’ll talk about how Kafka names this file in an upcoming section). But these `log` files aren’t 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

There have been 8 records appended so far



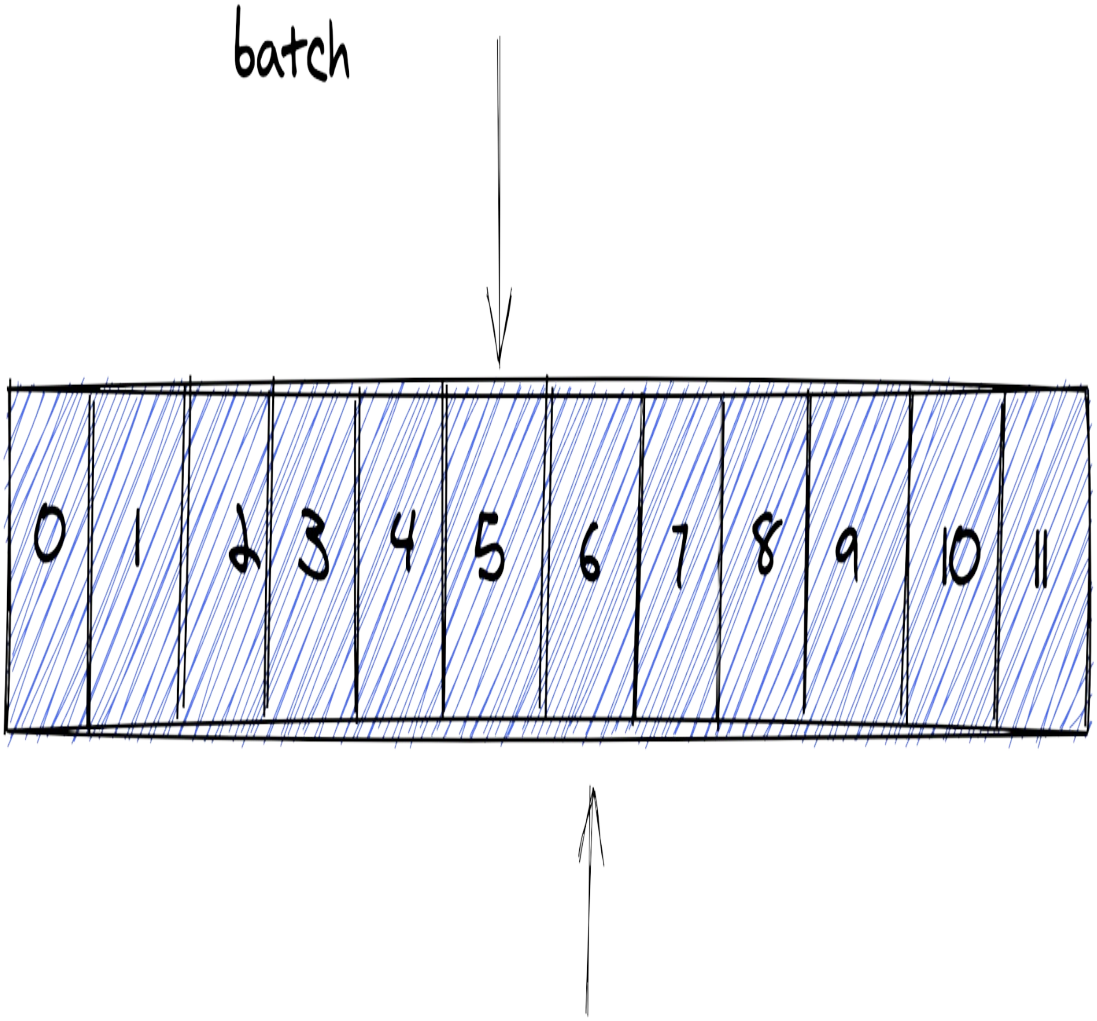
The broker will assign the next record appended an offset of 8

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

The consumer has a position of offset 5 from the previous batch



So the next batch for the consumer starts from offset 6

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\ #1
 --replication-factor 1\ #2
 --partitions 1 #3
```

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\ #1
  --broker-list localhost:9092\ #2
  --property parse.key=true\ #3
  --property key.separator=":" #4
```

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\ #1
 --bootstrap-server localhost:9092\ #2
 --from-beginning\ #3
 --property print.key=true\ #4
 --property key.separator="-" #5
```

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  #1
```

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.

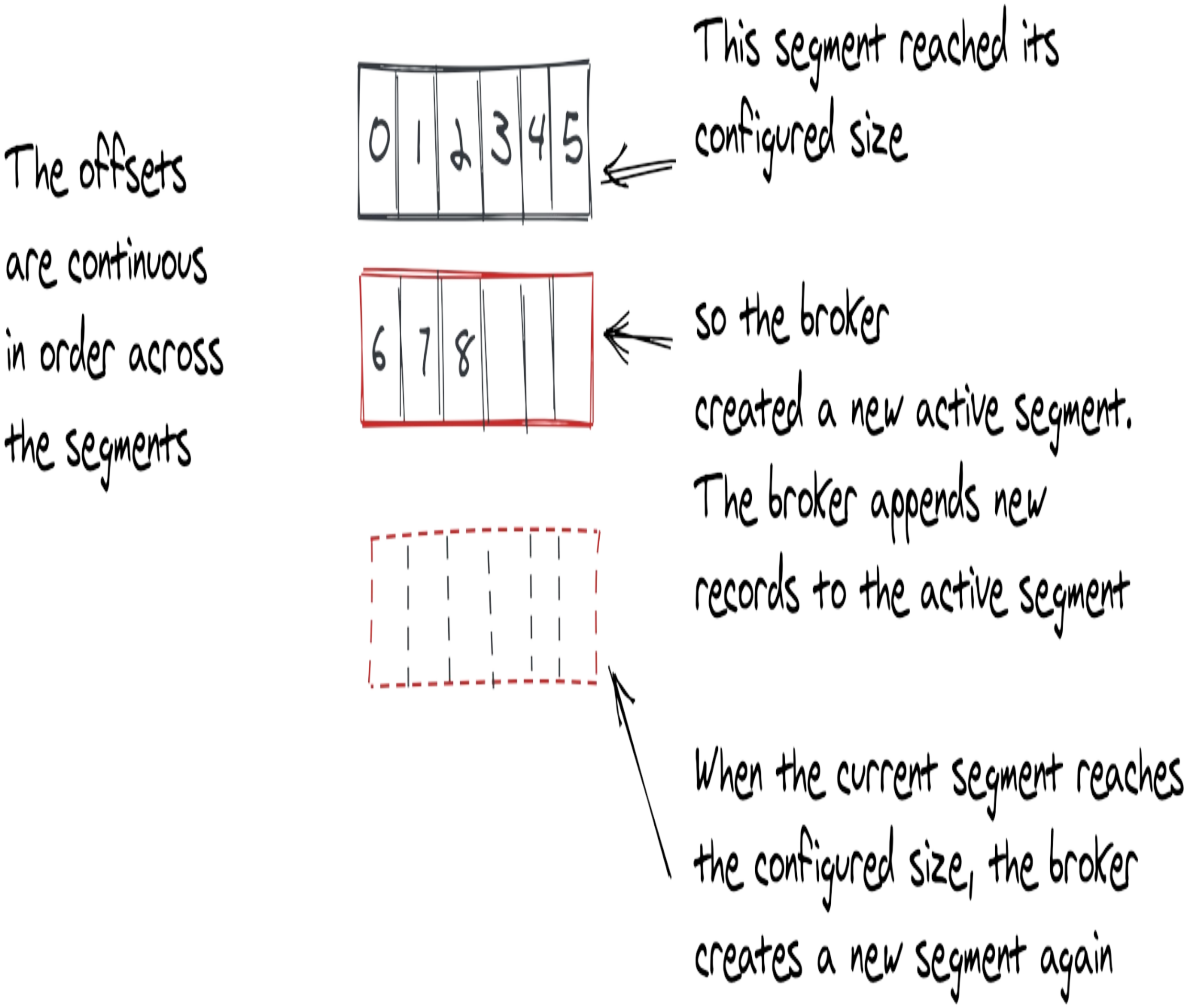
We’ve also covered the basic unit of storage the broker uses, partitions. We discussed how Kafka assigns each incoming record a unique, per partition id-the offset, and always appends records at the end of the topic partition log. But as more data flows into Kafka, do these files continue to grow indefinitely? 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



The offsets are continuous in order across the segments
0
1
2
3
4
5
6
7
8
This segment reached its configured size
so the broker created a new active segment. The broker appends new records to the active segment
When the current segment reaches the configured size, the broker creates a new segment again

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.

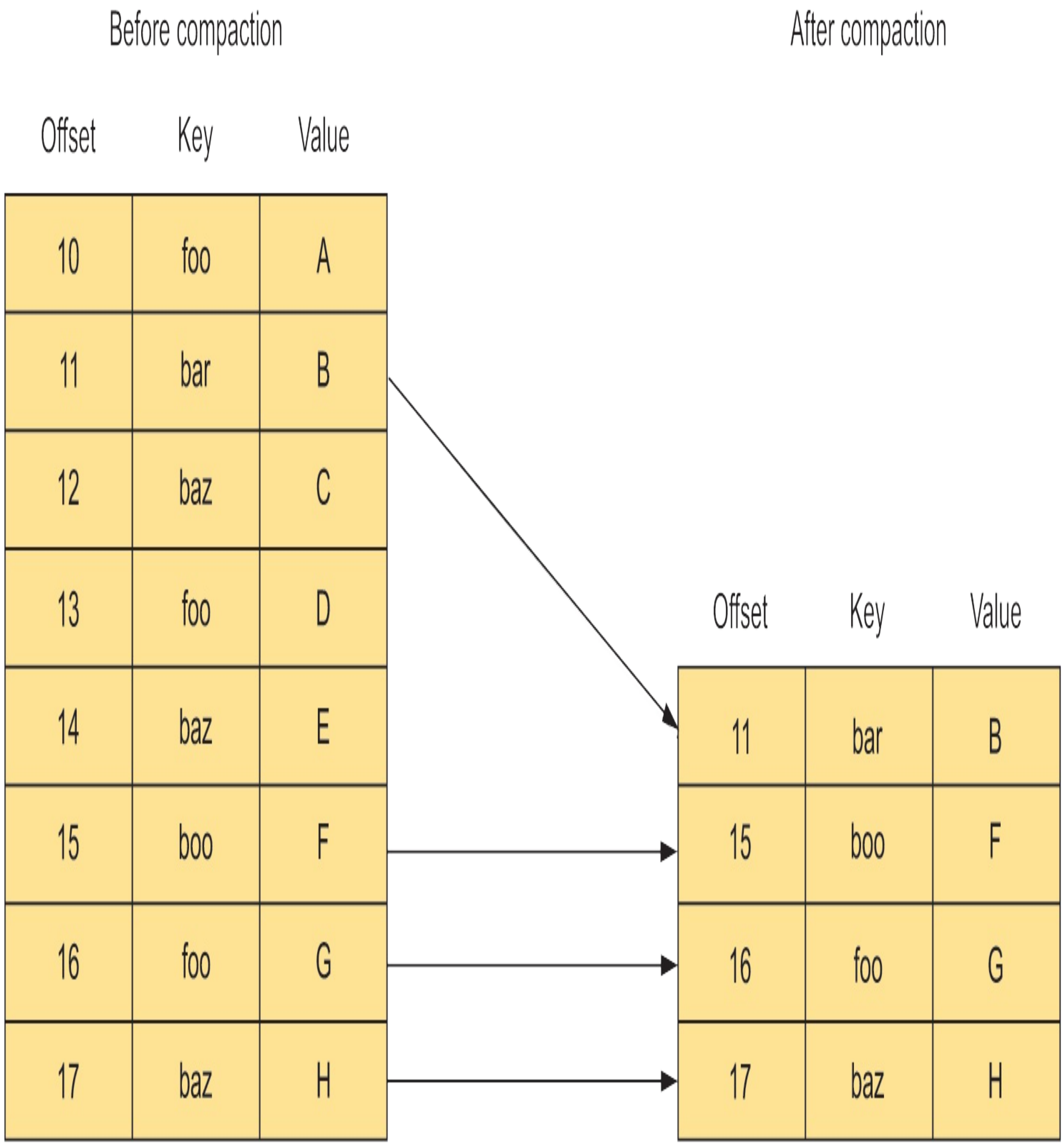
2.5.2 Compacted topics

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.



Before compaction

Offset	Key	Value
10	foo	A
11	bar	B
12	baz	C
13	foo	D
14	baz	E
15	boo	F
16	foo	G
17	baz	H

After compaction

Offset	Key	Value
11	bar	B
15	boo	F
16	foo	G
17	baz	H

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
```

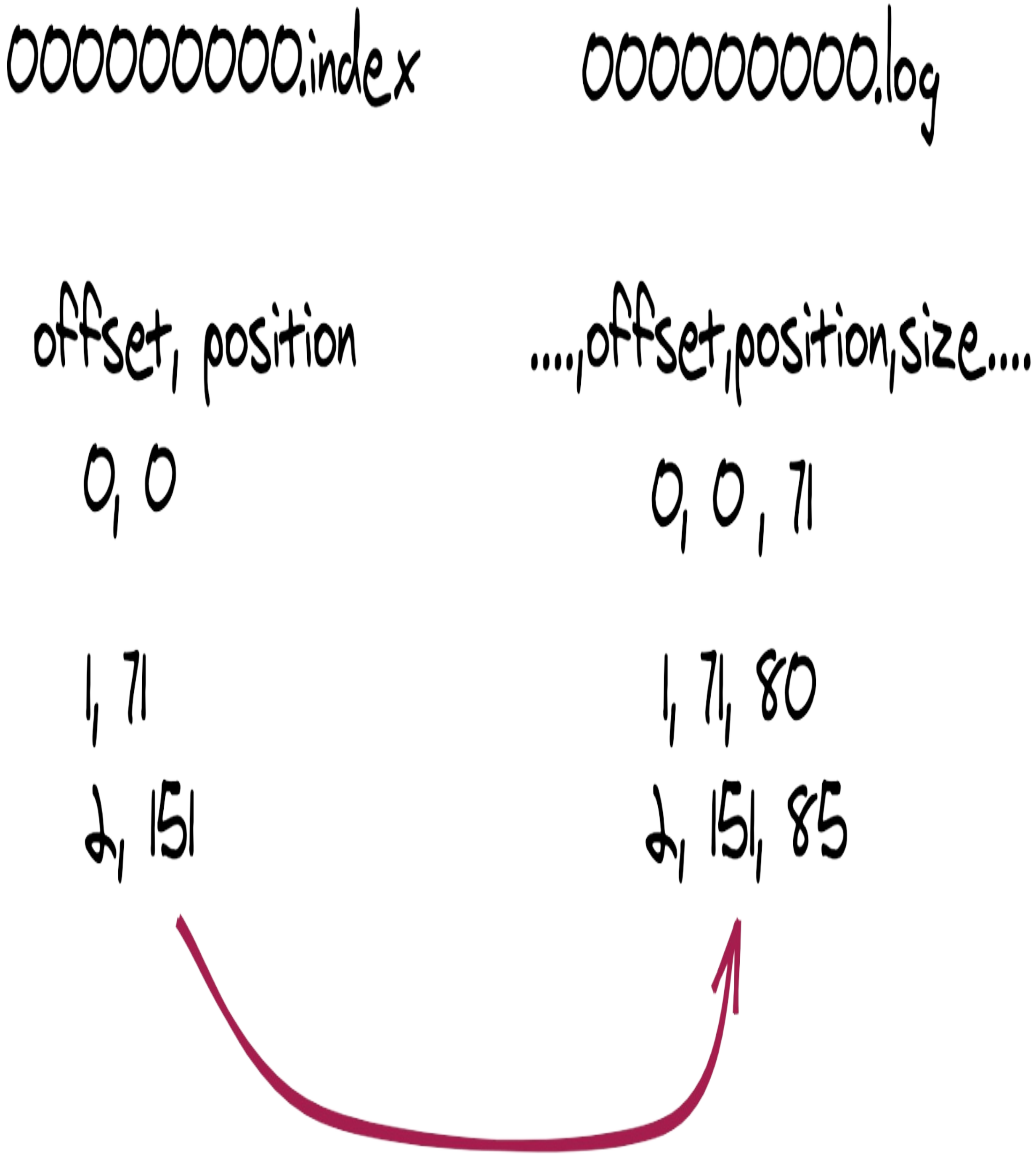
Note

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



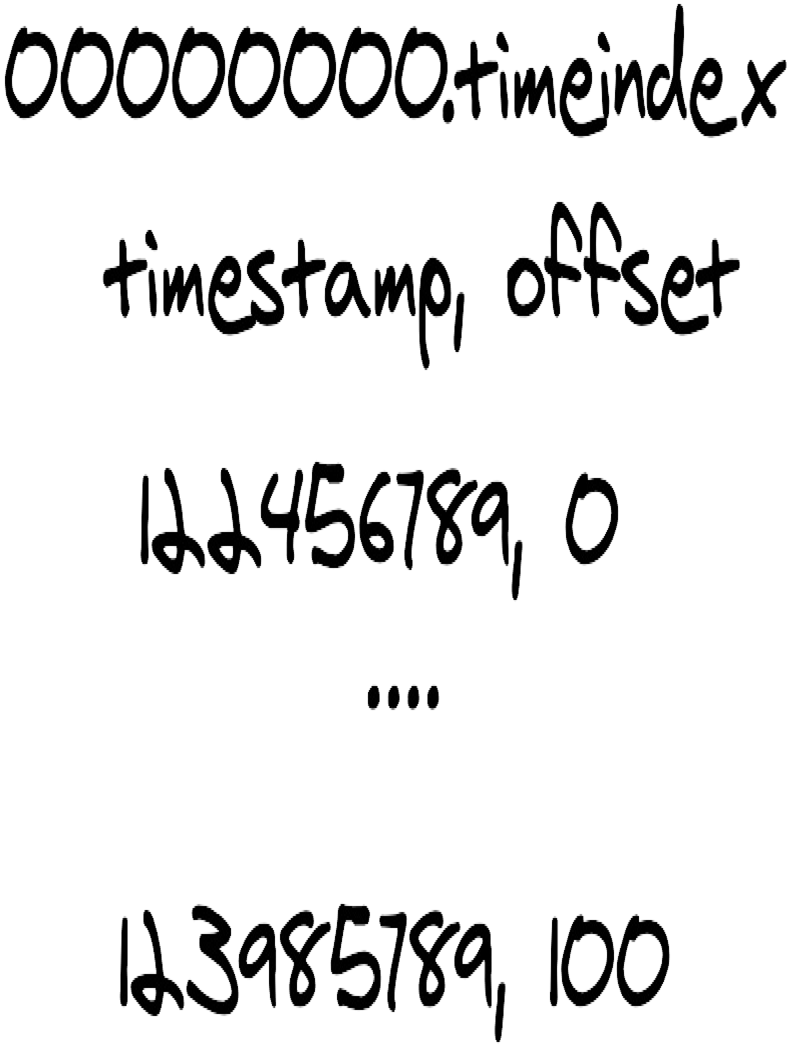
Brokers use the index files to find the starting point for retrieving records based on the given offset. The brokers do a binary search in the `index` file, looking for an index-position pair with the largest offset that is less than or equal to the target offset. The offset stored in the `index` file 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



```
00000000.timeindex
timestamp, offset
122456789, 0
....
123985789, 100
```

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, 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.

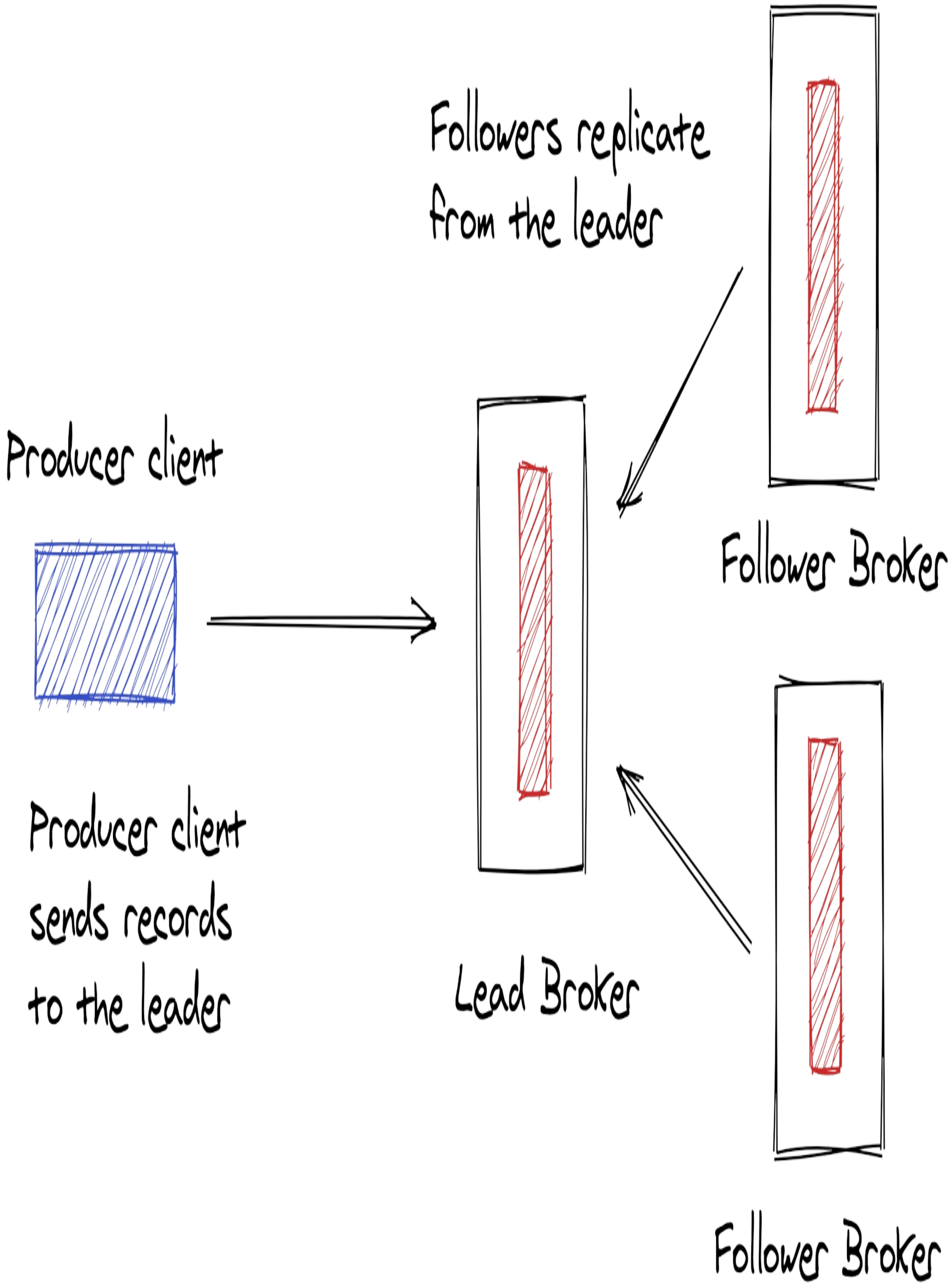
This has been a quick overview of how Kafka manages metadata. I don’t want to go into too much detail about metadata management as my approach to this book is more from the developer’s point of view and not someone who will manage a Kafka cluster. Now that we’ve briefly discussed Kafka’s need for metadata 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


Followers replicate from the leader
Producer client
Follower Broker
Producer client sends records to the leader
Lead Broker
Follower Broker

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.

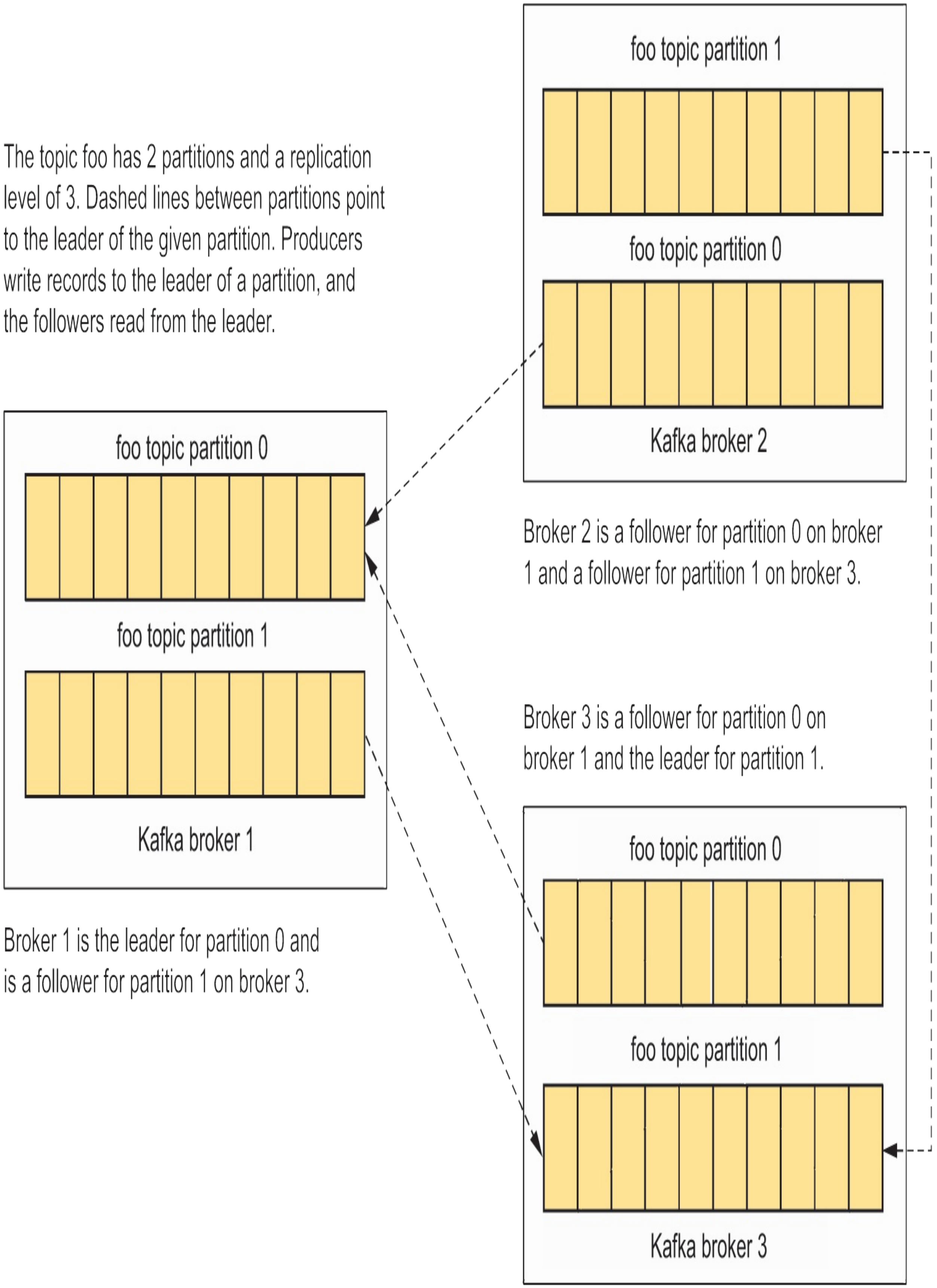
But before we discuss how leaders, followers, and replication work, we need to consider what Kafka does to achieve this.

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 topic foo has 2 partitions and a replication level of 3. Dashed lines between partitions point to the leader of the given partition. Producers write records to the leader of a partition, and the followers read from the leader.

Broker 2 is a follower for partition 0 on broker 1 and a follower for partition 1 on broker 3.

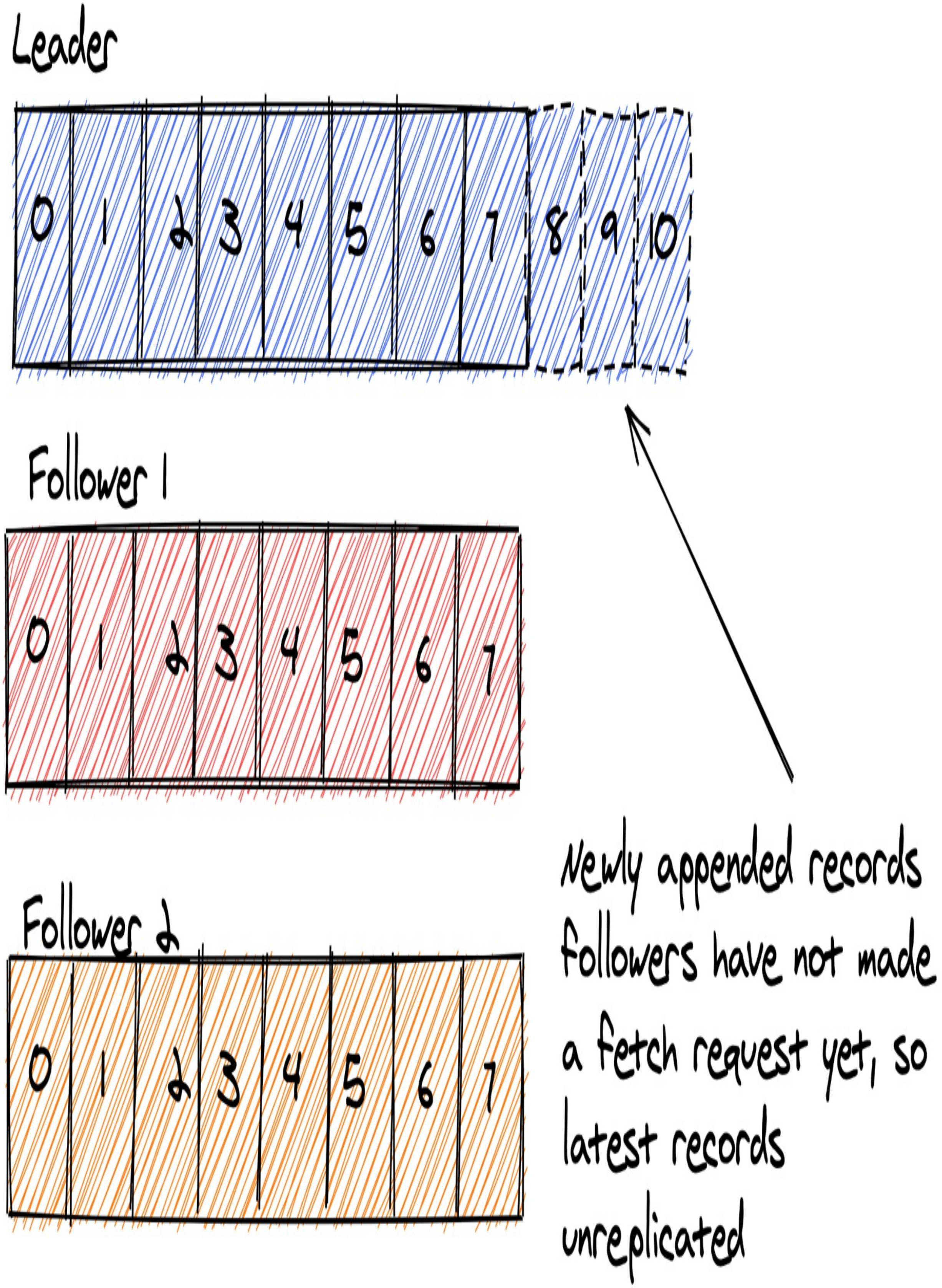
Broker 3 is a follower for partition 0 on broker 1 and the leader for partition 1.

Broker 1 is the leader for partition 0 and is a follower for partition 1 on broker 3.

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.

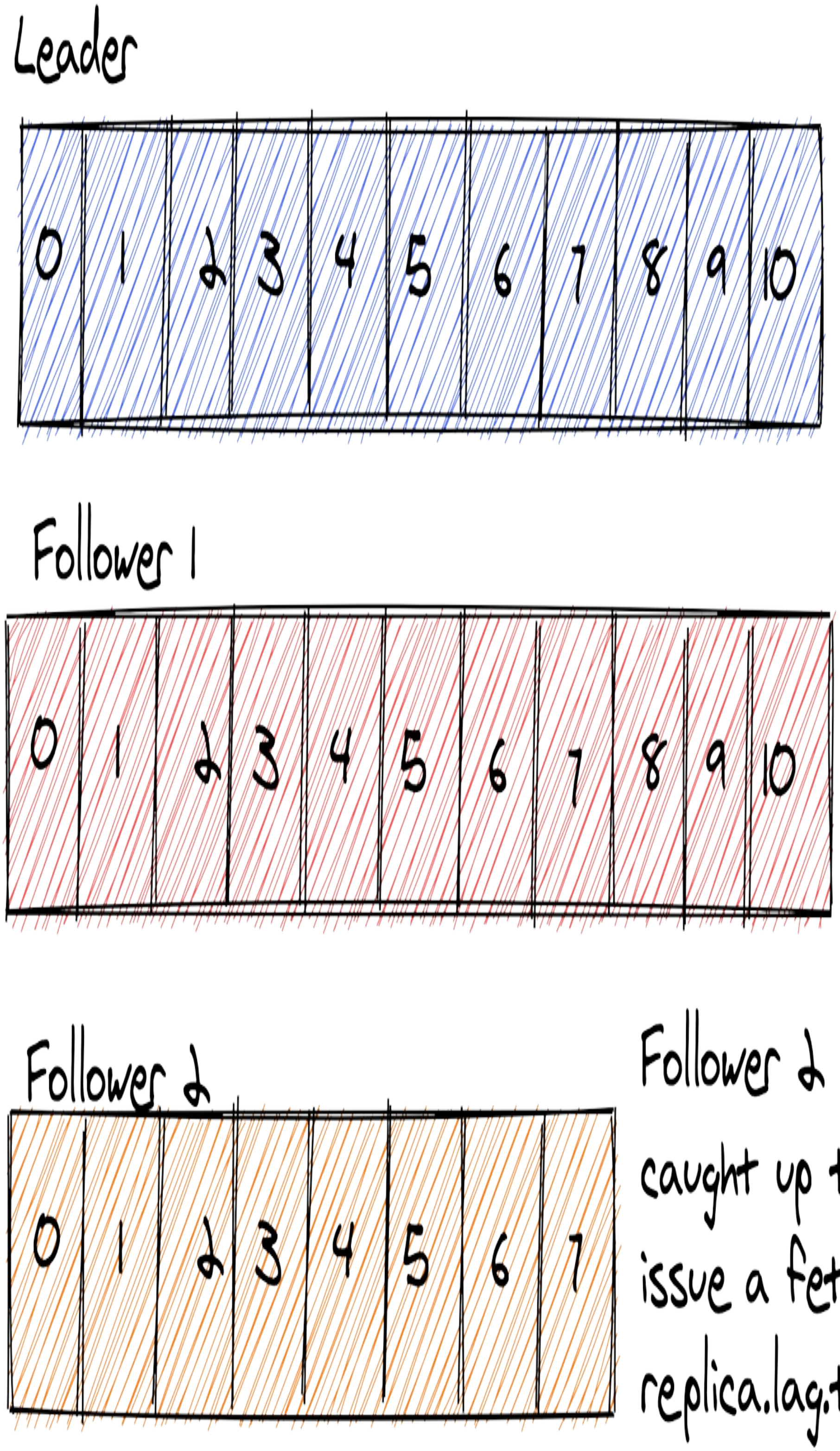
When a producer sends a batch of records, the leader must first append those records before the followers can replicate them. There is a small window of time where the leader will be ahead of the followers. This illustration demonstrates this concept:

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


Leader
0
1
2
3
4
5
6
7
8
9
10
Follower 1
0
1
2
3
4
5
6
7
Follower 2
0
1
2
3
4
5
6
7
Newly appended records followers have not made a fetch request yet, so latest records unreplicated

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



Follower 2 must be fully caught up to the leader or issue a fetch request within replica.lag.time.max.ms or it's considered out of sync

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.

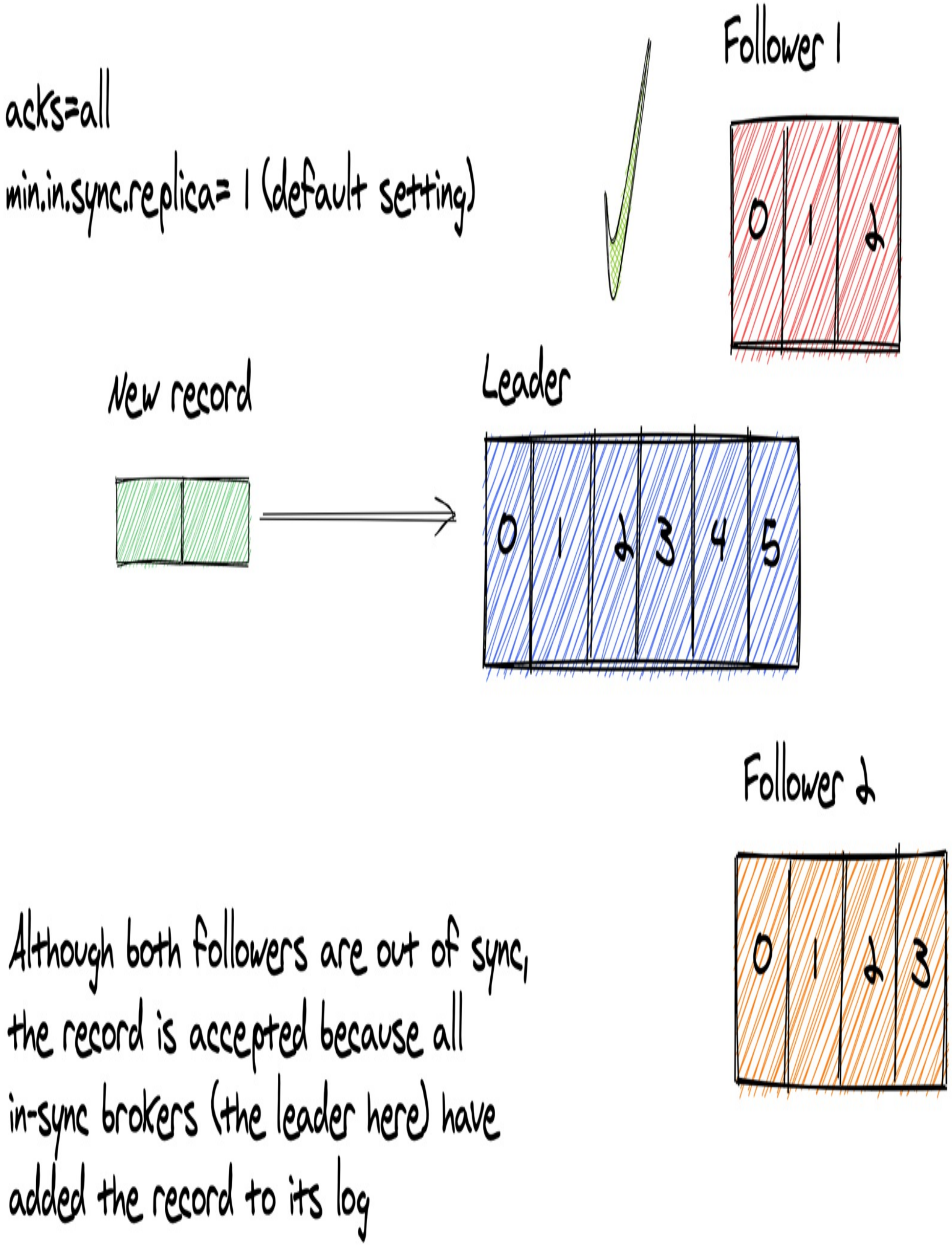
If the leader for a topic-partition fails, a follower has a complete replica of the leader’s log. But we should explore the relationship between leaders, followers, and replicas.

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



acks=all

min.in.sync.replica= 1 (default setting)

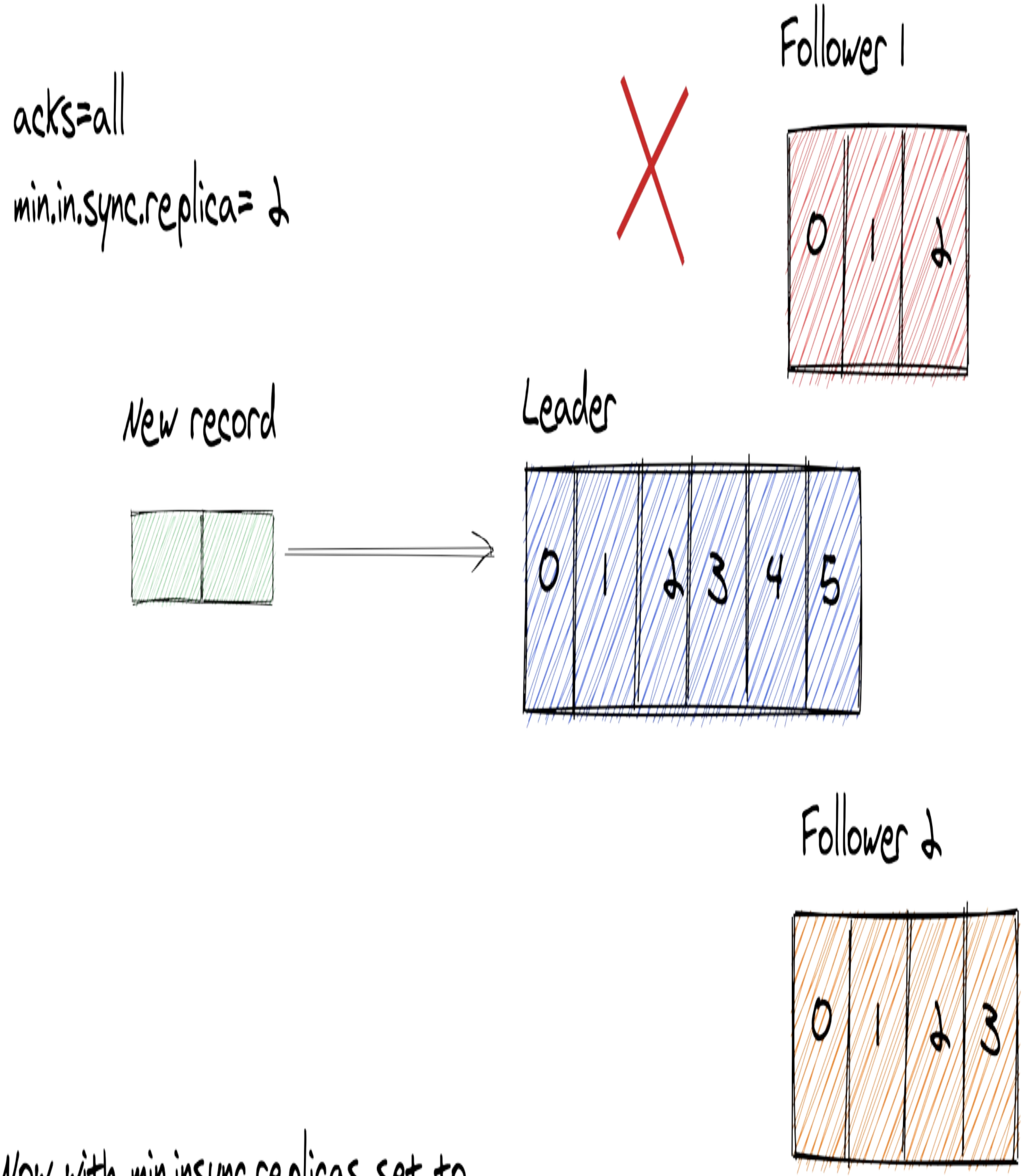
Although both followers are out of sync, the record is accepted because all in-sync brokers (the leader here) have added the record to its log

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



acks=all

min.in.sync.replica= 2

Now with min.insync.replicas set to two, new records rejected with a NotEnoughReplicas exception

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

[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.

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

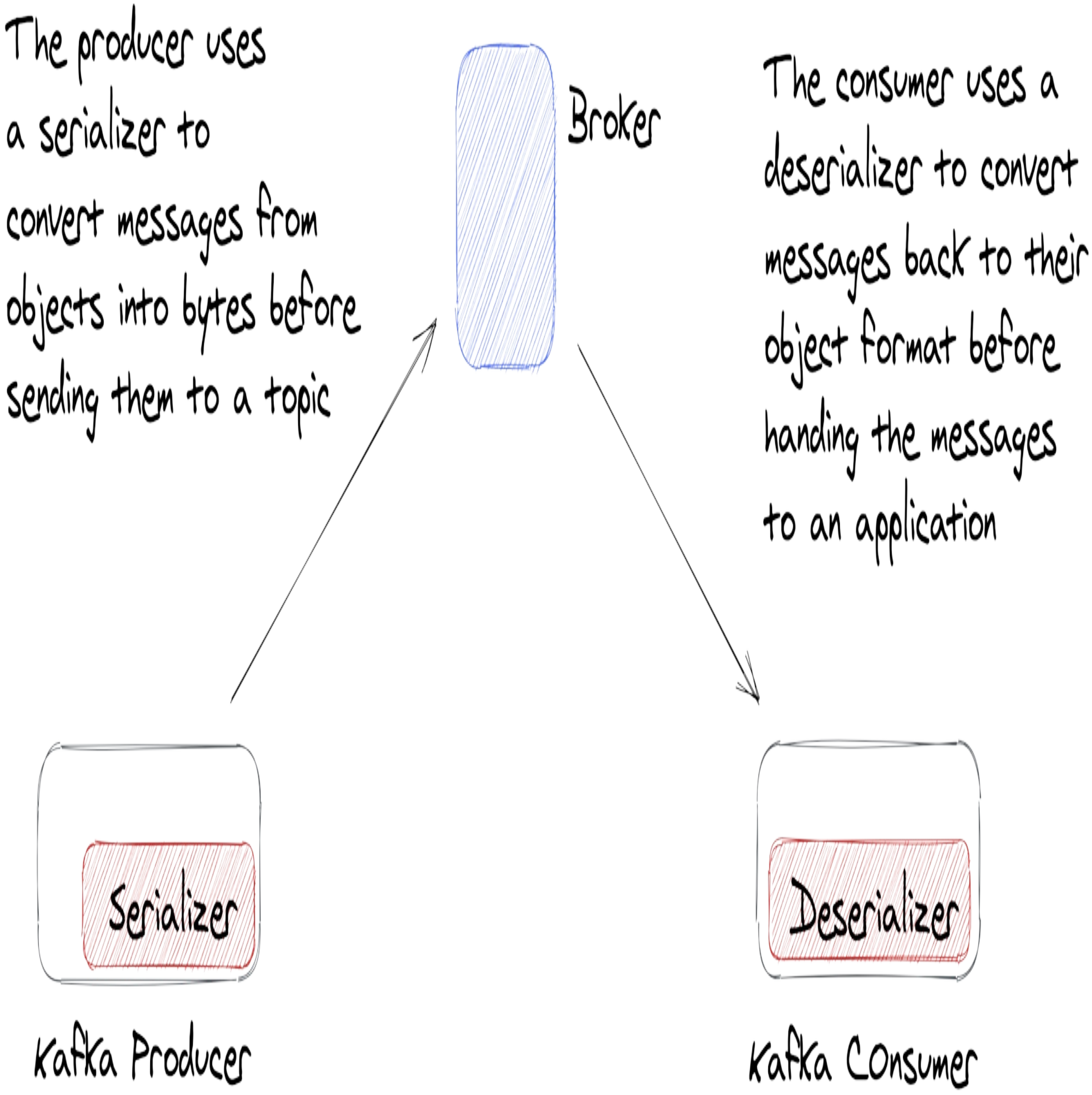
In chapter 2, you learned about the heart of the Kafka streaming platform, the Kafka broker. In particular, you learned how the broker is the storage layer appending incoming messages to a topic, serving as an immutable, distributed log of events. A topic represents the directory containing the log file(s).

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.

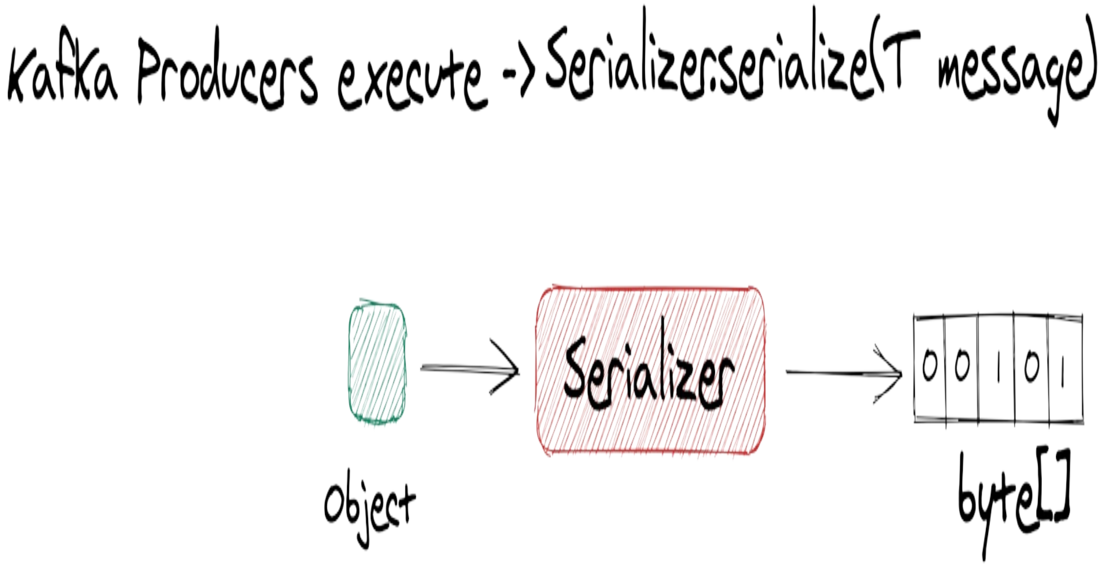
Figure 3.1. The conversion of objects to bytes and bytes to objects happens at the client level

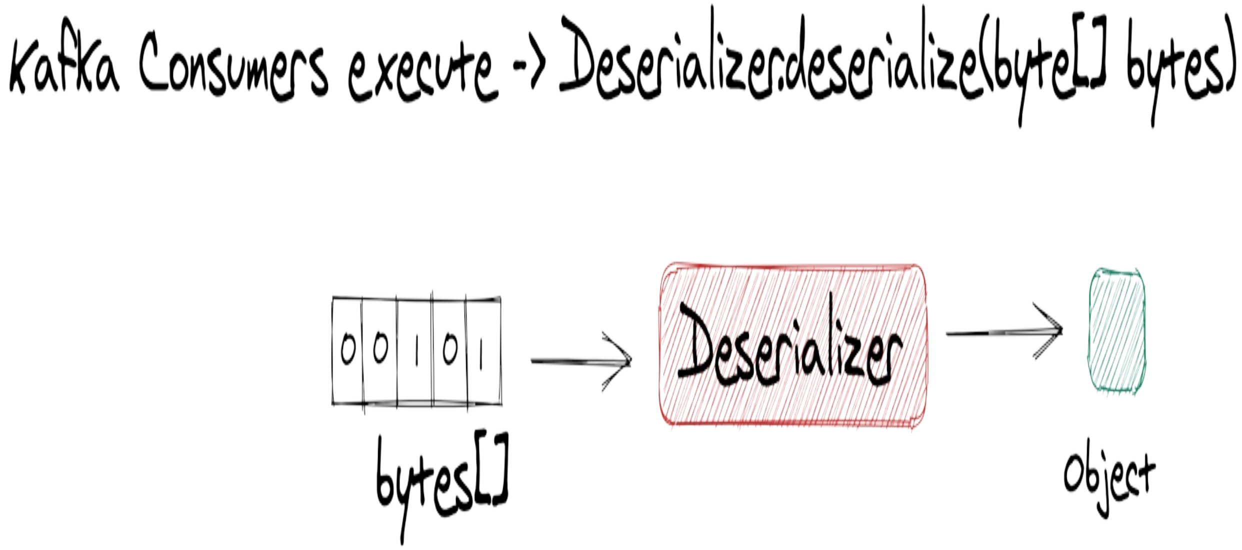

The producer uses a serializer to convert messages from objects into bytes before sending them to a topic
Broker
The consumer uses a deserializer to convert messages back to their object format before handing the messages to an application
Serializer
Kafka Producer
Deserializer
Kafka Consumer

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




Kafka Consumers execute -> Deserializer.deserialize(byte[] bytes)
0 0 1 0 1
bytes[]
Deserializer
Object

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",   #1
  "fields": [      #2
    {"name": "name", "type":"string"}, #3
    {"name": "age", "type": "int"},
    {"name": "email", "type":"string"}
  ]
}
```

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

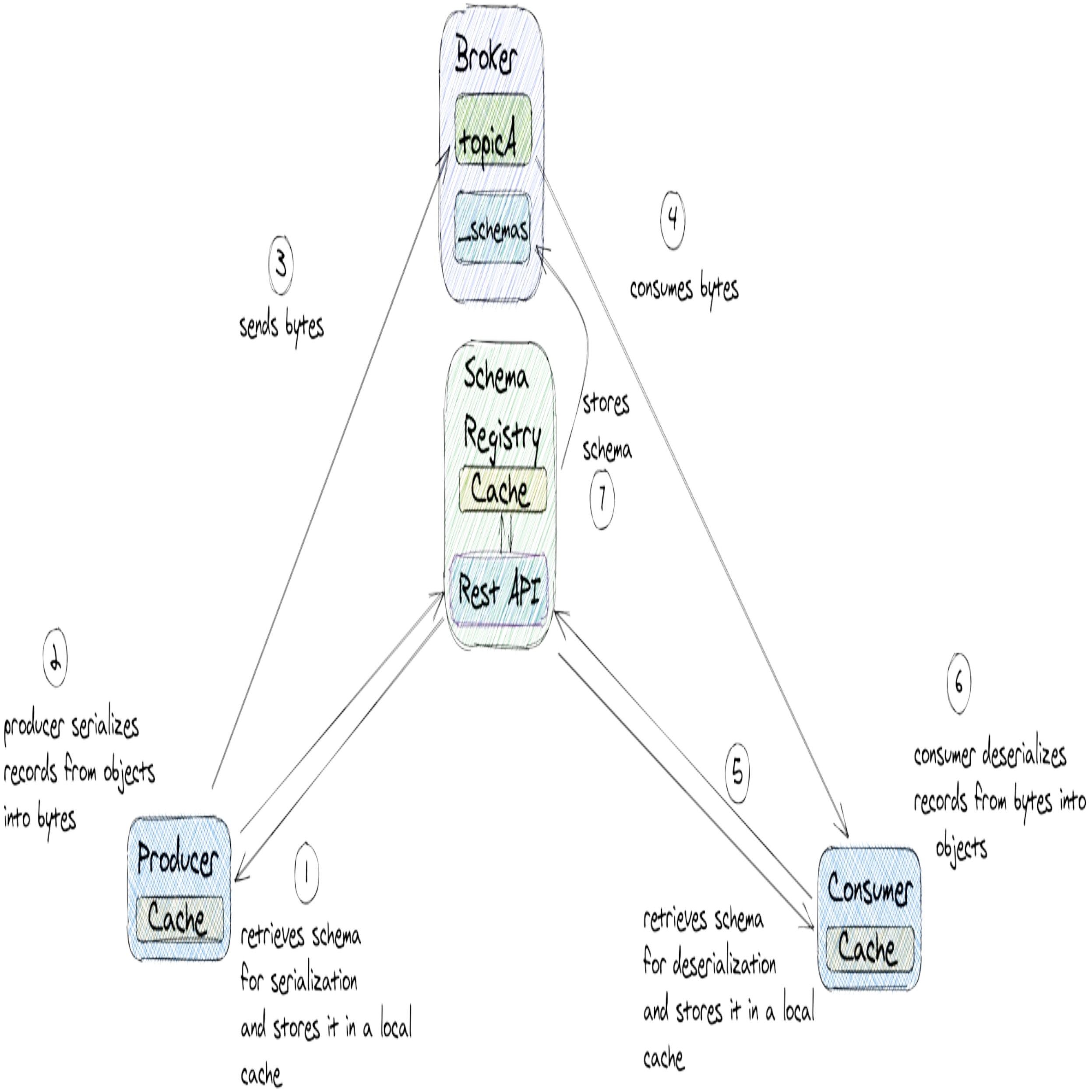
The answer to these *how* questions is Schema Registry.

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


Broker
topicA
_schemas
3
sends bytes
4
consumes bytes
Schema Registry
Cache
Rest API
stores schema
7
2
producer serializes records from objects into bytes
Producer
Cache
1
retrieves schema for serialization and stores it in a local cache
5
retrieves schema for deserialization and stores it in a local cache
6
consumer deserializes records from bytes into objects
Consumer
Cache

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.

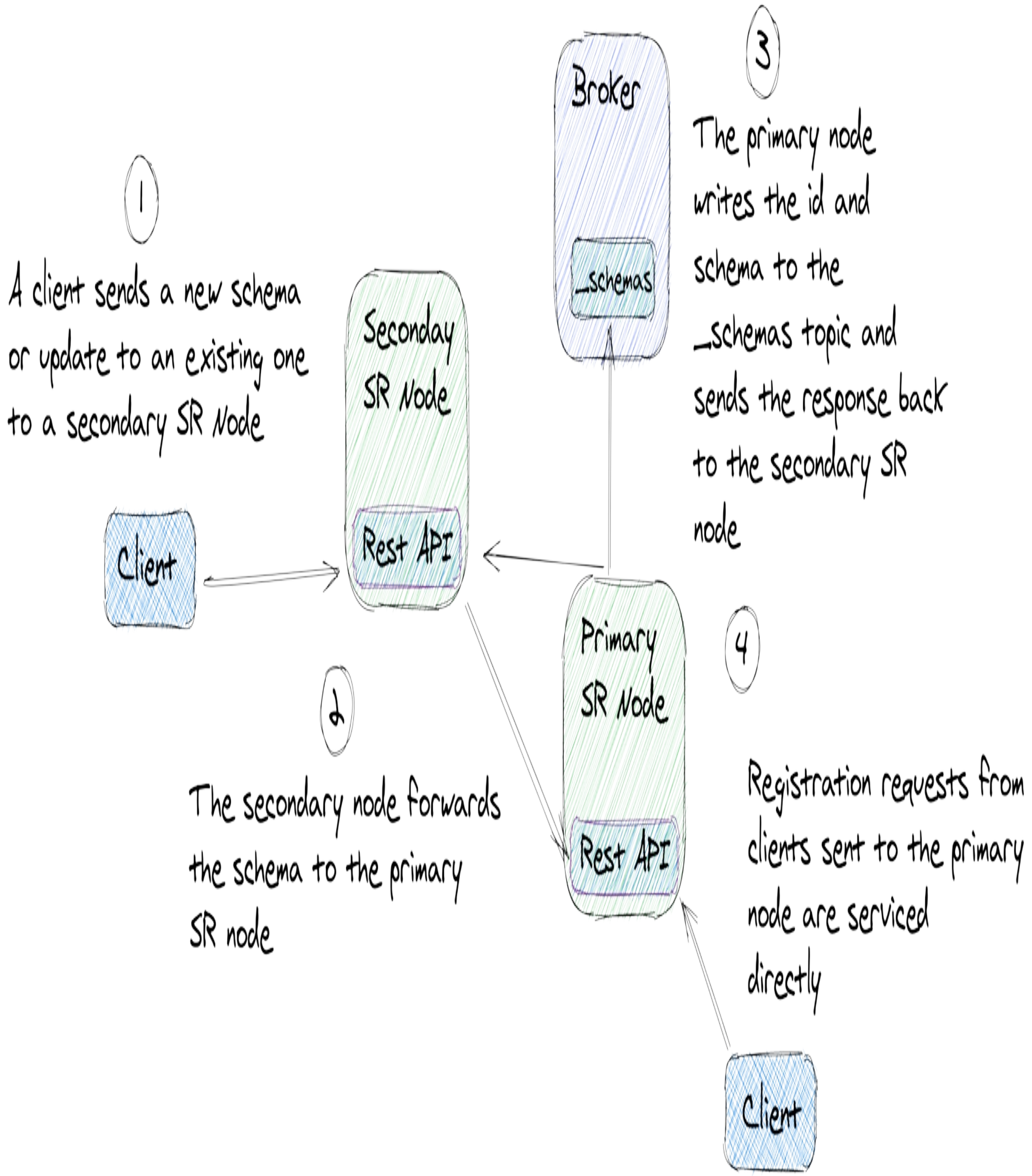
Schema Registry uses Kafka as storage (write-ahead-log) of all its schemas in `__schemas` which is a single partitioned, compacted topic. It has a primary architecture meaning there is one leader node in the deployment and the other nodes are secondary.

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


3
Broker
_schemas
The primary node writes the id and schema to the _schemas topic and sends the response back to the secondary SR node
1
A client sends a new schema or update to an existing one to a secondary SR Node
Seconday SR Node
Rest API
Client
Primary SR Node
4
2
The secondary node forwards the schema to the primary SR node
Rest API
Registration requests from clients sent to the primary node are serviced directly
Client

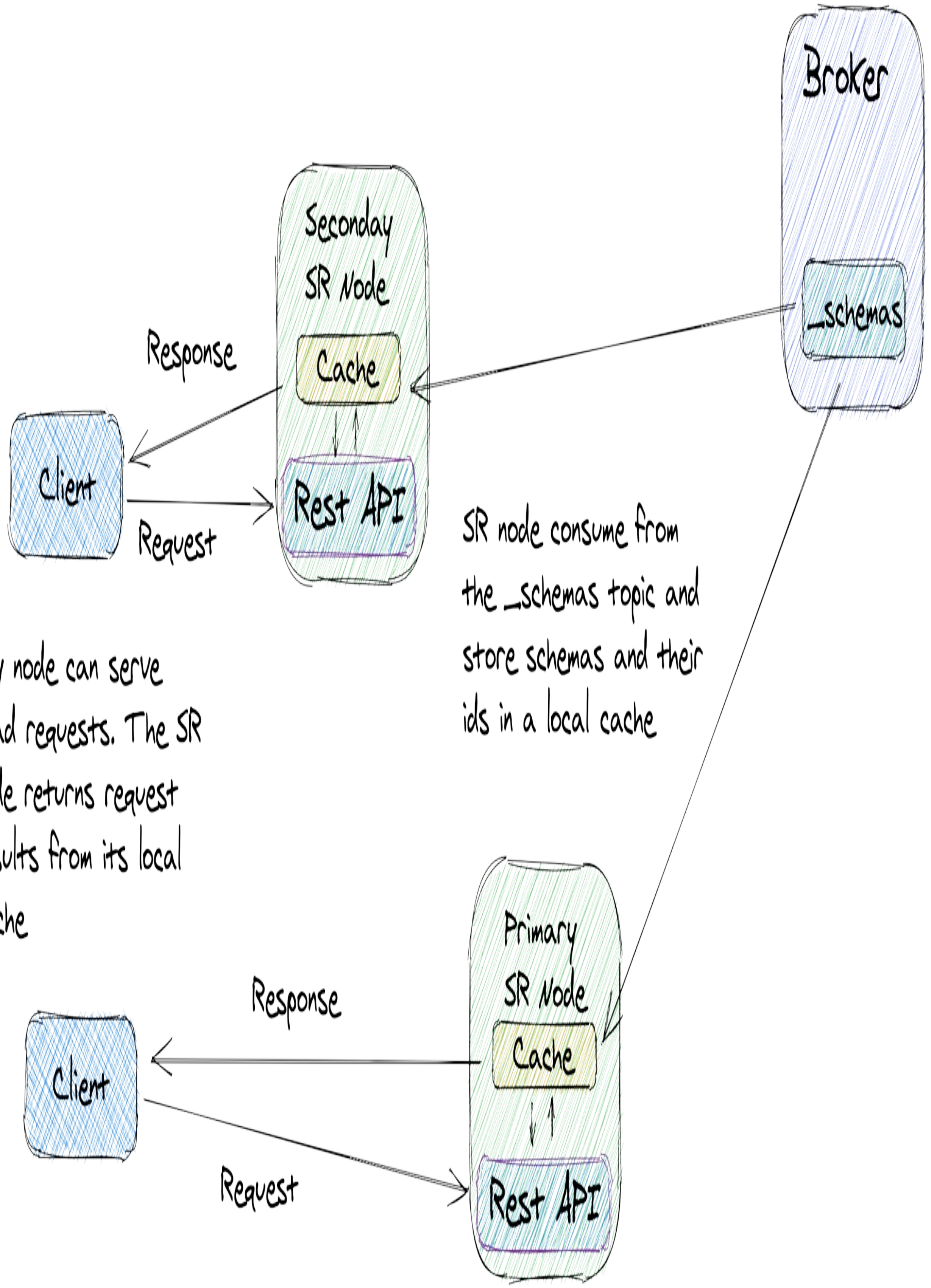
Anytime a client registers or updates a schema, the primary node produces a record to the {underscore}schemas topic. Schema Registry uses a Kafka producer for writing and all the nodes use a consumer for reading updates. So you can see that Schema Registry’s local state is backed up in a Kafka topic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



SR node consume from the _schemas topic and store schemas and their ids in a local cache

Any node can serve read requests. The SR node returns request results from its local cache

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", #1
 "type": "record",          #2
 "name": "Avenger",         #3
 "fields": [                #4
     {"name": "name", "type": "string"},
     {"name": "real_name", "type": "string"},         #5
     {"name": "movies", "type":
                      {"type": "array", "items": "string"},
      "default": []   #6
    }
  ]
}
```

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";  #1

package bbejeck.chapter_3.proto;  #2

option java_outer_classname = "AvengerProto"; #3

message Avenger {   #4
    string name = 1;  #5
    string real_name = 2;
    repeated string movies = 3;  #6
}
```

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#",  #1
  "title": "Avenger",
  "description": "A JSON schema of Avenger object",
  "type": "object",             #2
  "javaType": "bbejeck.chapter_3.json.SimpleAvengerJson", #3
  "properties": {     #4
    "name": {
      "type": "string"
    },
    "realName": {
      "type": "string"
    },
    "movies": {
      "type": "array",
      "items": {
        "type": "string"
      },
      "default": []  #5
    }
  },
  "required": [
    "name",
    "realName"
  ]
}
```

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 | \   #1
curl -s -X POST http://localhost:8081/subjects/avro-avengers-valu
         -H "Content-Type: application/vnd.schemaregistry.v1+json
         -d @-  \ #4
```

```
        | jq #5
```

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/versi
```

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/versi
 | 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 {   #1
    url = 'http://localhost:8081' #2


register {
        subject('avro-avengers-value', #3
                'src/main/avro/avenger.avsc', #4
                'AVRO') #5

     //other entries left out for clarity
    }

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

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("${project.projectDir}/src/main/json") #1
  targetDirectory = file("${project.buildDir}/generated-main-json
  // other configurations left out for clarity
}
```

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" #1
    protoc {
        artifact = 'com.google.protobuf:protoc:3.15.3' #2
    }
}
```

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 #1

./gradlew clean build #2
```

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); #1
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_
    "http://localhost:8081"); #2
```

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 - Producin
 [{"name": "Black Widow", "real_name": "Natasha Romanova", "movie
["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); #1
consumerProps.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READE
    true); #2
consumerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_
    "http://localhost:8081"); #3
```

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();   #1
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")) {  #2
    genericRecord.get("real_name");
}

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

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

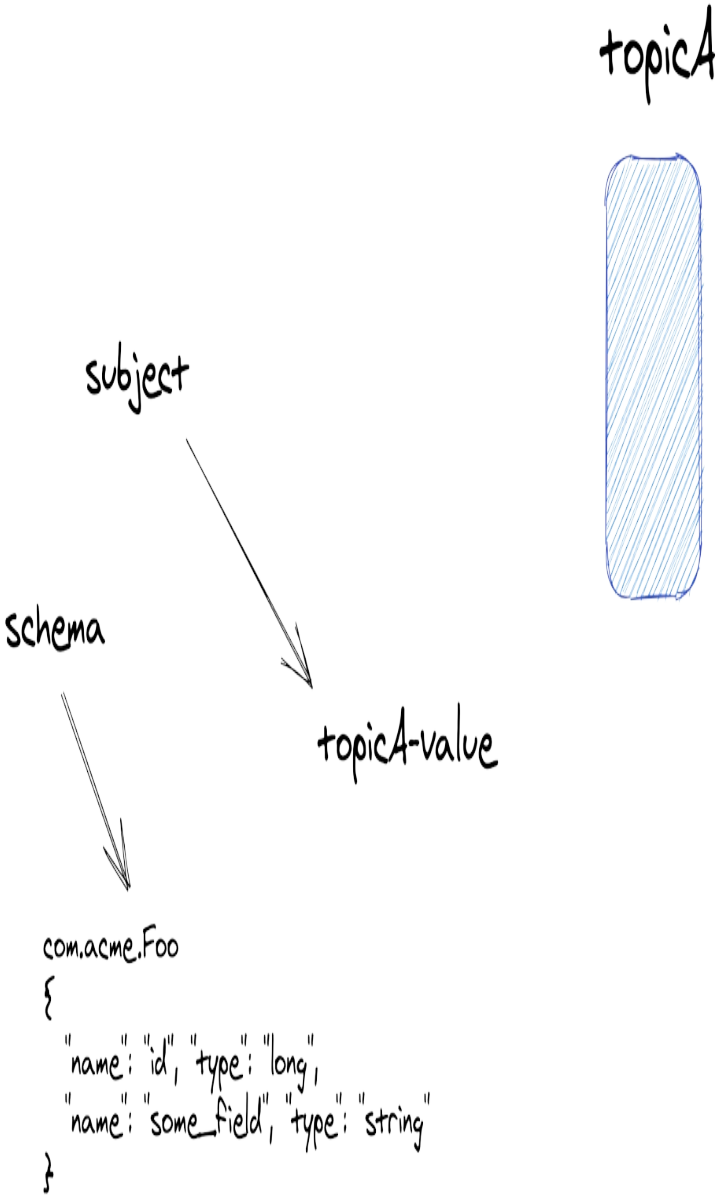
The `TopicNameStrategy` is the default subject in Schema Registry. The subject name comes from the name of the topic. You saw the `TopicNameStrategy` in action earlier in the chapter when you registered a schema with the gradle plugin. To be more precise the subject name is `topic-name-key` or `topic-name-value`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

TopicNameStrategy



```
com.acme.Foo
{
  "name": "id", "type": "long",
  "name": "some_field", "type": "string"
}
```

Here the registered schema is <topic-name>-value. This restricts the type contained in the topic to that of the registered schema for the value type.

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.

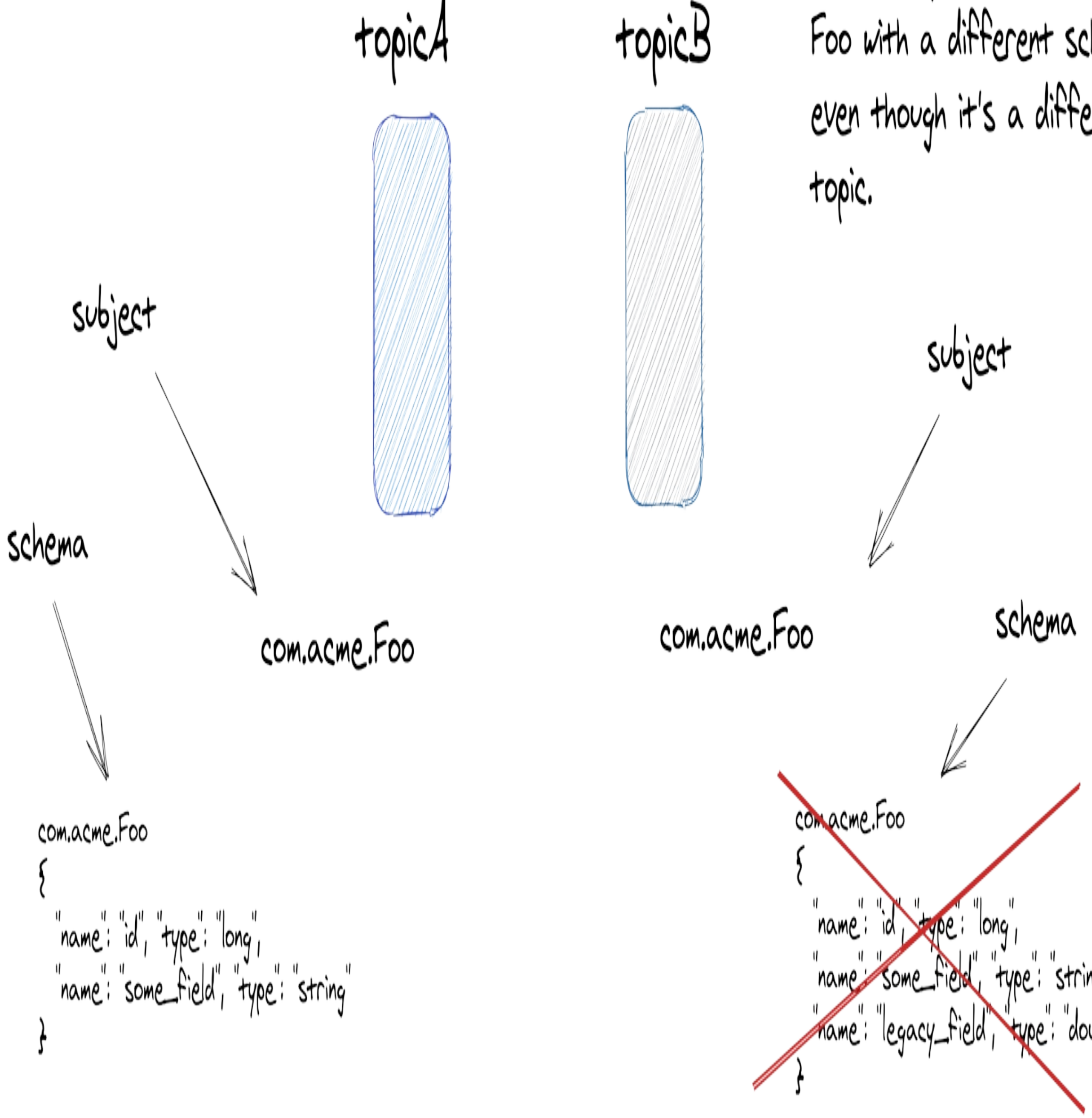
3.2.2 RecordNameStrategy

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

RecordNameStrategy



You can't produce records for Foo with a different schema even though it's a different topic.

```
com.acme.Foo
{
  "name": "id", "type": "long",
  "name": "some_field", "type": "string"
}
```

~~com.acme.Foo~~
~~{~~
~~"name": "id", "type": "long",~~
~~"name": "some_field", "type": "string",~~
~~"name": "legacy_field", "type": "double"~~
~~}~~

The domain object "Foo" has different schemas and since you're using the RecordNameStrategy, Schema Registry enforces schema compatibility across all topics for the given record subject name

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/aveng
```

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_
 RecordNameStrategy.class);
producerConfig.put(KafkaAvroSerializerConfig.KEY_SUBJECT_NAME_ST
 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_STRATE
 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.

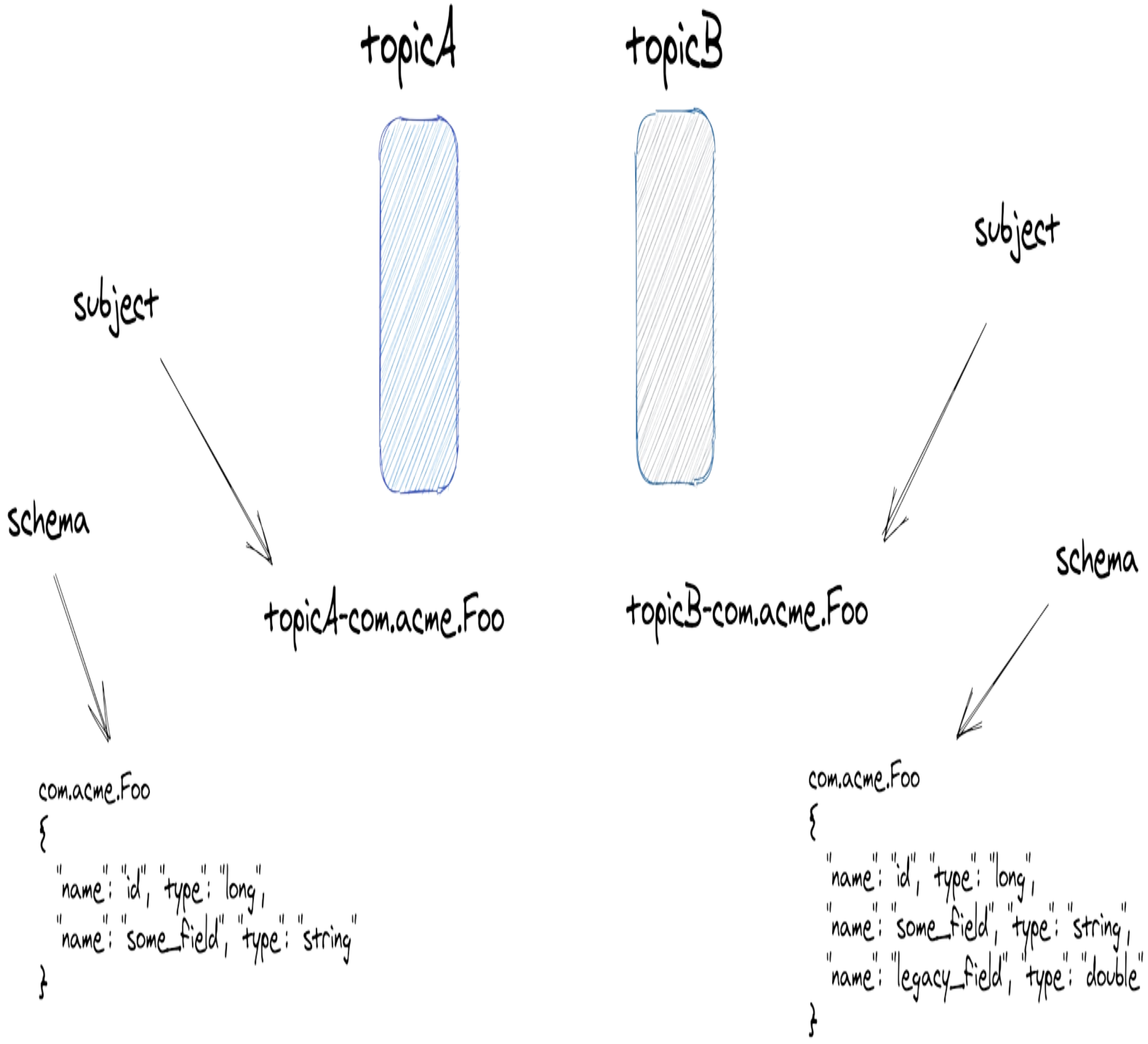
One thing to consider is that by using only the record name, all topics must use the same schema. If you want to use different records in a topic, but want to only consider the schemas for that particular topic, then you’ll need to use another strategy.

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

TopicRecordNameStrategy



```
com.acme.Foo
{
  "name": "id", "type": "long",
  "name": "some_field", "type": "string"
}
```

```
com.acme.Foo
{
  "name": "id", "type": "long",
  "name": "some_field", "type": "string",
  "name": "legacy_field", "type": "double"
}
```

The domain object "Foo" has different schemas but since you're using the TopicRecordNameStrategy this is allowed as Schema Registry only checks for schema compatibility for the record type within the scope of the topic.

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_
  TopicRecordNameStrategy.class);
 producerConfig.put(KafkaAvroSerializerConfig.KEY_SUBJECT_NAME_ST
  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_STRATE
 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.

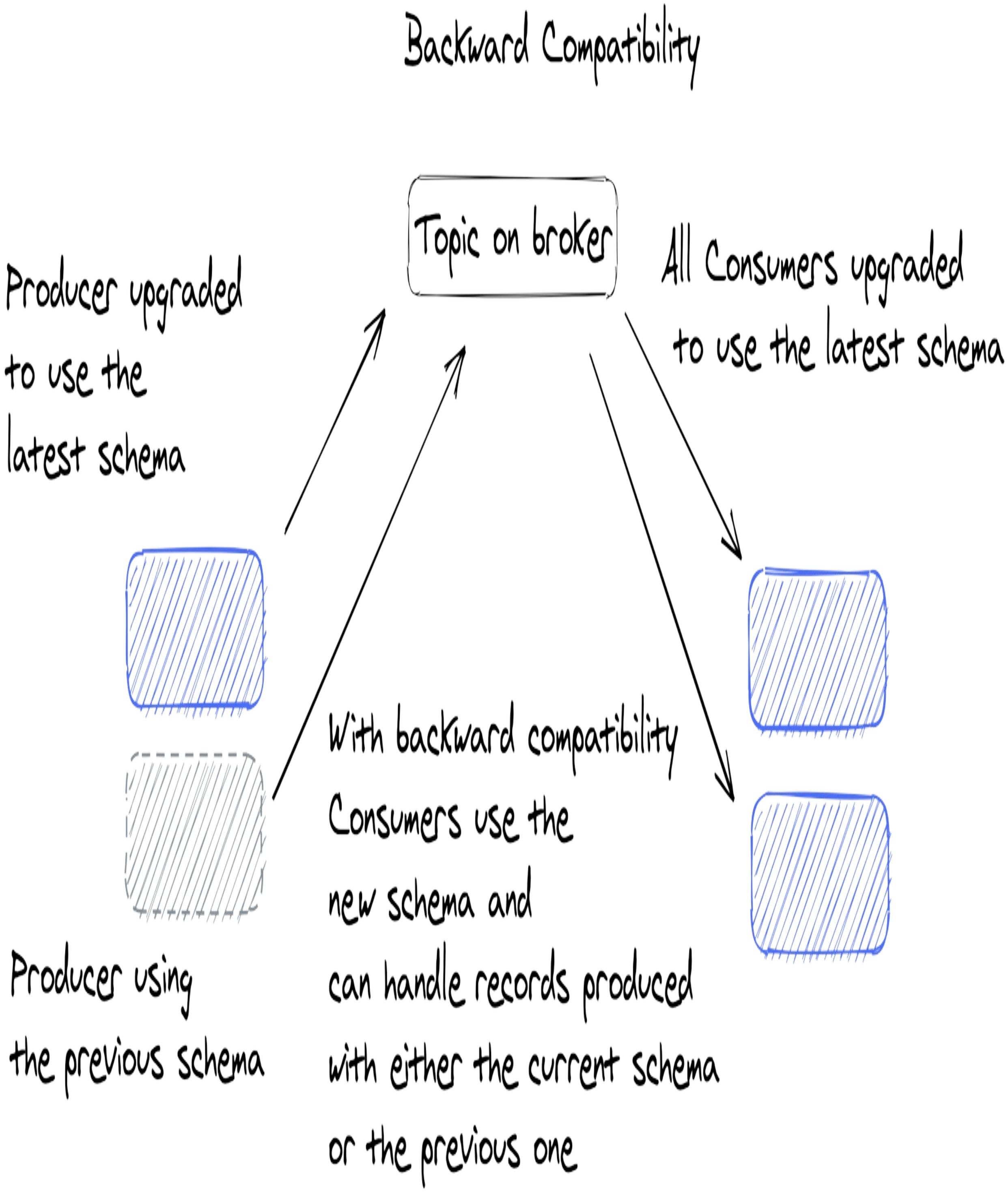
You can specify a global compatibility level or a compatibility level per subject.

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



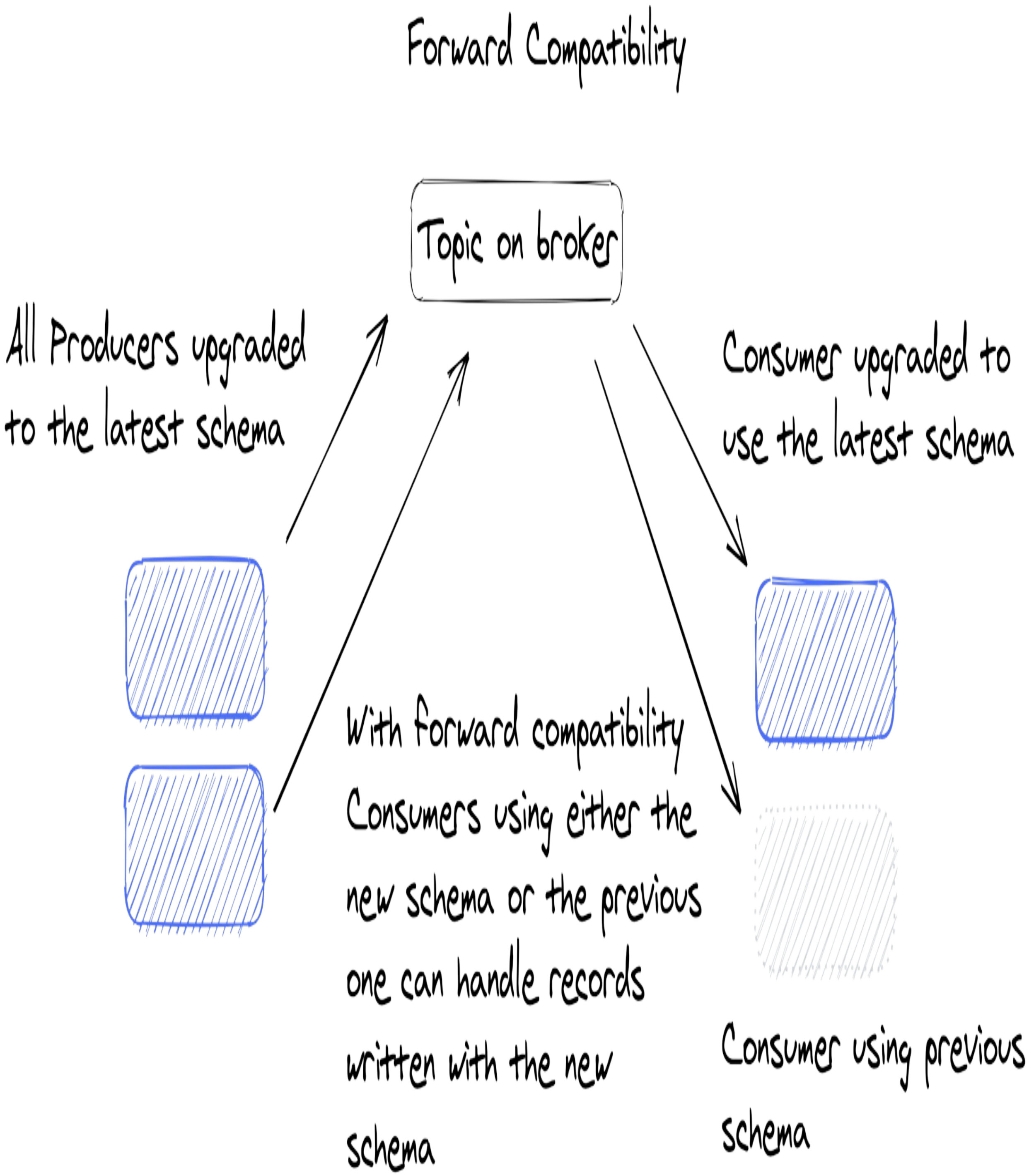
Backward Compatibility
Topic on broker
Producer upgraded to use the latest schema
All Consumers upgraded to use the latest schema
Producer using the previous schema
With backward compatibility Consumers use the new schema and can handle records produced with either the current schema or the previous 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


Forward Compatibility
Topic on broker
All Producers upgraded to the latest schema
Consumer upgraded to use the latest schema
With forward compatibility Consumers using either the new schema or the previous one can handle records written with the new schema
Consumer using previous schema

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.

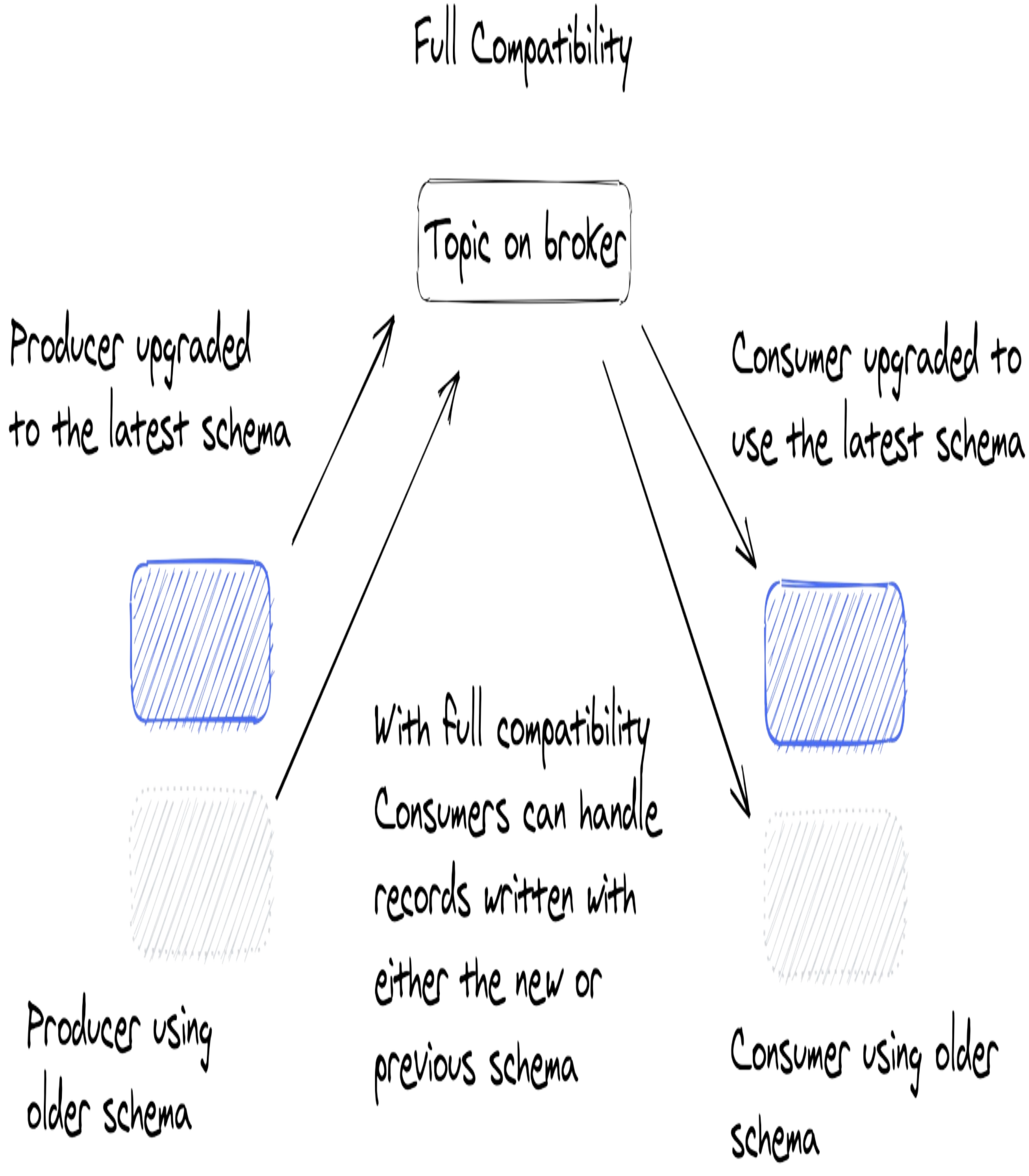
3.3.3 Full compatibility

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.

ℹ **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


Full Compatibility
Topic on broker
Producer upgraded to the latest schema
Consumer upgraded to use the latest schema
With full compatibility Consumers can handle records written with either the new or previous schema
Producer using older schema
Consumer using older schema

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": { #1
      "namespace": "bbejeck.chapter_3.avro",
      "name":"PersonAvro",       #2
      "fields": [
        {"name": "name", "type":"string"},
        {"name": "address", "type": "string"},
        {"name": "age", "type": "int"}
      ]
    }},
      "default": []
    }
  ]
}
```

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": {  #1
      "type":"record",
      "namespace": "bbejeck.chapter_3.avro",
      "name":"PersonAvro",        #2
      "fields": [
        {"name": "name", "type":"string"},
        {"name": "address", "type": "string"},
        {"name": "age", "type": "int"}
      ]
    }},
      "default": []
    }
  ]
}
```

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
      "default": []
    }
  ]
}
```

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.PersonA
        "default": []
       }
  ]
}
```

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" #1
      }
    }
  }
}
```

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";  #1

option java_outer_classname = "CompanyProto";

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

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')       #
    subject('college-value','src/main/avro/college.avsc', 'AVRO')
        .addReference("bbejeck.chapter_3.avro.PersonAvro", "perso
    subject('company-value','src/main/avro/company.avsc', 'AVRO')
        .addReference("bbejeck.chapter_3.avro.PersonAvro", "perso
    }
```

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",    #1
  "bbejeck.chapter_3.avro.PlaneEvent",
  "bbejeck.chapter_3.avro.DeliveryEvent"
]
```

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')
subject('plane_event','src/main/avro/plane_event.avsc', 'AVRO')
subject('delivery_event','src/main/avro/delivery_event.avsc', 'AV

subject('inventory-events-value', 'src/main/avro/all_events.avsc'
    .addReference("bbejeck.chapter_3.avro.TruckEvent", "truck_eve
    .addReference("bbejeck.chapter_3.avro.PlaneEvent", "plane_eve
    .addReference("bbejeck.chapter_3.avro.DeliveryEvent", "delive
```

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";  #1
import "login_event.proto";
import "search_event.proto";

option java_outer_classname = "EventsProto";

message Events {

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

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",  #1

  "fields" : [
    {"name": "event", "type"[   #2
      "bbejeck.chapter_3.avro.TruckEvent",  #3
      "bbejeck.chapter_3.avro.PlaneEvent",
      "bbejeck.chapter_3.avro.DeliveryEvent"
    ]}
  ]
}
```

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); #1
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaAvroSerializer.class); #2
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_
  "http://localhost:8081"); #3

//consumer properties these are set separately on the consumer
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  StringDeserializer.class); #4
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaAvroDeserializer.class); #5
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG
  true); #6
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
  "http://localhost:8081"); #7
```

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); #1
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaProtobufSerializer.class); #2
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_
  "http://localhost:8081"); #3

// consumer properties again set separately on the consumer
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  StringDeserializer.class);#4
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaProtobufDeserializer.class);#5
props.put(KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE
  AvengerSimpleProtos.AvengerSimple.class);#6
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
  "http://localhost:8081"); #7
```

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_
  "http://localhost:8081"); #1
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class); #2
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaJsonSchemaSerializer.class); #3

// consumer configuration
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONF
  "http://localhost:8081"); #4
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  StringDeserializer.class); #5
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaJsonSchemaDeserializer.class); #6
props.put(KafkaJsonDeserializerConfig.JSON_VALUE_TYPE,
  SimpleAvengerJson.class); #7
```

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); #1
    } catch (JsonProcessingException e) {
        throw new SerializationException(e);
    }
}
```

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); #1
    } catch (IOException e) {
        throw new SerializationException(e);
    }
}
```

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

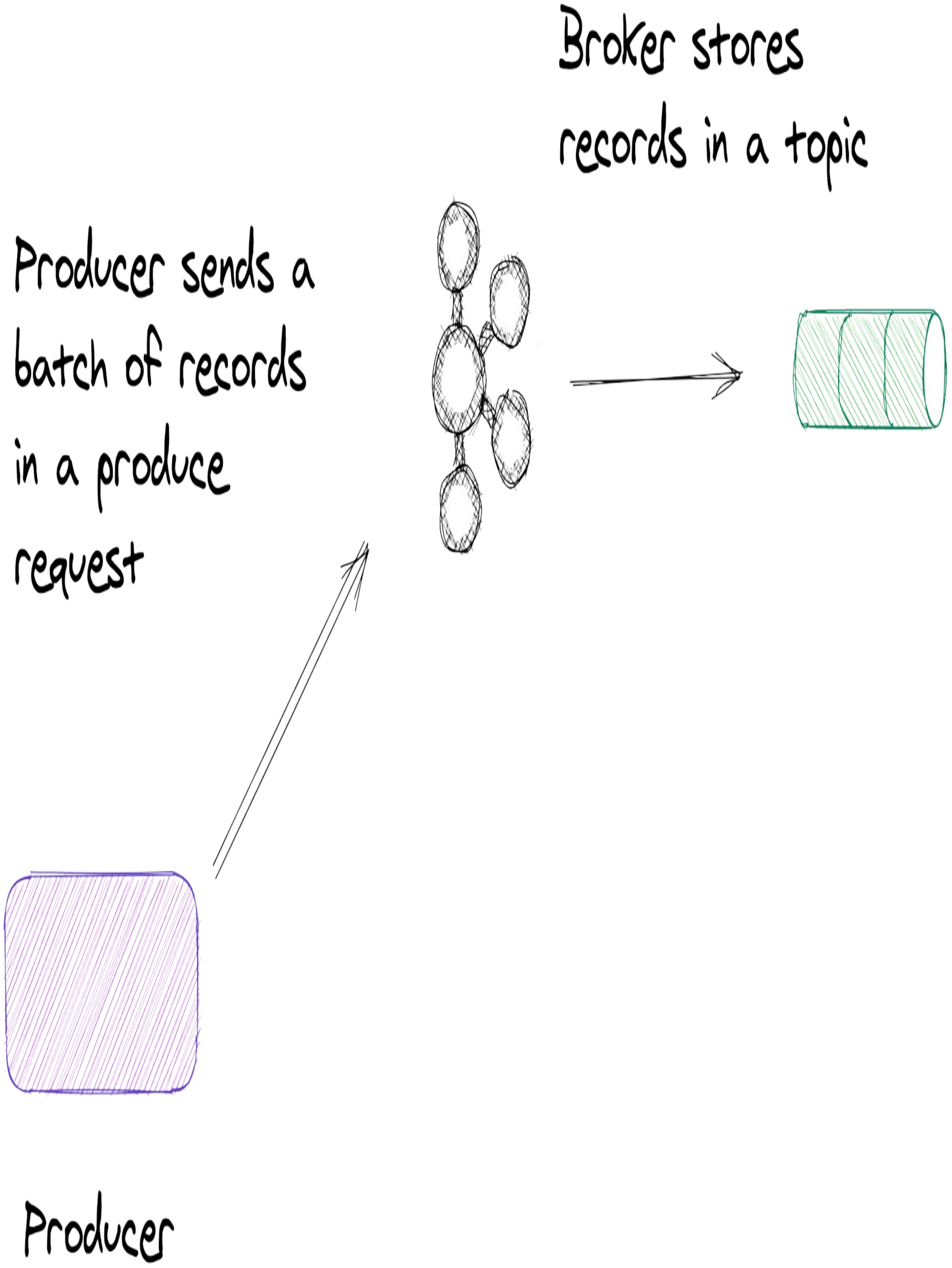
This chapter covers

- 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.

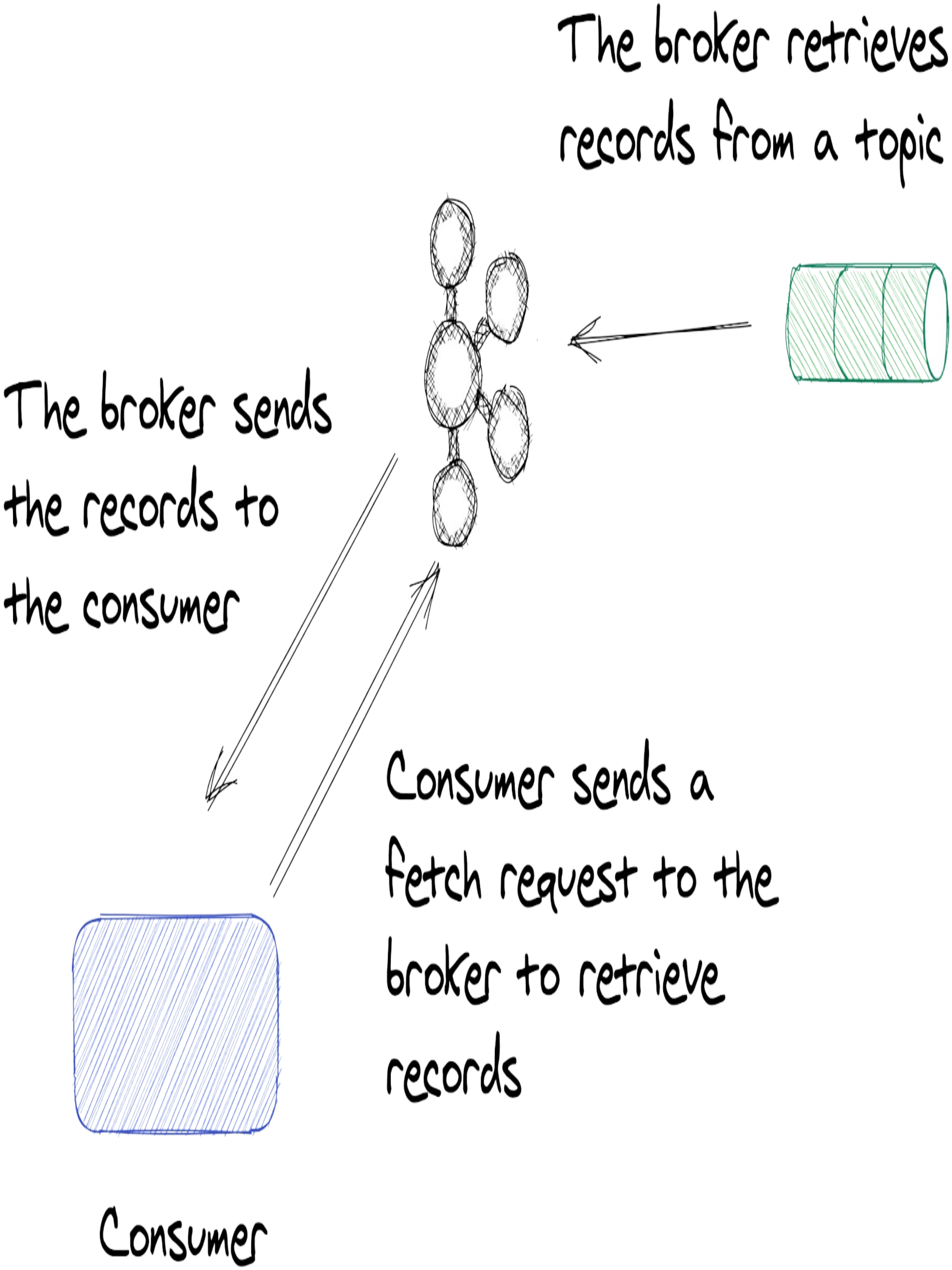
In their simplest form, clients operate like this: producers send records (in a produce request) to a broker and the broker stores them in a topic and consumers send a fetch request and the broker retrieves records from the topic to fulfill that request. When we talk about the Kafka event streaming platform, 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


Broker stores records in a topic
Producer sends a batch of records in a produce request
Producer

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 broker retrieves records from a topic
The broker sends the records to the consumer
Consumer sends a fetch request to the broker to retrieve records
Consumer

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.

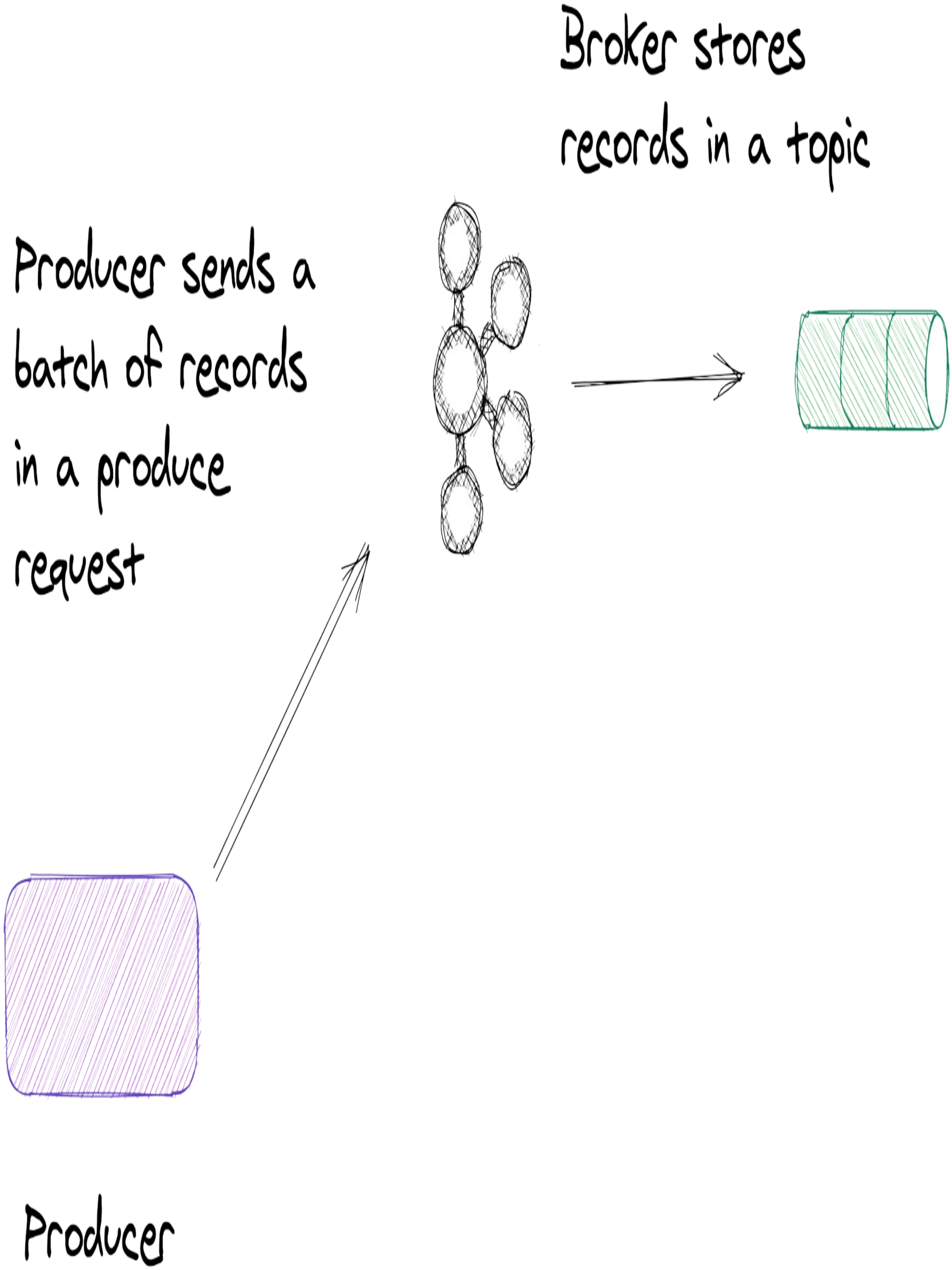
Say you work on the data ingest team for a medium sized wholesale company. You get transaction data delivered via a point of sale service and several different departments within the company want access to the data for things such as reporting, inventory control, detecting trends etc.

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


Broker stores records in a topic
Producer sends a batch of records in a produce request
Producer

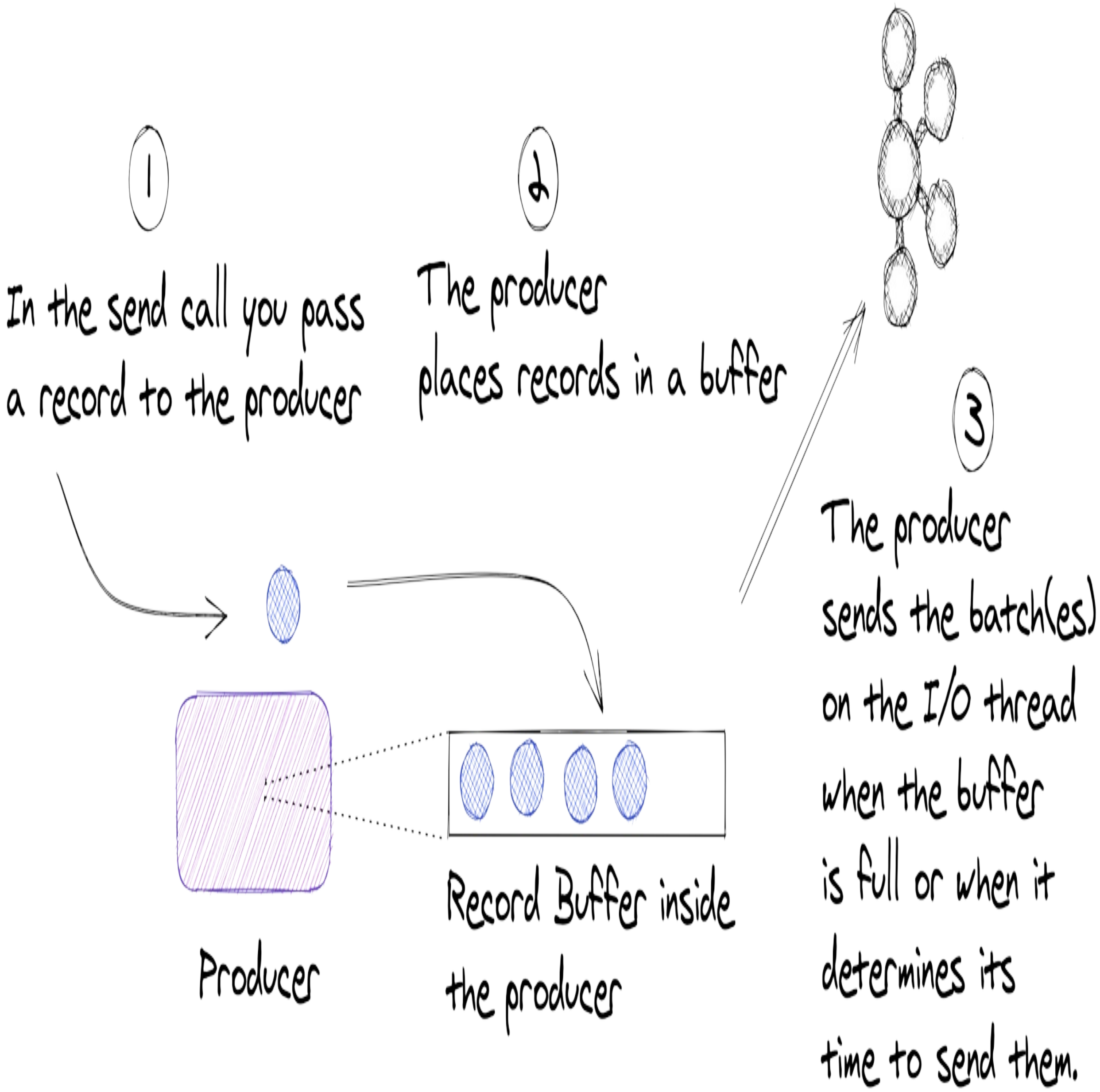
To make sure everyone is on the same page with the structure of the data, you’ve modeled the records with a schema and published it to Schema Registry. All that’s left is for you to do write the `KafkaProducer` code to take the sales records and send them into Kafka. Here’s what your code looks like

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)) { #1
   while(keepProducing) {
    Collection<ProductTransaction> purchases = salesDataSource.fe
     purchases.forEach(purchase -> {
        ProducerRecord<String, ProductTransaction> producerRecord
             new ProducerRecord<>(topicName, purchase.getCustomer
              purchase); #3
        producer.send(producerRecord,
          (RecordMetadata metadata, Exception exception) -> {
              if (exception != null) {   #5
                  LOG.error("Error producing records ", exception
            } else {
              LOG.info("Produced record at offset {} with timesta
                          metadata.offset(), metadata.timestamp()
              }
          });
      });
```

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


1
In the send call you pass a record to the producer
2
The producer places records in a buffer
3
The producer sends the batch(es) on the I/O thread when the buffer is full or when it determines its time to send them.
Producer
Record Buffer inside the producer

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 the section called “Idempotent producer” 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).

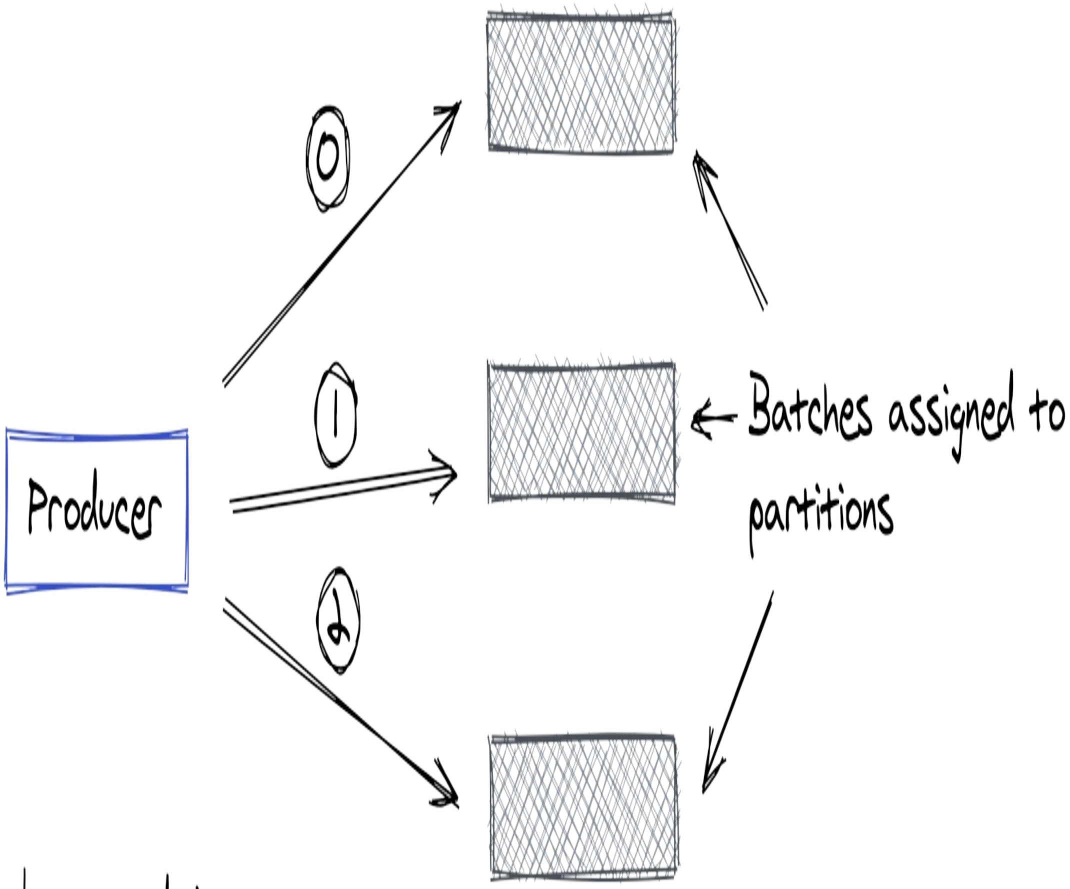
4.1.3 Partition assignment

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

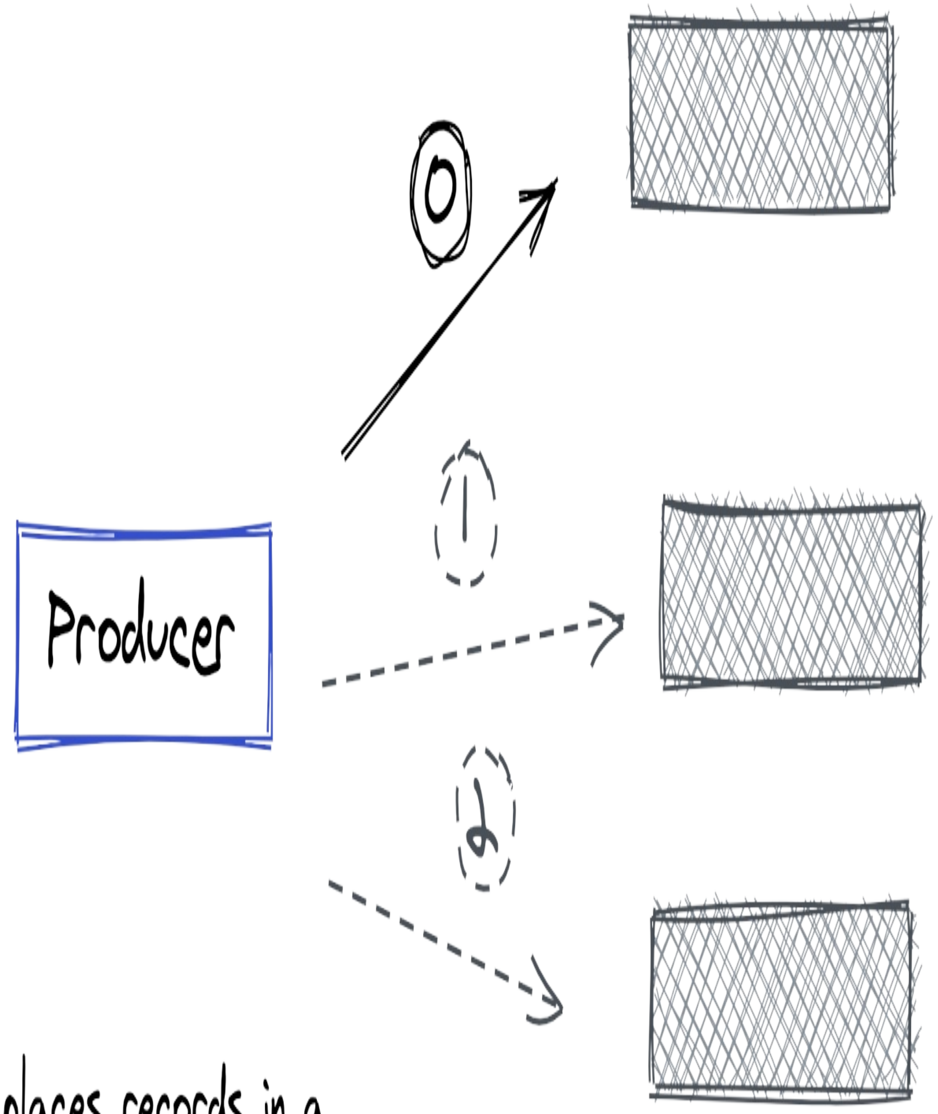


The producer places records in a batch for partitions in the following order 0, 1, and then 2. Then the order starts over again at 0.

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



Now the producer places records in a batch for partition 0. Once the producer sends the batch, all records will go in a batch for partition 1 until it's sent. Then the producer creates a batch for partition 2. After that the cycle repeats.

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 the section called “Producing records with the KafkaProducer” 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); #1
    String strKey = (String) key;
    int partition;

    if (strKey.equals("CUSTOM")) {
        partition = 0;  #2
    } else {
        byte[] bytes = strKey.getBytes(StandardCharsets.UTF_8);
        partition = Utils.toPositive(Utils.murmur2(bytes)) %
                                       (numPartitions - 1) + 1; #
    }
    return partition;
  }
}
```

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.

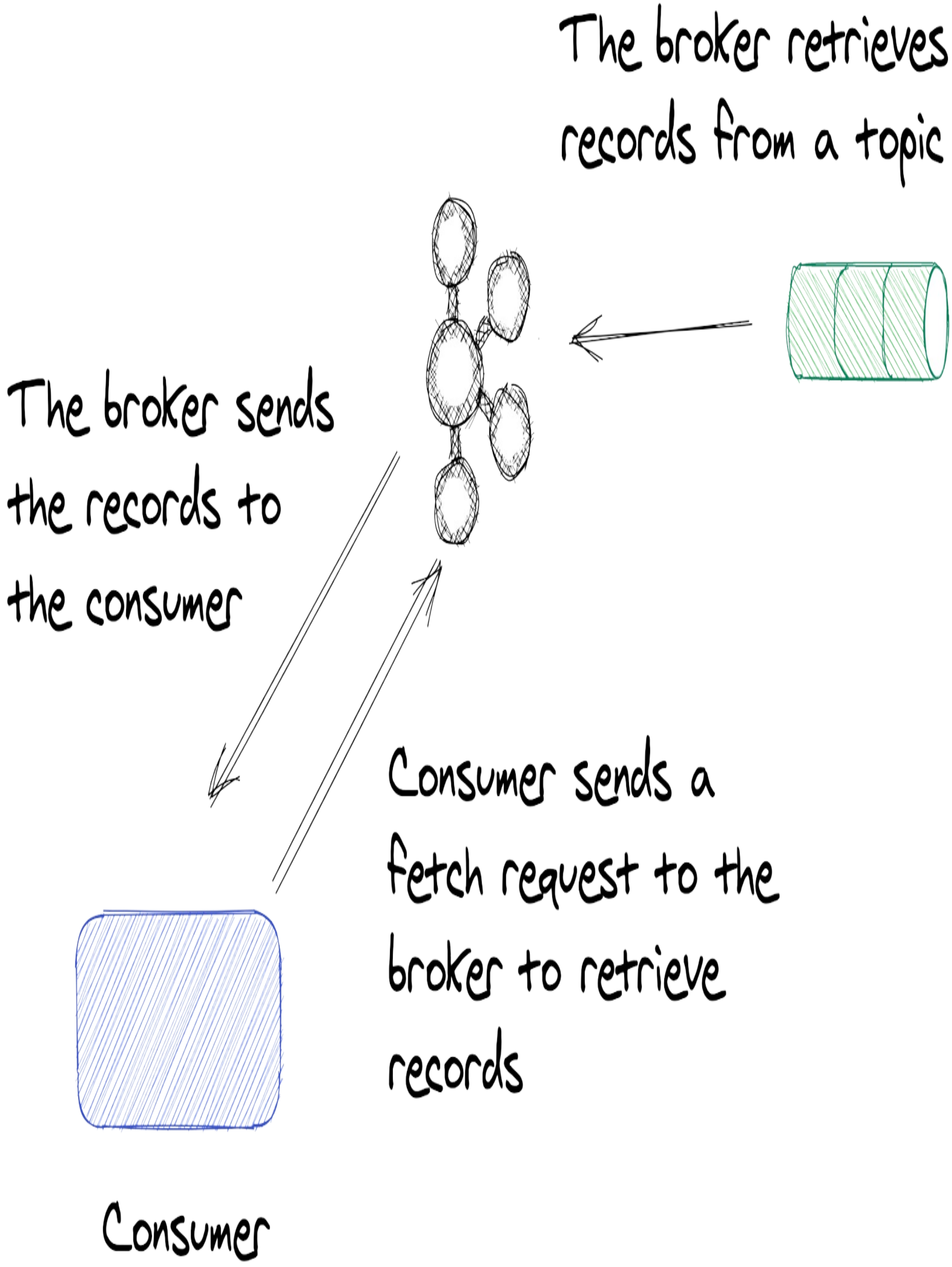
Another consideration is that you can embed the timestamp of the event in payload of the record value when you are creating it.

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 broker retrieves records from a topic

The broker sends the records to the consumer

Consumer sends a fetch request to the broker to retrieve records

Consumer

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 Kafka
    consumerConfigs)) { #1
    consumer.subscribe(topicNames);   #2
    while (keepConsuming) {
        ConsumerRecords<String, ProductTransaction> consumerRecor
          consumer.poll(Duration.ofSeconds(5)); #3
        consumerRecords.forEach(record -> {     #4
            ProductTransaction pt = record.value();
            LOG.info("Sale for {} with product {} for a total sal
                    record.key(),
                    pt.getProductName(),
                    pt.getQuantity() * pt.getPrice());
        });
    }
}
```

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.

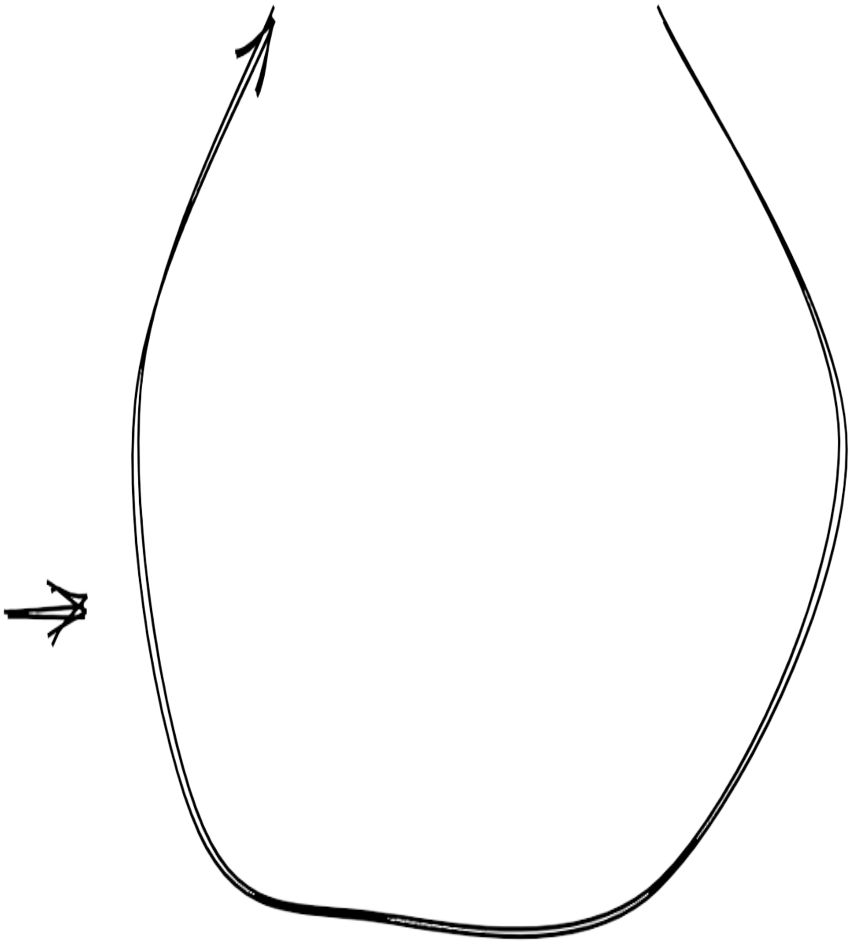
It’s more important to grasp how the `KafkaConsumer` works and the implications of the different configurations. By having this understanding you’ll be in a better position to know how to best write the code for performing the desired operations on the consumed records. So just as we did in the producer 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

The consumer calls poll
and retrieves a batch of records



The Process loop

The consumer has 5 minutes
(the default time) to fully process
records and return to make
another poll call.

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.

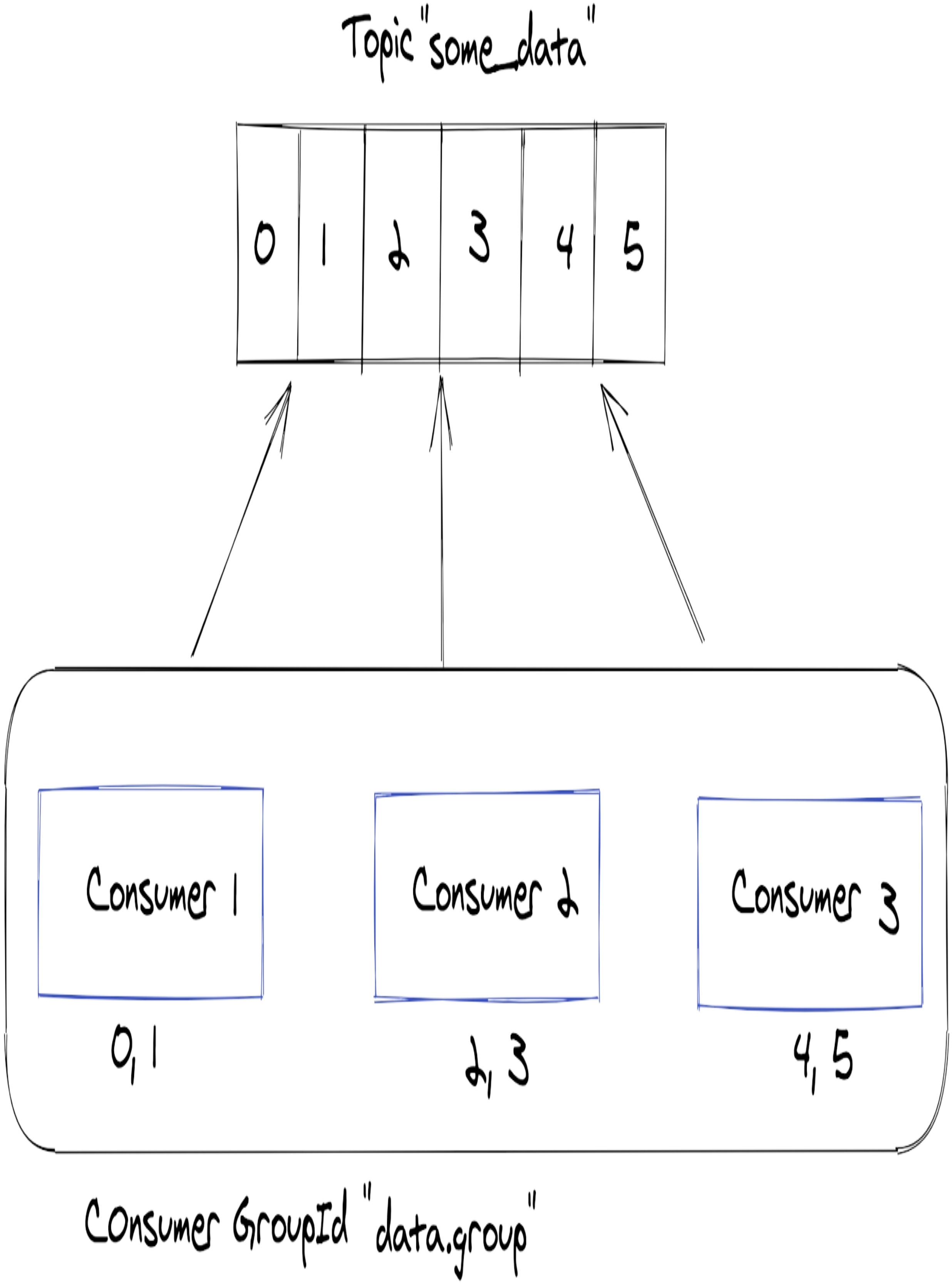
Going through this illustration, iterating over all the records and execution for each record must complete before the `max.poll.interval.ms` time elapses. By default this value is five minutes, so if your processing of returned records takes longer, then that individual consumer is considered dead and 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:

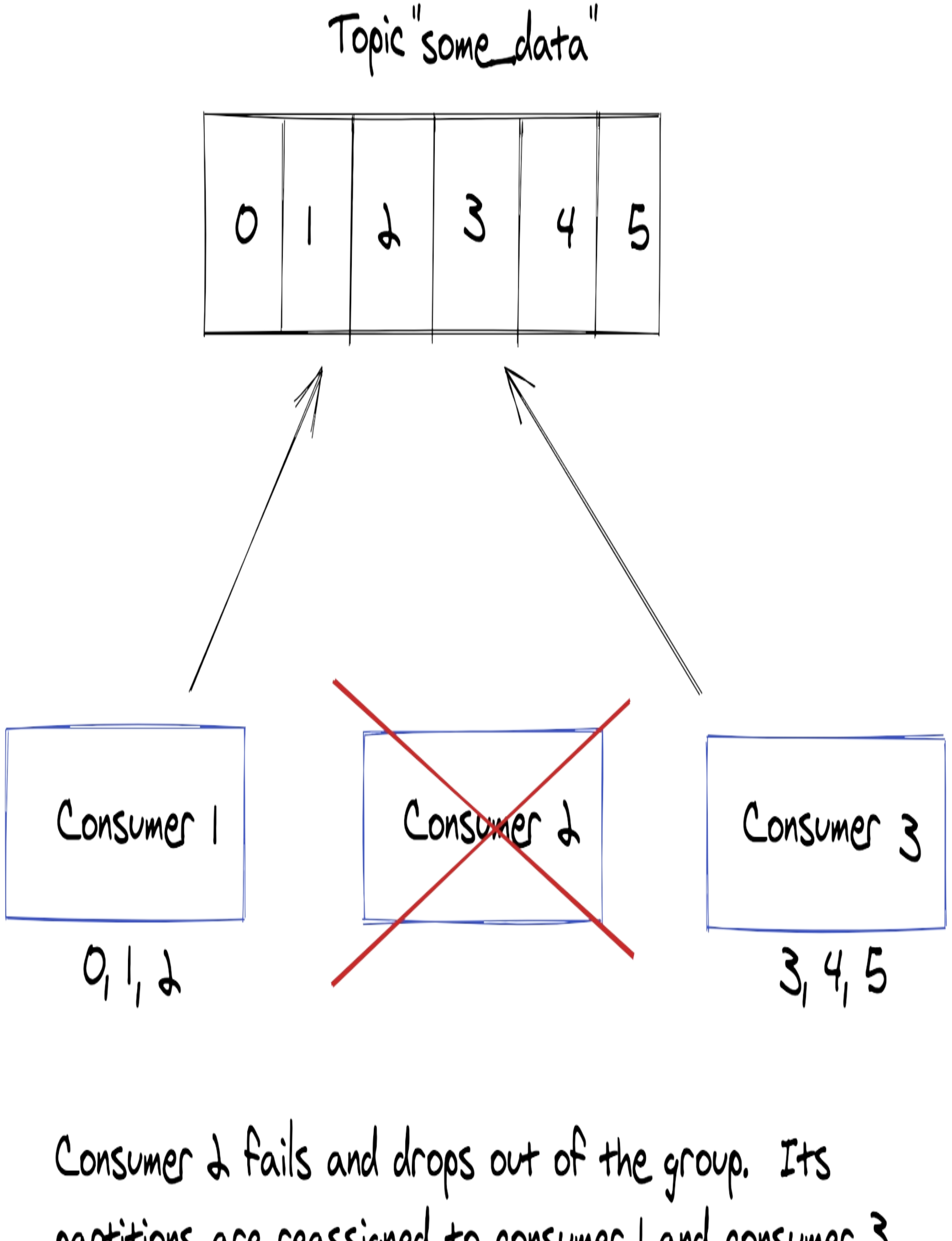
Figure 4.9. Consumer groups allow for assigning topic-partitions across multiple consumers


Topic "some_data"
0
1
2
3
4
5
Consumer 1
Consumer 2
Consumer 3
0, 1
2, 3
4, 5
Consumer GroupId "data.group"

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



Consumer 2 fails and drops out of the group. Its partitions are reassigned to consumer 1 and consumer 3

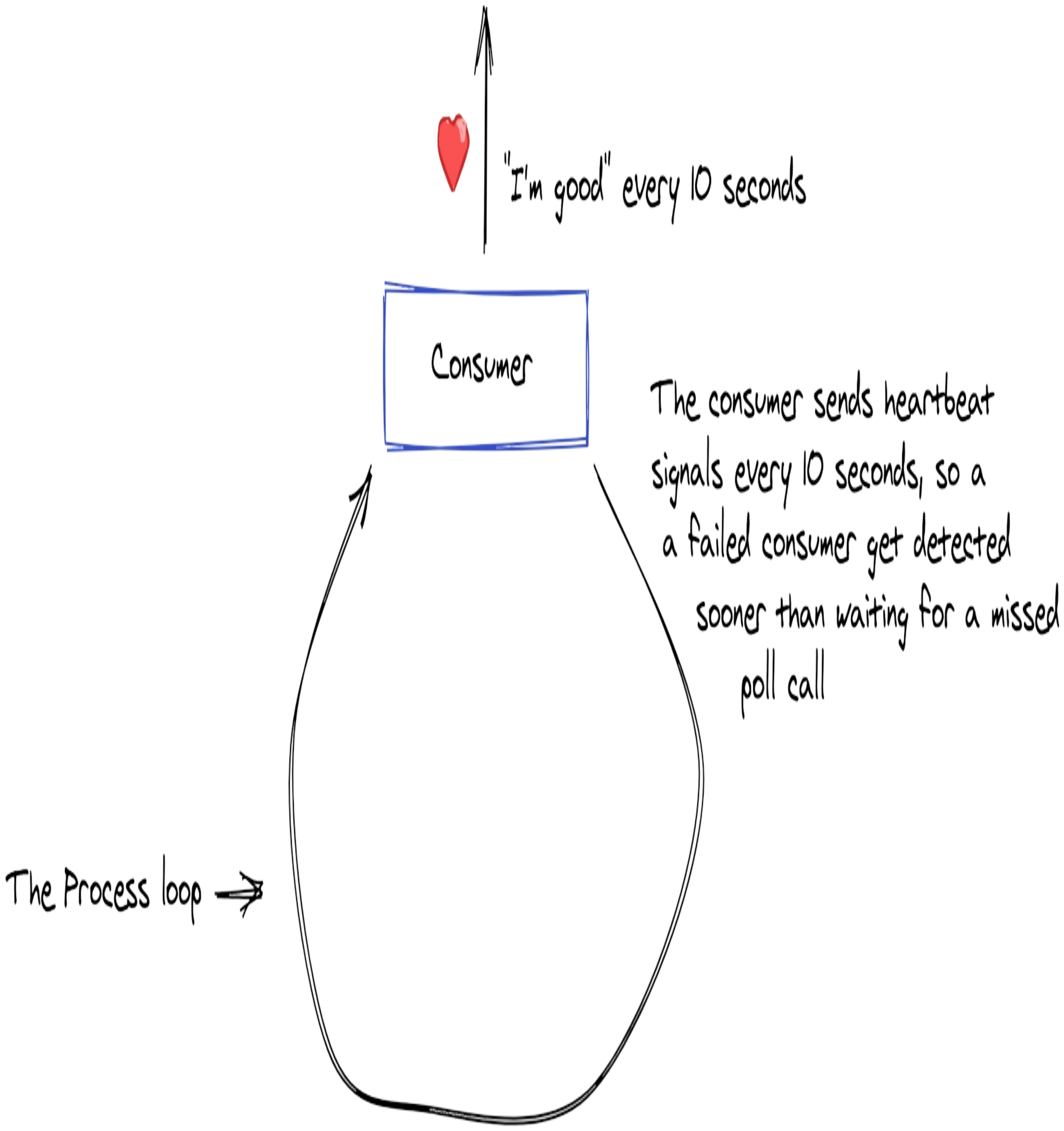
What we see here is that consumer-two fails and can longer function. So rebalancing takes the topic-partitions owned by consumer-two and reassigns one topic-partition each to other active consumers in the group. Should consumer-two become active again (or another consumer join the group), then another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

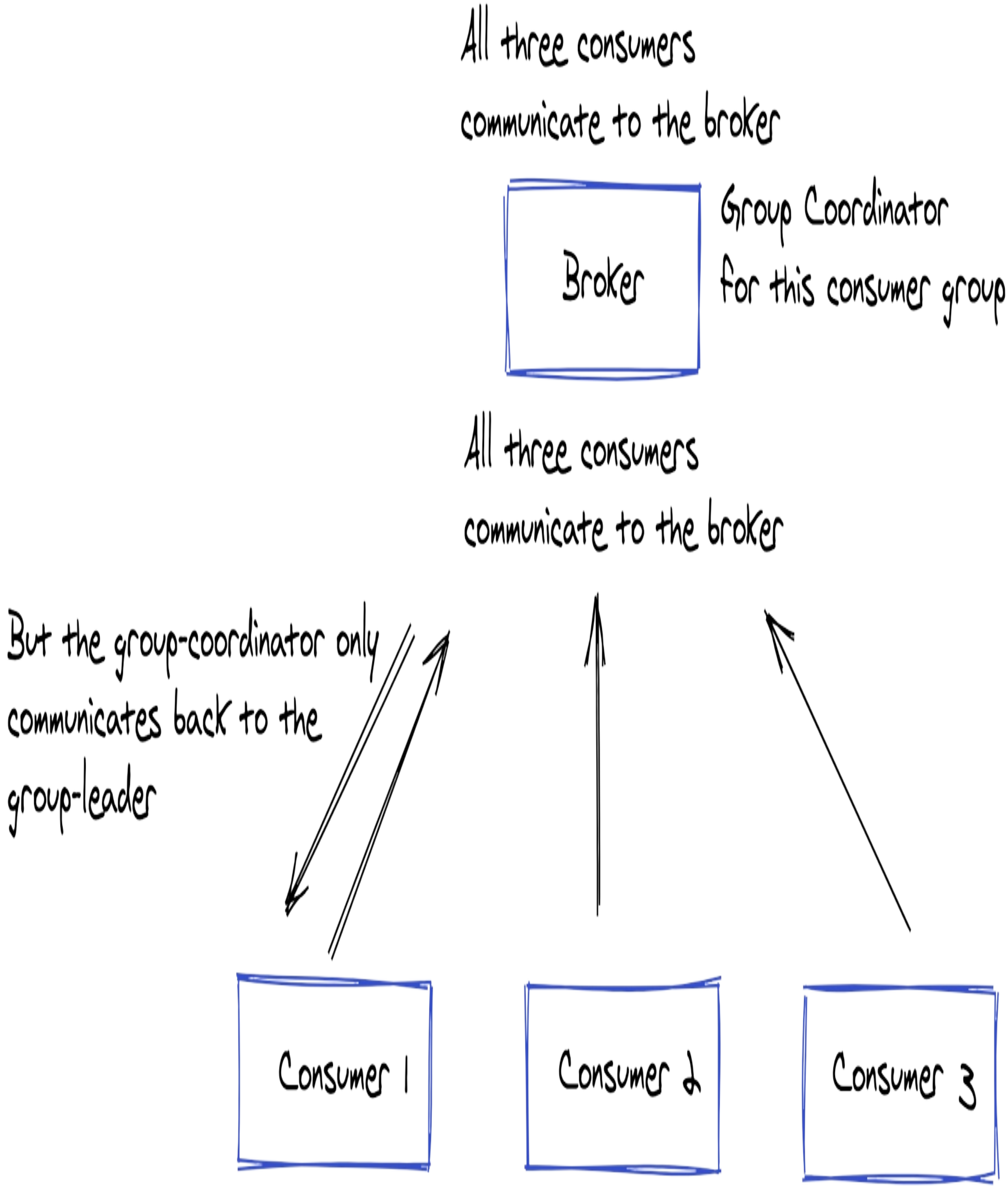
Figure 4.11. In addition to needing to call poll within the timeout, a consumer must send a heartbeat every ten seconds


"I'm good" every 10 seconds
Consumer
The consumer sends heartbeat signals every 10 seconds, so a a failed consumer get detected sooner than waiting for a missed poll call
The Process loop

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.

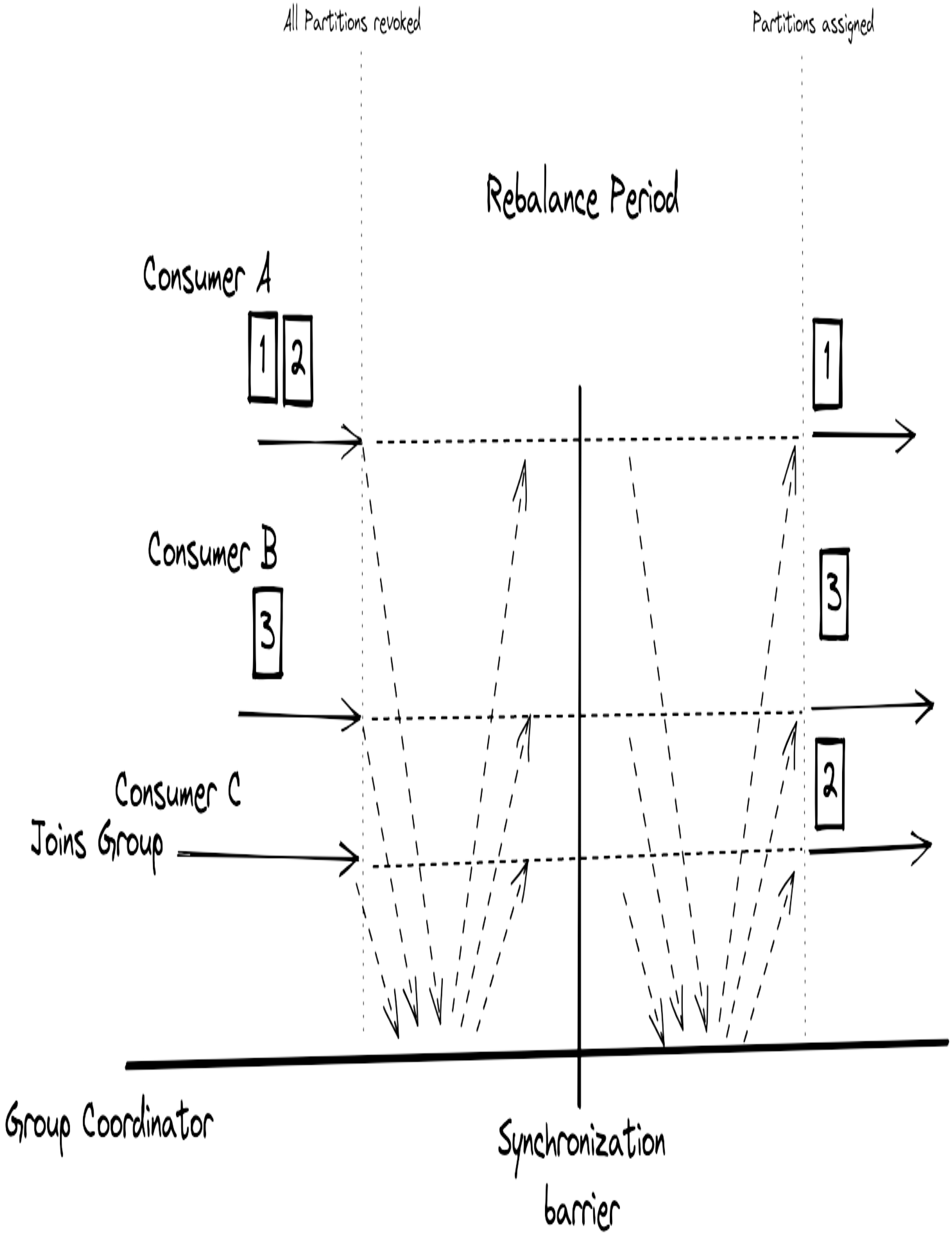
A rebalance is the process of having all members of the group rejoin so that group resources (topic-partitions) can be evenly (as possible) distributed to the other members. When a new member joins, then some topic partitions are removed from some or all members of the existing group and are assigned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


All Partitions revoked
Partitions assigned
Rebalance Period
Consumer A
1
2
1
Consumer B
3
3
Consumer C
Joins Group
2
Group Coordinator
Synchronization
barrier

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.

ℹ **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.

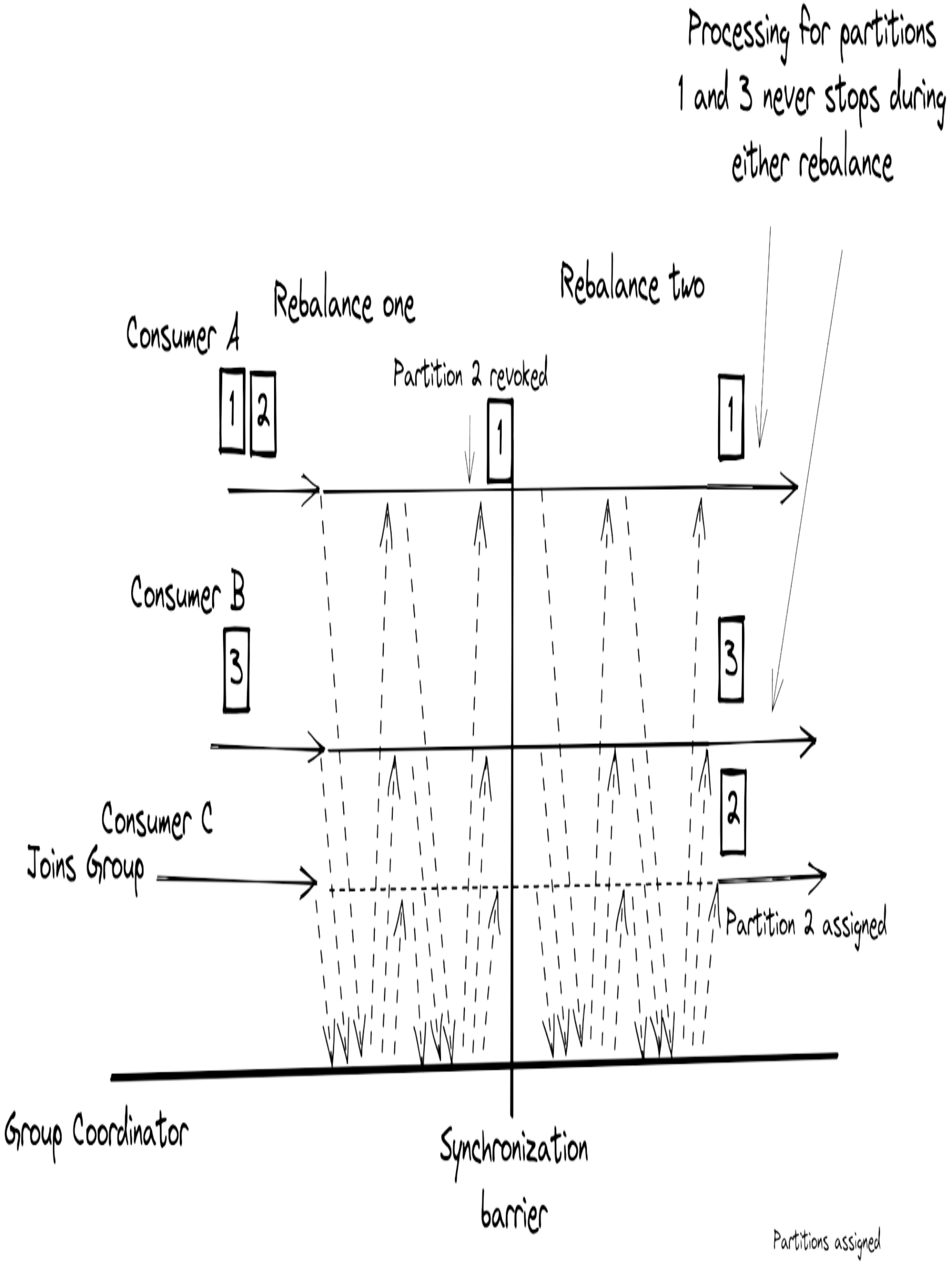
ℹ **Note**

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


Processing for partitions
1 and 3 never stops during
either rebalance
Rebalance one
Rebalance two
Consumer A
1
2
Partition 2 revoked
1
1
Consumer B
3
3
2
Consumer C
Joins Group
Partition 2 assigned
Group Coordinator
Synchronization
barrier
Partitions assigned

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.

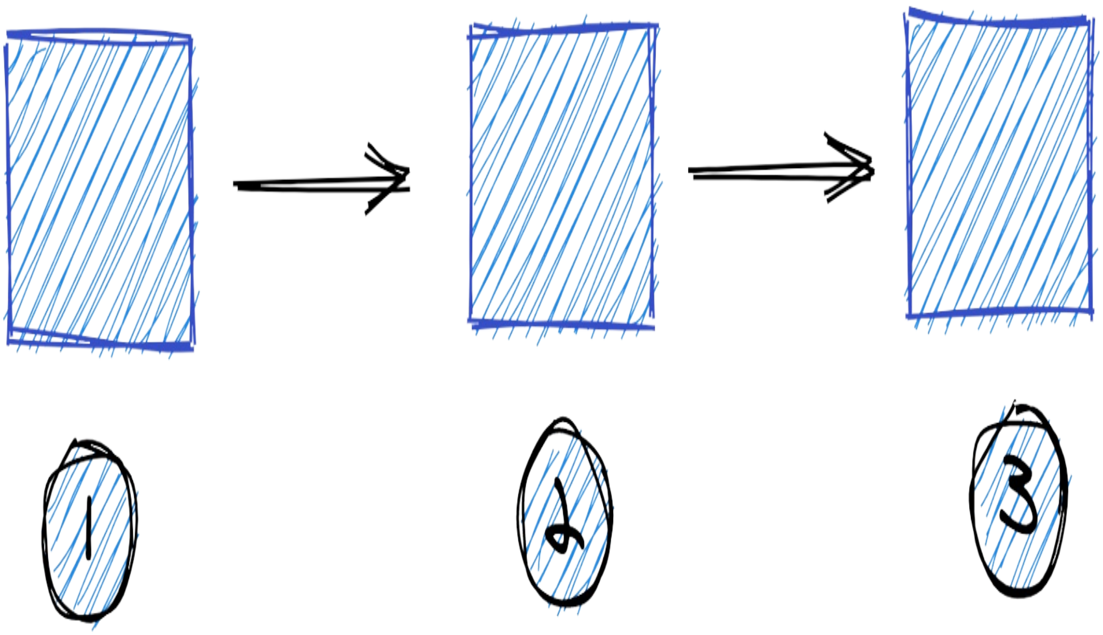
4.2.3 Static membership

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

Rolling upgrade each application is shut down, upgraded then restarted

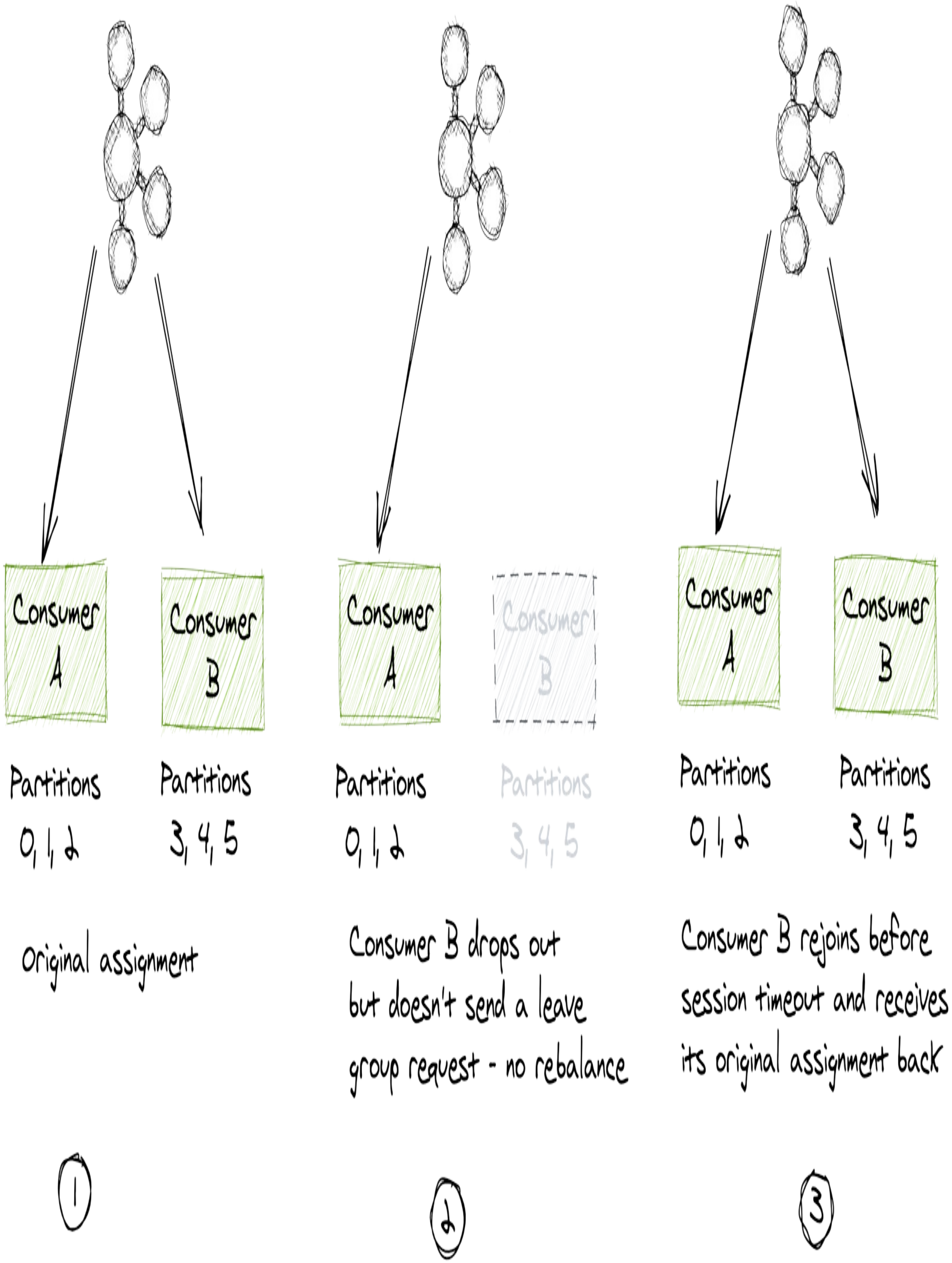


Each shutdown sends a "leave group" request then restarting issues a "join group" request So each restart results in 2 rebalances for all instances in the group

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



Original assignment

Consumer B drops out but doesn't send a leave group request - no rebalance

Consumer B rejoins before session timeout and receives its original assignment back

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.

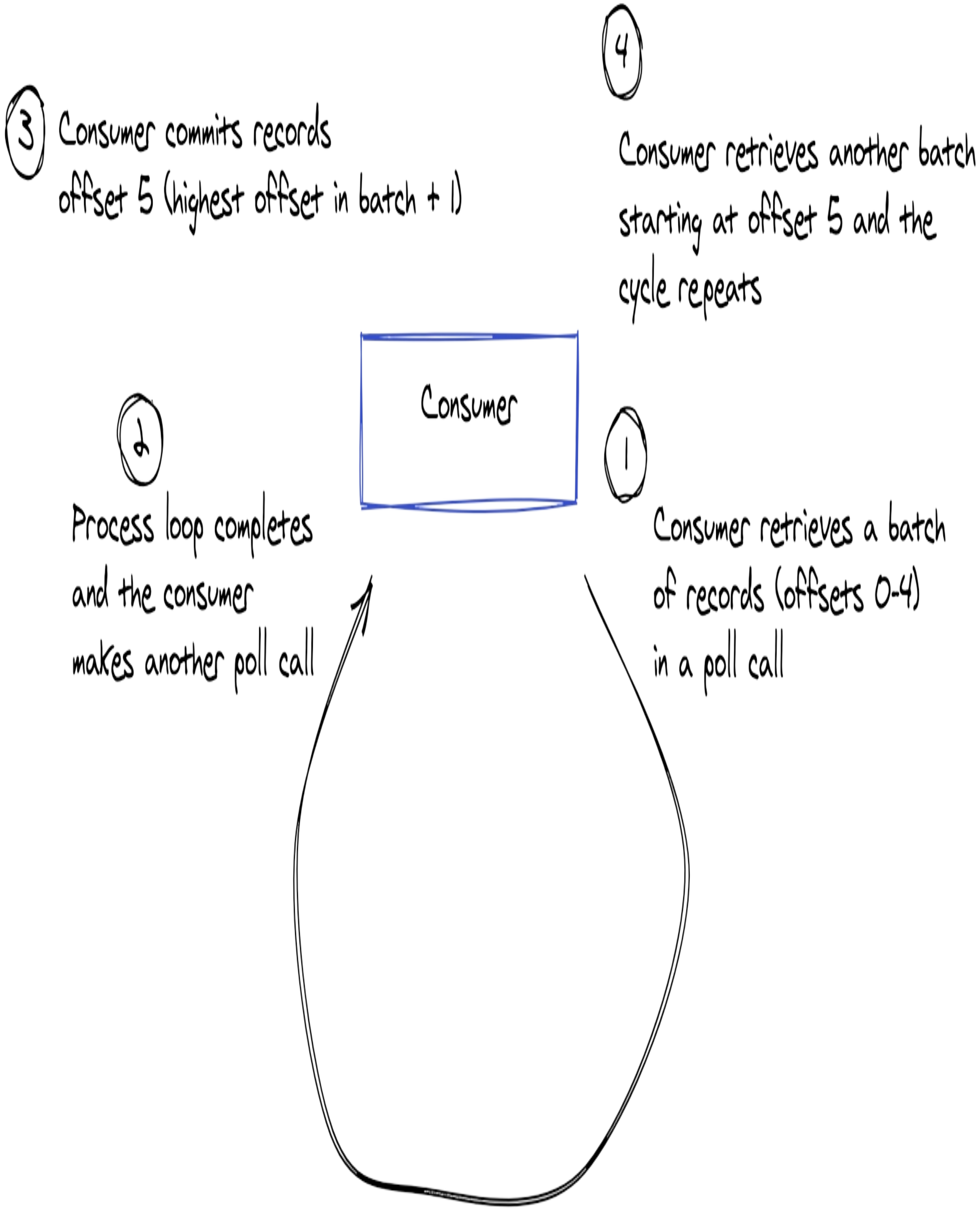
4.2.4 Committing offsets

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

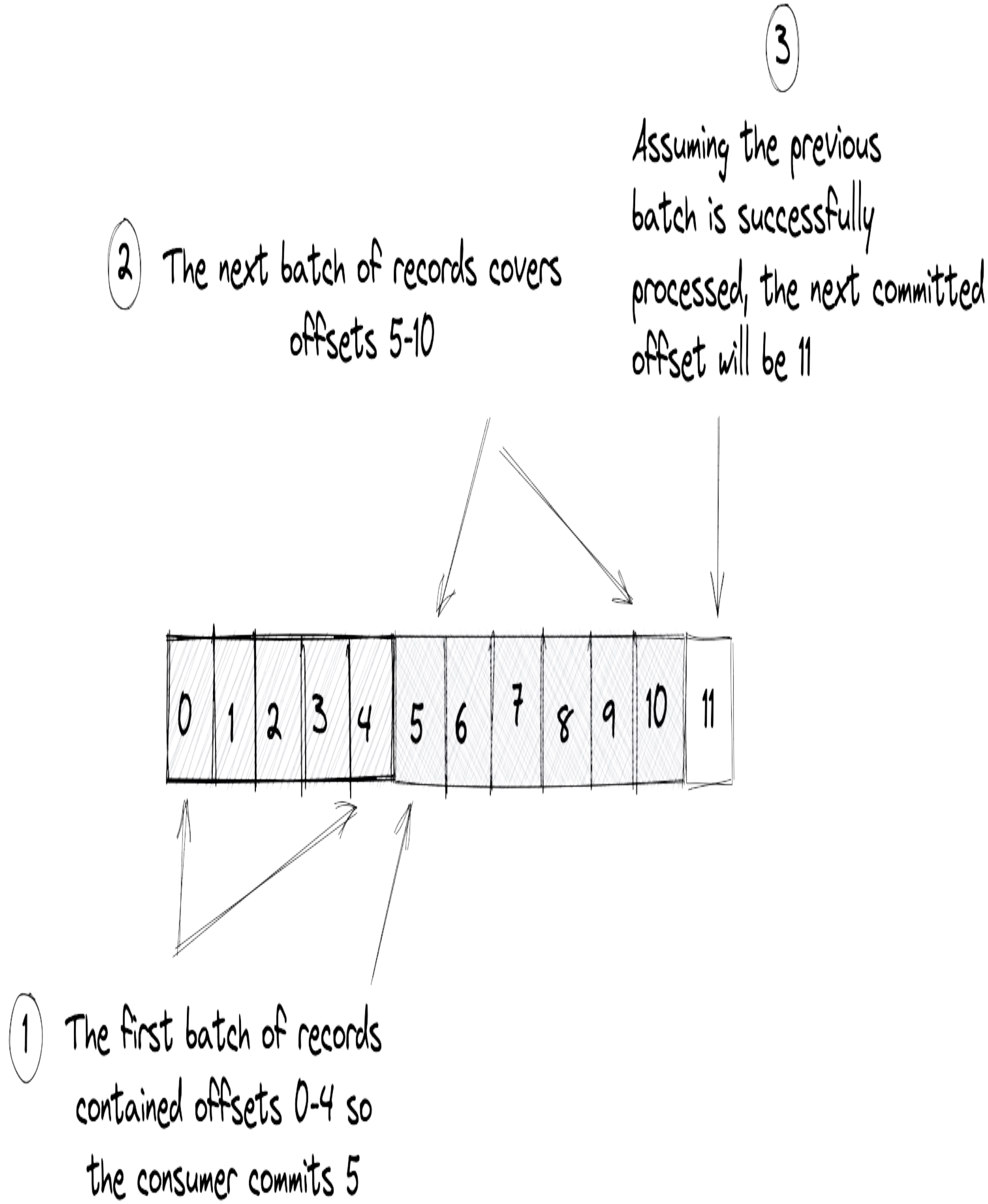

3 Consumer commits records offset 5 (highest offset in batch + 1)
4 Consumer retrieves another batch starting at offset 5 and the cycle repeats
Consumer
2 Process loop completes and the consumer makes another poll call
1 Consumer retrieves a batch of records (offsets 0-4) in a poll call

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



Assuming the previous batch is successfully processed, the next committed offset will be 11

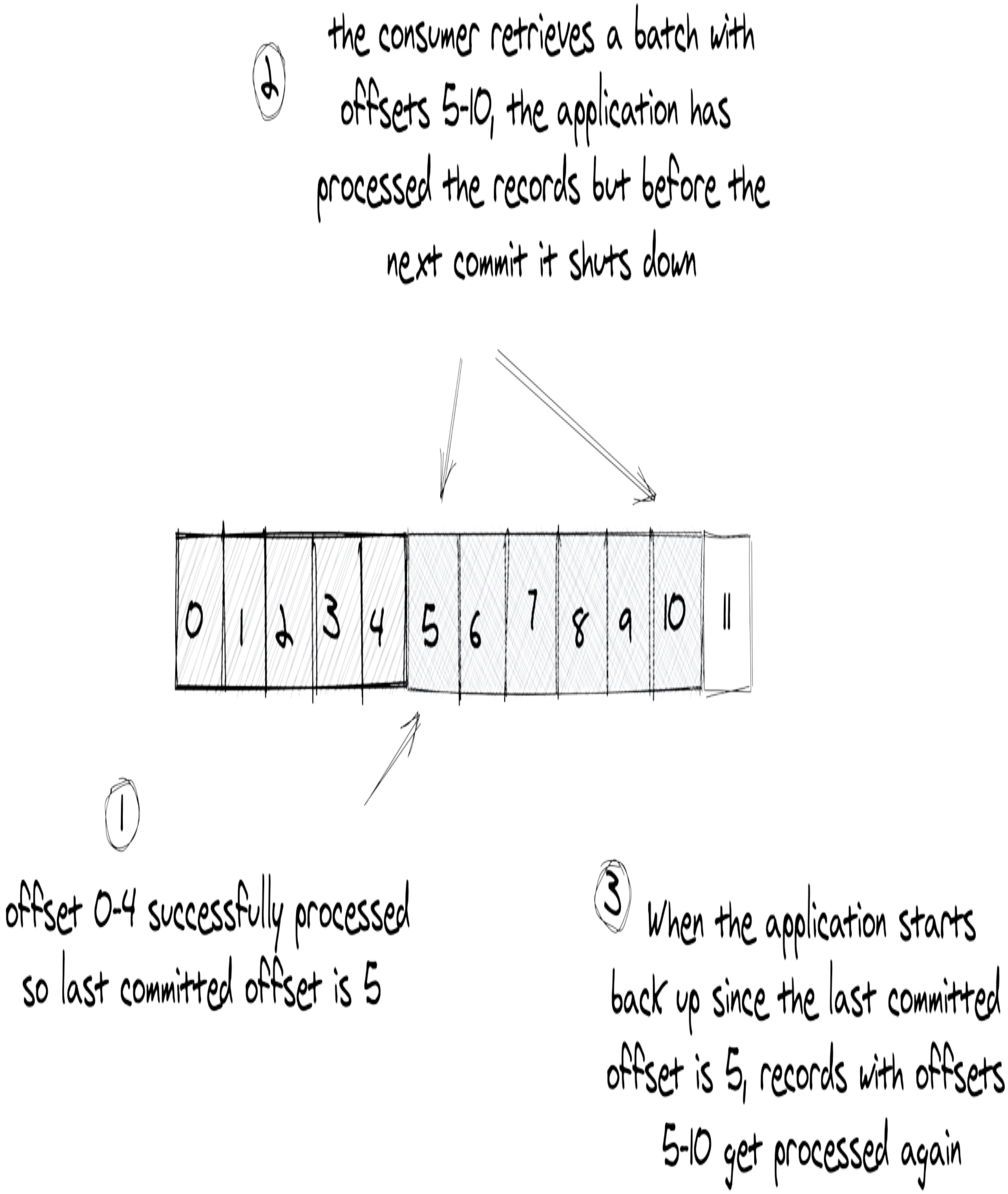
The next batch of records covers offsets 5-10

The first batch of records contained offsets 0-4 so the consumer commits 5

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


2 the consumer retrieves a batch with offsets 5-10, the application has processed the records but before the next commit it shuts down
0
1
2
3
4
5
6
7
8
9
10
11
1 offset 0-4 successfully processed so last committed offset is 5
3 When the application starts back up since the last committed offset is 5, records with offsets 5-10 get processed again

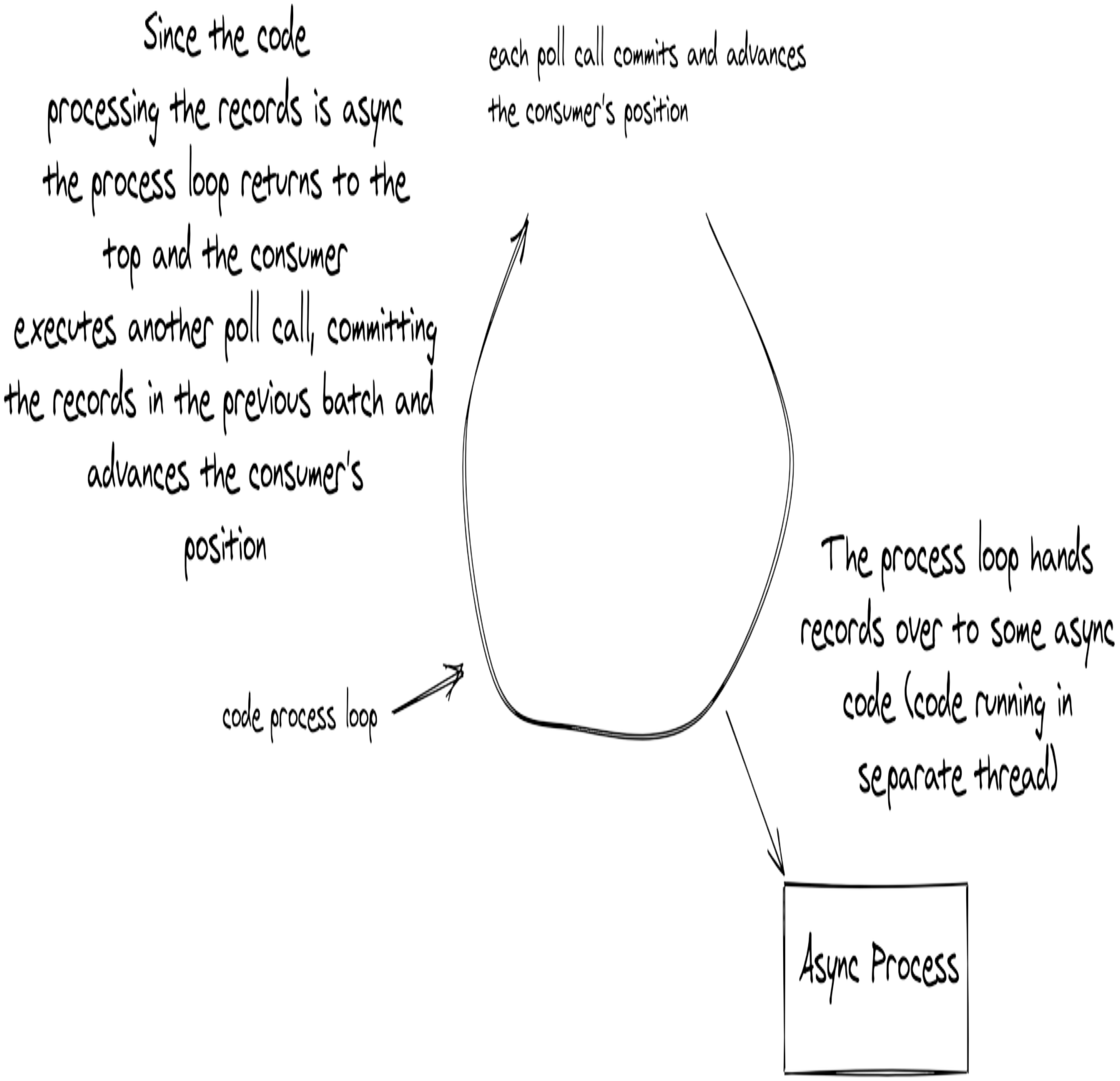
If you processed some of the records with offsets larger than the latest one committed, but your consumer failed to commit for whatever reason, this means when you resume processing, you start with records from the committed offset, so you’ll reprocess some of the records. This potential for reprocessing 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


Since the code processing the records is async the process loop returns to the top and the consumer executes another poll call, committing the records in the previous batch and advances the consumer's position
each poll call commits and advances the consumer's position
The process loop hands records over to some async code (code running in separate thread)
code process loop
Async Process

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 = con
  Duration.ofSeconds(5));
if (!consumerRecords.isEmpty()) {
    recordProcessor.processRecords(consumerRecords);     #1
    Map<TopicPartition, OffsetAndMetadata> offsetsAndMetadata =
      recordProcessor.getOffsets(); #2
    if (offsetsAndMetadata != null) {
        consumer.commitSync(offsetsAndMetadata); #3
    }
```

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<>(
consumerRecords.partitions().forEach(topicPartition -> {  #2
       List<ConsumerRecord<String,ProductTransaction>> topicPar
  consumerRecords.records(topicPartition); #3
       topicPartitionRecords.forEach(this::doProcessRecord); #4
       long lastOffset = topicPartitionRecords.get(
  topicPartitionRecords.size() - 1).offset(); #5
       offsets.put(topicPartition, new OffsetAndMetadata(lastOf
      });
 ....
 offsetQueue.offer(offsets); #7
```

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, "someh
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, ...
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, .

//Configs related to idempotence
producerProps.put(ProducerConfig.ACKS_CONFIG, "all"); #1
producerProps.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true)
producerProps.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALU
producerProps.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNE
```

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.

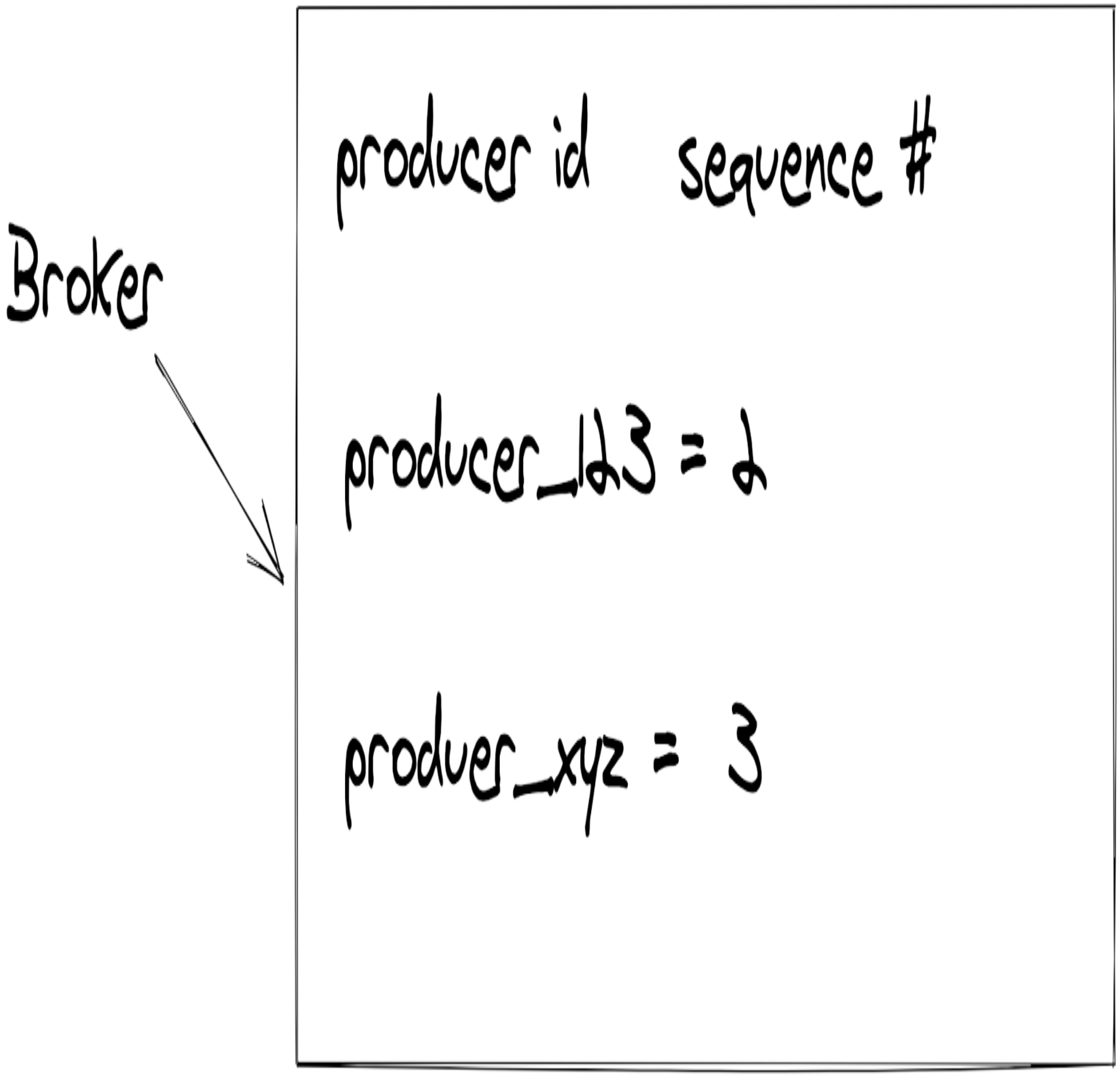
Note

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

Tracking producer id to next expected sequence number



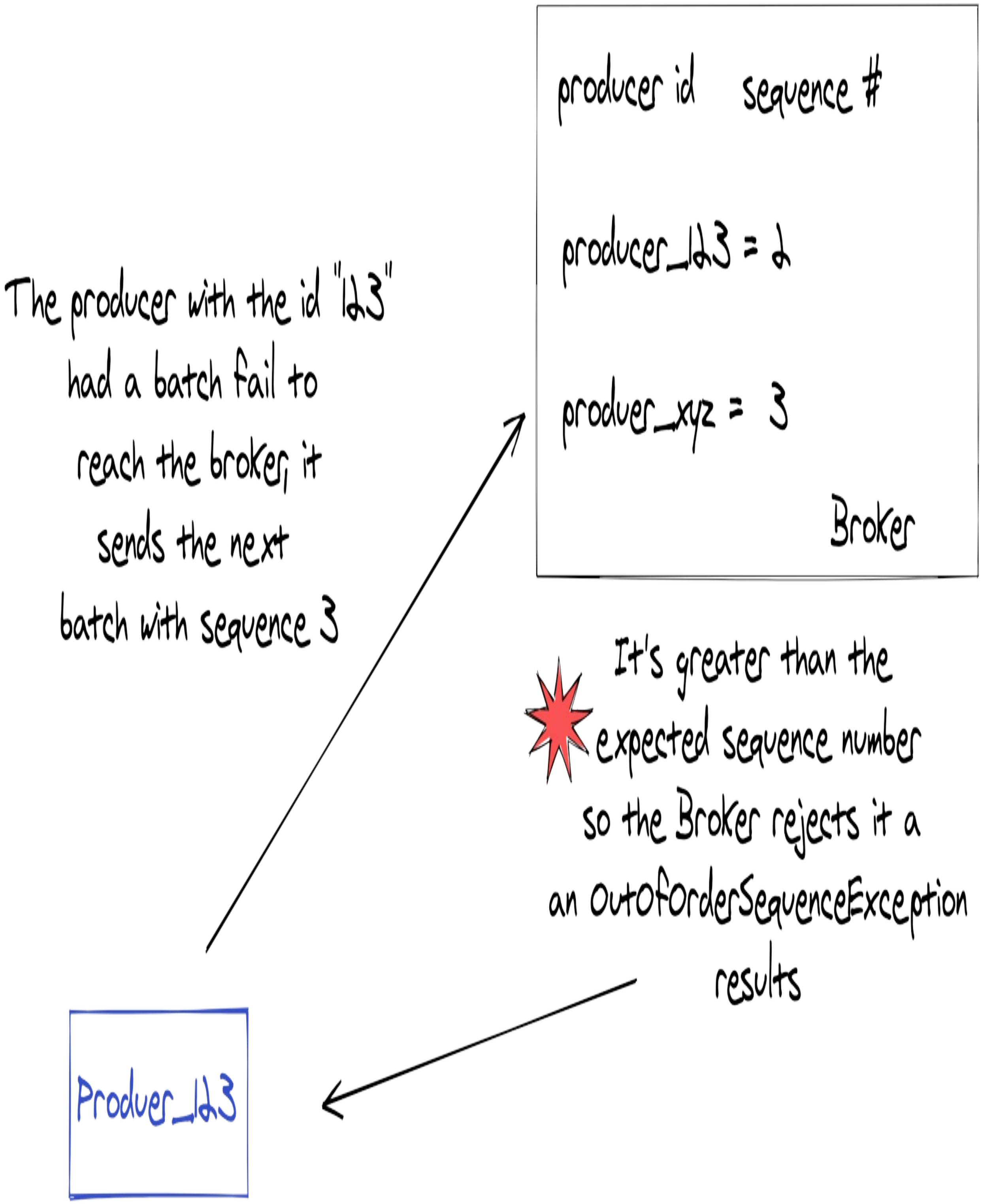
producer id   sequence #

producer_123 = 2

produer_xyz = 3

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


producer id
sequence #
producer_123 = 2
produer_xyz = 3
Broker
The producer with the id "123" had a batch fail to reach the broker, it sends the next batch with sequence 3
It's greater than the expected sequence number so the Broker rejects it a an OutofOrderSequenceException results
Produer_123

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#KIP98ExactlyOnc

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

Producer<String, String> producer = new KafkaProducer<>(producerP
producer.initTransactions(); #1

try {
    producer.beginTransaction(); #2
    producer.send(topic, "key", "value"); #3
    producer.commitTransaction();   #4
} catch (ProducerFencedException | OutOfOrderSequenceException
  | AuthorizationException e) {  #5
      producer.close();
} catch (KafkaException e) {     #6
     producer.abortTransaction();
     // safe to retry at this point #7
}
```

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.

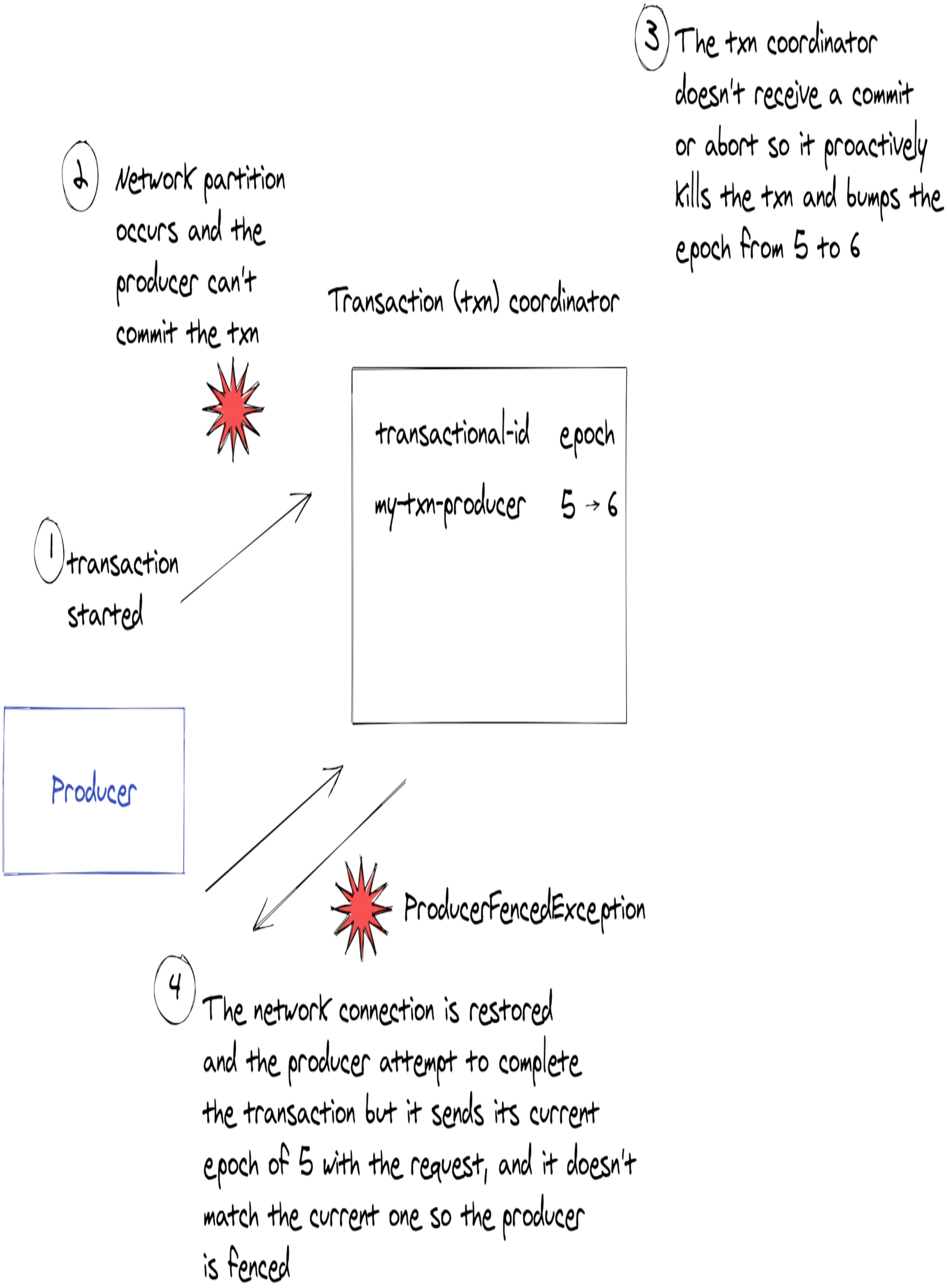
Before you start sending records you call `beginTransaction()`, which starts the transaction for the producer. Once the transaction starts,The transaction coordinator will only wait for a period of time defined by the `transaction.timeout.ms` (one minute by default) and it without an update (a commit 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



transactional-id	epoch
my-txn-producer	5 → 6

1 transaction started

2 Network partition occurs and the producer can't commit the txn

3 The txn coordinator doesn't receive a commit or abort so it proactively kills the txn and bumps the epoch from 5 to 6

4 The network connection is restored and the producer attempt to complete the transaction but it sends its current epoch of 5 with the request, and it doesn't match the current one so the producer is fenced

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`.

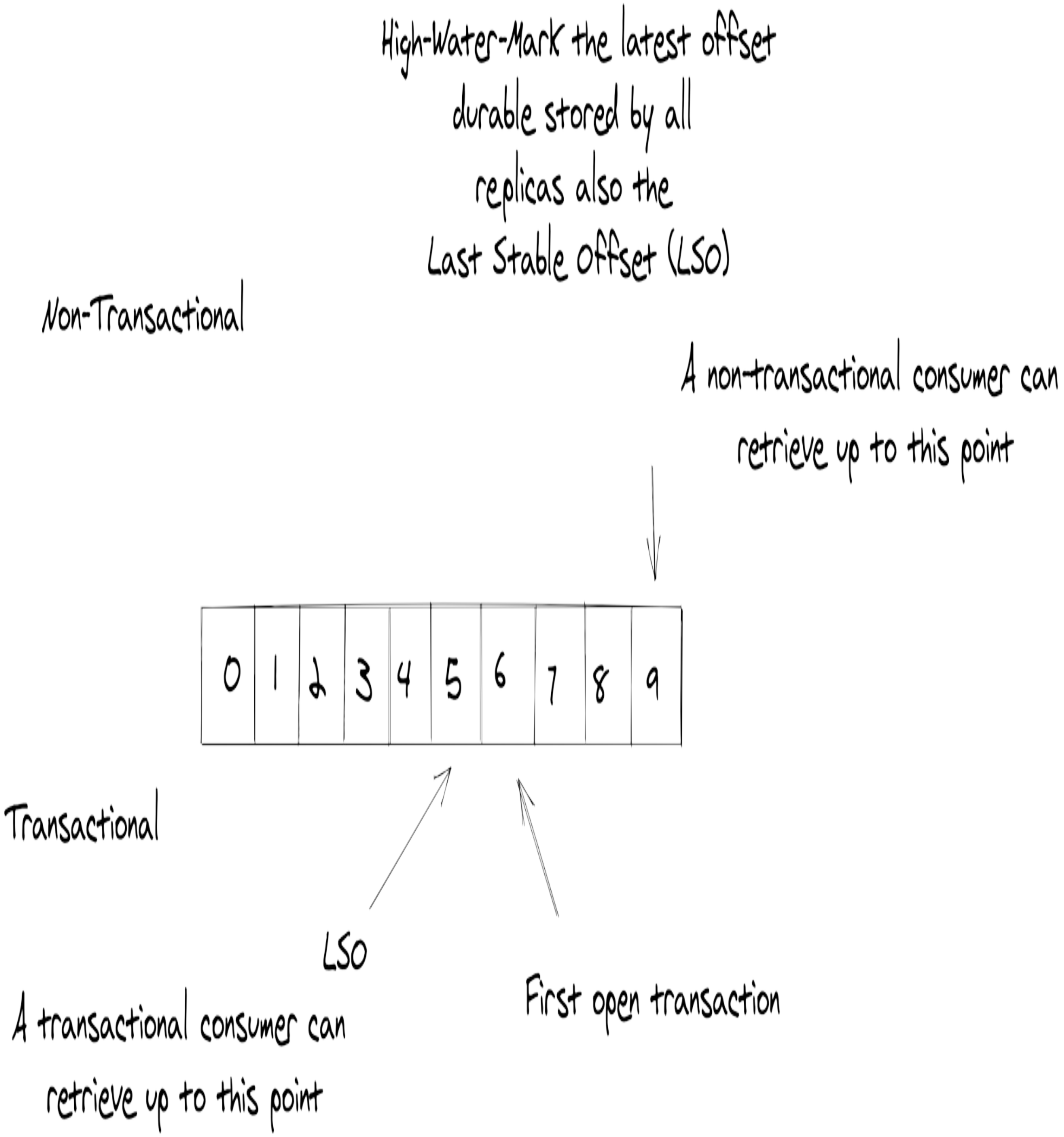
Listing 4.8. KafkaConsumer configuration for transactions

```
// Several details omitted for clarity

HashMap<String, Object> consumerProps = new HashMap<>();
consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "local
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, "the-group-id")

consumerProps.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_co

consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, S
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
```

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


High-Water-Mark the latest offset durable stored by all replicas also the Last Stable Offset (LSO)
Non-Transactional
A non-transactional consumer can retrieve up to this point
0
1
2
3
4
5
6
7
8
9
Transactional
LSO
First open transaction
A transactional consumer can retrieve up to this point

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<>();
producer.beginTransaction();                #2
consumerRecords.partitions().forEach(topicPartition -> {
    consumerRecords.records(topicPartition).forEach(record -> {
        lastOffset.set(record.offset());
        StockTransaction stockTransaction = record.value();
        BrokerSummary brokerSummary = BrokerSummary.newBuilder()


        producer.send(new ProducerRecord<>(outputTopic, brokerSum
    });
    offsets.put(topicPartition,
      new OffsetAndMetadata(lastOffset.get() + 1L)); #4
});
try {
    producer.sendOffsetsToTransaction(offsets,
      consumer.groupMetadata()); #5
    producer.commitTransaction();  #6
 }
```

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)) { #1

        final List<NewTopic> topics = new ArrayList<>)();  #2

    topics.add(new NewTopic("topic-one", 1, 1)); #3
    topics.add(new NewTopic("topic-two", 1, 1));

    adminClient.createTopics(topics); #4
}
```

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(
     System.out.println(topicNames); #2
     adminClient.deleteTopics(Collections.singletonList("topic-tw
}
```

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); #1
...

Producer<String, EventsProto.Events> producer = new KafkaProducer
```

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); #1
producerConfigs.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_
  false); #2
producerConfigs.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VER
  true); #3
```

```
Producer<String, SpecificRecord> producer = new KafkaProducer<>(
  producerConfigs()) #4
```

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); #1
consumerProps.put(
  KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE_TYPE,
    EventsProto.Events.class); #2

Consumer<EventsProto.Events> consumer = new KafkaConsumer<>(
  consumerProps); #3
```

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 -> {   #1
        logins.add(event.getLoginEvent()); #2
    }
    case SEARCH_EVENT -> {
        searches.add(event.getSearchEvent());
    }
    case PURCHASE_EVENT ->  {
        purchases.add(event.getPurchaseEvent());
    }
}
```

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); #1
consumerProps.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READE
  true); #2

Consumer<SpecificRecord> consumer = new KafkaConsumer<>(consumerP
```

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", #1

  "fields" : [
    {"name": "txn_type", "type": [  #2
      "bbejeck.chapter_4.avro.TruckEvent",
      "bbejeck.chapter_4.avro.PlaneEvent",
      "bbejeck.chapter_4.avro.DeliveryEvent"
    ]}
  ]
}
```

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.

ℹ **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.

5 Kafka Connect

This chapter covers

- Getting started with Kafka Connect
- Applying Single Message Transforms
- Building and deploying your own Connector
- Making your Connector dynamic with a monitoring thread
- Creating a custom transformation

In this chapter we’re going to learn how to easily move events into and out of Apache Kafka. While Kafka can function as a central nervous system for data, it’s primary function is to provide a decoupled and centrilized approach to data access, but there are other important services, like full-text searching, report generation, and data analysis that can only be serviced by applications specific to those purposes. No one technology or application can satisfy the needs of a business or organization.

We’ve established in earlier chapters that Kafka simplifies the architecture for a technology company by ingesting events once and any group within the organization can consume events independent of each other. In the cases mentioned above where the consumer is another application you’re left writing consumers for the application. If you have more than one, you’ll end up repeating a lot of code. A better approach would be to have an established framework that you can deploy that handles either getting data from an application into Kafka or getting data out of Kafka into an external application. That framework does exist and it’s a key component of Apache Kafka - Kafka Connect.

Kafka Connect is part of the Apache Kafka project and provides the avilibity to integrate Kafka with other systems like relational databases, search engines, NoSql stores, and Cloud object stores and data wharehouses systems. By using Connect you can easily stream large amounts of data in and out of Kafka. The ability to perform this streaming integration between Kafka is critical to include legacy systems into todays event streaming applications.

Actually, by using Connect you can achieve a two-way flow between existing architectures and new applications. For example you can take incoming event data from Kafka and migrate it a typcial Model-View-Controller (MVC) application by using a connector to write results to a relational database. So you could consider Kafka Connect as a sort of "glue" that enables you to seemlessly integrate different applications with new sources of event data.

One concrete example of using Kafka Connect is to capture changes in a database table as they occur called Change Data Capture or CDC. CDC works by exporting changes to a database table (INSERT, UPDATE, and DELETE) to other applications. By using CDC you are taking the database changes and storing them in a Kafka topic, making them available to any downstream consumers with low-latency. One of the best parts of this integration is that the old application doesn't need to change, it's business as usual.

Now you could implement this type of work yourself using producer and consumer clients. But saying that statement is over simplification of all the work you would be required to put in getting that applcation production ready. Not to mention the fact that each system you are consuming from or producing to would require different handling.

Consuming changes from a relational database isnt the same as consuming from a NoSQL store and producing records to ElasticSearch is different from producing records to Amazon's S3 storage. So it makes sense to go with a proven solution that has serveral off-the-shelf components ready for you to use. There are literally hundreds of connectors available from Confluent at https://www.confluent.io/hub/ - while some are commercial and require payment, there are over one hundred connectors that are freely available for you to use.

Connect runs in a separate process from a Kafka cluster, either as single application or several instances as a distributed application. The different connectors are supplied as plugins that you configure on the classpath. There is no coding invovled to run a connector, you supply a JSON configuration file to the connect server. Having said that, if there isn't a connector available to cover your use case, you have the ability to implement your own connector as well. We’ll cover creating a custom connector in the last section of this chapter.

When it comes to Connect it’s important to understand there are two kinds of connectors sources and sinks. A source connector will consume data from an external source such as a Postgres or MySql database, MongoDB or an S3 bucket into Kafka topics. A sink connector does the opposite, it produces event data from a Kafka topic into an external applciation like ElasticSearch or Google BigQuery. Additionally, due to Kafka’s design you can have more than one sink connector exporting data from a Kafka topic at one time. This pattern of multiple sink connectors means that you can take an existing application and with a combination of source and sink connectors, share its data with other systems without having to make any changes to the original applicaiton.

Finally, Connect provides a way to modify data either coming into or out of Kafka. Single Message Transforms or SMT allow you modify the format of records from the source system when reading it into a Kafka. Or if you need to change the format of a record to match the target system you can use an SMT there as well. For example, when importing data from a database table with customer data, you may want to mask sensitive data fields with the `MaskField` SMT. There’s also the ability to transform the format of the data.

Let’s say you’re using Protobuf format in your Kafka cluster. You have a database table that feeds a topic via source connector and for the target topic you also have a sink connector writing records out to a Redis key-value store. In both cases neither the database or Redis works with Protobuf format. But by using a value converter you can seemlessly transform the incoming records into Protobuf format from the source connector while the sink connector will use another value converter to change the outgoing records back into plain text from Protobuf.

To wrap up our introduction, in this chapter you’ll learn how to deploy and use Kafka Connect for integrating external systems with Kafka, enabling you to build event streaming data pipelines. You’ll learn how to apply message transforms to make changes to incoming or outging records and how to build your own transformations. Finally we’ll cover how to build your own connector, just in case there’s not an existing one that will meet your needs.

5.1 Integrating external applications into Kafka

We’ve learned so far what Connect is and that it acts as a sort of "glue" that can stitch different systems together, but let’s see this in acton with an example.

Imagine you are responsible for coordinating new student signups for college orientation. The students indicate which orientation session the will attend through a web form that the college has been using for some time. It used to be that when the student arrived for orientation, other departments, housing, food service, guidance would need to meet with the students. In order to use the information the students applied on the form, a staff member would print out the information and hand out paper copies to the other staff members from each department.

But this process was error prone and could be time consuming as other staff members were seeing the student information for the first time and figuring out the logistics for each student on the spot is time consuming. It would be better if there was a process in place to share the information as soon as as student signed up. You’ve learned that the university has recently embarked on a technology modernization approach and have adopted Kafka at the center of this effort.

The office of student affairs, the department responsible for getting orientation information, is set on their current application structure which is a basic web application that ends up feeding an PostgreSQL database. The reluctance of the department to change at first presents an issue, but then you realize there’s a way to integrate their data into the new event streaming architecture.

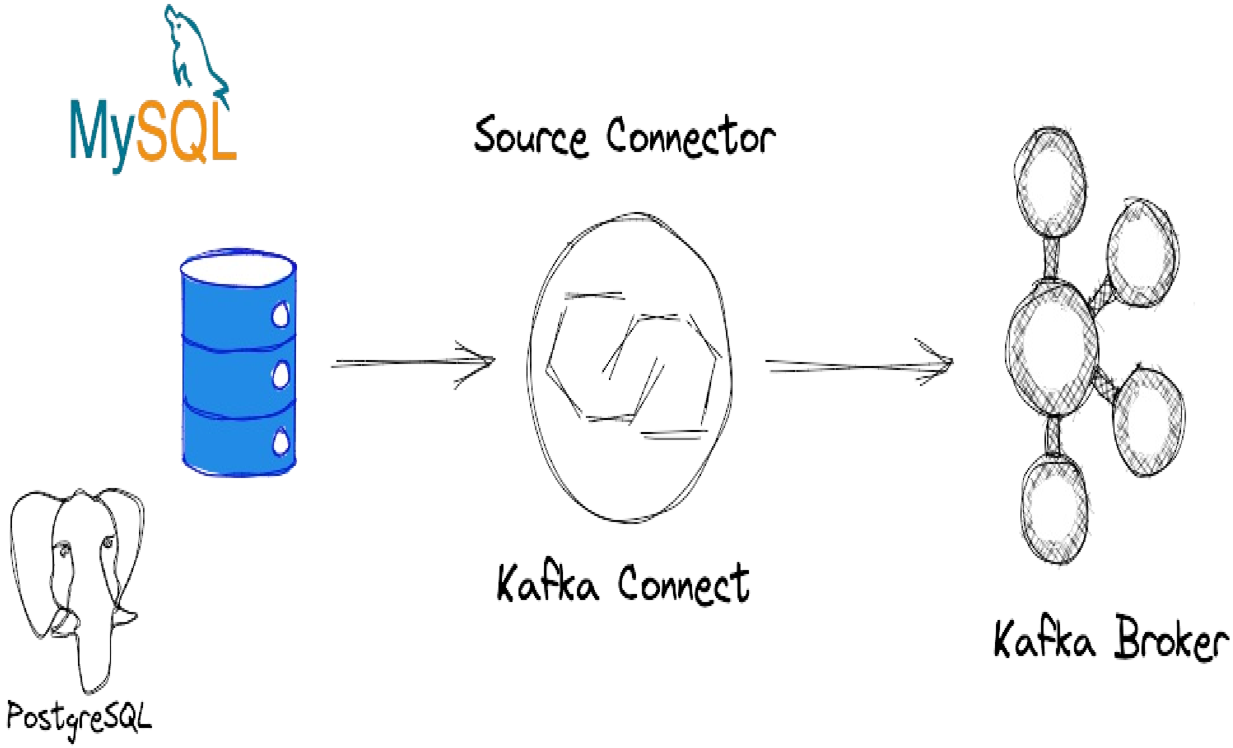
By using a JDBC source connector you can directly send orientation registrant information directy into a Kafka topic. Then the other departments can set up their own consumer applications to recieve the data immediately upon registration.

Note

When you use Kafka Connect to bring in data from other sources, the integration point is a Kafka topic. This means *any* application using a `KafkaConsumer` (including Kafka Streams) can use the imported data.

Figure 5.1 shows how this integration between the database and Kafka works. In this case, you'll use Kafka Connect to monitor a database table and stream updates into a Kafka topic.

Figure 5.1. Kafka Connect integrating a database table and a Kafka topic



Now that you've decided to use Kafka Connect to help you integrate the orientation student signups with the new event streaming platform rolled out at the university, let's dive in with a simple working example in the next section. After that we'll go into more details on how connect works and the main concepts.

5.2 Getting Started with Kafka Connect

Kafka Connect runs in two flavors: distributed and standalone mode. For most production environments, running in distributed mode makes sense, because you can take advantage of the parallelism and fault tolerance available when you run multiple Connect instances. Standalone mode is good for development on a local machine or your laptop.

Note

For our purposes, instead of downloading Kafka Connect and running it locally in your laptop, we'll use a connect docker image. Remember, Kafka Connect runs separately from a Kafka broker, so setting up a new `docker-compose` file adding a connect service makes things simpler for development and allows you focus on learning.

When setting up Kafka Connect there's two levels of configuration you'll provide. One set of configurations are for the connect server (or worker) itself and the other set is for the individual connector. I've just mentioned a new term, connect worker, that we haven't discussed yet but we'll cover what a worker is along with other core connect concepts in the next section.

Let's look at the some of the key configuration parameters you'll work with for Kafka Connect:

- `key.converter`—Class of the converter that controls serialization of the key from Connect format to the format written to Kafka. In this case you'll use the built-in `org.apache.kafka.connect.storage.StringConverter`. The value specified here sets a default converter for connectors created on this worker, but individual connectors can override this value.
- `value.converter`—Class of the converter controlling serialization of the value from Connect format to the format written to Kafka. For this example, you'll use the built-in `org.apache.kafka.connect.json.JsonConverter`. Just like key converter, this is a default setting and connectors can use a different

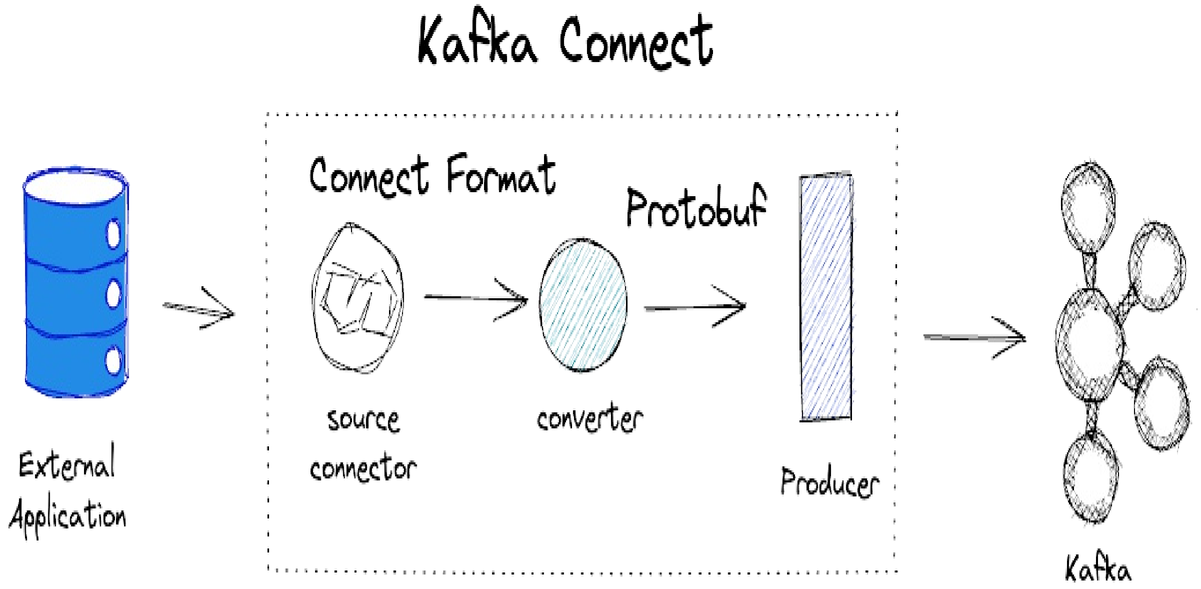
setting.

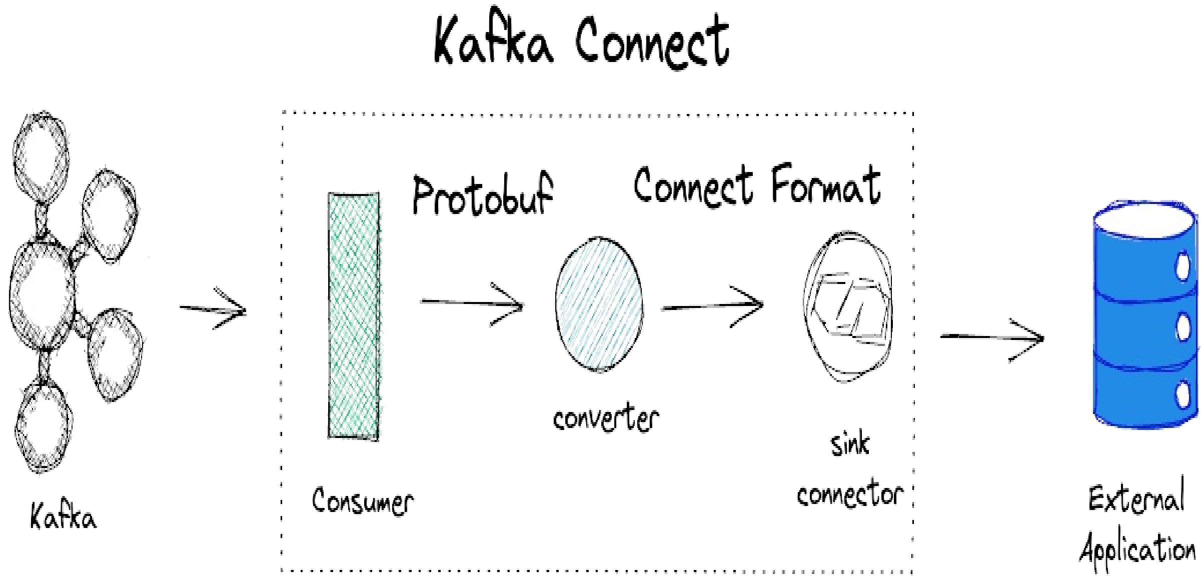
- `plugin.path`—Tells Connect the location of plugins such as connectors and converters, and their dependencies, on the worker.
- `group.id`—The id used for all the consumers in the connect cluster, this setting is used for source connectors only.

Note

I’m discussing these configuration paramters here for completeness. You won’t have set any of these configurations yourself as they are provided in the docker-compose file under the connect service. The connect paramaters in the file will take form similar to `CONNECT_PLUGIN_PATH` which will set the `plugin.path` configuration. Additionally the docker-compose file will also start a Postgres DB instance and will automatically populate a table with values needed to run the example.

Figure 5.2. Connect coverter changes data format before it gets into Kafka or after it leaves Kafka





I’d like to briefly explain what a converter is here. Connect uses a converter to switch the form of the data that has been captured and produced by a source connector into Kafka format or convert from Kafka format to an expected format of an external system as in the case of a sink connector. Since this setting can be changed for individual connectors, it allows for any connector to work in any serialization format. For example one connector could use Protobuf while another could use JSON.

Next, let’s take a look at some configuraton you’ll provide for a connector. In this case we’re using a JDBC connector for our first example, so you’ll need to provide the information needed for the connector to connect to the database, like username and passwor and you’ll also need to specify how the connector will determine which rows to import into Kafka. Let’s take a look at few of the more important configs for the JDBC connector:

- `connector.class`—Class of the connector.
- `connection.url`—URL used to connect to the database.
- `mode`—Method the JDBC source connector uses to detect changes.
- `timestamp.column.name`—Name of the column tracked for detecting changes.
- `topic.prefix`—Connect writes each table to a topic named *topic.prefix+Table name*.

Most of these configurations are straightforward, but we need to discuss two of them—`mode` and `timestamp.column.name`—in a little more detail because they have an active role in how the connector runs. The JDBC source connector uses `mode` to detect which rows it needs to load.

For this example, you’ll use the `timestamp` setting for the `mode`, which relies on column containing a timestamp, named in the `timestamp.column.name` config. While it’s should be obvious an that insert sets the timestamp, we’ve also added a trigger to the database docker image that refreshes the timestamp with update statements. By using a timestamp, the connector will pull any values from the database table whose timestamp is greater than the last one from the previous import.

Another value for the mode could be `incrementing` which relies on column with an auto-incrementing number, which would focus only in new inserts. The JDBC connector provides several configuration items the connector will use for determining which rows need to be imported. Under the covers, the connector will issue a query to the database and the result of the query is what get produced into Kafka. I’m not going to go into any more detail on the JDBC connector in this chapter. It’s not that we’ve covered everything there is to know, the information on the JDBC connector could fill an entire chapter on its own. The more important point is to see that you need to provide configurations for the individual connector, and most of them provide a rich set to control the connector's behavior.

Now let's take a look at how you start an individual connector. It's easily achieved by using Connect's provided REST API. Here's a sample of what you'll use to launch a connector:

Listing 5.1. REST API call to start a connector

```
curl -i -X PUT http://localhost:8083/connectors/jdbc_source_conne
     -H "Content-Type: application/json" \
     -d '{
            "connector.class": "io.confluent.connect.jdbc.JdbcSou
            "connection.url": "jdbc:postgresql://postgres:5432/po
            "connection.user": "postgres",
            "connection.password": "postgres",
            "mode":"timestamp",
            "timestamp.column.name":"ts",
            "topic.prefix":"postgres_",
            "value.converter":"org.apache.kafka.connect.json.Json
            "value.converter.schemas.enable": "false", #2
            "tasks.max":"1" #3
        }'
```

So this REST call will start the JDBC connector running. There's a few configurations I'd like to call your attention to here. At annotation one you set the `value.converter` which will convert the incoming records from the database into JSON format. But if you'll look at annotation two you'll see a `value.converter.schemas.enable` configuration which is set to false, which means the converter will not preserve the schmea coming from the connector in the contents of the message.

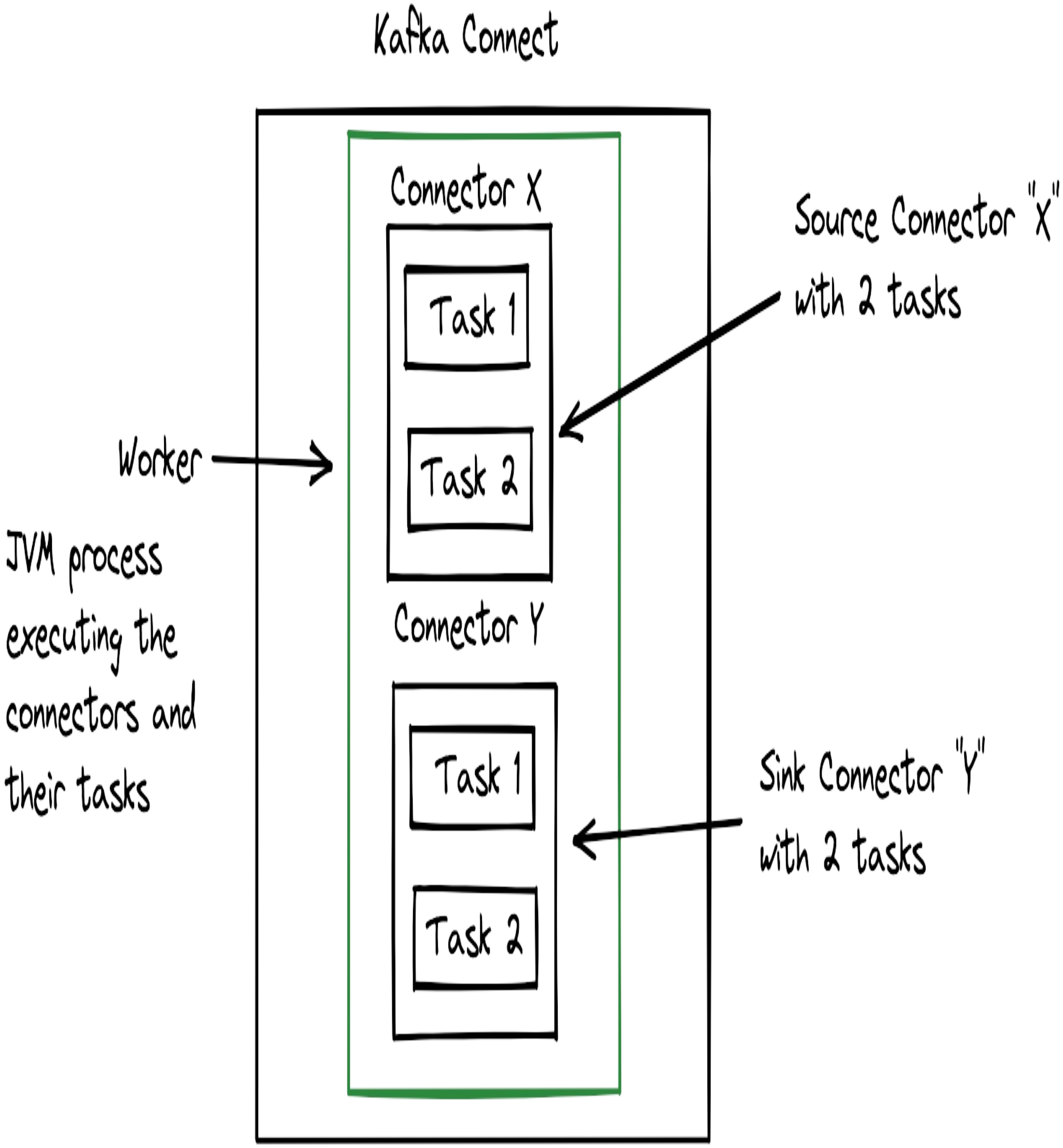
Keep in mind when using a JSON converter the schema is attached to each incoming record which can increase the size of it significantly. In this case, since we are producing records ***into*** Kafka we can disable the inferred schemas. But when conumsing from a Kafka topic to write to an external system, depending on the connector, you must enable the schema infering so Connect can understand the byte arrays that are stored in Kafk. A better approach would be to use Schema Registry, then you could use either Avro, Protobuf, or JSONSchema for the value converter. We covered schemas and

Schema Registry in chapter 3, so I won't go over those details again here.

At annotation three you see a `tasks.max` setting and to fully explain this configuration, let's provide some additional context. So far we've learned that you use Kafka Connect to pull data from external system into Kafka or to push data from Kafka out to another application. You just reviewed the JSON required to start a Connector, it does not do the pulling or pushing of data. Instead the Connector instance is responsible for starting a number of tasks whose job it is to move the data.

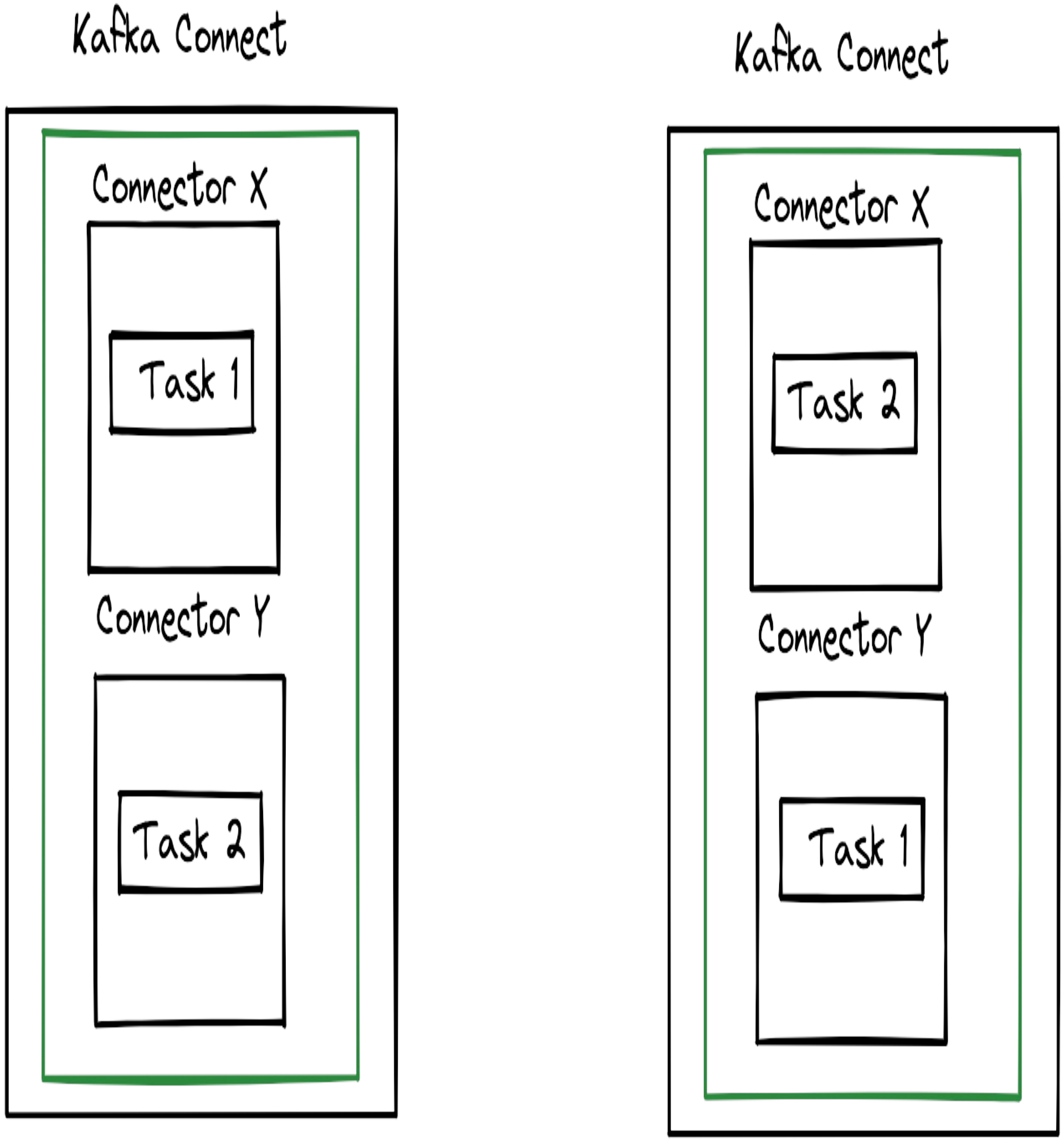
Earlier in the chapter we mentioned there are two types of a connectors - `SourceConnector` and `SinkConnector` and they use two corresponding types of tasks; the `SourceTask` and `SinkTask`. It's the Connector's main job to generate task configurations. When running in distributed mode, the Connect framework code will distribute and start them across the different workers in the connect cluster. Each task instance will run in its own thread. Note that setting the `tasks.max` doesn't guarantee that's the total number of tasks that will run, the connector will determine to correct number it needs ***up to*** the maximum number. At this point it would be helpful to look at an illustration of the relationship between workers connectors and tasks:

Figure 5.3. Connect in standalone mode all tasks reside in one worker


Kafka Connect
Connector X
Task 1
Task 2
Source Connector "X" with 2 tasks
Worker
JVM process executing the connectors and their tasks
Connector Y
Task 1
Task 2
Sink Connector "Y" with 2 tasks

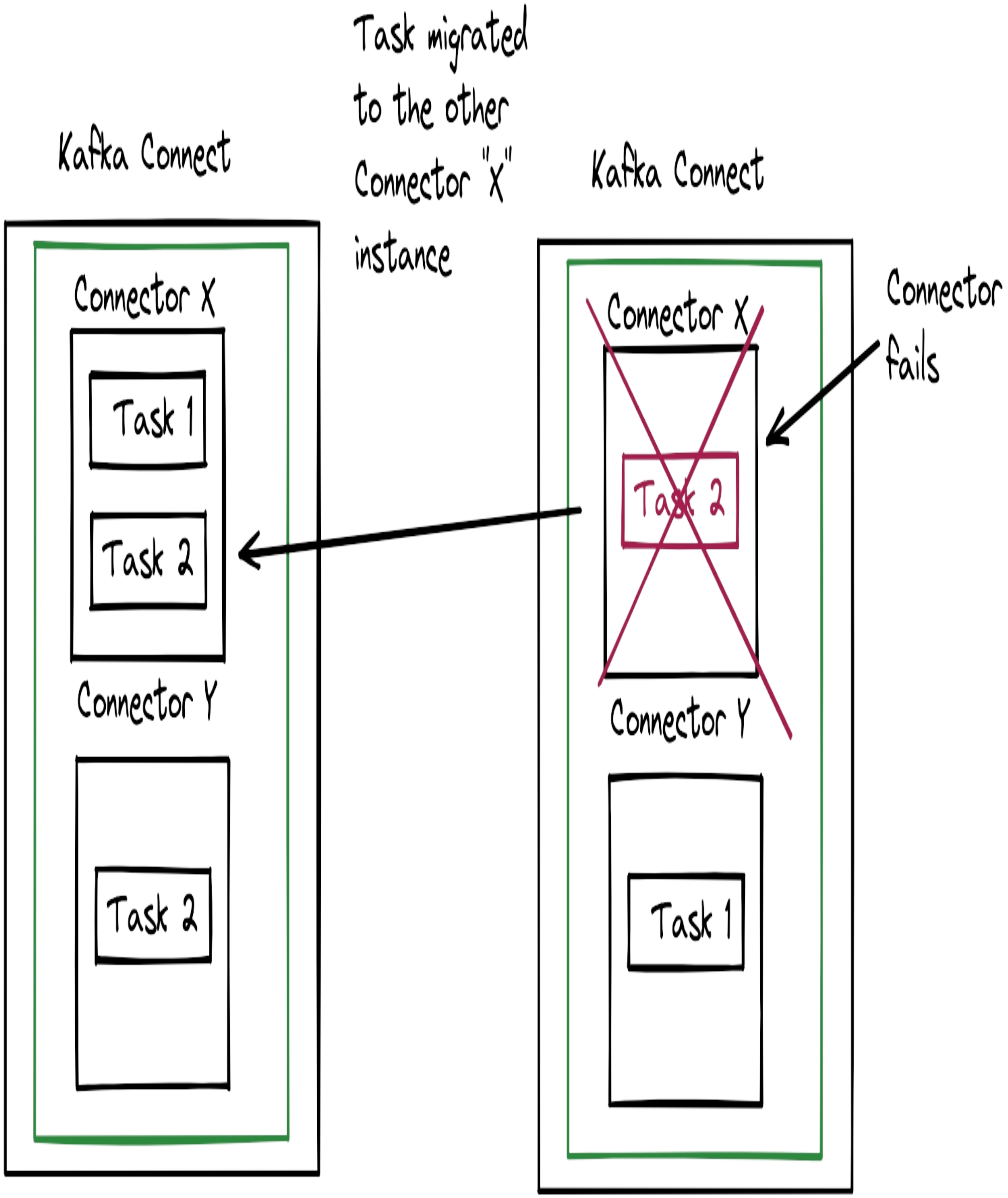
Here we’re looking at Kafka Connect in standalone mode, there’s a single worker which is a JVM process responible for running the connector(s) and their task(s). Now let’s take a look at distributed mode:

Figure 5.4. Connect in distributed mode, tasks get spread around to other connector instances



As you can see here in distributed mode the tasks are spread out to other workers in the connect cluster. Not only does distributed mode allow for higher througput due to spreading out the load, it also provides the ability to continue processing in the face of a connector failure. Let’s look at one more diagram to illustrate what this means:

Figure 5.5. Connect in distributed mode provides fault tolerance



From looking at this illustration should a Kafka Connect worker stop running the Connector and task instances on that worker will get assigned to other workers in the cluster. So while standalone mode is great for prototyping and getting up and running quickly with Kafka Connect, running in distributed mode is recommended for production systems as it provides fault tolerance; tasks from failed workers get assigned to the remaining workers in the cluster. Note that in distributed mode you’ll need to issue a REST API call to start a particular connector on each machine in the connect cluster.

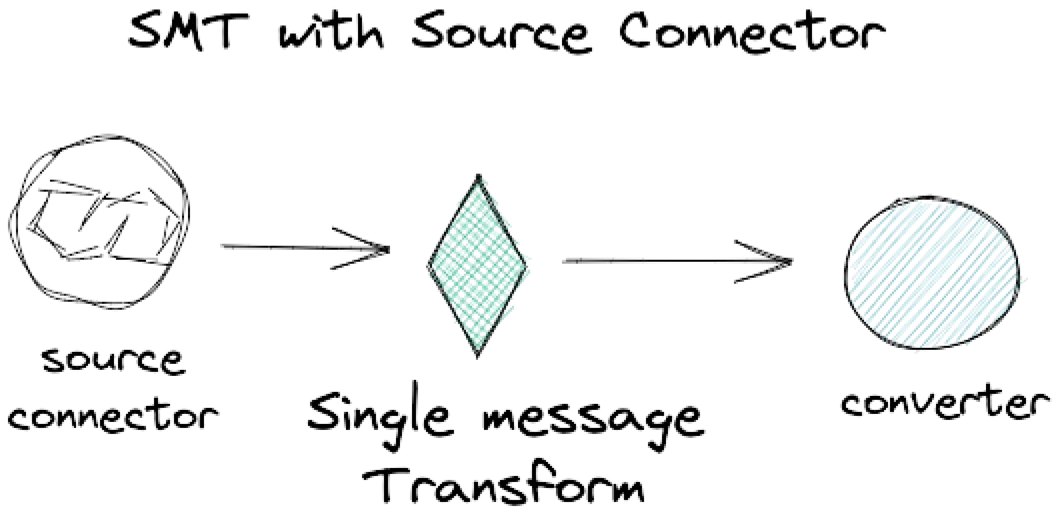
Now let’s get back to our example. You have your connector running but there are a few things you’d like to do differently. First of all, there are no keys for the incoming records. Having no keys is a problem because records for the same student will possibly end up on different partitions. Since Kafka only guarantees ordering within a partition, multiple updates made by a student could get processed out of order. Secondly the students input their social security number. It’s important to limit the exposure of the ssn, so it would be great to alter or mask it before it gets into Kafka.

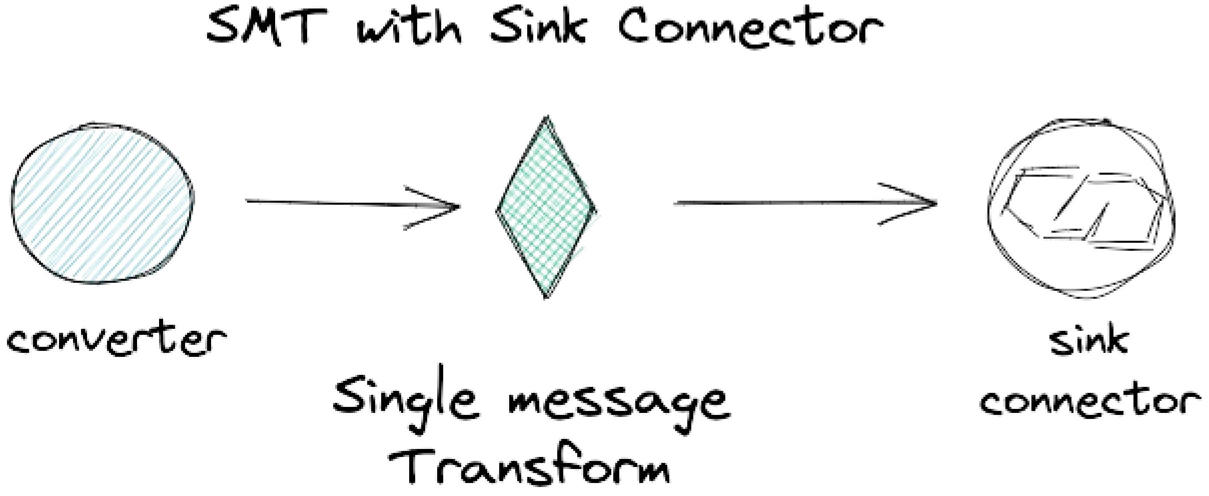
Forturnatly, Connect offers a simple, but powerful solution to this with transforms.

5.3 Applying Single Message Transforms

Simply put, single message transforms or SMTs provide a way to perform lightwieght changes to records either before they get into Kafka or on their way out to external systems, depending on if a source connector or a sink connector is used. The key point about transforms is that the work done should be simple, meaning they work on only one record at a time (no joins or aggregations) and the use case should be applicable solely for connectors and not custom producer or consumer applications. For anyhing more complext it’s better to use Kafka Streams or ksqlDB as they are purpose built for performing complex operations. Here’s an illustration depicting the role of the transformation:

Figure 5.6. Connect coverter changes data format before it gets into Kafka or after it leaves Kafka





As you can see here, the role of a SMT is to sit between the connector and the coverter. For a source connector it will apply its operation the record ***before*** it gets to the coverter and in the case of a sink connector it will perform the transformation ***after*** the converter. In both cases, SMTs operate on data in the same format, so that most SMTs will work equally well on a sink or a source connector.

Connect provides several SMTs out of the box that tackle a wide range of use cases. For example there are SMTs for fitering records, flattening nested structures, or removing a field. I won’t list them all here but for the full list of SMTs consult the Kafka Connect documentation -

https://kafka.apache.org/documentation.html#connect_included_transformatic

For our purposes we’ll use three transforms `ValueToKey`, `ExtractField`, and `MaskField`. Working with the provided SMTs is a matter of adding some JSON configuration, no code required. Should you have a use-case where the provided tranformations won’t provide what you need, you can of course write your own transform. We’ll cover creating a custom SMT a little later in the chapter.

Here’s the full JSON you’ll use to add the necessary transforms for your connector:

Listing 5.2. The JSON transform configuration

```
"transforms":"copyFieldToKey, extractKeyFromStruct, maskSsn", #1
"transforms.copyFieldToKey.type":
      "org.apache.kafka.connect.transforms.ValueToKey", #2
"transforms.copyFieldToKey.fields":"user_name", #3
"transforms.extractKeyFromStruct.type":
       "org.apache.kafka.connect.transforms.ExtractField$Key", #
"transforms.extractKeyFromStruct.field":"user_name",  #5
"transforms.maskSsn.type":
       "org.apache.kafka.connect.transforms.MaskField$Value", #6
"transforms.maskSsn.fields":"ssn",   #7
"transforms.maskSsn.replacement":"xxx-xx-xxxx" #8
```

Most of the JSON for the transform configuration should be straight forward. There’s a comma separated list of names, each one representing a single transform.

Tip

The order of names in the `transforms` entry is not arbitrary. They are in order of how the connector will apply them, so it’s importanc to consider how each transform will change the data going through the transformation chain and how it will affect the final outcome.

Then for each name in the list you provide the class of the transform, then the field that it will apply to. But there is one thing I’d like to point out that might not be readily apparent. With the `copyFieldToKey` transform you’ve indicated that you want to use the `user_name` column for the key of each resulting Kafka record. But the result produces a single field in `STRUCT` that looks like this:

Listing 5.3. Struct

```
Struct {"user_name" : "artv"}
```

But what you realy want is the value of the struct field `user_name` for the key, so you apply the `ExtractField` transform as well. In the configs you need to specify the transform should apply the extraction on the key like this `ExtractField$Key` and Connect applies the second transform and the key ends up with the raw single value applied to the key for the incoming record.

I’d like to point out something here about the transforms that could go unnoticed. You can chain mulitple transforms together to operate on the same field, this is demonstrated in this example here by first copying a field to the key, then extracting a value from the intermediate result of the field copy operation. But there is a balance you’ll need to strike here, if you find yourself building a transform chain the starts to go beyond 2, it might be good to consider using Kafka Streams to perform the transformational work as it will be more efficient.

The final transform you utilize is `MaskField`, which you’ll apply to the field containing the ssn of the student. Again you’ll see in the configuration how you specifed you want to apply the masking the value with the `MaskField$Value` setting. In this case you specify the replacement for the ssn as a string of x characters for each number resulting a value of `xxx-xx-xxxx`. With the `MaskField` transform you also have the option of not specifying a specific replacement and it will use an empty value based on the type of the field it’s replacing - an empty string for string field or a 0 for numerical ones.

Now you’ve completed a fully configured connector that will poll the database for changes or updates and import them into Kafka, making your relational database part of your event streaming platform!

Note

We’ve talked about the JDBC Connctor in this section. There are a few corner cases where the JDBC connector won’t get the latest changes into Kafka. I won’t go into those here, but I would reccomend taking a look at the Debezium Conntor for integration relational databases with Apache Kafka (https://debezium.io/documentation/reference/stable/connectors/index.html). Instead of using an incrementing value or timestamp field, debezium uses the database changelog to capture changes that need to go into Kafka.

To run the example we just detailed in this section consult the README file in the chapter_5 directory of the source code for this book. There’s also a sink connector as part of the example, but we won’t cover it here, as you deploy it in the same way, provide some JSON configurations and issue a REST call to the connect worker.

So far you’ve set up a source connector, but you’ll also want to send events from Kafka into an external system. To accomplish that you’ll use a sink connector, specifically an elastic search sink connector.

5.3.1 Adding a Sink Connector

One additional feature that you thought to add was the ability for incoming students to search for a potential roomate based in the input data gathered for the orientation process. The idea being that when students come for oritentation part of the process can be to enter a few key words and get hits from other students that have the same preferences.

When you originally pitched the idea everyone was enthusiastic, but questions on how to get the data seemed too difficult and the prospect of setting up a new pipeline for gathering information was not well receieved, so the idea was put on hold. But now with the adoption of Kafka and Connect for importing the incoming student information, exposing a search application with Elasitc Search (https://www.elastic.co/) is as simple now as providing a sink connector.

Fortunately, there’s already an elastic search sink connector, so all you need to do is install the jar files to your running Connect cluster and you’re all set. So getting started with the elastic sink connector is simply a matter of making a REST API call to get the connector up and running:

Listing 5.4. REST Call to install Elastic Connector

```
$ curl -i -X PUT localhost:8083/connectors/student-info-elasticse
    -H "Content-Type: application/json" \
        -d ‘{
                        "connector.class": "io.confluent.connect.
                        "connection.url": "http://elasticsearch:9
                        "tasks.max": "1",
                        "topics": "postgres_orientation_students"
                        "type.name": "_doc",
                        "value.converter": "org.apache.kafka.conn
                        "value.converter.schemas.enable": "false"
                        "schema.ignore": "true",
                        "key.ignore": "false",
                    "errors.tolerance":"all",  #2
                    "errors.deadletterqueue.topic.name":"orientat
                    "errors.deadletterqueue.context.headers.enabl
            "errors.deadletterqueue.topic.replication.factor": "1

            }’
```

For the most part, the configurations here are similar to what you saw for the JDBC connector with the exception for the configurations that are specific for the connector. At annotation one, we see the name of topic the sink connector will use to read records and write them to elastic search.

But there are three additional configurations, the ones starting with `errors`, that we haven’t seen before and we should discuss them now. Since a sink connector is attempting to write event records to an external system there’s a good possibility for errors. After all to work with distributed applications is to embrace failure and provide the mechanisms for how you want your application to respond to them.

Note

Annotation 5 in the configuration for the Elastic sink connector sets to the replication factor for the DLQ topic. In a single node cluster, such as our

Docker development environment you'll want to set this to 1, otherwise the connector won't start since there aren't enough brokers for the default replication factor of 3.

The `errors.tolerance` configuration at annotation two specifies how the connector will responde when it gets an error. Here you've used a setting of `all` meaning that regardless of errors during conversion and transformation, and regardless of the number of retriable or data-dependent errors encountered while writing records to the sink, the connector will keep running. Then you'd need to go review the logs for that specific connector to determine what went wrong and how to procede. While the `all` setting allows to connector to continue operating, you still want to know when a record fails. The configruation at annotation three creates a dead letter queue or DLQ, where Connect can set aside the records it can't deliver.

But with enabling a DLQ topic Connect only stores the failed record itself. To get the reason ***why*** the failure occured you'll need to enable storing some additional information in the record header which you've done at annoation four. We covered record headers previously so you can refer back to chapter four to review information on reading with record headers.

As with anything in life, there are trade-offs to consider with using a DLQ. While a setting of `none` for `errors.tolerance` and shutting down on error sounds harsh, if this were a production system, you'd probably find out very fast that something was amiss and needed to be fixed. Contrast this with the setting of `all` which will continue on regardless of any errors the connector may encounter, it's critical to track any errors because to run indefinitely with an error condition could arguably be worse than shutting down. In other words "if a tree falls in a forest and no one is there to hear it, does it make a sound?", if errors arrive in your DLQ, but no one is looking at it, it's the same as no errors ocurring at all.

So with enabling a DLQ, you'll want to set up some sort of monitoring (i.e. a KafkaConsumer) that can alert on any errors and possibly take action like shutting down the problematic connector in the face of continuous problems. The source code for the book will have a basic implmentation demonstrating this type of functionality.

So now you’ve learned about using a source Connector for bringing event data into Kafka and sink Connector for exporting from Kafka to an external system. While this is a simple workflow, don’t think this example is representative of the full usefulness of Kafka Connect. First of all, there could be several sink connectors writing the imported records out to external systems, not just one, and increasing the leverage of Kafka for creating an event streaming platorm, by effectively bringing in all external system into one central "data flow".

In practice you can have several different source connectors bringing data and more than likely there would be more processing of those incoming records, and not a simple data pipe demonstrated in this chapter.

For example consider that there could be any number of client applications relying on the topic with data a source connector provides. Each application could produce its own unique results to another topic. Then there could be any number of sink connectors producing the updated records back out external systems in your architecture.

As you can see from the this illustration, Connect plays an integral part in gluing together external data applications into the Kafka event streaming platform. By using Connect, essentially any application with a connector can plug into it and leverage Kafka as the central hub for all incoming data.

But what about if you have a source or a sink candidate that doesn’t have an existing connector. While there are literally hundreds of existing connectors today, it’s possible that the one you need might not exist. But there’s good news, there’s a Connect API that makes it possible for you to implement your own connector and that’s what you’ll do in the next section.

5.4 Building and deploying your own Connector

In this section we’re going to walk through the steps you’ll take to implement your own connector. Let’s say you work for a fintech firm and along with providing institutional-grade financial data to investors and investment firms, your company is branching out an wants provide analysys based on a subscription model.

In order to offer the analysis data, the company as stood up a few different departments that will ingest real-time stock ticker data. To get a full feed of live data is not cheap and the cost for each department to have their own feed would be prohibitively expensive. But you realize that if you created a source connector you could consume the feed once and every department could standup their own client application to consume the feed in near-realtime from your Kafka cluster. So with your idea firmly in mind, let’s move on to implementing your own connector.

5.4.1 Implementing a connctor

To develop your own connector your going to work the following interfaces `Connector` and `Task`. Specifically you’re going the extend the abstract class `SourceConnector` which itself extends the `Connector` class. There are several abstract methods that you’ll need to implement, but we’re not going to go over each one, just the most significant ones. You can consult the source code from the book to look at the full implementation.

Let’s start with configuration. The Connector class itself doesn’t do too much when it comes to moving data, it’s main responsibilty is to make sure each `Task` instance is configured properly, as it’s the `Task` that does the work of getting data either into or out of Kafka. So you’ll create a `StockTickerSourceConnectorConfig` class that you’ll instantiate directly inside your connector. The configuration class contains a `ConfigDef` instance to specify the expected configurations:

Listing 5.5. Setting up a ConfigDef instance

```
public class StockTickerSourceConnector extends SourceConnector {
private static final ConfigDef CONFIG_DEF = new ConfigDef() #1
            .define(API_URL_CONFIG,         #2
                   ConfigDef.Type.STRING,
                   ConfigDef.Importance.HIGH,
                   "URL for the desired API call")
            .define(TOPIC_CONFIG,    #3
                   ConfigDef.Type.STRING,
                   ConfigDef.Importance.HIGH,
                   "The topic to publish data to")

... followed by the rest of the configurations
}
```

Here you can see we’re adding the configurations using a fluent style where we chain the method calls `define` one after another. Note that there more configurations we’re adding but it’s not necessary to show all of them here, I think you get the point of how they all get added. By building the `ConfigDef` the connector will "know" what configurations to expect. The `ConfigDef.define` method allows you to provide a default value. So if an expected configuration isn’t provided, the default value gets populated in its place. But if you don’t provide a default value for any configuration, when starting the connector a `ConfigException` is thrown shutting the connector down. The next part of the connector we’ll want to take a look at is how it determines the configuration for each `Task` instance.

Listing 5.6. Configuring the tasks

```
@Override
public List<Map<String, String>> taskConfigs(int maxTasks) {
 List<Map<String, String>> taskConfigs = new ArrayList<>();
 List<String> symbols = monitorThread.symbols(); #1
 int numTasks = Math.min(symbols.size(), maxTasks);
 List<List<String>> groupedSymbols =
       ConnectorUtils.groupPartitions(symbols, numTasks); #2
 for (List<String> symbolGroup : groupedSymbols) {
    Map<String, String> taskConfig = new HashMap<>(); #3
    taskConfig.put(TOPIC_CONFIG, topic);
    taskConfig.put(API_URL_CONFIG, apiUrl);
    taskConfig.put(TOKEN_CONFIG, token);
    taskConfig.put(TASK_BATCH_SIZE_CONFIG, Integer.toString(batch
    taskConfig.put(TICKER_SYMBOL_CONFIG, String.join(",", symbolG
    taskConfigs.add(taskConfig);
 }
return taskConfigs;
}
```

You’ll notice at annotation one, you’re getting the list of symbols from an instance variable named `monitorThread`. I’d like to hold talking about this field until we get to the monitoring section later in this chapter. For now it’s enough to know this method returns the list of symbols need to run the ticker.

If you recall from earlier in the chapter, one of the configurations you set for the connector is `max.tasks`. This configuration determines the maximum number of tasks the connector may spin up to move data. Our Stock API service can take a comma separated list of up to 100 ticker symbols to retrieve information about those companies. To make sure work is evenly distributed across all tasks, you’ll want to partition the ticker symbols into separate lists.

For example if you have specified 2 as the maximum number of tasks and there are 10 ticker symbols, the connector will partition them into two lists of 5 ticker symbols each. This grouping of ticker symbols occurs at annotation 1 using the `ConnectionUtils.groupPartitions` utility method. That’s about as much that we want to cover for the connector implementation, so we’ll move on to the next section implmenting the task.

Since you’re creating a `SourceConnector` you’ll extend the abstract class `SourceTask` to build your task implmentation, `StockTickerSourceTask`. There’s a handful of method on the class you’ll need to override, but we’re going to focus on the `poll()` method since that is at the heart of what a `SourceTask` does, get data from an external source and load it into Kafka.

Let’s review the behavior of a `SourceTask` at a high level. Once you start a connector it will configure the correct number of tasks and start them in their own thread. Once the thread is executing the `SourceTask.poll()` is called periodically to retrieve records from the configured external application. There’s no overall configuration on how often the `poll` method is executed. While you don’t have explicit controll how often the `poll` method of your task is called, you can add some throttling inside your task implementation to wait a desired amount of time before executing the logic of your `SourceTask`.

There’s a basic implemtation of throttling in the our `SourceTask` example. Why would you want to add throttling to the source task exectution? In our case we’re going to issue a request to a HTTP endpoint for a stock service API. Many of the available API’s have limits on how often you can hit the service or give a total number requests per day. So by adding throttling you can make sure the calls to the API are spread out in way that stays within the plan limits.

Also, you’ll want to handle the case when there’s no records to return. It would be a good idea to wait a small amount of time, 1-2 seconds for example, to allow for the source to have something new to return. It’s important however to return control to the calling thread by returning `null` so the worker can respond to requests to pause or shutdown.

Now let’s take a look at the action of the `StockTickerSourceTask.poll` method. To make sure we fully understand what is going on with the polling logic, we’re going to view the code in sections.

Listing 5.7. Start of poll method for the source task

```
public List<SourceRecord> poll() throws InterruptedException {
    final long nextUpdate = lastUpdate.get() + timeBetweenPoll; #
    final long now = sourceTime.milliseconds();     #2
    final long sleepMs = nextUpdate - now; #3

    if (sleepMs > 0) {
        LOG.trace("Waiting {} ms to poll API feed next", sleepMs)
        sourceTime.sleep(sleepMs);
    }
```

At the very top of the method, we calculate what time our next API call should take place, remember we don’t want to overrun the limit set for how many calls we can make to the service. If the current time is less that the time of our last call plus the interval we want to have in between calls, we’ll have the polling thread sleep for that amount of time.

Now once the wait time passes, the next part of the method that executes is the core logic of retrieving the stock results:

Listing 5.8. Core polling logic

```
// Details left out for clarity

HttpRequest request = HttpRequest.newBuilder()  #1
                .uri(uri)
                .GET()
                .headers("Content-Type", "text/plain;charset=UTF-
                .build();
HttpResponse<String> response;

try {
    response = httpClient.send(request,
                  HttpResponse.BodyHandlers.ofString()); #2
    AtomicLong counter = new AtomicLong(0);
    JsonNode apiResult = objectMapper.readTree(response.body());
    ArrayNode tickerResults = (ArrayNode) apiResult.get(resultNod
    LOG.debug("Retrieved {} records", tickerResults.size());
    Stream<JsonNode> stockRecordStream =
              StreamSupport.stream(tickerResults.spliterator(), f

    List<SourceRecord> sourceRecords = stockRecordStream.map(entr
        Map<String, String> sourcePartition =
              Collections.singletonMap("API", apiUrl);
        Map<String, Long> sourceOffset =
              Collections.singletonMap("index", counter.getAndInc
       Schema schema = getValueSchema(entry);  #5
       Map<String, Object> resultsMap = toMap(entry); #6
       return new SourceRecord(sourcePartition,
                               sourceOffset,
                               topic,
                               null,
                               schema,
                               toStruct(schema, resultsMap)); #7

    }).toList();

    lastUpdate.set(sourceTime.milliseconds()); #8

   return sourceRecords; #9
}
```

The logic for the polling is pretty clear. You create a `HTTPRequest` object and then submit it to the ticker API endpoint, then you read the string repsponse body that API endpoint returns. The results are in JSON format, with the ticker symbol information in a nested array. You extract the array and flatten the results, mapping each one to a `SourceRecord` so that each entry in the array will become a record sent to Kafka.

There’s one part of this conversion that we need to cover here. The `SourceRecord` constructor accepts two parameters a source partition and a source offset, both represented by a `Map` instance. The notion of a source partition and offset might seem a bit out of context considering the source for a `SourceConnector` will not be Kafka. But the concept of a source partition indicates a general location description of where the connector sources the data - a database table name, file name or in our case here the API url. You’ll want to take note of annotation 5, we build the schema for the record returned from the ticker API call. Then at annotation six you convert the JSON record into a `Map`. We need to do these two steps to create a `Struct` used for the value of the `SourceRecord`. Additionally, we pass in the generated schema to the `SourceRecord` instance which may end up in the Kafka topic if you’ve configured the connector to include the schema.

With that gereralized definition of a source partition, the source offset is the position of the individual record in the retreived result. If you remember in Kafka a consumer commits the offset of the last fully consumed record so if it shuts down for any reason, it can resume operations from that committed offset.

Earlier in the section, your custom connector instance used a `monitoringThread` variable to get the list of ticker symbols to track. In the next section I’ll explain what this variable is and why the connector is using it.

5.4.2 Making your connector dynamic with a monitoring thread

In this example connector it would be reasonable to assume the list of ticker symbols is static. But what if you wanted or needed to change them? Of course you can use the Connect REST API to update the configurations but that means you have to keep track of any changes and manually update the connector. But you can provide a monitoring thread in conjuction with your custom connector to keep track of any changes and update the connector tasks automatically.

To be clear, a monitoring thread is not something unique to the Connector, you’ll implement a class that extends a regular `java.lang.Thread` class. Conceptually, you’ll start the thread when your connector starts and it will contain the logic needed to check for any changes and reconfigure the connector tasks if needed.

Imagine you have a separate microservice that handles which information needs to be included in the source connector configuration. The micro-service generates a file containing a comma separated list of the ticker symbols. So what you’ll need is for your monitoring thread to periodically read in the file and compare its contents against the current list of symbols. If there’s a change, then it will trigger a reconfiguration of the connector tasks. A better example of why you’d want to use a monitoring thread is the JDBC Connector. You can use the JDBC Connector to import an entire relational database, possibly consisting of many tables. In any organization a relational database is not a static resource, it will change with some frequency. So you’ll want to automatically pick up those changes to make sure the connector is importing the latest data into Kafka.

Let’s start the monitoring thread analysis by looking that class declaraton and constructor:

Listing 5.9. Class declaration and constructuor of monitoring thread

```
public StockTickerSourceConnectorMonitorThread(
  final ConnectorContext connectorContext,  #1
  final int monitorThreadCheckInterval, #2
  final String symbolUpdatesPath) {  #3
        this.connectorContext = connectorContext;
        this.monitorThreadCheckInterval = monitorThreadCheckInter
        this.symbolUpdatePath = Paths.get(symbolUpdatesPath);
    }
```

The constructor parameters are fairly self-explanatory, but the `ConnectorContext` deserves a quick discussion. Any class exteding the abstract `Connector` class will have access to the `ConnectorContext`, so you don’t need to concerned about where it comes from. It’s used to interact with the connect runtime, requesting task reconfiguration when the source changes for example.

Now for the behavior of the monitoring thread, you’ll override the `Thread.run()` method and provide the logic you want to execute should there be any relevant changes:

Listing 5.10. Overridden run method of the `StockTickerSourceConnectorMonitorThread` class

```
public void run() {
 while (shutDownLatch.getCount() > 0) {  #1
```

```
  try {
       if (updatedSymbols()) { #2

          connectorContext.requestTaskReconfiguration(); #3
       }
boolean isShutdown = shutDownLatch.await(monitorThreadCheckInter
                                     TimeUnit.MILLISECONDS);
if (isShutdown) { #5
     return;
}
 ....

 // Details left out for clarity
```

All that the `StockTickerSourceConnectorMonitorThread` does is check if the stock symbols have changed and if they have, request a configuration change for the connector task(s). After checking for changes, the `shutDownLatch` waits the designated time specified by the `monitorThreadCheckInterval` instance variable set when the connector created the thread. If the `shutDownLatch` counted down while waiting, then the `isShutdow` variable returns `true` and the monitoring thread stops running, otherwise it continues to monitor for changes.

Note

The `CountDownLatch` is a class from the `java.util.concurrent` package and is a synchronization tool that gives you the ability to have one or more threads wait until a certain condition is reached. I won’t go into any more details here but consult the Javadoc for a full explaination - https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurre

To conclude the discussion on the monitoring thread, let’s quickly review how it determines if there are any changes.

Listing 5.11. Logic used for detecting changes to symbols

```
//Details omitted for clarity

 List<String> maybeNewSymbols = symbols(); #1
 boolean foundNewSymbols = false;
 if (!Objects.equals(maybeNewSymbols, this.tickerSymbols)) { #2
```

```
     tickerSymbols = new ArrayList<>(maybeNewSymbols); #3
     foundNewSymbols = true;
}
return foundNewSymbols;
```

To determine that there are changes, the `symbols()` method reads a file and returns a `List<String>` representing the comma separated list of symbols contained in the file. If the retrieved list contents differs from the current one, we update the instance list and set the `foundNewSymbols` boolean to `true`, which will triggers a reconfiguration when the method returns.

So this wraps up our coverage of the custom `SourceConnector`, the code presented here isn't production ready, but it's meant to give you a good understanding of how you would implement your own connector. There are instructions in the book's source code that describe how to run this custom connector locally with a docker image.

The API returns a rich set of fields for the stock ticker symbol, but you may only interested in storing a subset of those fields into Kafka. Of course you could extract the fields of interest right at the source when the connector task retrieves the API results, but if you switched API services that had a different JSON structor it would mean you'd need to change the logic of your connector. A better approach would be to use a transform that could extract the exact fields you need before the records make it to Kafka. For that you'll need a custom transformation that can take artbitrary fields from a JSON object and return one with just those fields. That's exactly what we are going to cover in the next section, create a custom transformation.

5.4.3 Creatign a custom transformation

Although Kafka Connect provides serveral transformations out of the box, given the unique structure of data created by deployed production systems, the idea that those transforms will handle every case isn't always possible. While the need for a custom connector is less likely, due to the number of connectors availble for popular external systems, it's more likely that you'll need to write your own transformation.

The stock API feed we used for our custom Connector produces a result containing 70+ fields of different metrics for each stock symbol. While each one is useful, for our purposes we only want to keep a fraction of those, 5-6 at most. So you’re going to create a transformation that keep only the fields you specify via a configured comma separated list of field names.

To complete the transformation you’re going to implement the `Transformation` interface. There are a few methods on the interface you’ll need to implement, but we’re going to focus on one in particular the `apply` method as that’s where all the action of dropping the fields you’re not interested in happens.

But there’s a bit of a twist with implementing a `Tranformation` object. If you remember from earlier in the chapter, when you launch a connector, one of the configurations you set specifies if you want to include the schema for each record. So you’ll need to take that into account when implementing your custom SMT. We’ll see how this works in the upcoming section. Additionally, you’ll need to account for the fact that a user may want to apply this transformation to either the key or the value, and you’ll see how we handle that as well.

Let’s get into the implementation for the `Transformation` now:

Listing 5.12. Implementation of the custom transformation

```
// Details left out for clarity

public abstract class MultiFieldExtract<R extends ConnectRecord<R
                                       implements Transformation<

 @Override
public R apply(R connectRecord) {     #2
  if (operatingValue(connectRecord) == null) {  #3
    return connectRecord;
  } else if (operatingSchema(connectRecord) == null) { #4
    return applySchemaless(connectRecord);
  } else {
    return applyWithSchema(connectRecord); #5
  }
}
```

The first thing you’ll probably notice about our `Transformation` implementation is that it’s an `abstract` class. If you go back to the earlier part of this chapter you’ll remember when we configured SMTs you needed to specify that the transform was either for the key or the value with configuration like this `MaskField$Value` if you recall in Java the `$` indicates an inner class. So what’s happening is you declare the tranformation class as abstract since it will have by convention three abstract methods `operatingSchema`, `operatingValue`, and `newRecord`. You’ll implement these methods with two inner classes `Key` and `Value` which represent the transformation for the respective part of the connect record. We won’t go into any more details here as so we can continue moving forward and discuss the action happening in the `apply` method.

At annotation three, you’ll see the simplest case the underlying value for the connect record is null. Remember in Kafka it’s acceptable to have a null key or value. Keys are optional and with compacted topics a null value represents a tombstone and indicates to the log cleaner to delete the record from the topic. For the rest of this section I’m going to assume that we’re only working with values only.

Next at annotation four, we check if the record has an embedded schema. Again earlier in the chapter we discussed the `value.converter.schemas.enable` configuration which, if enabled, embeds the schema for each record coming through the connector. If this branch evaluates to `true` we’ll use the `applySchemaless` method to complete the tranformation:

Listing 5.13. Transforming without a schema

```
private R applySchemaless(R connectRecord) {
       final Map<String, Object> originalRecord =
               requireMap(operatingValue(connectRecord), PURPOSE
       final Map<String, Object> newRecord = new LinkedHashMap<>
       List<Map.Entry<String,Object>> filteredEntryList =
     originalRecord.entrySet().stream()
     .filter(entry -> fieldNamesToExtract.contains(entry.getKey(
     .toList();

       filteredEntryList.forEach(entry -> newRecord.put(entry.ge
                                                entry.getValue()
       return newRecord(connectRecord, null, newRecord);
```

```
    }
```

Since there’s no schema we can create a `Map` with `String` keys (for the field name) and `Obejct` for the values for the current record. Then you create an empty map for the new record and then filter the existing records by checking if the list of configured field names contains it. If is in the list the key and value are place in the map representing the new record.

If the record does have a schema, we follow a very similar process, but we first have to adjust the schema to only contian the fields we’re intrested in keeping:

Listing 5.14. Transforming with a schema

```
private R applyWithSchema(R connectRecord) {
final Struct value =
       requireStruct(operatingValue(connectRecord), PURPOSE);

Schema updatedSchema = schemaUpdateCache.get(value.schema());
if(updatedSchema == null) {
    updatedSchema = makeUpdatedSchema(value.schema());
    schemaUpdateCache.put(value.schema(), updatedSchema);
}
final Struct updatedValue = new Struct(updatedSchema);

updatedValue.schema().fields()
.forEach(field -> updatedValue.put(field.name(),
                                  value.get(field.name())));
return newRecord(connectRecord, updatedSchema, updatedValue);
}
```

Here with the schema-record combo we first need to get a `Struct` which is very similar to the `HashMap` in the schemaless version, but it contains all the type information for the fields in the record. First we check if we’ve already created an updated schema, if not, we create it and store it in a cache. Once we’ve created the updated schema there’s no need to create another one since the structure for all records will be the same. Then we simply iterate over the field names of our updated schema using each one to extract the value from the old record.

You now know how to implement your own custom connect

`Transformation`. I’ve not covered all the details here, so be sure to consult the source code in `bbejeck.chapter_5.transformer.MultiFieldExtract` for all the details.

I’d like close this chapter on Kafka Connect out by saying the custom `Connector` and `Transformation` you’ve created here aren’t necesarily meant for production use, they are simply meant to be teaching tools on ***how*** you can go about creating custom variations of those two classes when the ones provided by Kafka Connect won’t fit your needs.

5.5 Summary

- Kafka Connect is the lynch pin that will move data from an external system into Kafka and it can move data from Kafka out into an external system as well. This ability move data in and out of Kafka is critical for getting existing applications outside of Kafka involved in an event streaming platform.
- Using an existing connector doesn’t require any code, you just need to upload some JSON configurations to get it up and running. There are literally hundreds of existing connectors, so chances are there’s one already developed that can you use out of the box.
- The connect framework also provides something called a single message transform or SMT that can apply lightweight changes to incoming or outgoing records.
- If the exisiting connectors or transformations don’t do what you need them to do, you can implement your own.

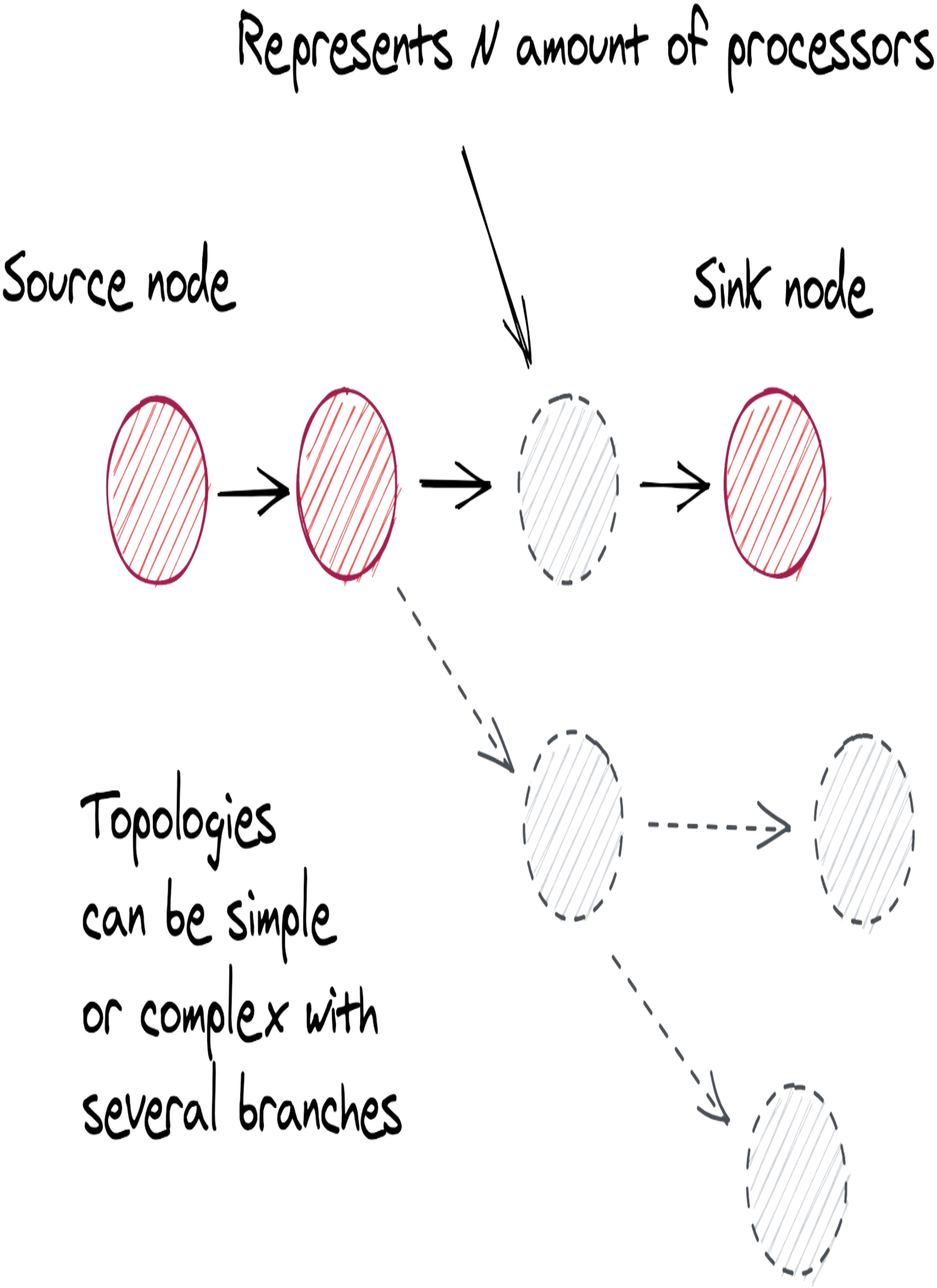
6 Developing Kafka Streams

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


Represents N amount of processors
Source node
Sink node
Topologies
can be simple
or complex with
several branches

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.

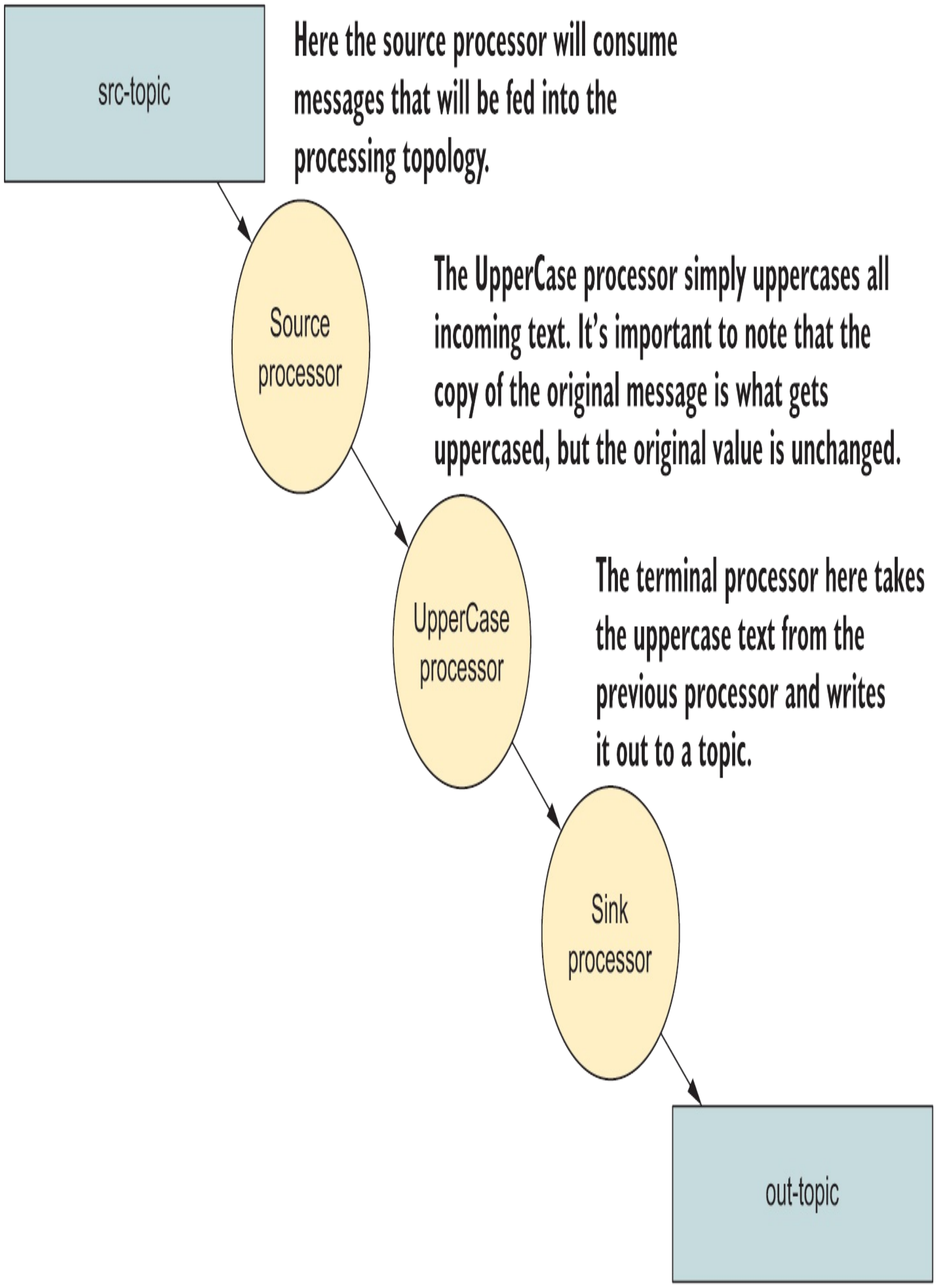
There’s also a lower-level API, the Processor API, which isn’t as succinct as the Kafka Streams DSL but allows for more control. We’ll cover the Processor API in a later chapter. With that introduction out of the way, let’s dive into the requisite Hello World program for Kafka Streams.

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



Here the source processor will consume messages that will be fed into the processing topology.

The UpperCase processor simply uppercases all incoming text. It’s important to note that the copy of the original message is what gets uppercased, but the original value is unchanged.

The terminal processor here takes the uppercase text from the previous processor and writes it out to a topic.

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:

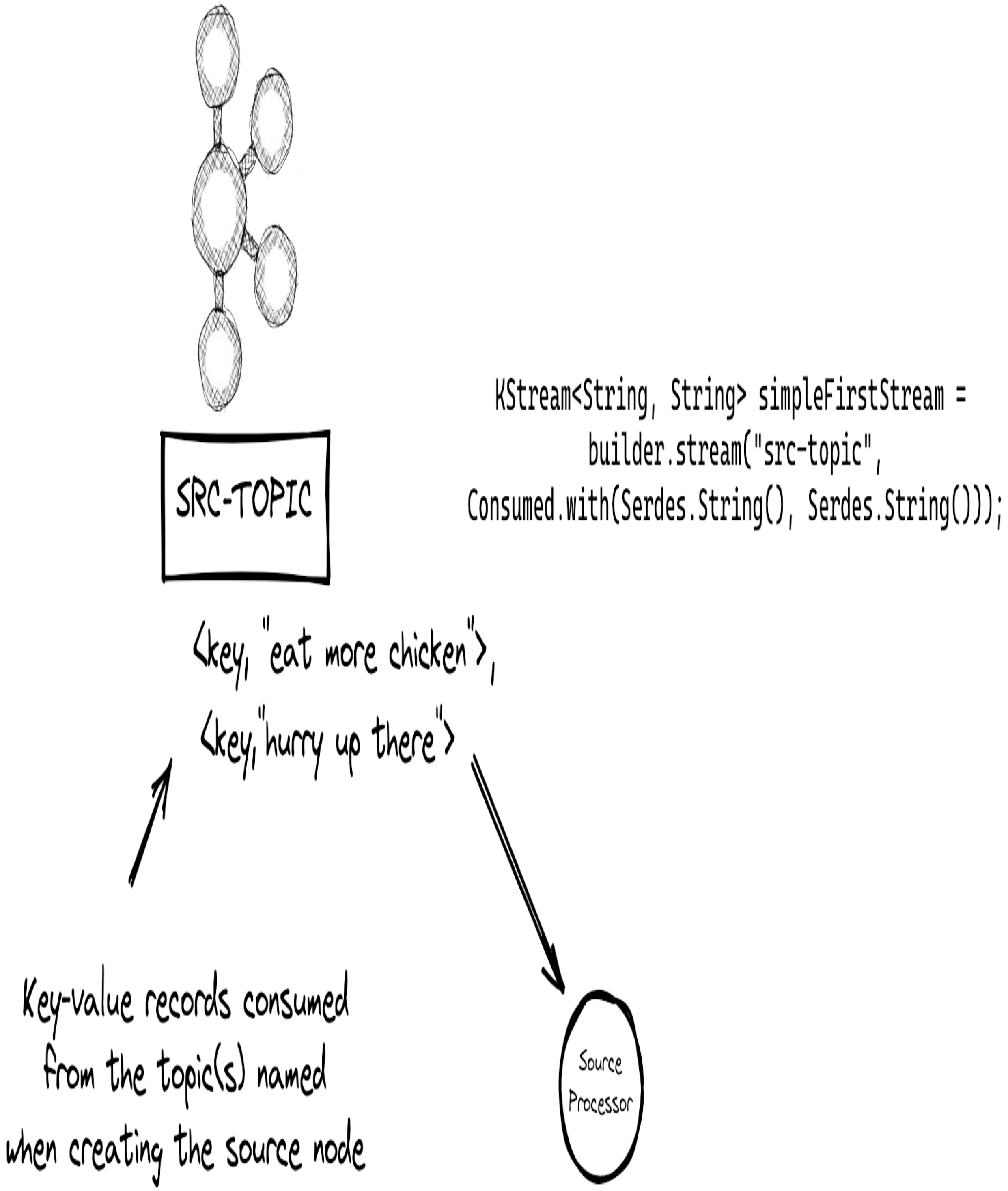
1. Define the configuration items.
2. Create `Serde` instances, either custom or predefined, used in deserialization/serializtion of records.
3. Build the processor topology.
4. Create and start the `Kafka Streams`.

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


SRC-TOPIC
<key, "eat more chicken">,
<key, "hurry up there">
Key-value records consumed from the topic(s) named when creating the source node
Source Processor

```
KStream<String, String> simpleFirstStream =
builder.stream("src-topic",
Consumed.with(Serdes.String(), Serdes.String()));
```

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-t
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

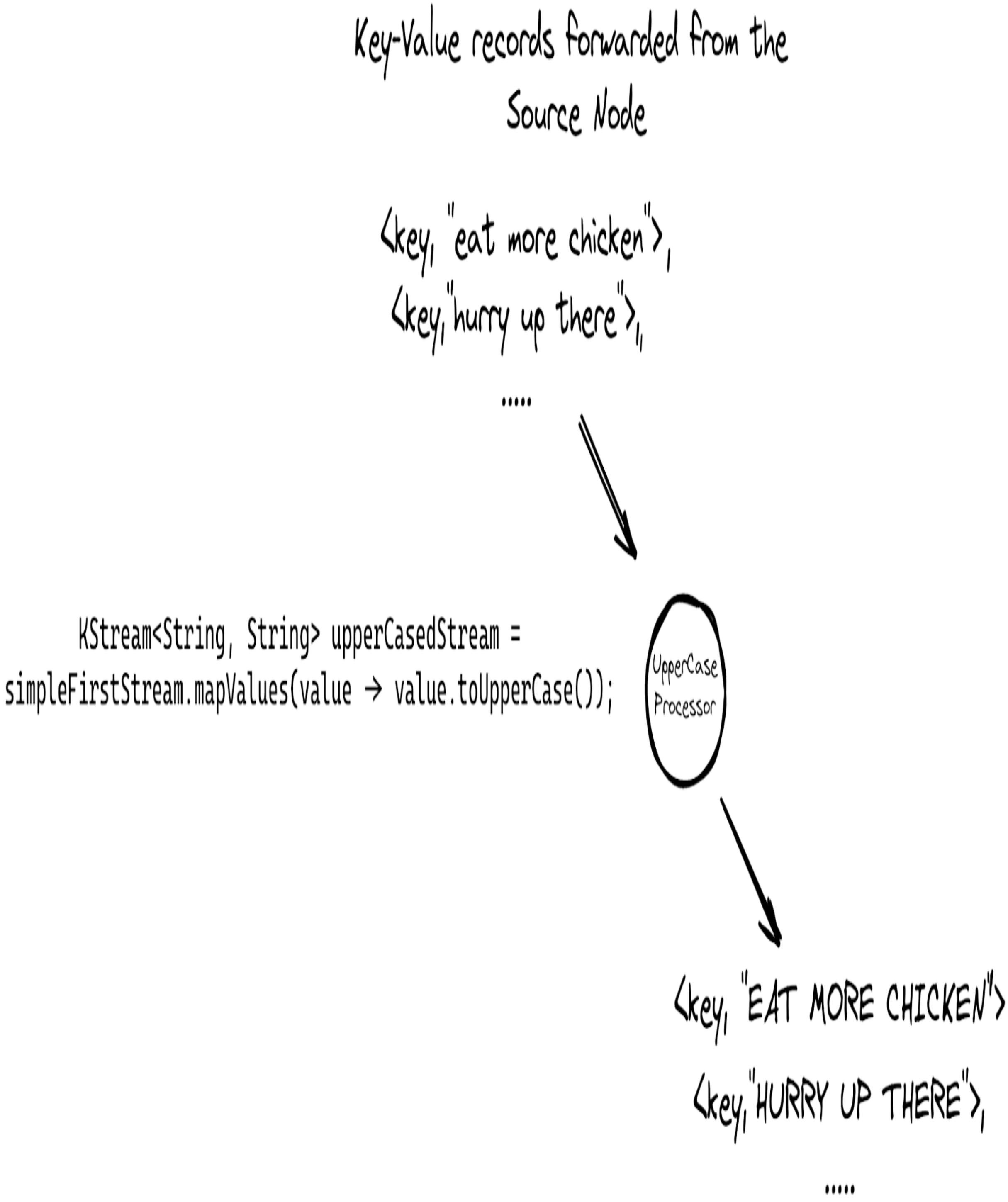
```
KStream<String, String> simpleFirstStream =
   buider.source(Pattern.compile("topic[A-C]"),
      Consumed.with(Serdes.String(), Serdes.String()))
```

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


Key-Value records forwarded from the Source Node
<key, "eat more chicken">,
<key,"hurry up there">,,
.....
KStream<String, String> upperCasedStream =
simpleFirstStream.mapValues(value -> value.toUpperCase());
UpperCase Processor
<key, "EAT MORE CHICKEN">
<key,"HURRY UP THERE">,
.....

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 forwarded along.

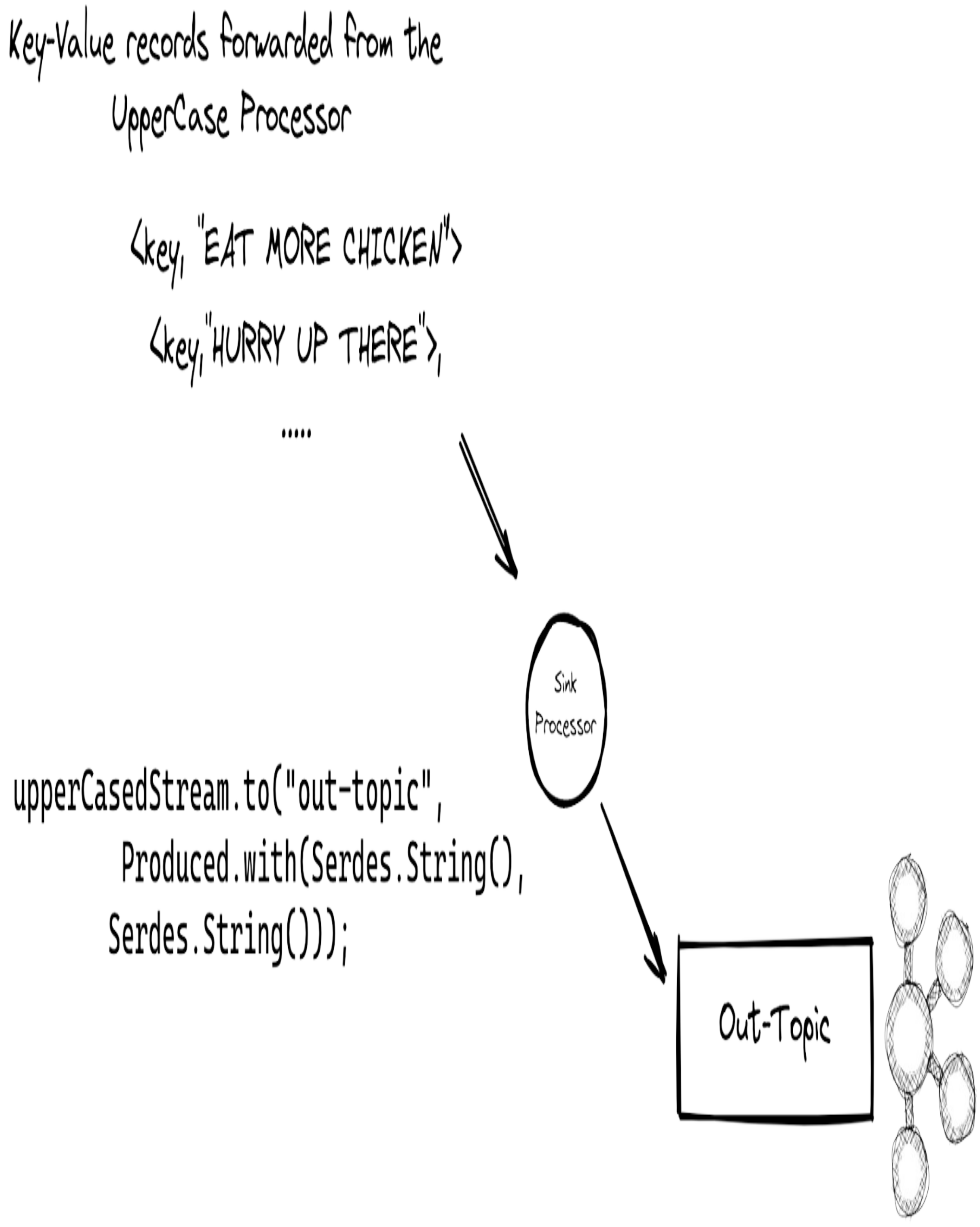
The `mapValues()` method takes an instance of the `ValueMapper<V, V1>` interface. The `ValueMapper` interface defines only one method, `ValueMapper.apply`, making it an ideal candidate for using a lambda expression, which is exactly what you’ve done here with `value → value.toUpperCase()`.

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


Key-Value records forwarded from the UpperCase Processor
<key, "EAT MORE CHICKEN">
<key,"HURRY UP THERE">,
.....
Sink Processor
Out-Topic

```
upperCasedStream.to("out-topic",
          Produced.with(Serdes.String(),
         Serdes.String()));
```

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

KStream<String, String> upperCasedStream =
simpleFirstStream.mapValues(value -> value.toUpperCase());
upperCasedStream.to("out-topic", Produced.with(Serdes.String(), S
```

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
 .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 BaseStreamsApplicatio

private static final Logger LOG =
  LoggerFactory.getLogger(KafkaStreamsYellingApp.class);
```

```
@Override
public Topology topology(Properties streamProperties) {

  Serde<String> stringSerde = Serdes.String(); #1
  StreamsBuilder builder = new StreamsBuilder(); #2

  KStream<String, String> simpleFirstStream = builder.stream("src
            Consumed.with(stringSerde, stringSerde)); #3
  KStream<String, String> upperCasedStream =
  simpleFirstStream.mapValues(value)-> value.toUpperCase()); #4

  upperCasedStream.to("out-topic",
      Produced.with(stringSerde, stringSerde)); #5

  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 KafkaStreamsYellingAp
    Topology topology = yellingApp.topology(streamProperties);

    try(KafkaStreams kafkaStreams =
                    new KafkaStreams(topology, streamProperties))
        LOG.info("Hello World Yelling App Started");
        kafkaStreams.start(); #6

        LOG.info("Shutting down the Yelling APP now");
    }
 }
}
```

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.

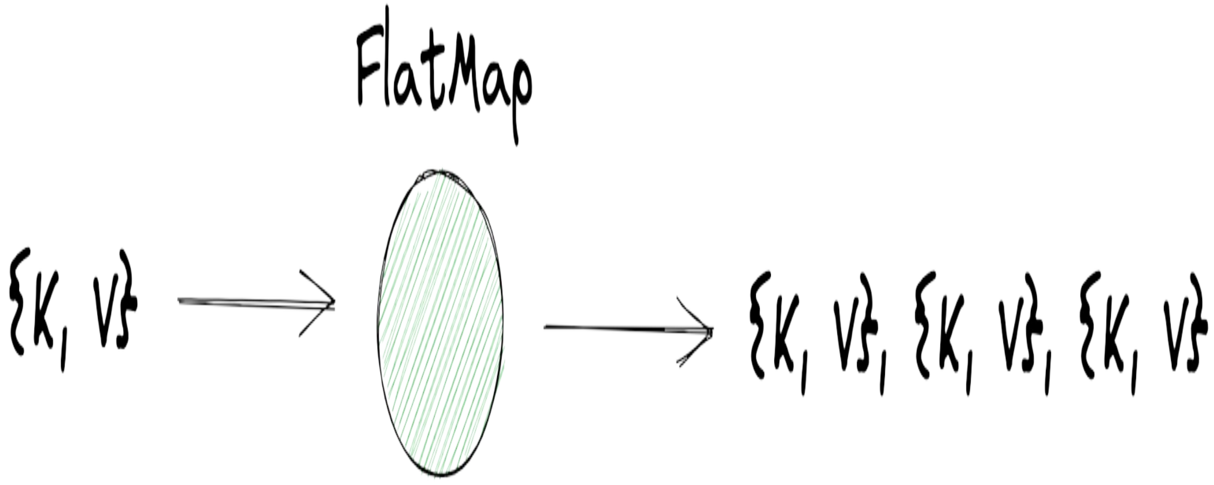
6.3.2 Adding the patterns processor

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>>> retailTransactionToPu
    (key, value) -> {
      String zipcode = value.getZipCode(); #1
      return value.getPurchasedItemsList().stream() #2
                .map(purchasedItem ->
                    KeyValue.pair(zipcode, purchasedItem))
                .collect(Collectors.toList());
}
```

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) #1
                .mapValues(patternObjectMapper);  #2

patternKStream.print(Printed.<String,Pattern>toSysOut()
              .withLabel("patterns")); #3
patternKStream.to("patterns",
        Produced.with(stringSerde,purchasePatternSerde)); #4
```

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> reward
       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> retailPurchas
        streamsBuilder.stream("transactions",
            Consumed.with(stringSerde, retailPurchaseSerde))
        .mapValues(creditCardMapper);  #1

KStream<String, PatternProto.Pattern> patternKStream =
   retailPurchaseKStream
        .flatMap(retailTransactionToPurchases)
        .mapValues(patternObjectMapper);  #2

patternKStream.to("patterns",
      Produced.with(stringSerde,purchasePatternSerde));

KStream<String, RewardAccumulatorProto.RewardAccumulator> rewards
      retailPurchaseKStream.mapValues(rewardObjectMapper); #3

rewardsKStream.to("rewards",
      Produced.with(stringSerde,rewardAccumulatorSerde));
retailPurchaseKStream.to("purchases",
      Produced.with(stringSerde,retailPurchaseSerde));

return streamsBuilder.build(streamProperties);

}
```

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> purchaseDeserial
      new ProtoDeserializer<>(); #1
Serializer<RetailPurchaseProto.RetailPurchase> purchaseSerializer
      new ProtoDeserializer<>(); #2
Map<String, Class<RetailPurchaseProto.RetailPurchase>> configs
            = new HashMap<>();
   configs.put(false, RetailPurchaseProto.RetailPurchase.class);
          deserializer.configure(configs,isKey);  #3
Serde<RetailPurchaseProto.RetailPurchase> purchaseSerde =
     Serdes.serdeFrom(purchaseSerializer,purchaseDeserializer); #
```

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); #1
String url = "https://...";                                 #2
Map<String, Object> configMap = new HashMap<>();
configMap.put(
  AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  url);                                             #3
protobufSerde.configure(configMap, false);   #4
```

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)      #1
  .mapValues(value -> value.toUpperCase())
  .peek(sysout)      #2
  .to( "out-topic",
  Produced.with(stringSerde, stringSerde));
```

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(...);
```

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.j

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> reward
    retailPurchaseKStream #1
    .mapValues(rewardObjectMapper) #2
    .filter((key,potentialReward) ->
            potentialReward.getPurchaseTotal() > 10.00); #3
```

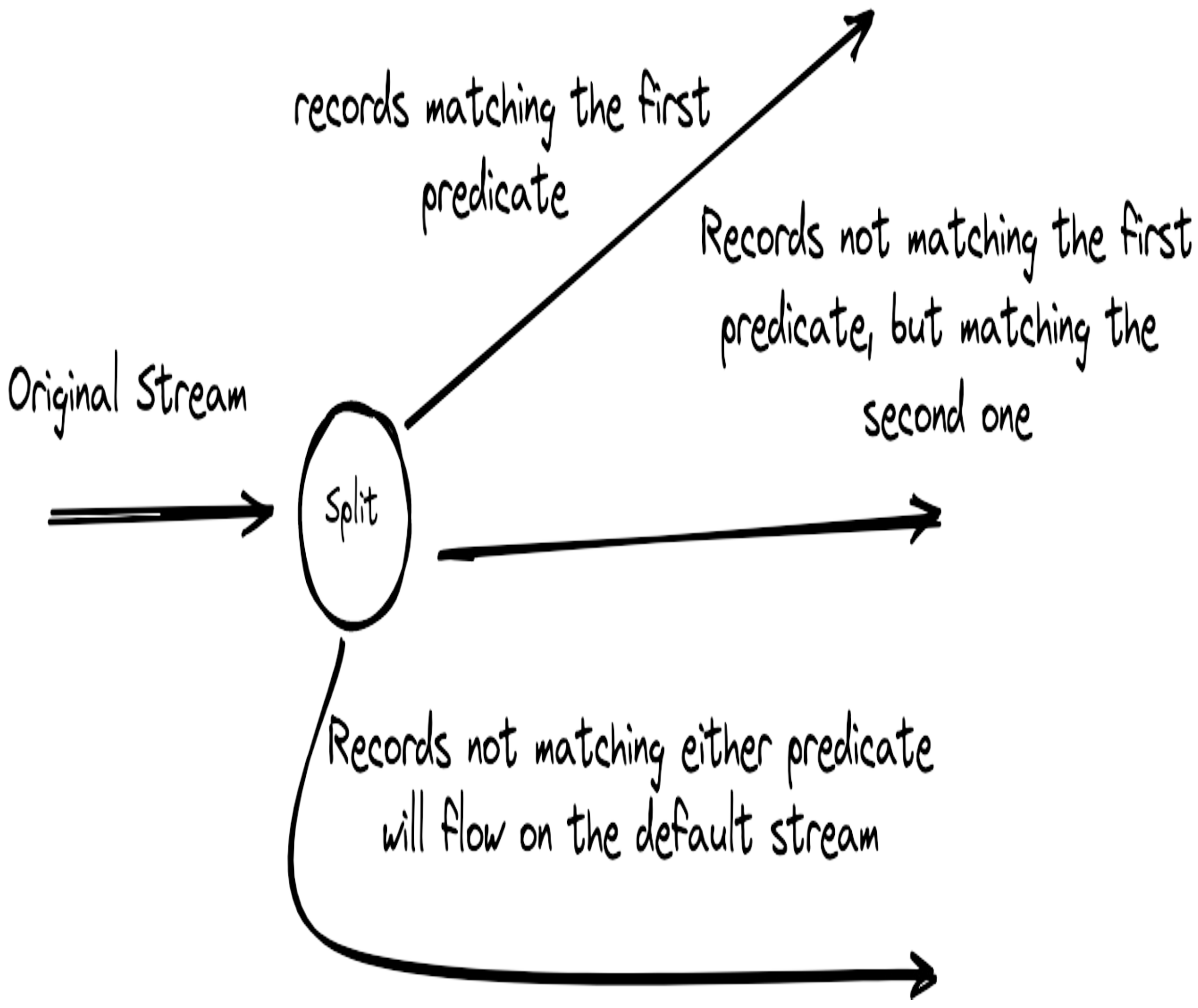
You have now successfully updated the rewards stream to drop purchases that don’t qualify for reward points.

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


records matching the first predicate
Records not matching the first predicate, but matching the second one
Original Stream
Split
Records not matching either predicate will flow on the default stream

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"); #1

Predicate<String, Purchase> isElectronics =
  (key, purchase) ->
  purchase.getDepartment().equalsIgnoreCase("electronics"); #1

purchaseKStream.split())  #2
.branch(isCoffee,
 Branched.withConsumer(coffeeStream -> coffeeStream.to("coffee-to
.branch(isElectronics,
  Branched.withConsumer(electronicStream ->
  electronicStream.to("electronics"))  #4
.defaultBranch(Branched.withConsumer(retailStream ->
                retailStream.to("purchases")); #5
```

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-")) #1
        .branch(isTemperatureSensor, Branched.as("temperature"))
        .branch(isProximitySensor,
            Branched.withFunction(
                ps -> ps.mapValues(feetToMetersMapper), "proximit
        .noDefaultBranch(); #4
```

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 =  #1
      builder.stream("temperature-sensors", sensorConsumed);

KStream<String, SensorProto.Sensor> proximitySensorStream =  #2
      builder.stream("proximity-sensors", sensorConsumed);

temperatureSensorStream.merge(sensorMap.get("sensor-temperature")
      .to("temp-reading", Produced.with(stringSerde, sensorSerde)

proximitySensorStream.merge(sensorMap.get("sensor-proximity"))
     .to("proximity-reading", Produced.with(stringSerde, sensorSe
```

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])  #1
      --> KSTREAM-MAPVALUES-0000000001  #2
    Processor: KSTREAM-MAPVALUES-0000000001 (stores: []) #3
      --> KSTREAM-SINK-0000000002
      <-- KSTREAM-SOURCE-0000000000  #4
    Sink: KSTREAM-SINK-0000000002 (topic: out-topic) #5
      <-- KSTREAM-MAPVALUES-0000000001
```

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")) #1
        .mapValues((key, value) -> value.toUpperCase(),
                  Named.as("Convert to Yelling")) #2
        .to("out-topic",
            Produced.with(stringSerde, stringSerde)
                    .withName("Application Output")) #3
```

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")) { #1
        return department;
    } else {
        return "purchases";             #2
    }
}
```

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(); #1
       if (headers != null) {
        Iterator<Header> routingHeaderIterator =
          headers.headers("routing").iterator();

        if (routingHeaderIterator.hasNext()) {
             Header routing = routingHeaderIterator.next(); #2

             return new String(routing.value(),
                                StandardCharsets.UTF_8); #3
        }
      }
       return defaultTopicName; #4
    }
```

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`!

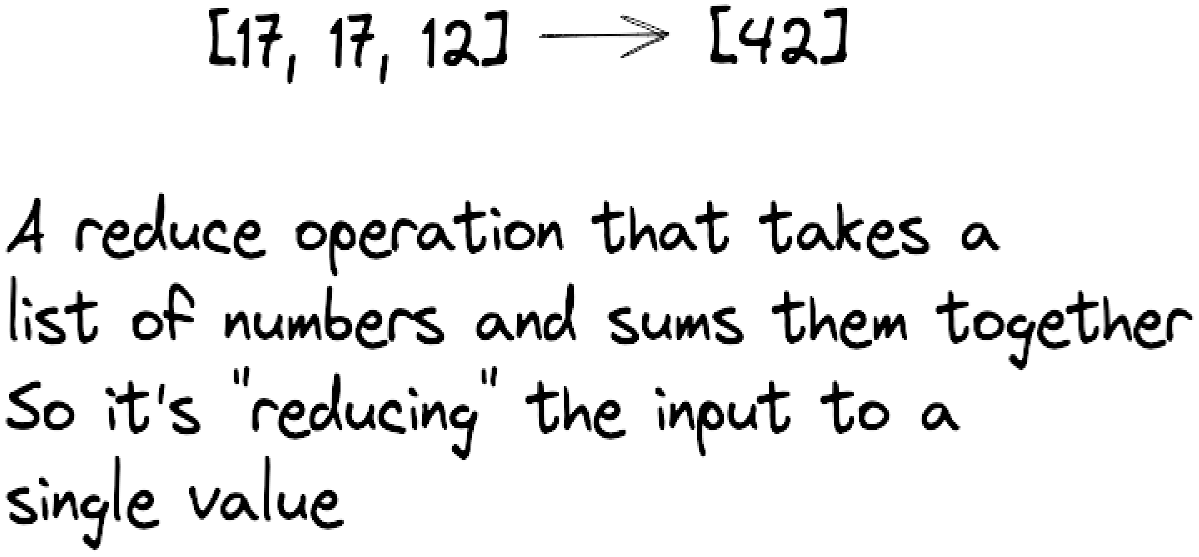
7.2 Adding stateful operations to Kafka Streams

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

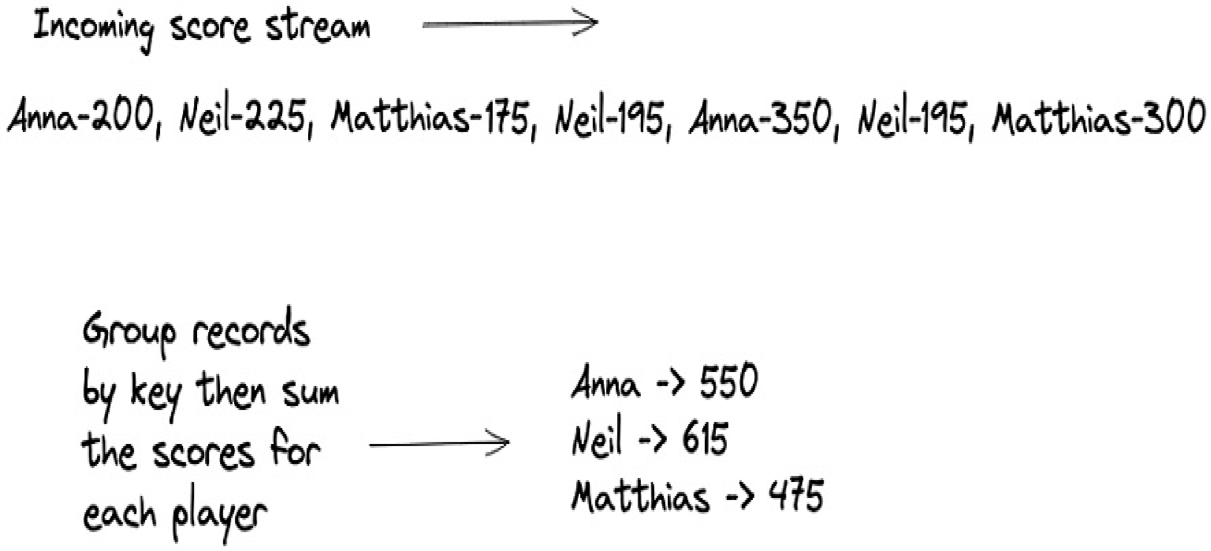

[17, 17, 12] → [42]
A reduce operation that takes a list of numbers and sums them together
So it's "reducing" the input to a single value

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


Incoming score stream
Anna-200, Neil-225, Matthias-175, Neil-195, Anna-350, Neil-195, Matthias-300
Group records by key then sum the scores for each player
Anna -> 550
Neil -> 615
Matthias -> 475

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-
        Consumed.with(Serdes.String(), Serdes.Double()));

pokerScoreStream
        .groupByKey() #1
        .reduce(Double::sum,  #2
                Materialized.with(Serdes.String(), Serdes.Double(
        .toStream()  #3
        .to("total-scores",
                Produced.with(Serdes.String(), Serdes.Double()));
```

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.

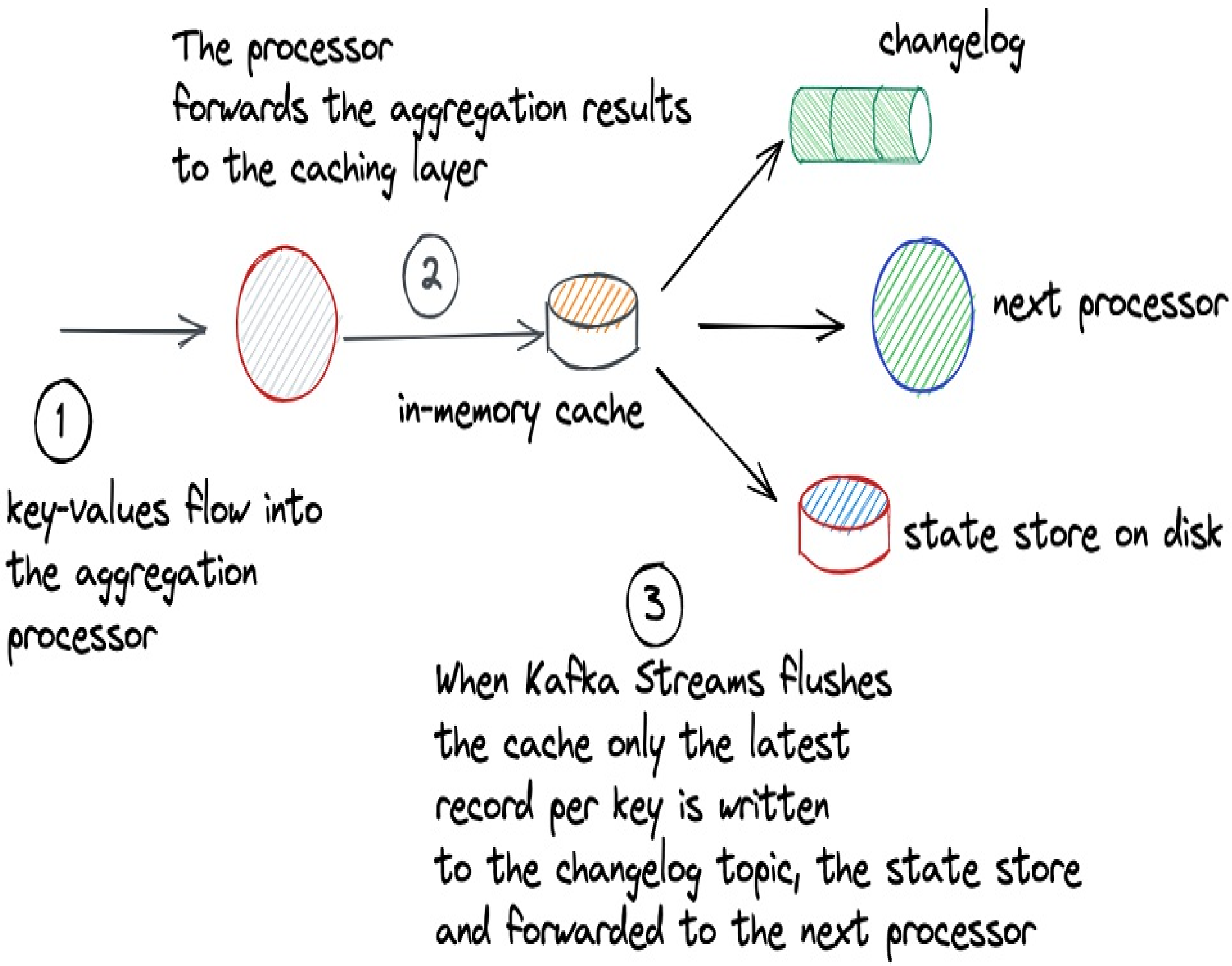
The `Materialized` object provides the serdes used by the state store for (de)serializing keys and values. Under the covers, Kafka Streams uses local storage to support stateful operations. The stores store key-value pairs as byte arrays, so you need to provide the serdes to serialize records on input 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


The processor forwards the aggregation results to the caching layer
changelog
2
next processor
1
in-memory cache
key-values flow into the aggregation processor
state store on disk
3
When Kafka Streams flushes the cache only the latest record per key is written to the changelog topic, the state store and forwarded to the next processor

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) { #1

   Aggregate.Builder currAggregate =
                                aggregate.toBuilder(); #2

    double transactionDollars =
           transaction.getNumberShares()
         * transaction.getSharePrice(); #3

    if (transaction.getIsPurchase()) {       #4
        long currentPurchaseVolume =
             currAggregate.getPurchaseShareVolume();
        currAggregate.setPurchaseShareVolume(
                      currentPurchaseVolume
                     + transaction.getNumberShares());

        double currentPurchaseDollars =
                currAggregate.getPurchaseDollarAmount();

        currAggregate.setPurchaseDollarAmount(
                      currentPurchaseDollars
                      + transactionDollars);
    }
    //Further details left out for clarity
```

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)); #1


transactionKStream.groupBy((key, value) -> value.getSymbol(), #2
      Grouped.with(Serdes.String(), txnSerde))
  .aggregate(() -> initialAggregate, #3
            new StockAggregator(),
            Materialized.with(stringSerde, aggregateSerde))
  .toStream() #4
  .peek((key, value) -> LOG.info("Aggregation result {}", value))
  .to("stock-aggregations", Produced.with(stringSerde, aggregateS
```

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, #1
            new StockAggregator(),
            Materialized.with(stringSerde, aggregateSerde))
  .toStream() #2
  .peek((key, value) -> LOG.info("Aggregation result {}", value))
  .to("stock-aggregations", Produced.with(stringSerde, aggregateS
```

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.

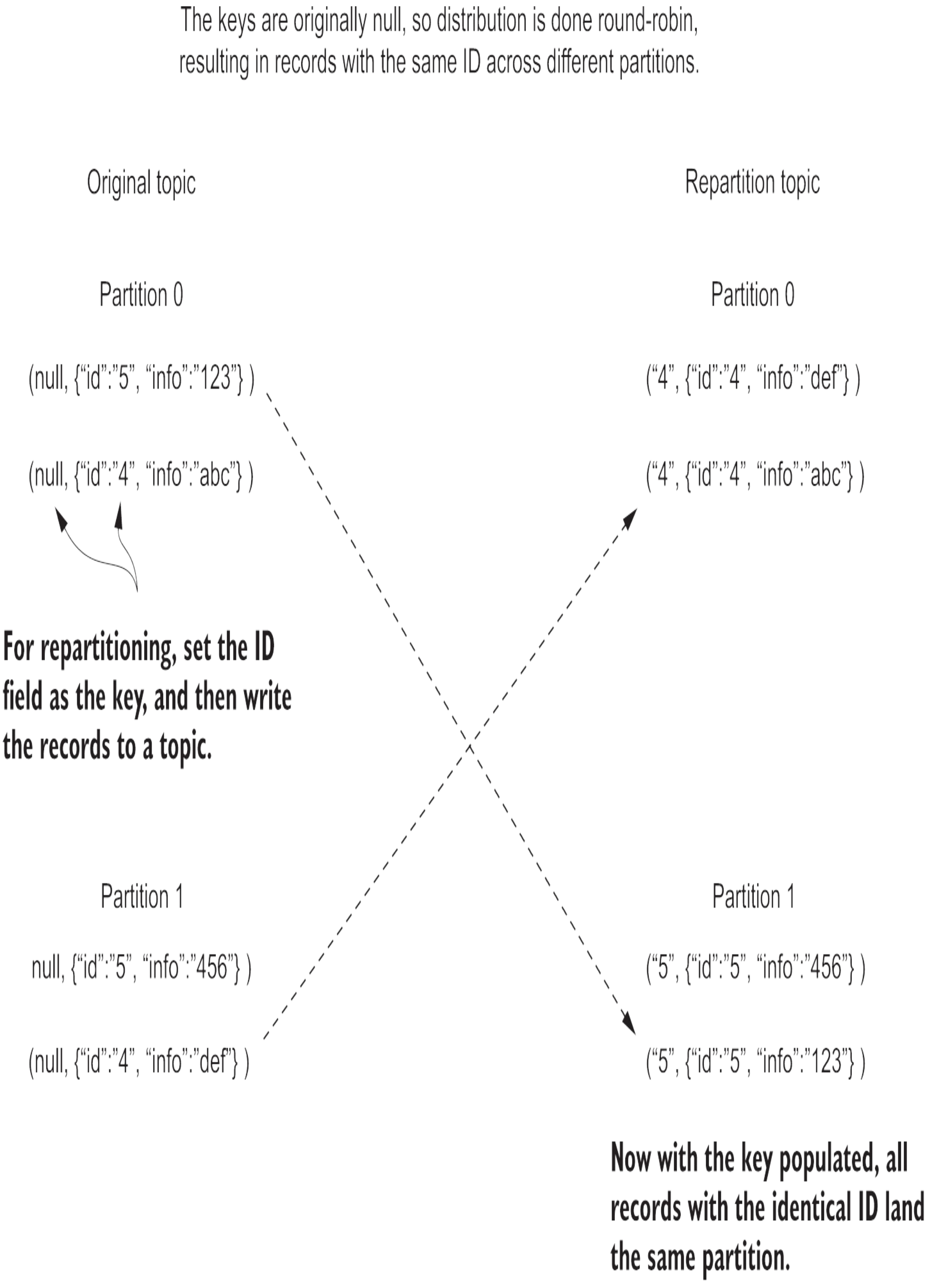
In this previous example here where we built stock transaction aggregates, I mentioned that changing the key for an aggregation requires a repartitioning of the data, let’s discuss this in a little more detail in the next section.

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



Original topic
Repartition topic
Partition 0
(null, {“id”:”5”, “info”:”123”})
(null, {“id”:”4”, “info”:”abc”})
Partition 1
null, {“id”:”5”, “info”:”456”})
(null, {“id”:”4”, “info”:”def”})
Partition 0
(“4”, {“id”:”4”, “info”:”def”})
(“4”, {“id”:”4”, “info”:”abc”})
Partition 1
(“5”, {“id”:”5”, “info”:”456”})
(“5”, {“id”:”5”, “info”:”123”})

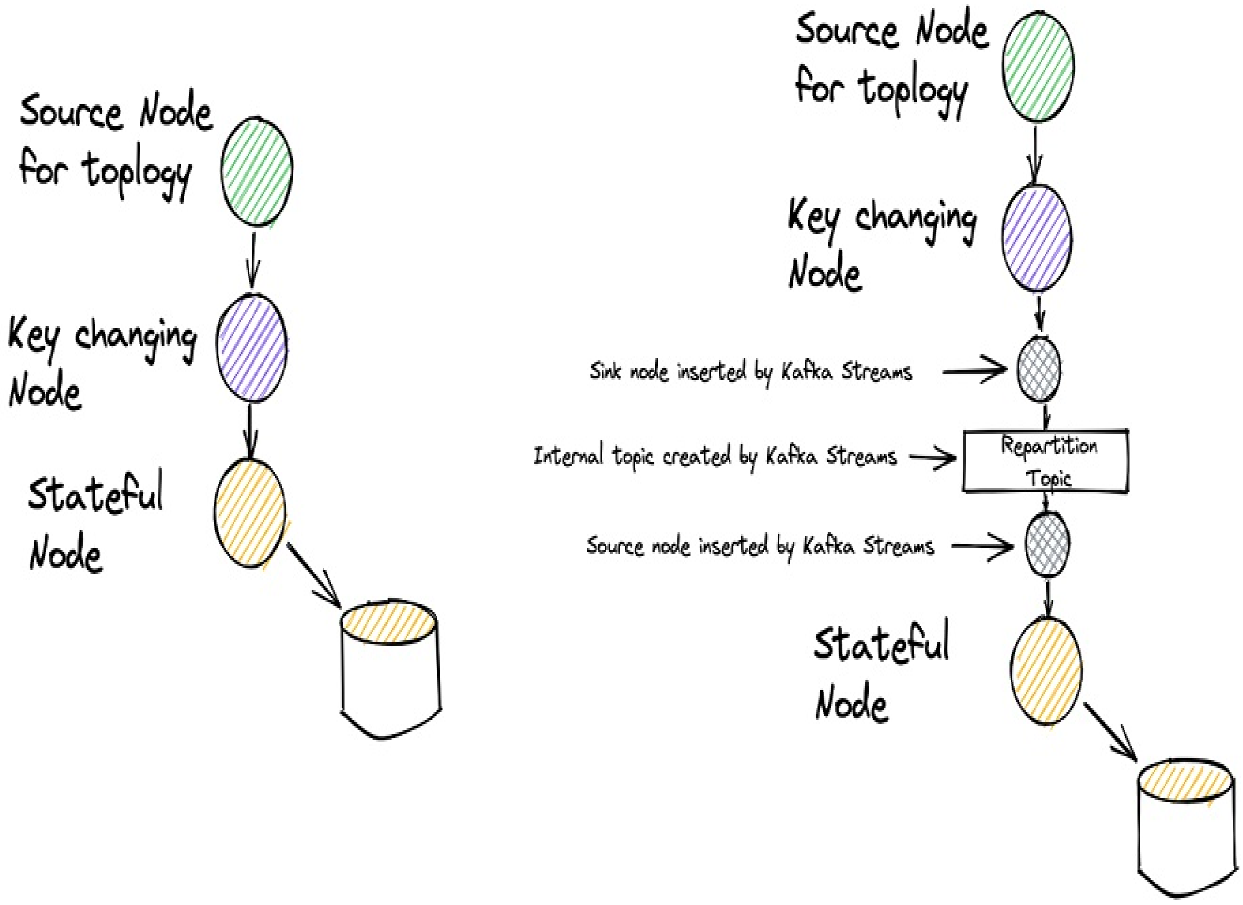
The keys are originally null, so distribution is done round-robin, resulting in records with the same ID across different partitions.

For repartitioning, set the ID field as the key, and then write the records to a topic.

Now with the key populated, all records with the identical ID land the same partition.

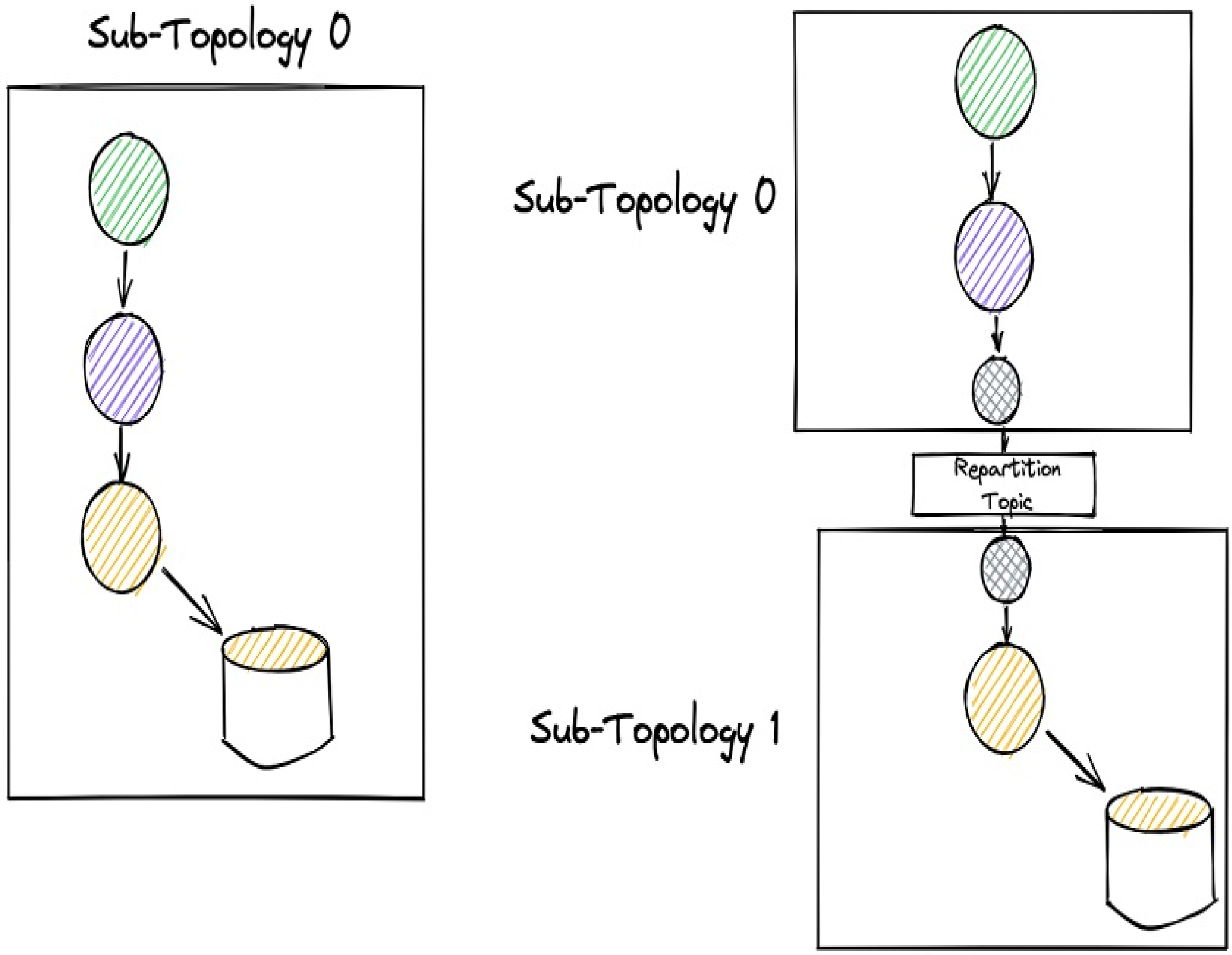
As you can see here, repartitioning is nothing more than producing records out to a topic and then immediately consuming them again. When the Kafka Streams embedded producer writes the records to the broker, it uses the updated key to select the new partition. Under the covers, Kafka Streams inserts 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(...)... #1
myStream.map(...).groupByKey().reduce(...)... #2

filteredStream = myStream.selectKey(...).filter(...); #3
....
filteredStreaam.groupByKey().aggregate(...)... #3
```

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(...); #1

myStream.selectKey(...).filter(...).to(...); #2
```

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:

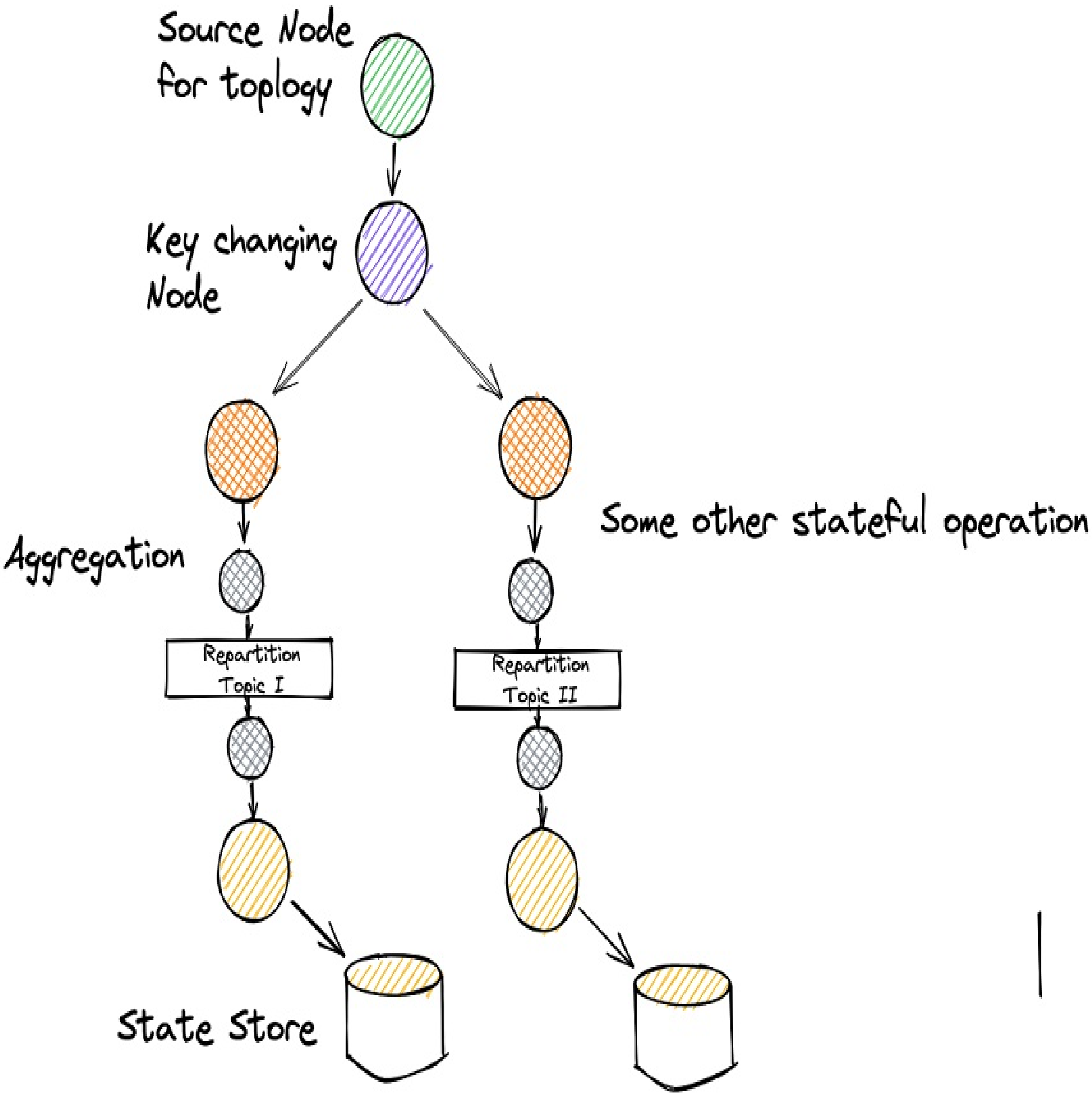
Listing 7.9. Changing the key then aggregate and join

```
// Several details omitted for clarity

KStream<String, String> originalStreamOne = builder.stream(...);

KStream<String, String> inputStreamOne = originalStreamOne.select

KStream<String, String> inputStreamTwo = builder.stream(...); #2

inputStreamOne.groupByKey().count().toStream().to(...); #3

KStream<String, String> joinedStream =       #4
  inputStreamTwo.join(inputStreamOne,
                (v1, v2)-> v1+":"+v2,
                JoinWindows.ofTimeDifferenceWithNoGrace(...),
                StreamJoined.with(...);

....
```

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

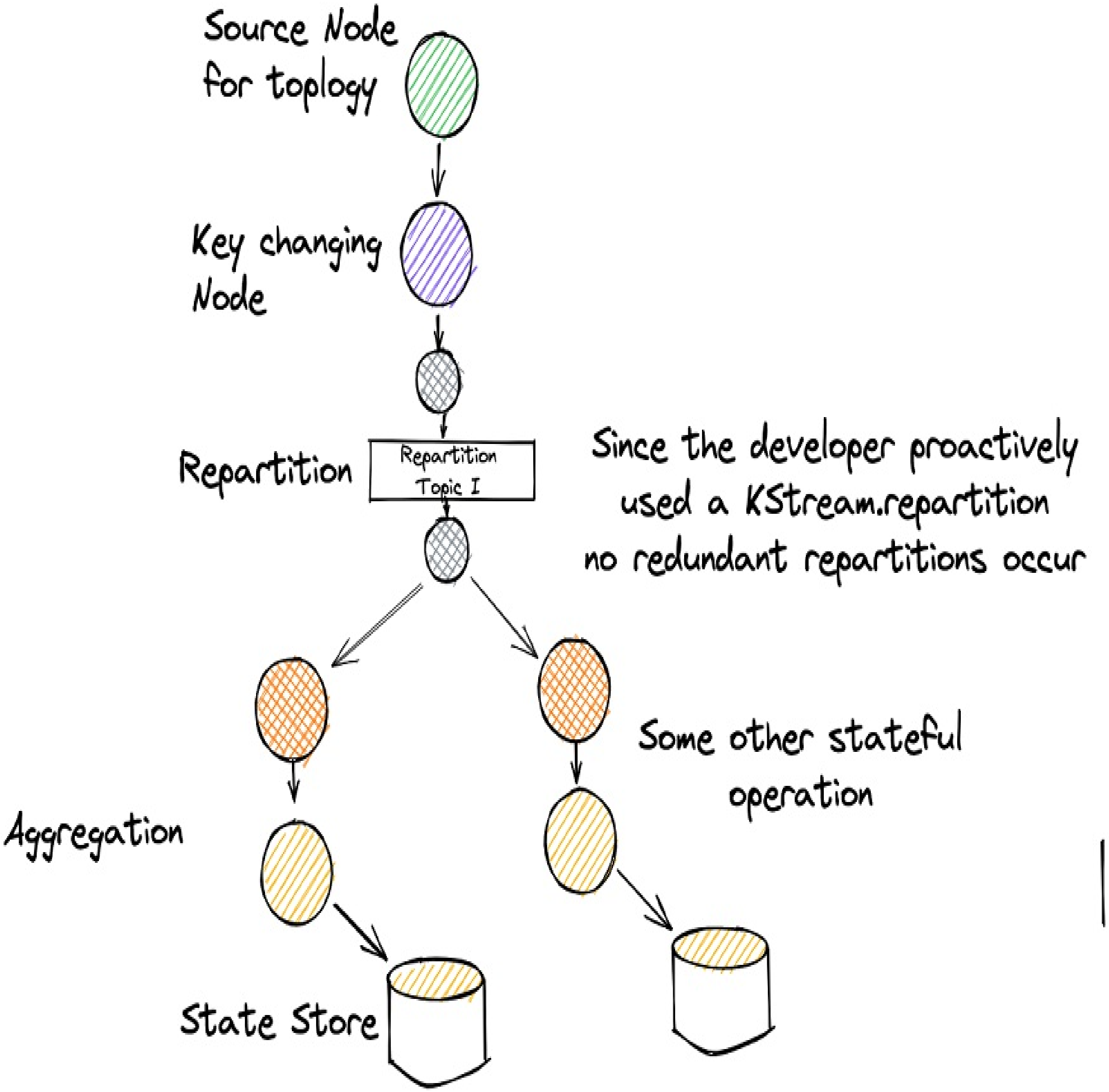
For the times when you might need to repartition the data yourself, the KStream API provides the `repartition` method. Here's how you use it to manually repartition after a key change:

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.select

KStream<String, String> inputStreamTwo = builder.stream(...);

KStream<String, String> repartitioned =
  inputStreamOne.repartition(Repartitioned   #2
                .with(stringSerde, stringSerde)
                .withName("proactive-repartition"));

repartitioned.groupByKey().count().toStream().to(...); #3

KStream<String, String> joinedStream = inputStreamTwo.join(...) #

.....
```

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


Source Node for toplogy
Key changing Node
Repartition
Repartition Topic I
Since the developer proactively used a KStream.repartition no redundant repartitions occur
Some other stateful operation
Aggregation
State Store

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.

ℹ **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)); #1
```

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.

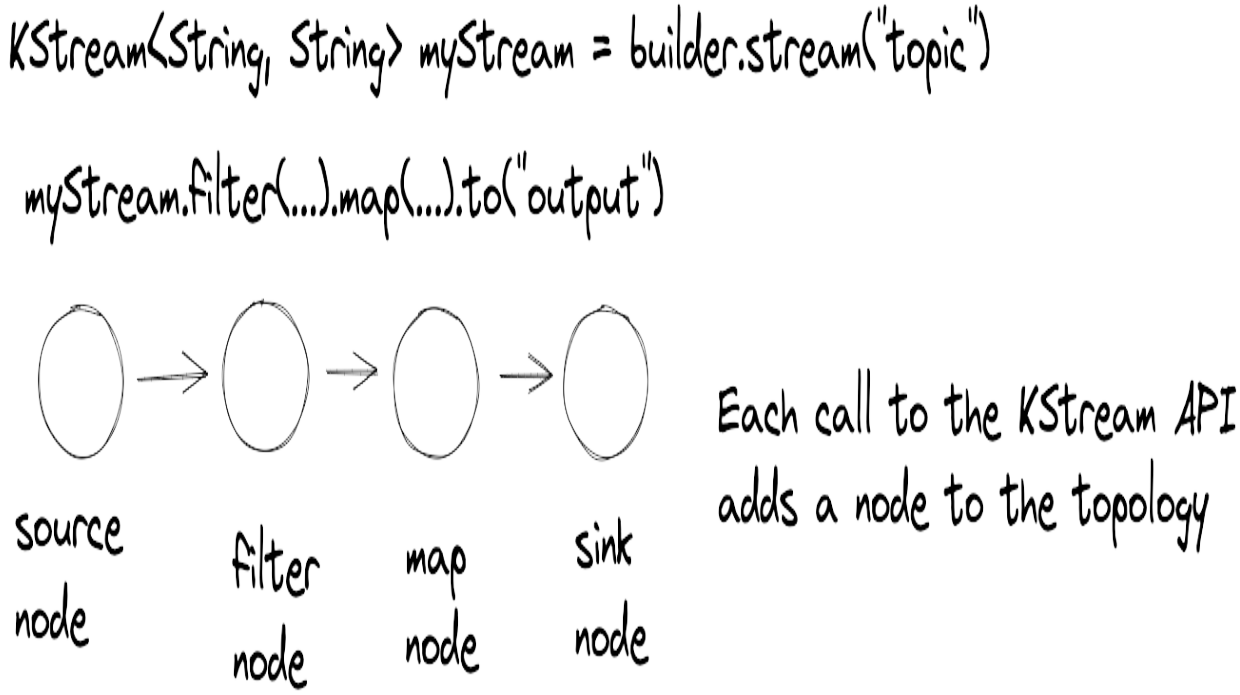
So far we've covered how you can proactively repartition when you've changed the key. But this requires you to know when to repartition and always remember to do so, but is there a better approach, maybe have Kafka Streams take care of this for you automatically? Well there is a way, by enabling optimizations.

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


KStream<String, String> myStream = builder.stream("topic")
myStream.filter(...).map(...).to("output")
source node
filter node
map node
sink node
Each call to the KStream API adds a node to the topology

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);   #1
builder.build(streamProperties);  #2
```

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.

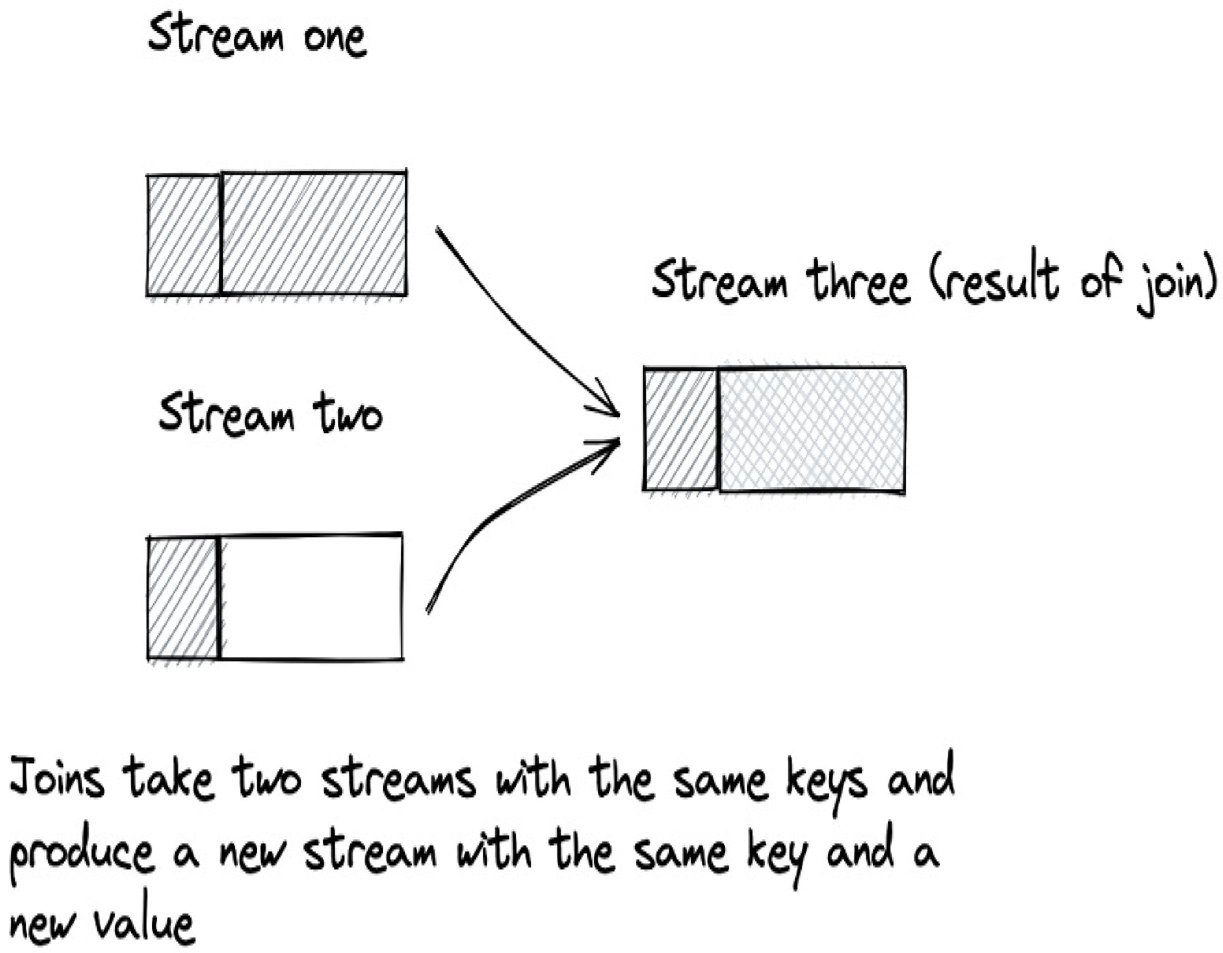
Now we’ve covered repartitioning, let’s move on to our next stateful operation, joins

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


Stream one
Stream two
Stream three (result of join)
Joins take two streams with the same keys and produce a new stream with the same key and a new value

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(...)

KStream<String, RetailPurchase>
                     retailPurchaseKStream = builder.stream(...)

ValueJoiner<CoffeePurchase,
            RetailPurchase,
            Promotion> purchaseJoiner =
                                          new PromotionJoiner();

JoinWindows thirtyMinuteWindow =
     JoinWindows.ofTimeDifferenceWithNoGrace(Duration.minutes(30)

KStream<String, Promotion> joinedKStream =
    coffeePurchaseKStream.join(retailPurchaseKStream, #4
                               purchaseJoiner,
                               thirtyMinuteWindow,
                                StreamJoined.with(stringSerde, #5
                                                  coffeeSerde,
                                                  storeSerde)
                                      .withName("purchase-join")
                                      .withStoreName("join-stores
```

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

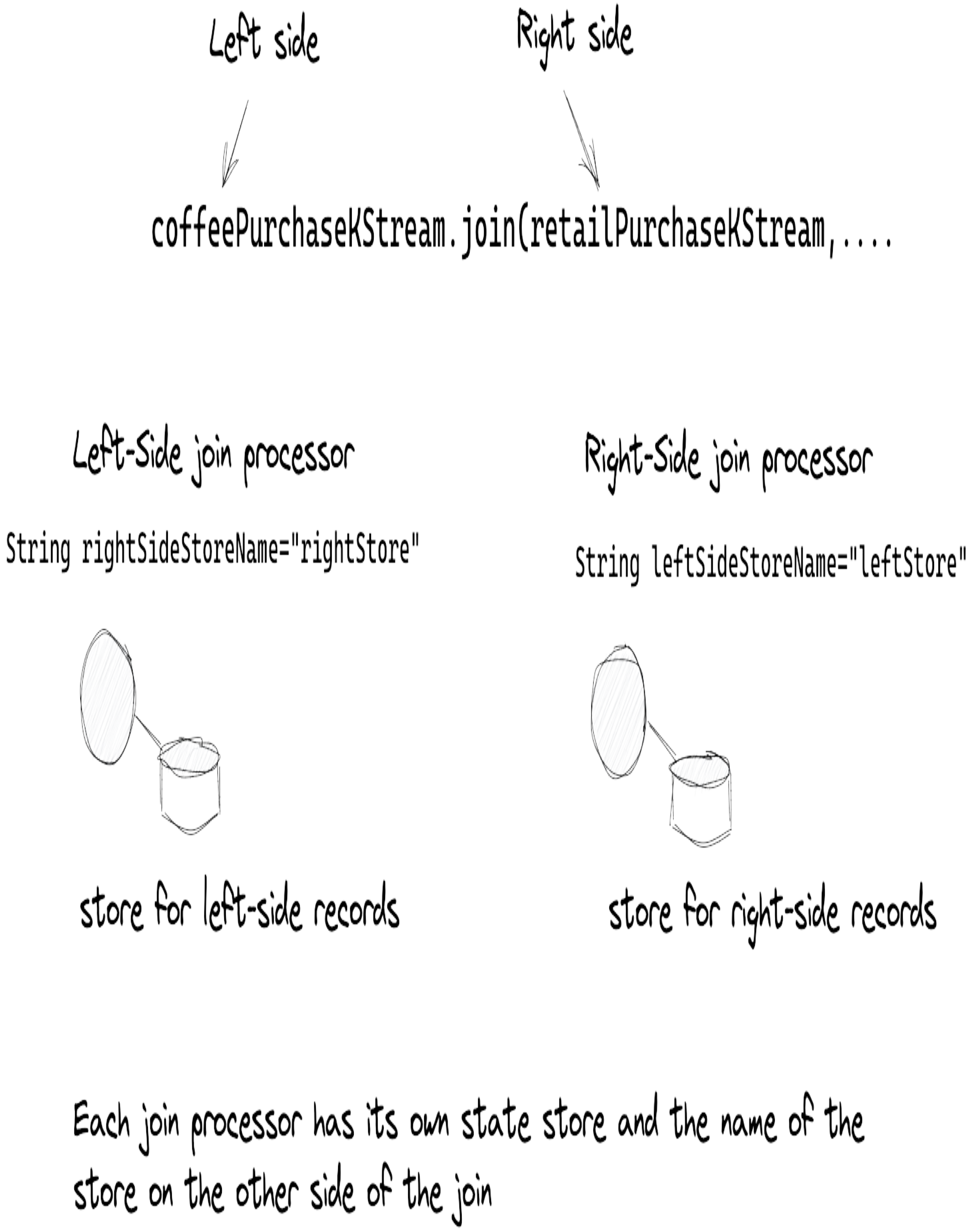
`Serde` objects are required for joins because join participants are materialized in windowed state stores. You provide only one `Serde` for the key, because both sides of the join must have a key of the same type.

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


Left side
Right side
Left-Side join processor
Right-Side join processor
store for left-side records
store for right-side records
Each join processor has its own state store and the name of the store on the other side of the join

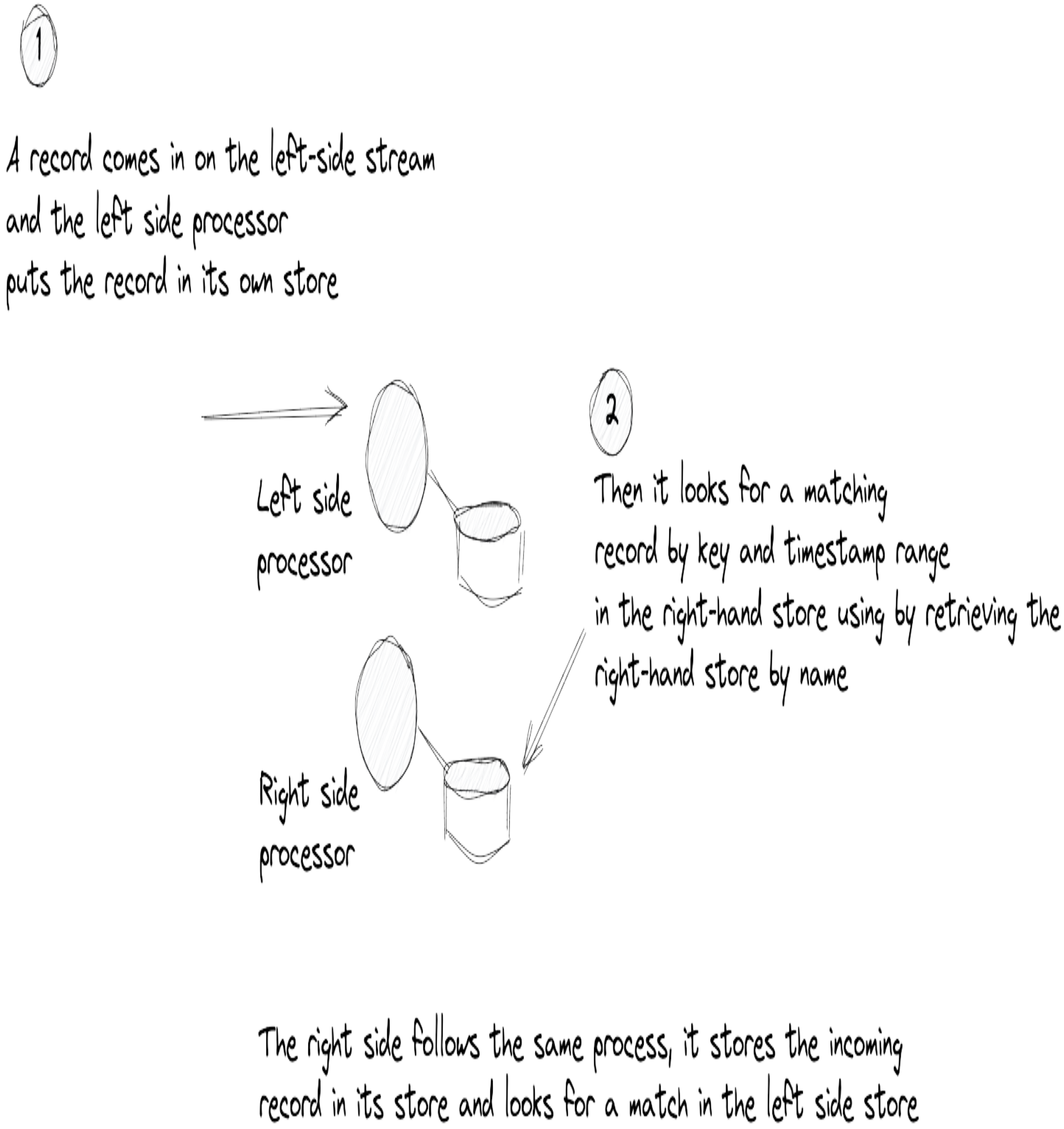
```
coffeePurchaseKStream.join(retailPurchaseKStream,....
```

```
String rightSideStoreName="rightStore"
```

```
String leftSideStoreName="leftStore"
```

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:

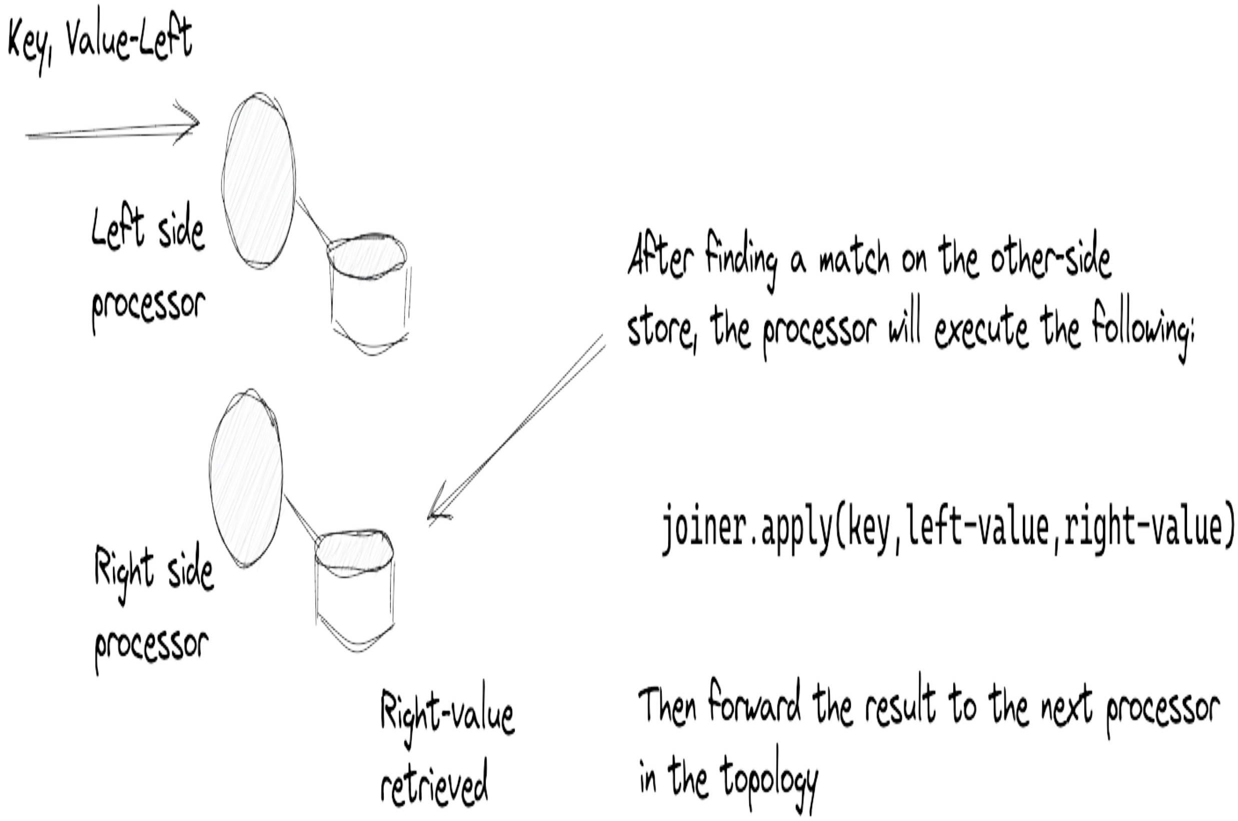
Figure 7.12. Join processors look in the other side’s state store for matches when a new record arrives


1
A record comes in on the left-side stream
and the left side processor
puts the record in its own store
Left side
processor
2
Then it looks for a matching
record by key and timestamp range
in the right-hand store using by retrieving the
right-hand store by name
Right side
processor
The right side follows the same process, it stores the incoming
record in its store and looks for a match in the left side store

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


Key, Value-Left
Left side processor
After finding a match on the other-side store, the processor will execute the following:
Right side processor
joiner.apply(key,left-value,right-value)
Right-value retrieved
Then forward the result to the next processor in the topology

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(); #1
    double storeSpend = retailPurchase.getPurchasedItemsList() #2
            .stream()
            .mapToDouble(pi -> pi.getPrice() * pi.getQuantity()).
    double promotionPoints = coffeeSpend + storeSpend;  #3
    if (storeSpend > 50.00) {
        promotionPoints += 50.00;
    }
    return Promotion.newBuilder()  #4
            .setCustomerId(retailPurchase.getCustomerId())
            .setDrink(coffeePurchase.getDrink())
            .setItemsPurchased(retailPurchase.getPurchasedItemsCo
            .setPoints(promotionPoints).build();
}
```

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

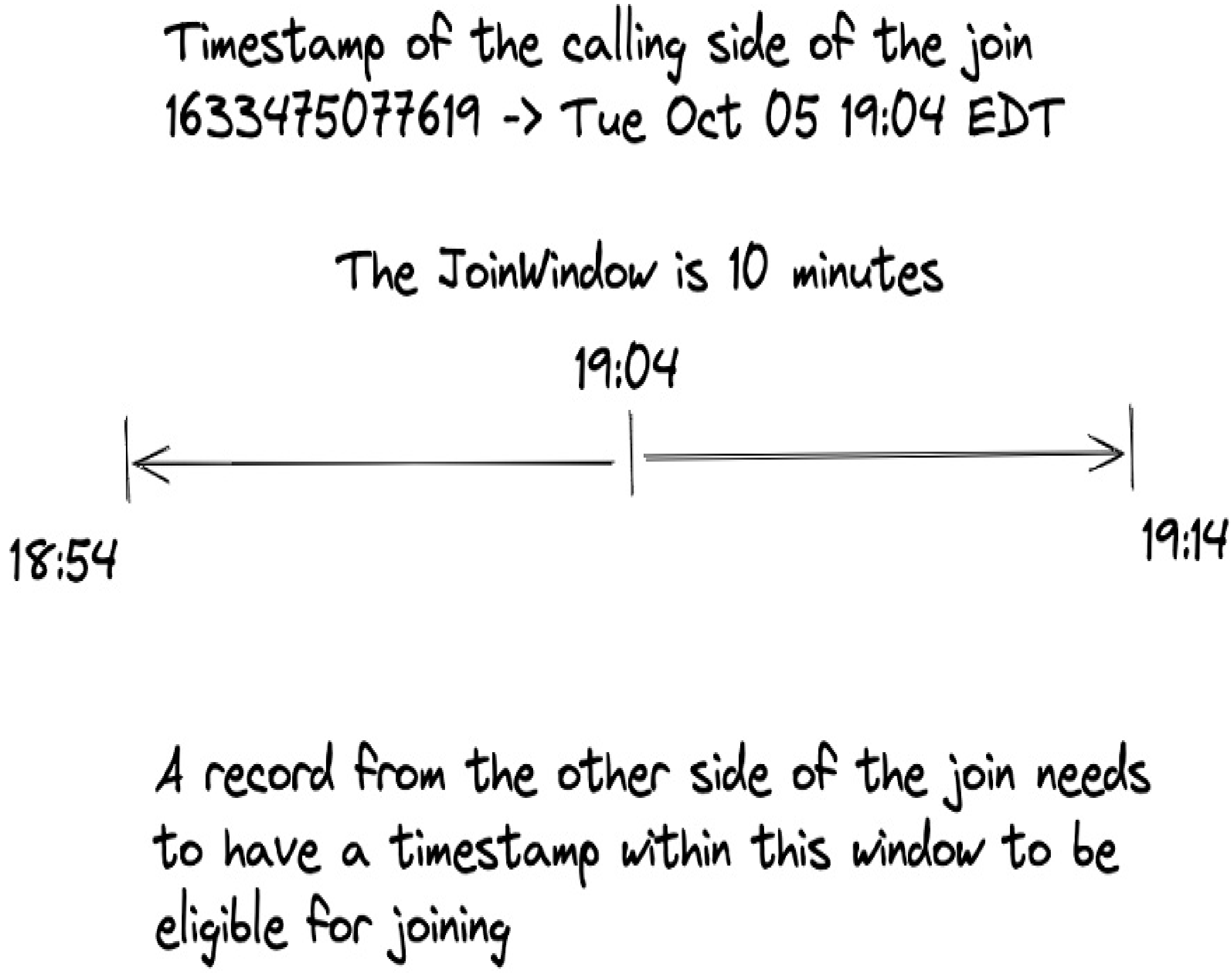
Kafka Streams also provides a `ValueJoinerWithKey` interface which provides access to the key for calculating the value of the join result. However the key is considered ***read-only*** and making changes to it in the joiner implementation will lead undefined behavior.

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.


Timestamp of the calling side of the join
1633475077619 -> Tue Oct 05 19:04 EDT
The JoinWindow is 10 minutes
19:04
18:54
19:14
A record from the other side of the join needs to have a timestamp within this window to be eligible for joining

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 the section called "Naming Stateful operations" 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.

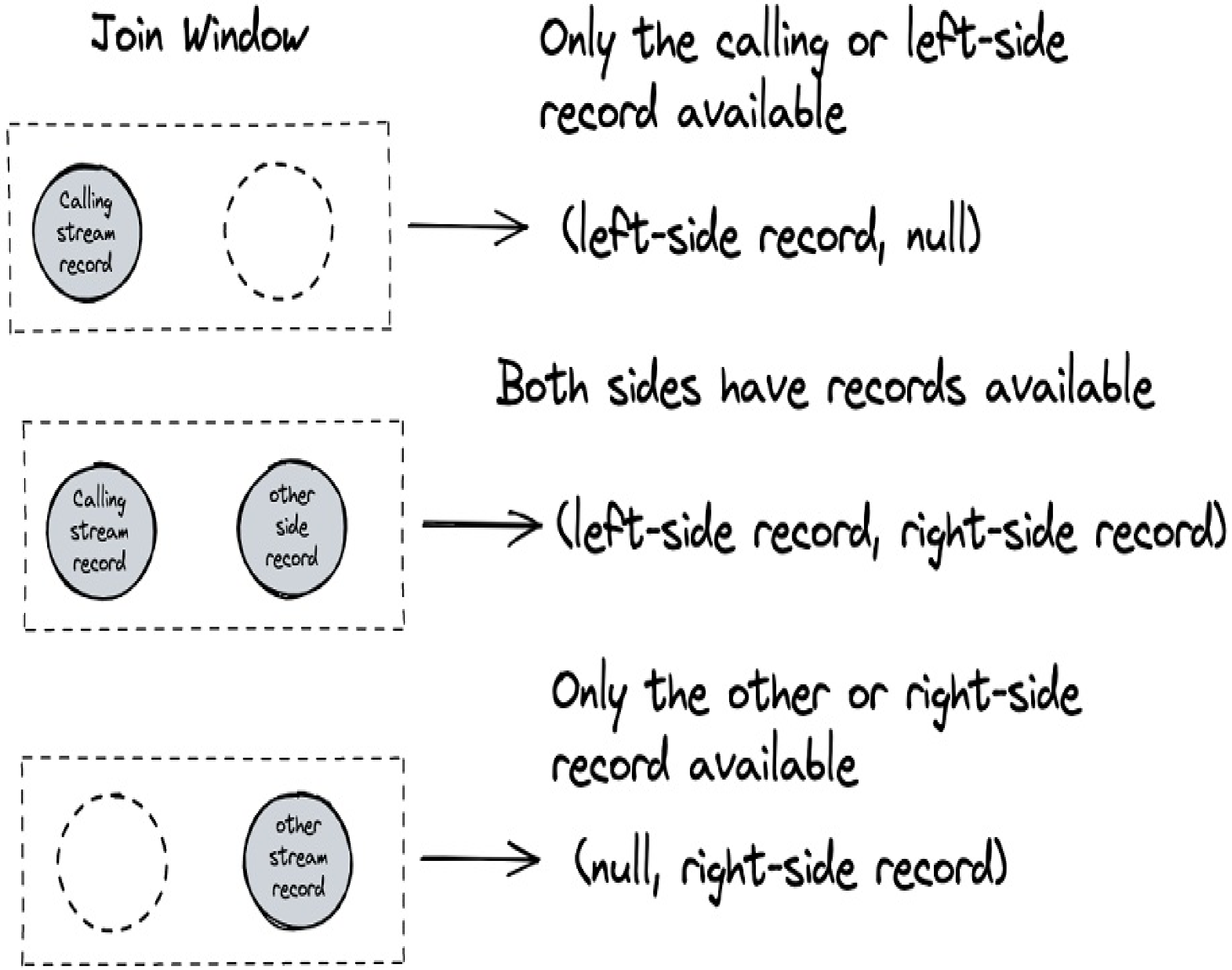
7.3.7 Outer joins

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.


Join Window
Only the calling or left-side record available
Calling stream record
(left-side record, null)
Both sides have records available
Calling stream record
other side record
(left-side record, right-side record)
Only the other or right-side record available
other stream record
(null, right-side record)

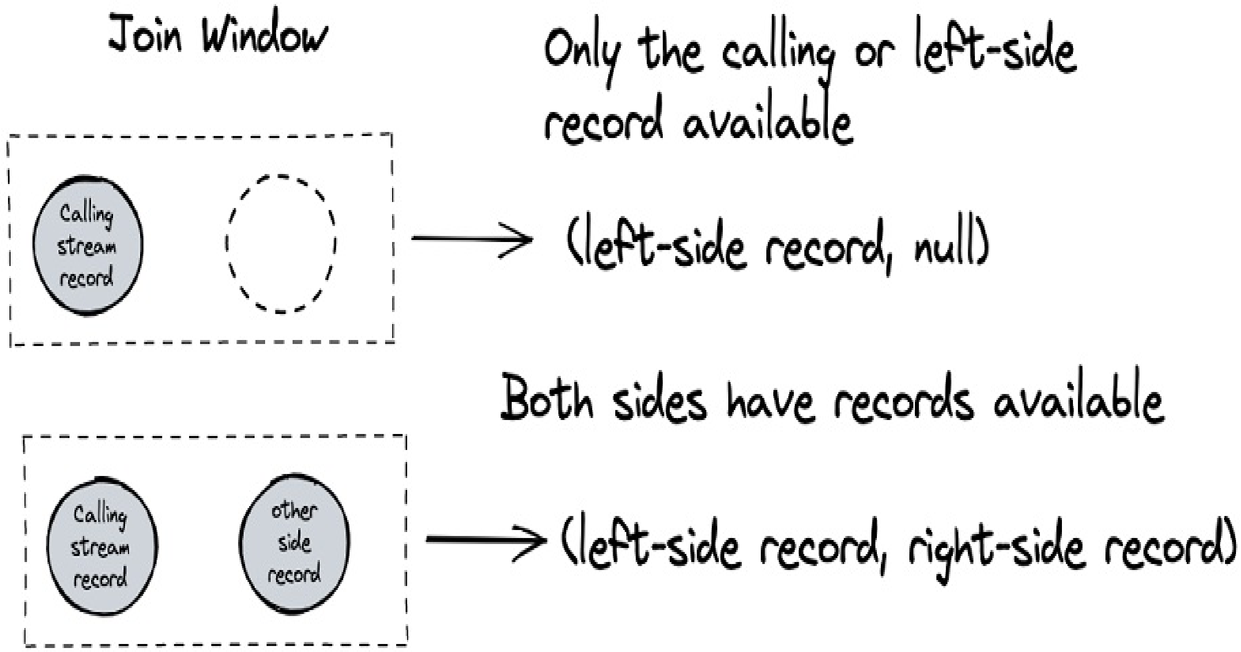
7.3.8 Left-outer join

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.


Join Window
Only the calling or left-side record available
Calling stream record
(left-side record, null)
Both sides have records available
Calling stream record
other side record
(left-side record, right-side record)

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.

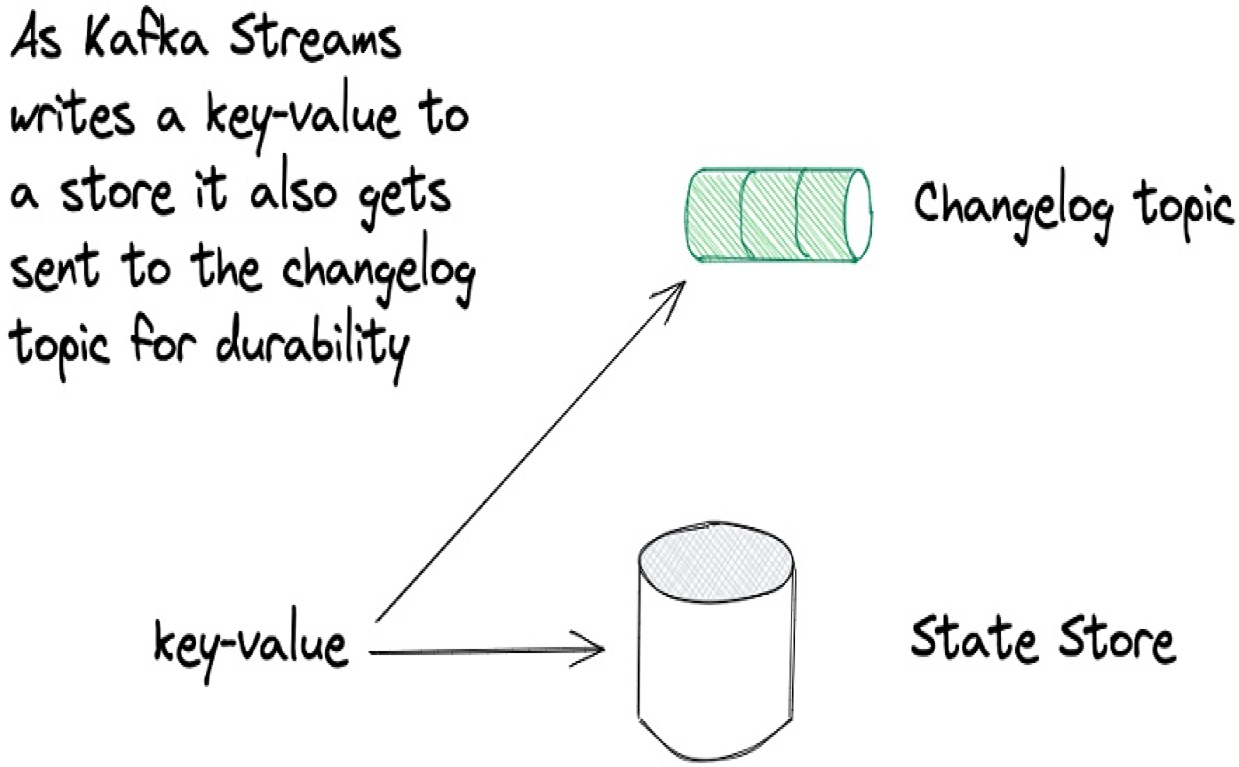
Before I go into any specifics, let’s cover some general information. At a high-level, the state stores in Kafka Streams are key-value stores and they fall into two categories, persistent and in-memory. Both types are durable due to the fact that Kafka Streams uses changelog topics to back the stores. 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


As Kafka Streams writes a key-value to a store it also gets sent to the changelog topic for durability
Changelog topic
key-value
State Store

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

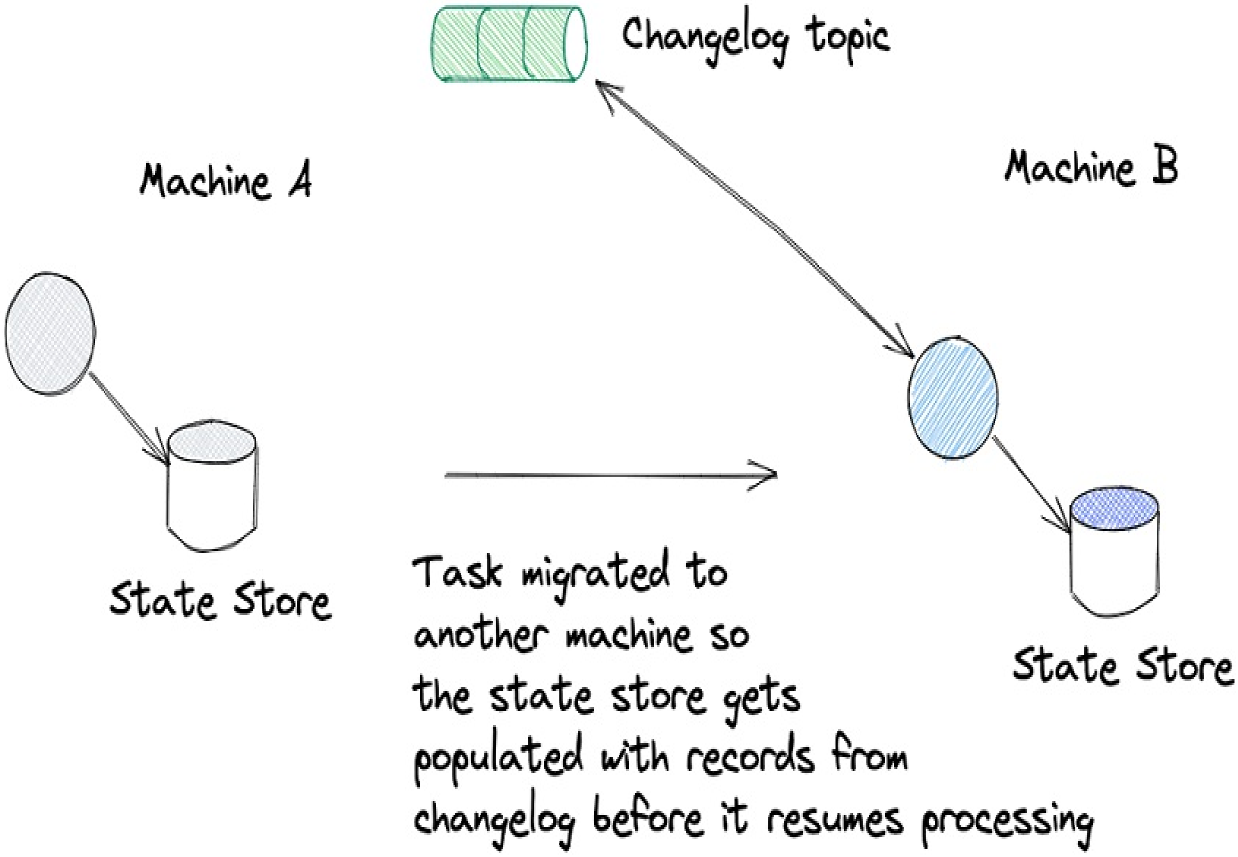
An exception to using a checkpoint file for restoration is when you run Kafka Streams in EOS mode (either `exactly_once` or `exactly_once_v2` is enabled) as state stores are fully restored on startup to ensure the only records in the stores are ones that were included in successful transactions.

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

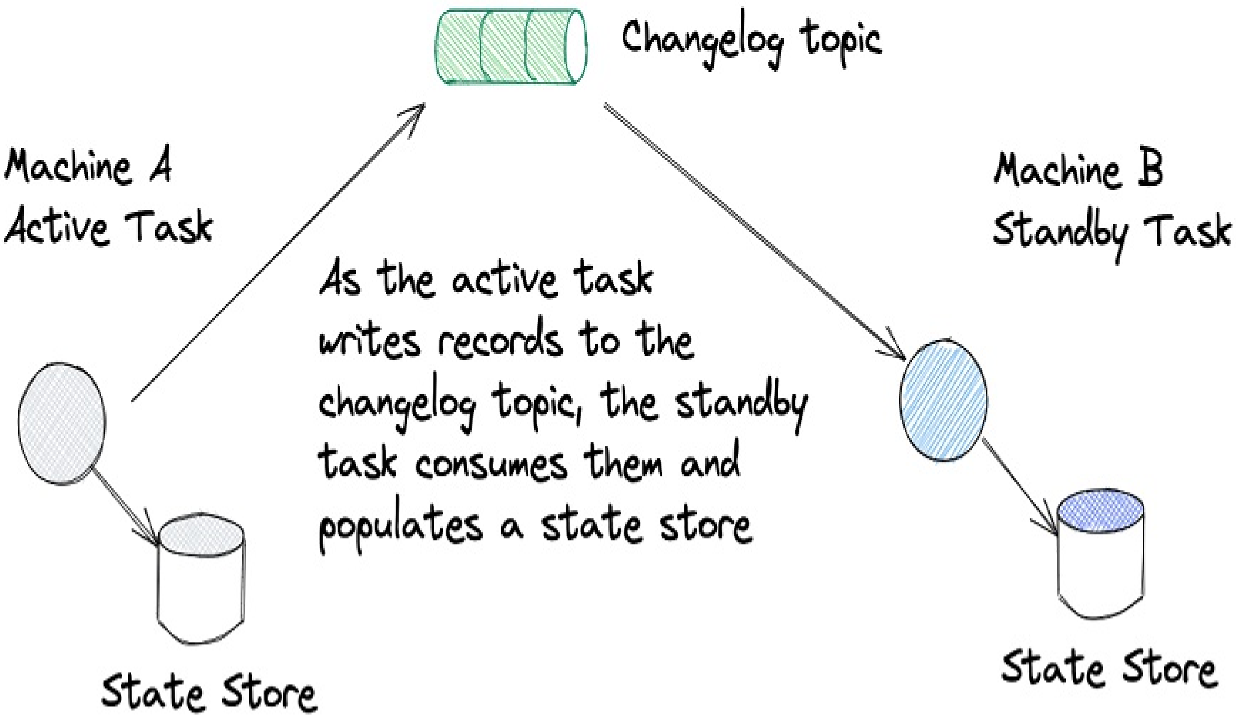
To enable fast failover from an application instance dropping out of the group Kafka Streams provides the standby task. A standby task "shadows" an active task by consuming from the changelog topic into a state store local to the standby. Then should the active task drop out of the group, the standby 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


Changelog topic
Machine A
Active Task
Machine B
Standby Task
As the active task writes records to the changelog topic, the standby task consumes them and populates a state store
State Store
State Store

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.

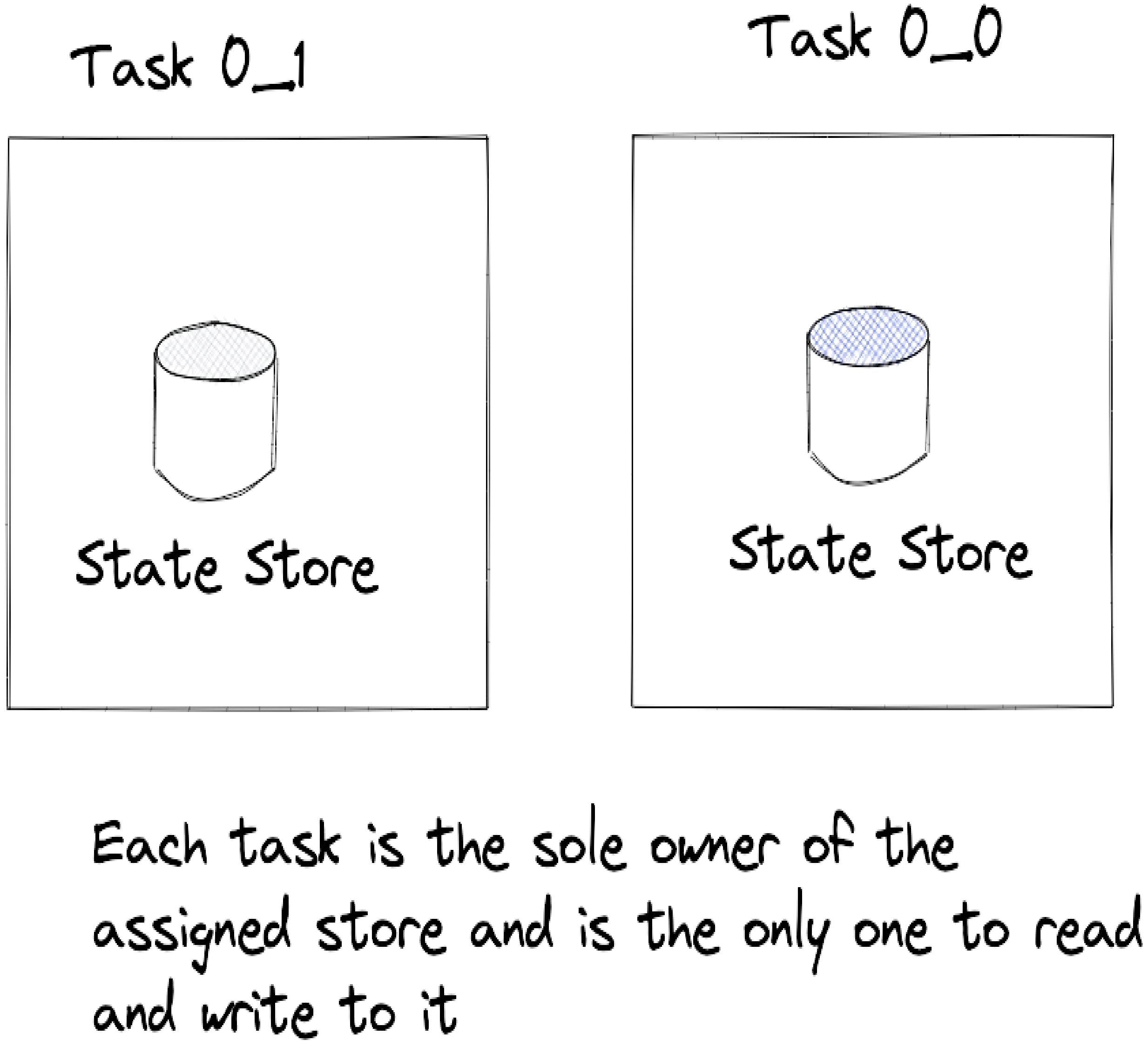
So far we've covered how state stores enable stateful operations and how the stores are robust due to changelog topic and using standby tasks to enable quick failover. But we still have some more ground to cover. First we'll go over state store assignment, from there you'll learn how to configure state 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


Task 0_1
Task 0_0
State Store
State Store
Each task is the sole owner of the assigned store and is the only one to read and write 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 #1
     (stores: [KSTREAM-AGGREGATE-STATE-STORE-0000000001]) #2
      --> 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
```

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_0/rocksdb/KSTREAM-AGGREGATE-STATE-STORE-0000000001`

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 #1
     (stores: [KSTREAM-AGGREGATE-STATE-STORE-0000000002]) #2
      --> 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
```

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")) #1
       .toStream()
       .to("output")
```

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-
        Consumed.with(Serdes.String(), Serdes.Double()));

pokerScoreStream
        .groupByKey()
        .reduce(Double::sum,
                Materialized.<String, Double>as(
            Stores.inMemoryKeyValueStore("memory-poker-score-stor
                  .withKeySerde(Serdes.String())  #2
                  .withValueSerde(Serdes.Double())) #3
        .toStream()
        .to("total-scores",
                      Produced.with(Serdes.String(), Serdes.Double()));
```

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/kafkastreams/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)) #1
        .toStream()
        .to("output")
```

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 Windowing and Time Stamps

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.

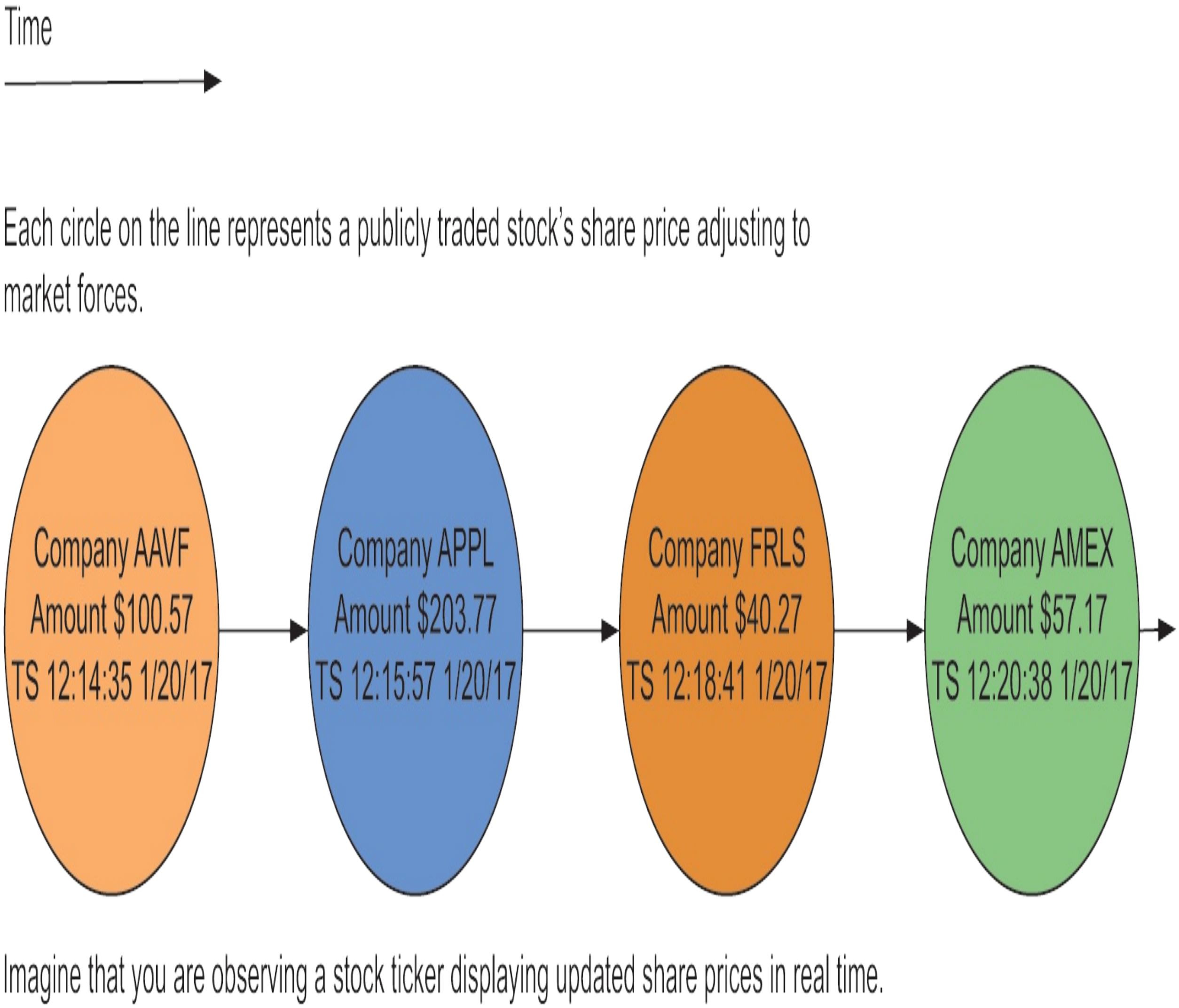
After covering the table abstractions we’ll get into how to "bucket" your aggregations into specific time periods using windowing. For example, how many purchases have there been over the past hour, updated every ten minutes? Windowing allows you to place data in discrete blocks of time, as opposed to 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



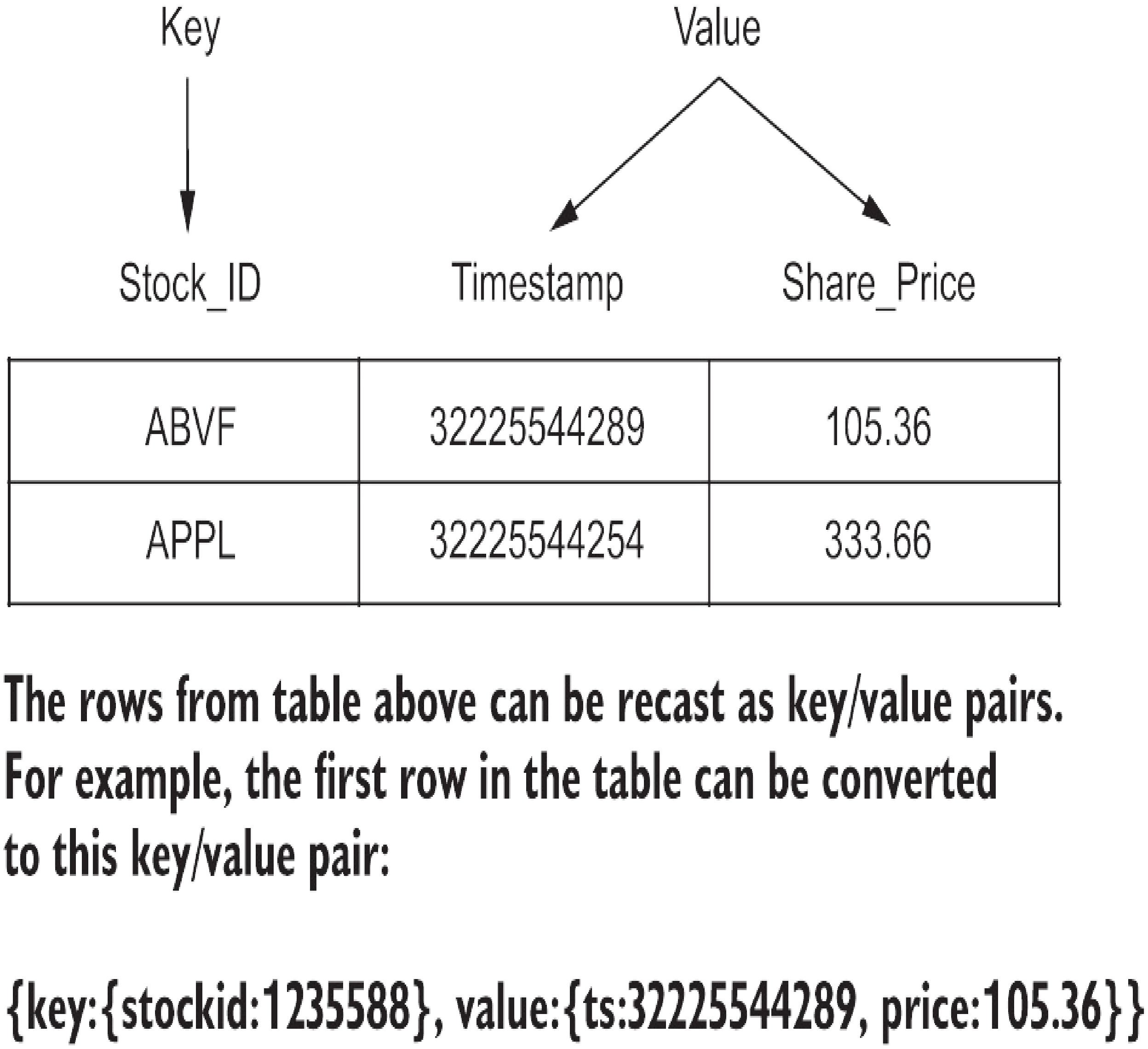
Each circle on the line represents a publicly traded stock's share price adjusting to market forces.

Imagine that you are observing a stock ticker displaying updated share prices in real time.

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.

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.



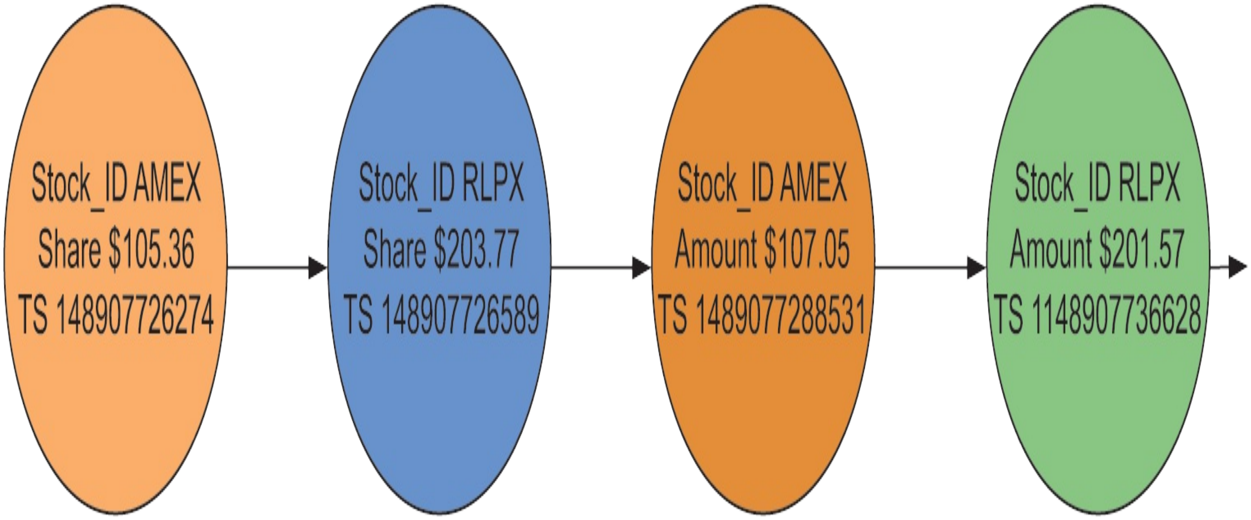
Stock_ID	Timestamp	Share_Price
ABVF	32225544289	105.36
APPL	32225544254	333.66

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:32225544289, price:105.36}}
```

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.

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.



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.

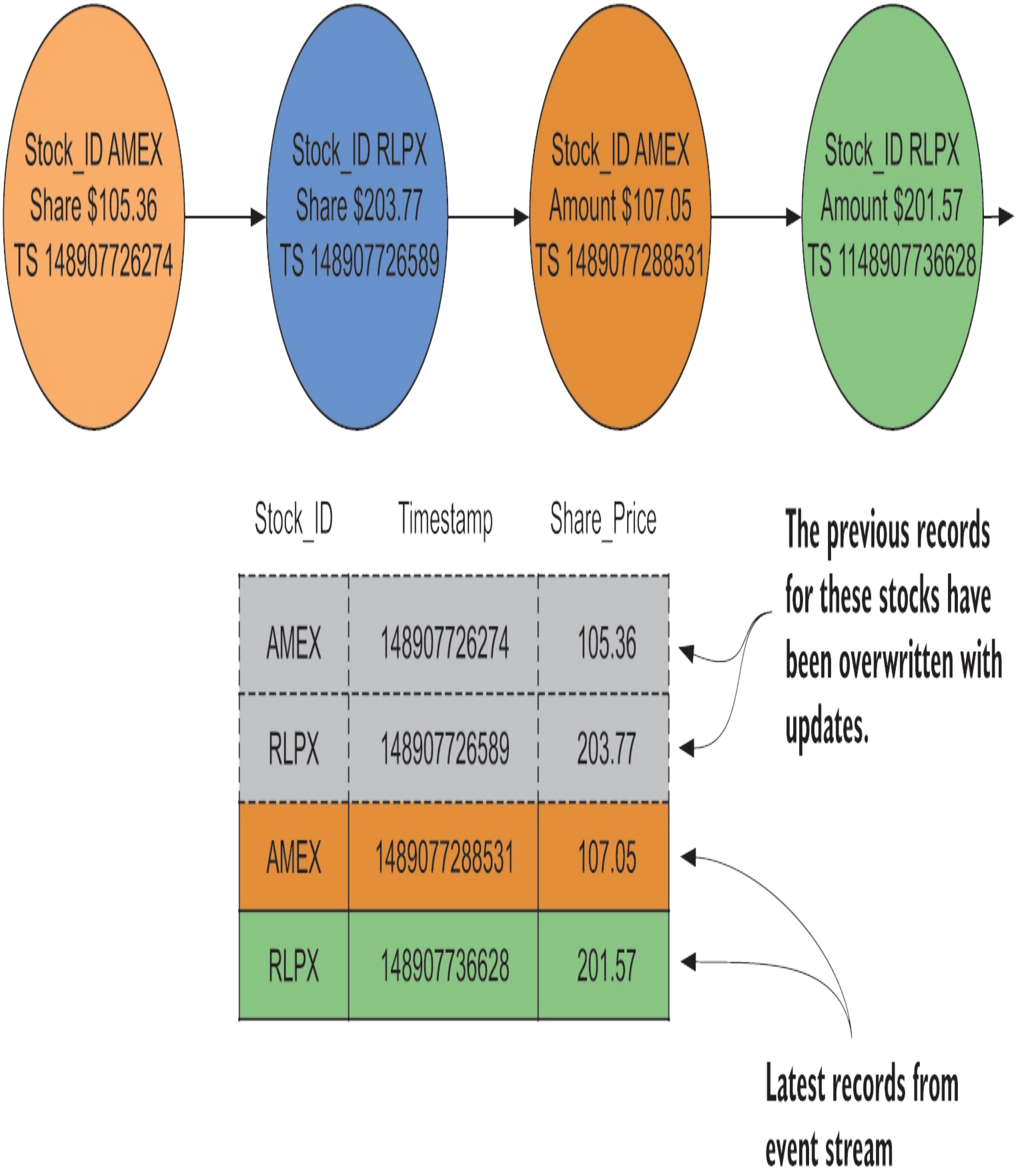
What’s important here is that you can view a stream of events in the same light as inserts into a table, which can help give you a deeper understanding of using streams for working with events. The next step is to consider the case where events in the stream *are* related to one another.

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.

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.



Stock_ID	Timestamp	Share_Price
AMEX	148907726274	105.36
RLPX	148907726589	203.77
AMEX	1489077288531	107.05
RLPX	148907736628	201.57

The previous records for these stocks have been overwritten with updates.

Latest records from event stream

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.

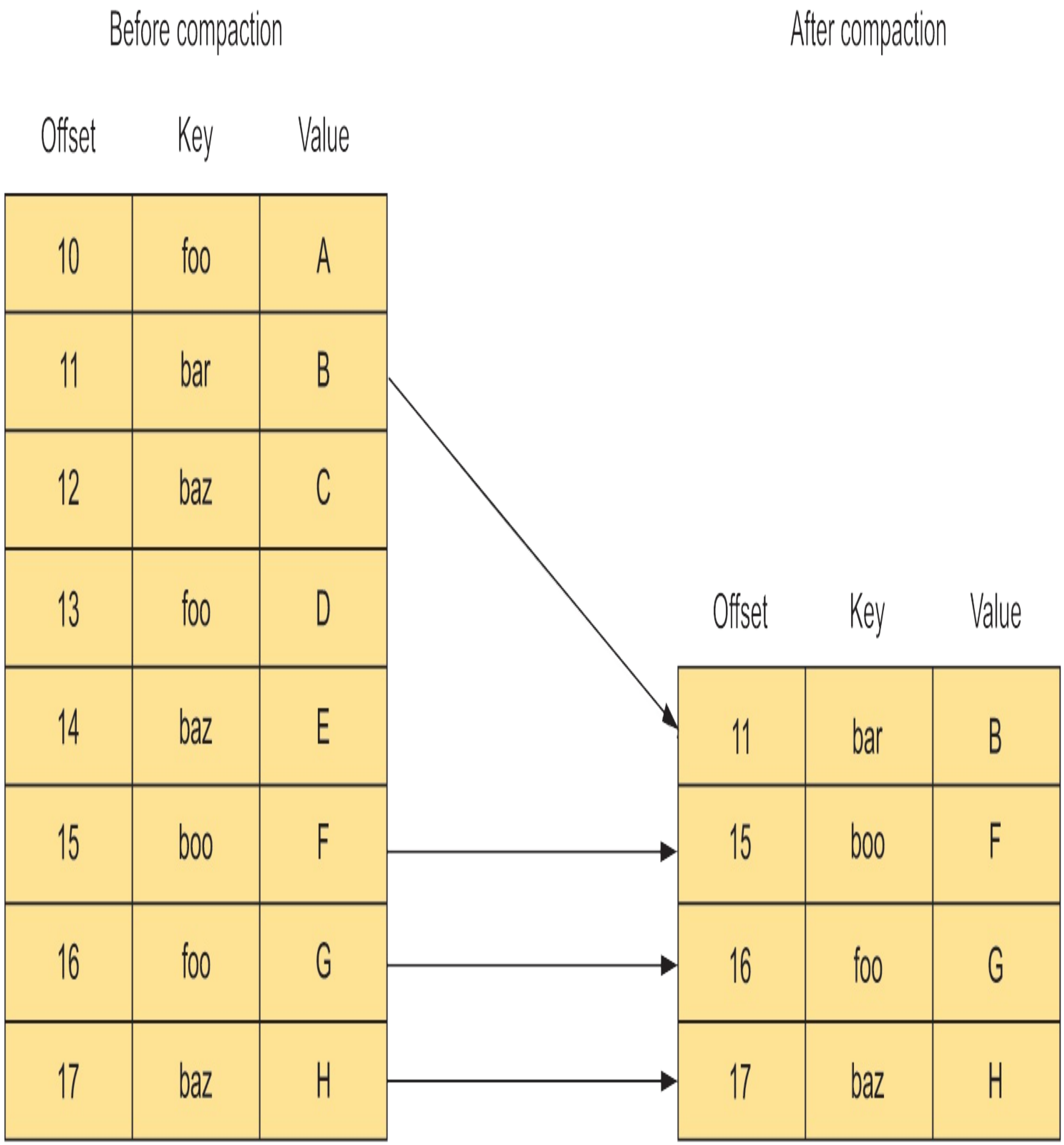
Here, you can see the relationship between a stream of updates and a database table. Both a log and a changelog represent incoming records appended to the end of a file. In a log, you see all the records; but in a changelog, you only keep the latest record for any given key.

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.



Before compaction

Offset	Key	Value
10	foo	A
11	bar	B
12	baz	C
13	foo	D
14	baz	E
15	boo	F
16	foo	G
17	baz	H

After compaction

Offset	Key	Value
11	bar	B
15	boo	F
16	foo	G
17	baz	H

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.

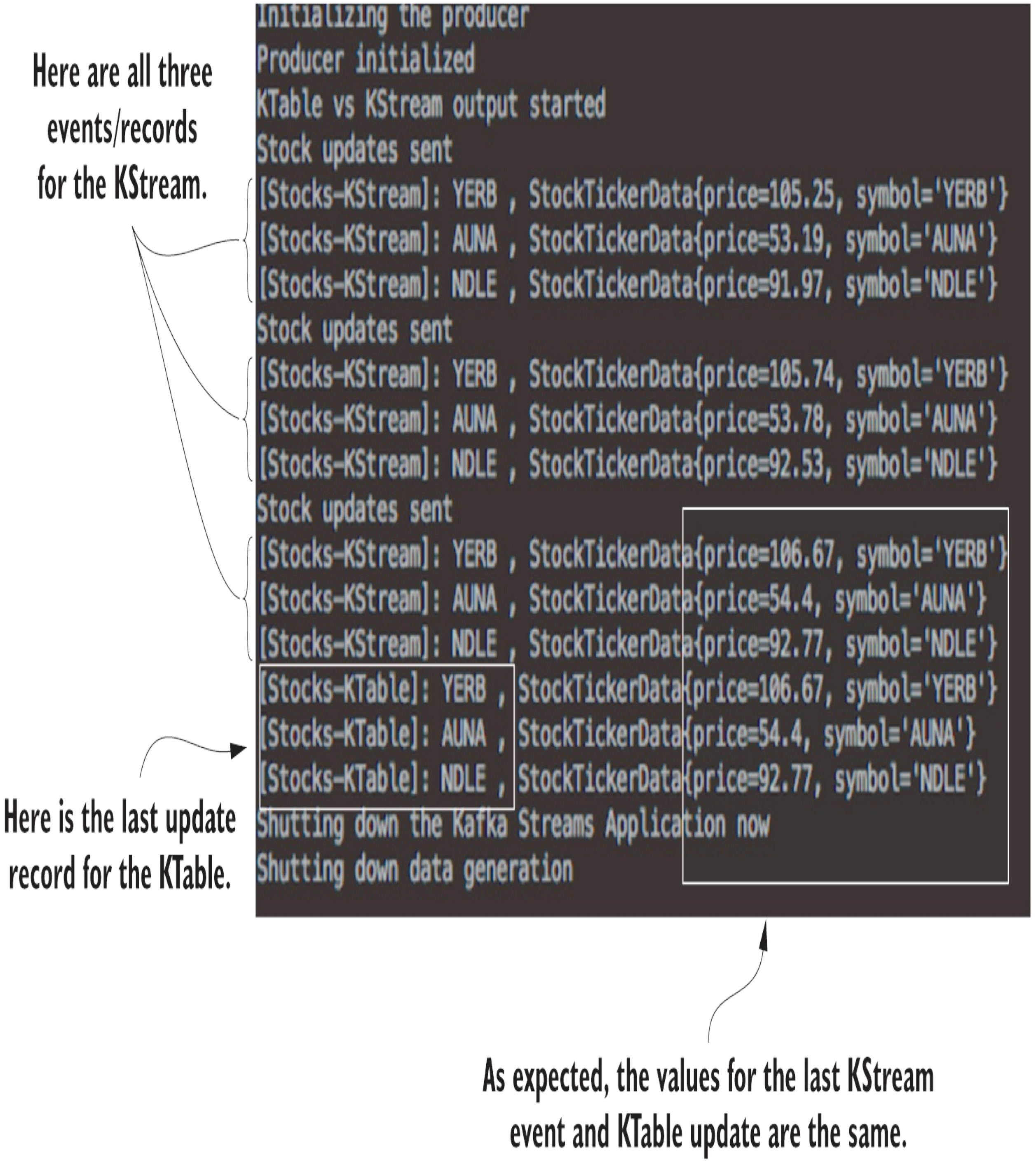
Note

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.

Figure 8.6. `KTable` versus `KStream` printing messages with the same keys

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.



```
Initializing the producer
Producer initialized
KTable vs KStream output started
Stock updates sent
[Stocks-KStream]: YERB , StockTickerData{price=105.25, symbol='YERB'}
[Stocks-KStream]: AUNA , StockTickerData{price=53.19, symbol='AUNA'}
[Stocks-KStream]: NDLE , StockTickerData{price=91.97, symbol='NDLE'}
Stock updates sent
[Stocks-KStream]: YERB , StockTickerData{price=105.74, symbol='YERB'}
[Stocks-KStream]: AUNA , StockTickerData{price=53.78, symbol='AUNA'}
[Stocks-KStream]: NDLE , StockTickerData{price=92.53, symbol='NDLE'}
Stock updates sent
[Stocks-KStream]: YERB , StockTickerData{price=106.67, symbol='YERB'}
[Stocks-KStream]: AUNA , StockTickerData{price=54.4, symbol='AUNA'}
[Stocks-KStream]: NDLE , StockTickerData{price=92.77, symbol='NDLE'}
[Stocks-KTable]: YERB , StockTickerData{price=106.67, symbol='YERB'}
[Stocks-KTable]: AUNA , StockTickerData{price=54.4, symbol='AUNA'}
[Stocks-KTable]: NDLE , StockTickerData{price=92.77, symbol='NDLE'}
Shutting down the Kafka Streams Application now
Shutting down data generation
```

Here are all three events/records for the KStream.

Here is the last update record for the KTable.

As expected, the values for the last KStream event and KTable update are the same.

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);#1
KStream<String, StockTickerData> stockTickerStream =
builder.stream(STOCK_TICKER_STREAM_TOPIC);#2

stockTickerTable.toStream()
  .print(Printed.<String, StockTickerData>toSysOut()
  .withLabel("Stocks-KTable")); #3

stockTickerStream
  .print(Printed.<String, StockTickerData>toSysOut()
  .withLabel("Stocks-KStream"));#4
```

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

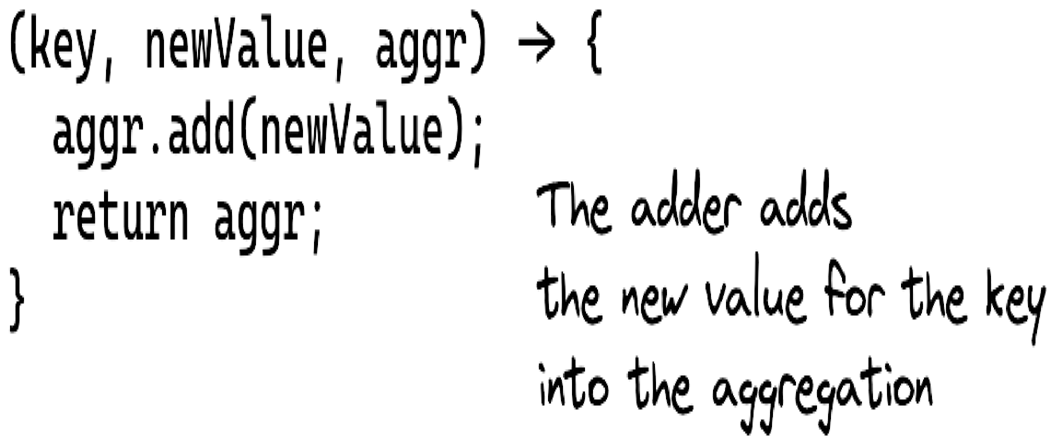
```
KTable<String, StockAlertProto.StockAlert> stockTable =
                builder.table("stock-alert",
                 Consumed.with(stringSerde, stockAlertSerde)); #1

stockTable.groupBy((key, value) ->
                    KeyValue.pair(value.getMarketSegment(), value
                Grouped.with(stringSerde, stockAlertSerde)) #2
        .aggregate(segmentInitializer, #3
                adderAggregator,                 #4
                subtractorAggregator,            #5
                Materialized.with(stringSerde, segmentSerde))
        .toStream()

        .to("stock-alert-aggregate",
                Produced.with(stringSerde, segmentSerde));
```

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.

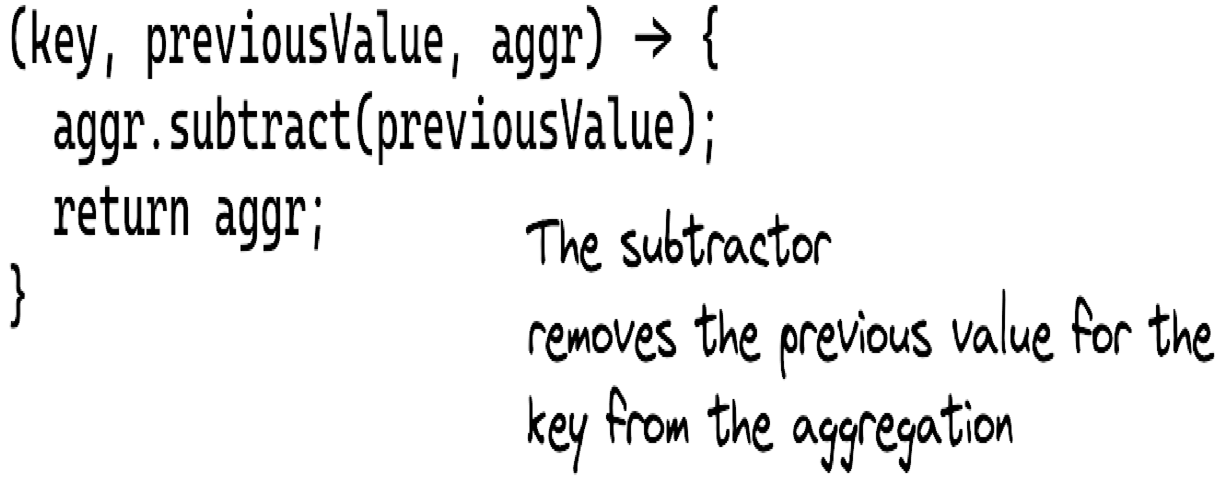
The second difference occurs with annotations four and five. With the `KTable` aggregations, just like with the `KStream` the first parameter you provide is an `Initializer` instance, to provide the default value for the first aggregation. However you then supply ***two aggregators*** one that adds the 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



```
(key, newValue, aggr) → {
  aggr.add(newValue);
  return aggr;
}
```

The adder adds
the new value for the key
into the aggregation



```
(key, previousValue, aggr) → {
  aggr.subtract(previousValue);
  return aggr;
}
```

The subtractor
removes the previous value for the
key from the aggregation

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> adderAgg
                       (key, newStockAlert, currentAgg) -> {

            long currentShareVolume =
                    newStockAlert.getShareVolume(); #1
            double currentDollarVolume =
                    newStockAlert.getShareVolume() * newStockAler

            aggBuilder.setShareVolume(currentAgg.getShareVolume()
            aggBuilder.setDollarVolume(currentAgg.getDollarVolume
}
```

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.getSharePr

aggBuilder.setShareVolume(currentAgg.getShareVolume() - prevShare
aggBuilder.setDollarVolume(currentAgg.getDollarVolume() - prevDol
```

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     : -> key textiles stock alert symbol: "PXLW" share_pric
Adder     : <- updated aggregate dollar_volume: 10.08 share_volum

Subtractor: -> key textiles stock alert symbol: "PXLW" share_pric
               and aggregate dollar_volume: 54.57 share_volume: 1

Subtractor: <- updated aggregate dollar_volume: 44.49 share_volum
```

```
Adder     : -> key textiles stock alert symbol: "PXLW" share_pric
                 and aggregate dollar_volume: 44.49 share_volume: 1

Adder     : <- updated aggregate dollar_volume: 64.83 share_volum
```

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.

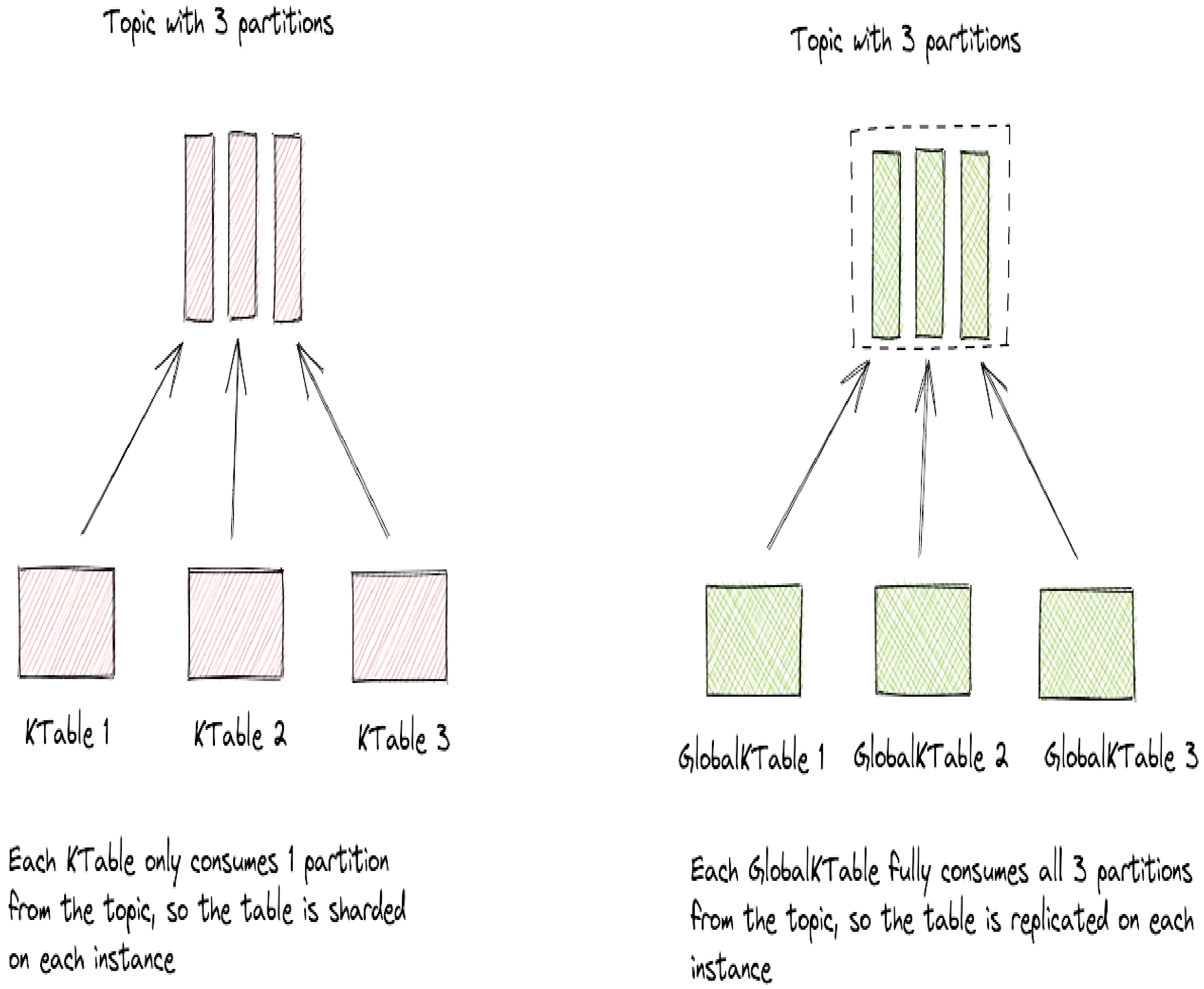
This wraps up our coverage of the `KTable` API but before we move on to more advanced stateful subjects, I'd like to go over another table abstraction offered by Kafka Streams, the `GlobalKTable`.

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


Topic with 3 partitions
Topic with 3 partitions
KTable 1
KTable 2
KTable 3
GlobalKTable 1
GlobalKTable 2
GlobalKTable 3
Each KTable only consumes 1 partition from the topic, so the table is sharded on each instance
Each GlobalKTable fully consumes all 3 partitions from the topic, so the table is replicated on each 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, clickEventSerd

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,   #1
               foreignKeyExtractor, #2
               joiner);  #3
```

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, #1
                keySelector, #2
                valueJoiner); #3
```

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
            .advanceBy(Duration.ofSeconds(10)))     #2
   .count(Materialized.as("hopping-window-counting-store"))
   .toStream()          #3
   .map((windowedKey, value) -> KeyValue.pair(windowedKey.key(),
   .to("counting-output", Produced.with(stringSerde, longSerde));
```

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.

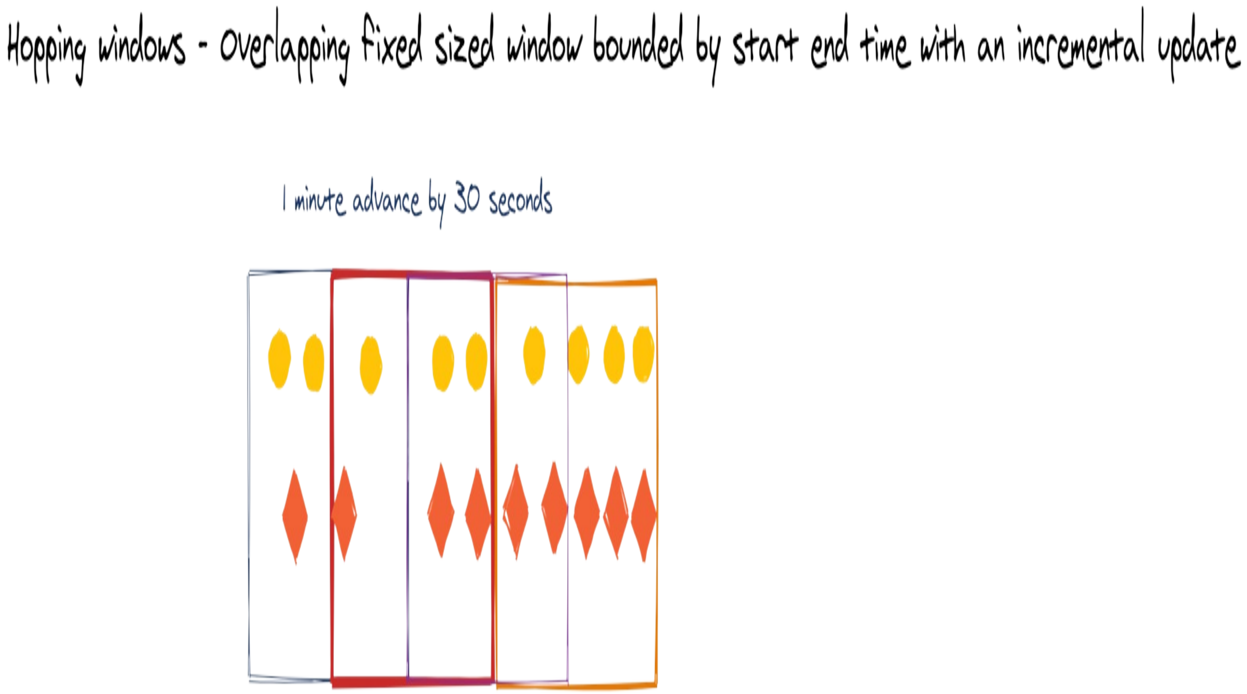
At annotation two, you use the `advanceBy` call which determines the interval that the aggregation will occur. Since the `advanceBy` is less than the window size, it is a hopping window. At annotation three we convert the `KTable` to a `KStream` as we need to convert from the update stream to an event 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


Hopping windows - Overlapping fixed sized window bounded by start end time with an incremental update
1 minute advance by 30 seconds

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.

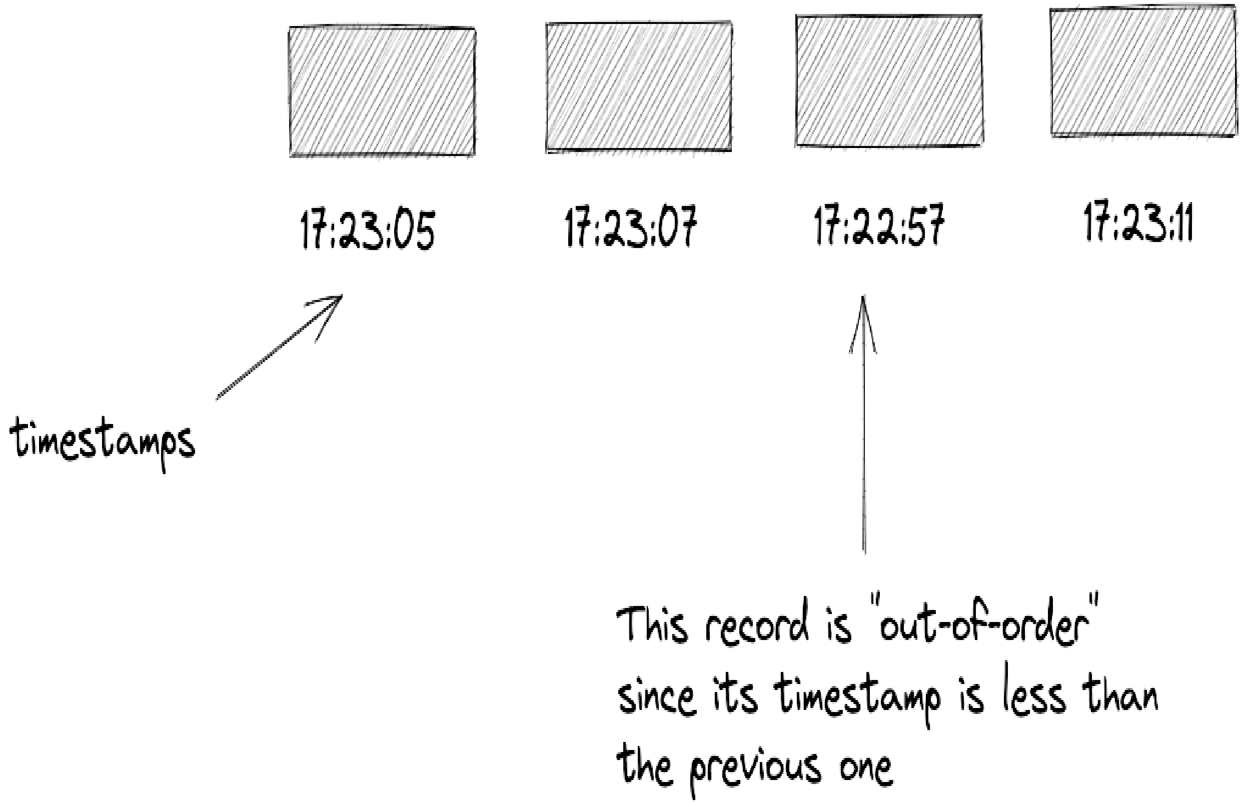
You now have learned about the hopping window, but so far we’ve assumed that records allways arrive in order. Suppose some of your records don’t arrive in order and you’d still like to include them (up to a point) in your count, what whould you do to handle that? Now’s a good time to circle back to the 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:

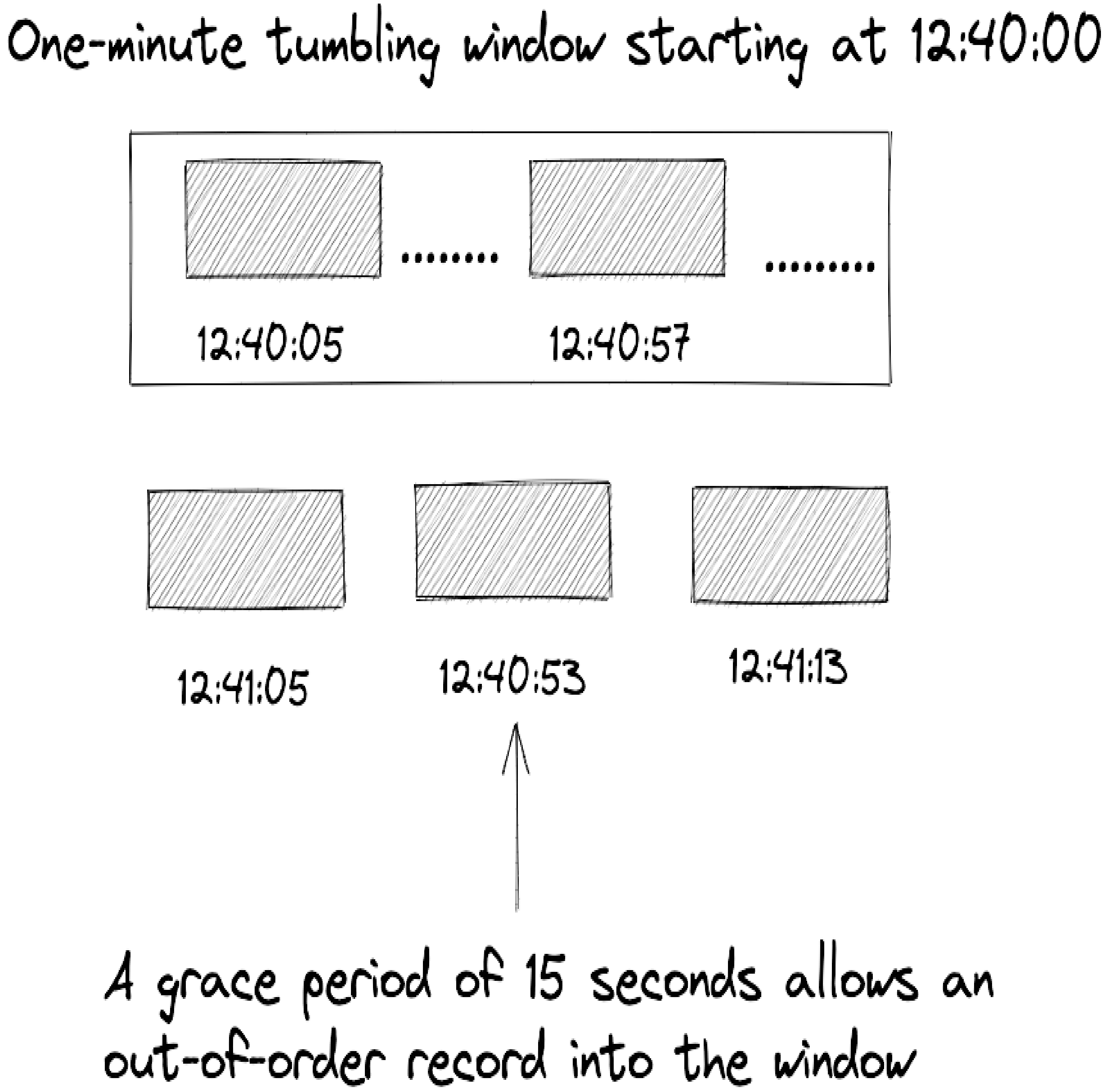
Figure 8.10. Out of order records didn’t arrive in the correct sequence


17:23:05
17:23:07
17:22:57
17:23:11
timestamps
This record is "out-of-order" since its timestamp is less than the previous one

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


One-minute tumbling window starting at 12:40:00
12:40:05
12:40:57
12:41:05
12:40:53
12:41:13
A grace period of 15 seconds allows an out-of-order record into the window

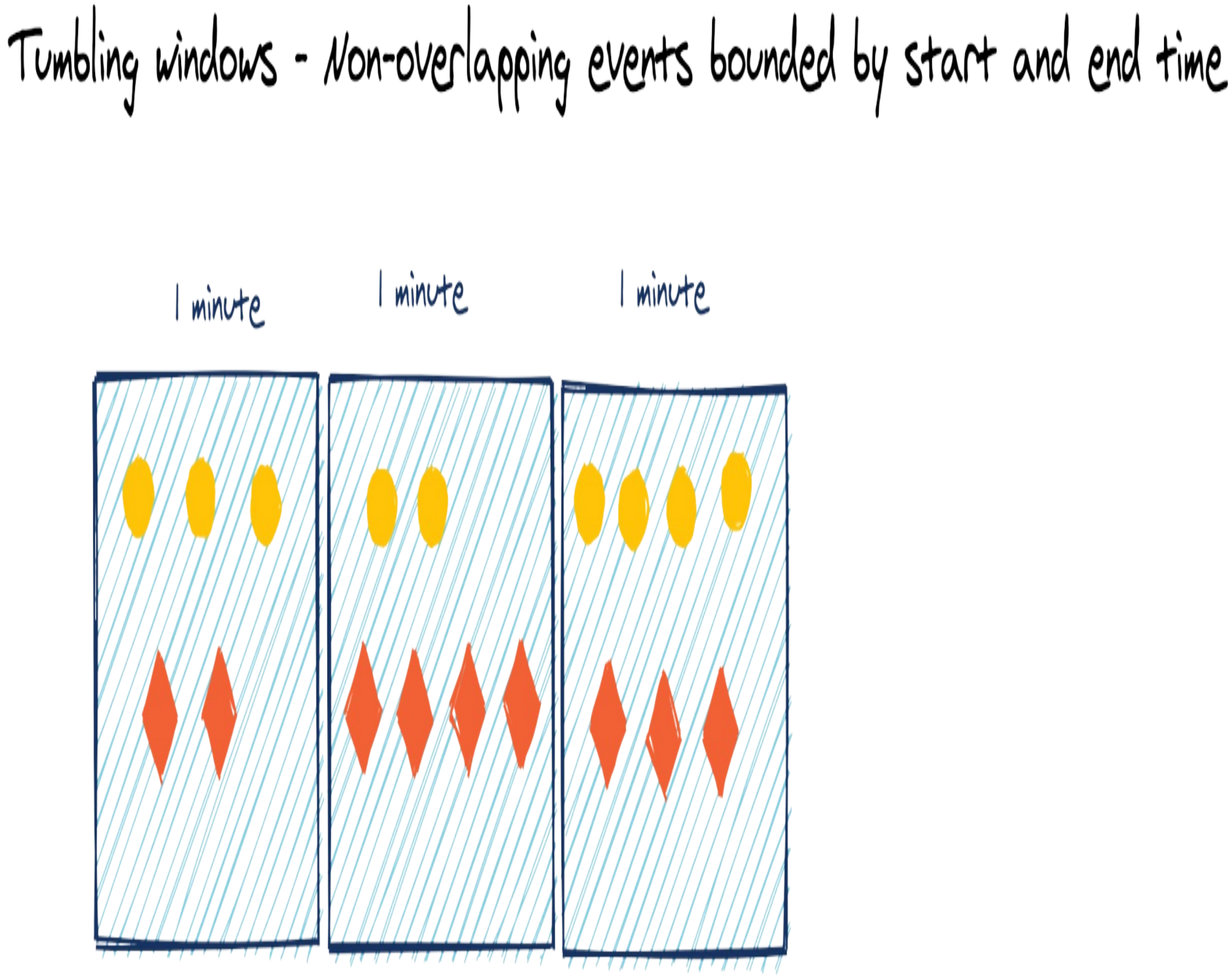
So from looking at the illustration, grace allows records into an aggregation that ***would have been included were they on-time***, and allows for a more accurate calculation. Once the grace period has expired, any out-of-order records are considered late and dropped. In the case of our example above, 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


Tumbling windows - Non-overlapping events bounded by start and end time
1 minute
1 minute
1 minute

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)
                            ,Duration.ofSeconds(30))) #2
    .count(Materialized.as("Tumbling-window-counting-store"))
```

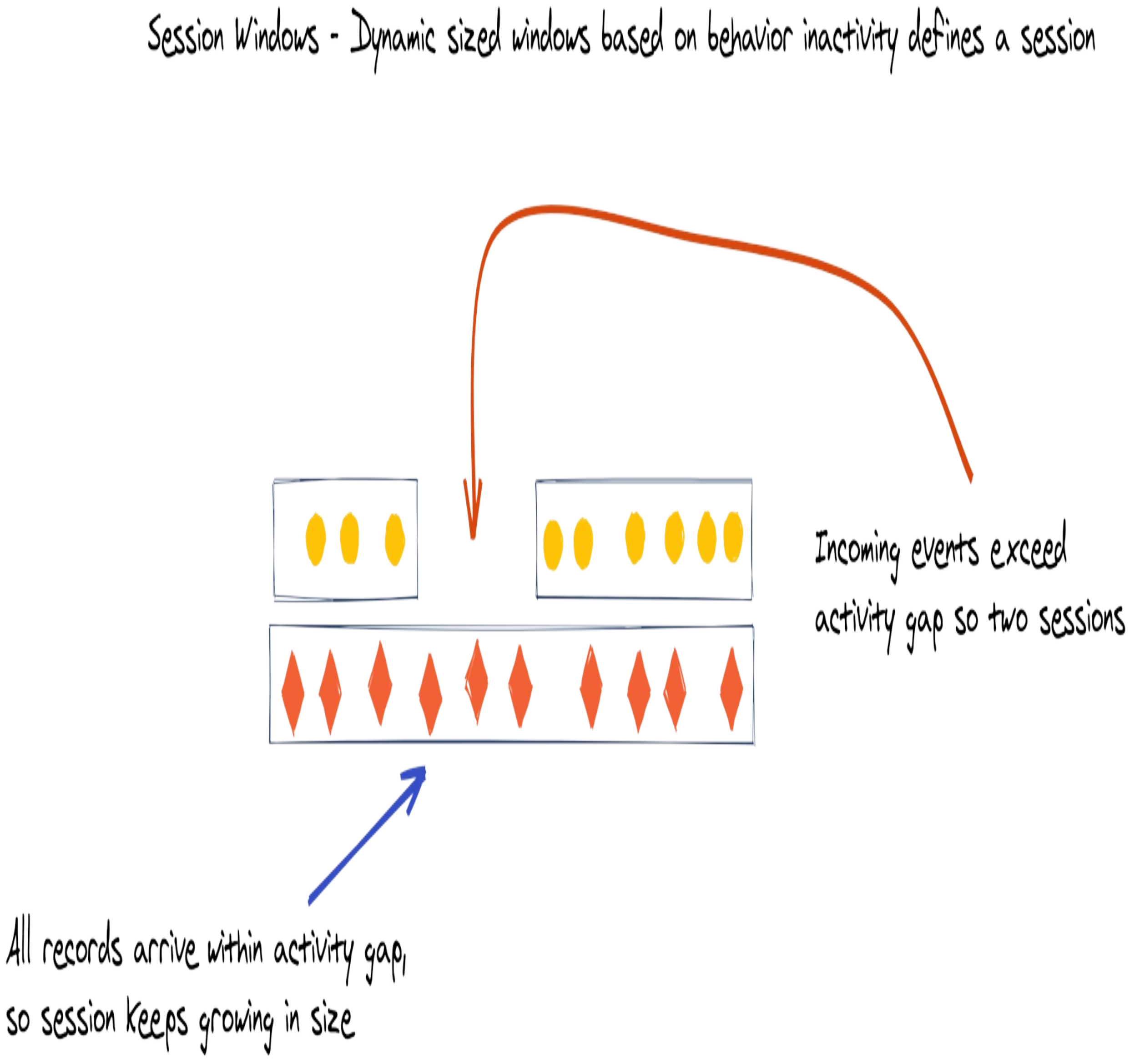
From looking at annotation one using a tumbling window is simply a matter not setting the advance time. Also I’d like to point out that in this example you are using a grace period of thirty seconds, so there’s a second `Duration` parameter passed into the `TimeWindows.ofSizeAndGrace` method. Note that 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


Session Windows - Dynamic sized windows based on behavior inactivity defines a session
Incoming events exceed activity gap so two sessions
All records arrive within activity gap, so session keeps growing in size

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.ofMi
                      Duration.ofSeconds(30)))  #2
 .count(Materialized.as("Session-window-counting-store"))
```

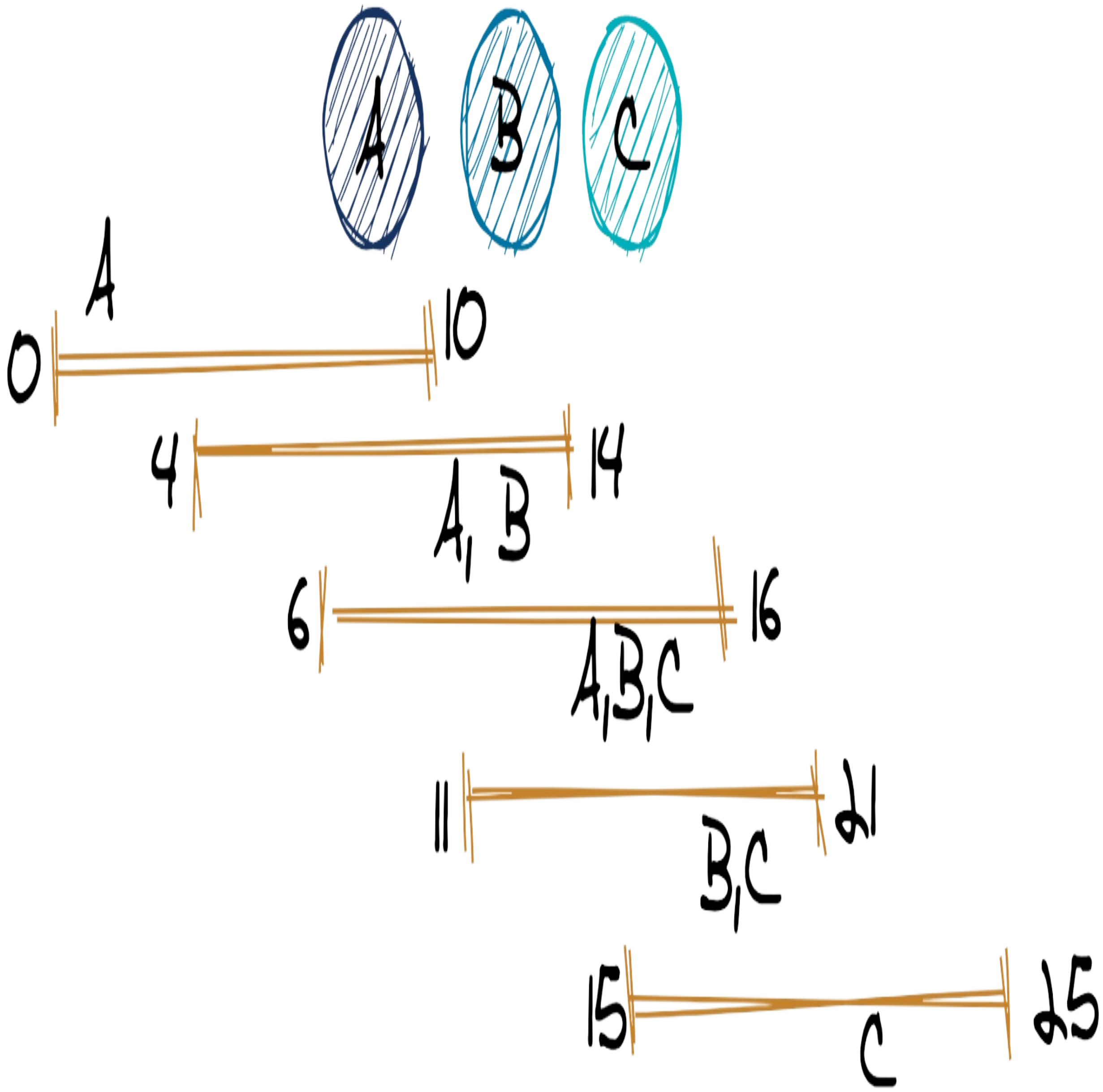
So to use sessions with your aggregation is to use a `SessionWindows` factory method. In this case you specify an inactivity period of one minute and you include a grace period as well. The grace period for session window works in the similar manner, it provides a time for Kafka Streams to include out-of-order 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))) #1
    .count(Materialized.as("Sliding-window-counting-store"))
```

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(
    .count(Materialized.as("Tumbling-window-suppressed-counting-s
    .suppress(untilWindowCloses(unbounded())) #2
```

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) #1
                           .shutDownWhenFull()) #2
```

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),  #1
                          maxRecords(1000)      #2
                         .emitEarlyWhenFull())) #3
```

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 Streams. 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

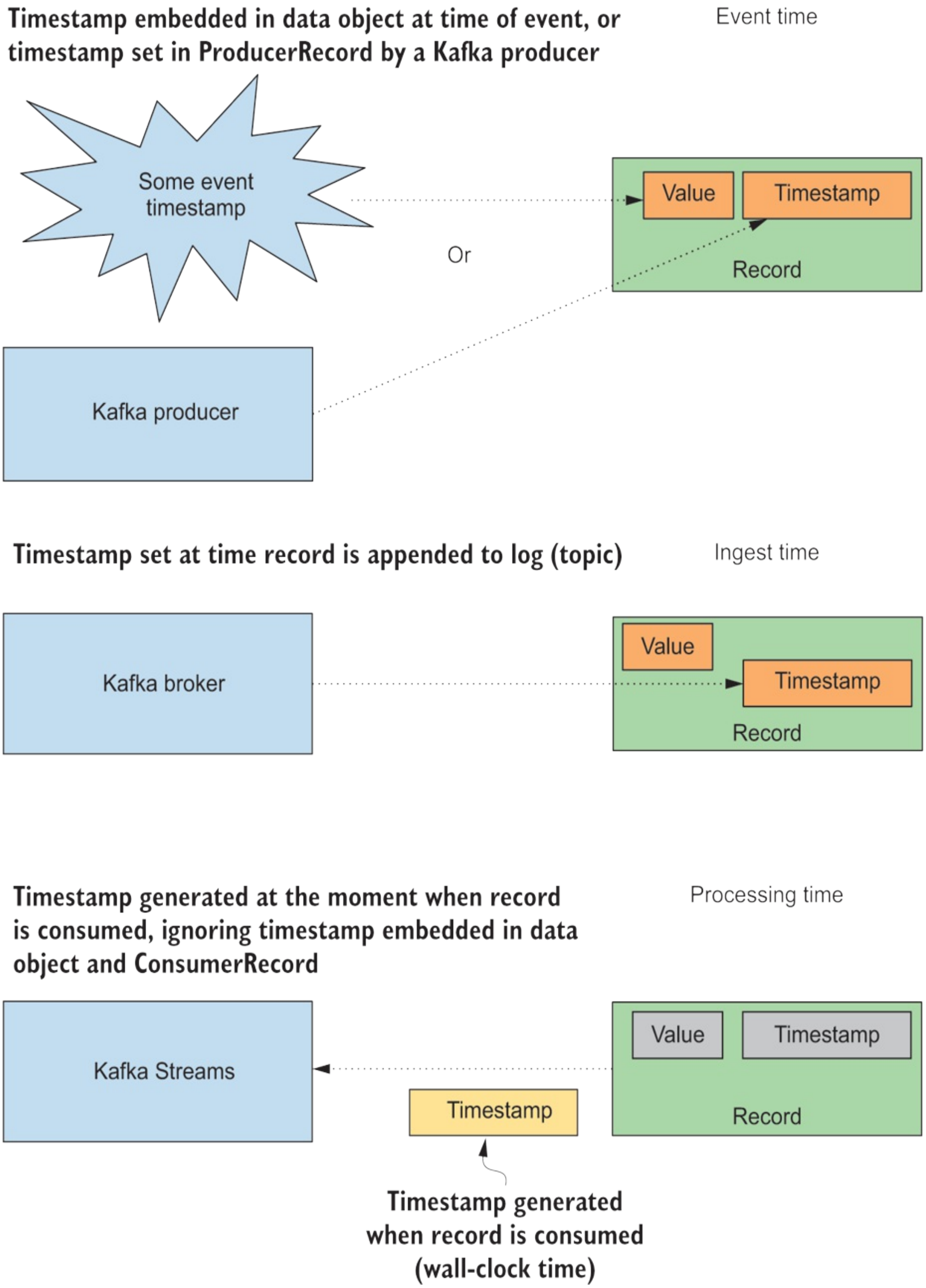
Earlier in the book, we discussed timestamps in Kafka records. In this section, we’ll discuss the use of timestamps in Kafka Streams. Timestamps play a role in key areas of Kafka Streams functionality:

- 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.



Timestamp embedded in data object at time of event, or timestamp set in ProducerRecord by a Kafka producer

Timestamp set at time record is appended to log (topic)

Timestamp generated at the moment when record is consumed, ignoring timestamp embedded in data object and ConsumerRecord

Timestamp generated when record is consumed (wall-clock 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:

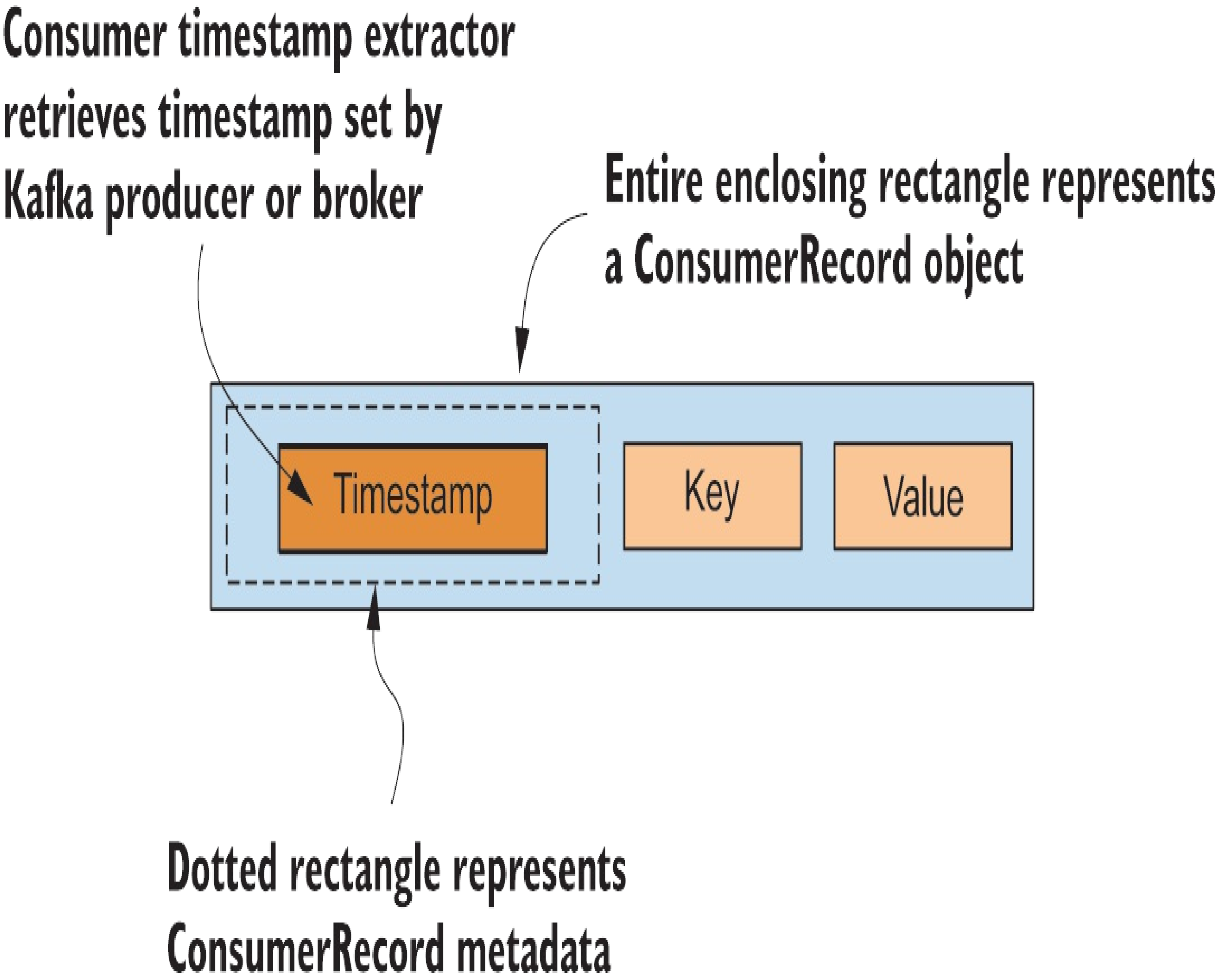
- You're sending messages to Kafka with events that have timestamps recorded in the message objects. There's some lag time in when these event objects are made available to the Kafka producer, so you want to consider only the embedded timestamp.
- You want to consider the time when your Kafka Streams 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.


Consumer timestamp extractor retrieves timestamp set by Kafka producer or broker
Entire enclosing rectangle represents a ConsumerRecord object
Timestamp
Key
Value
Dotted rectangle represents ConsumerRecord metadata

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.

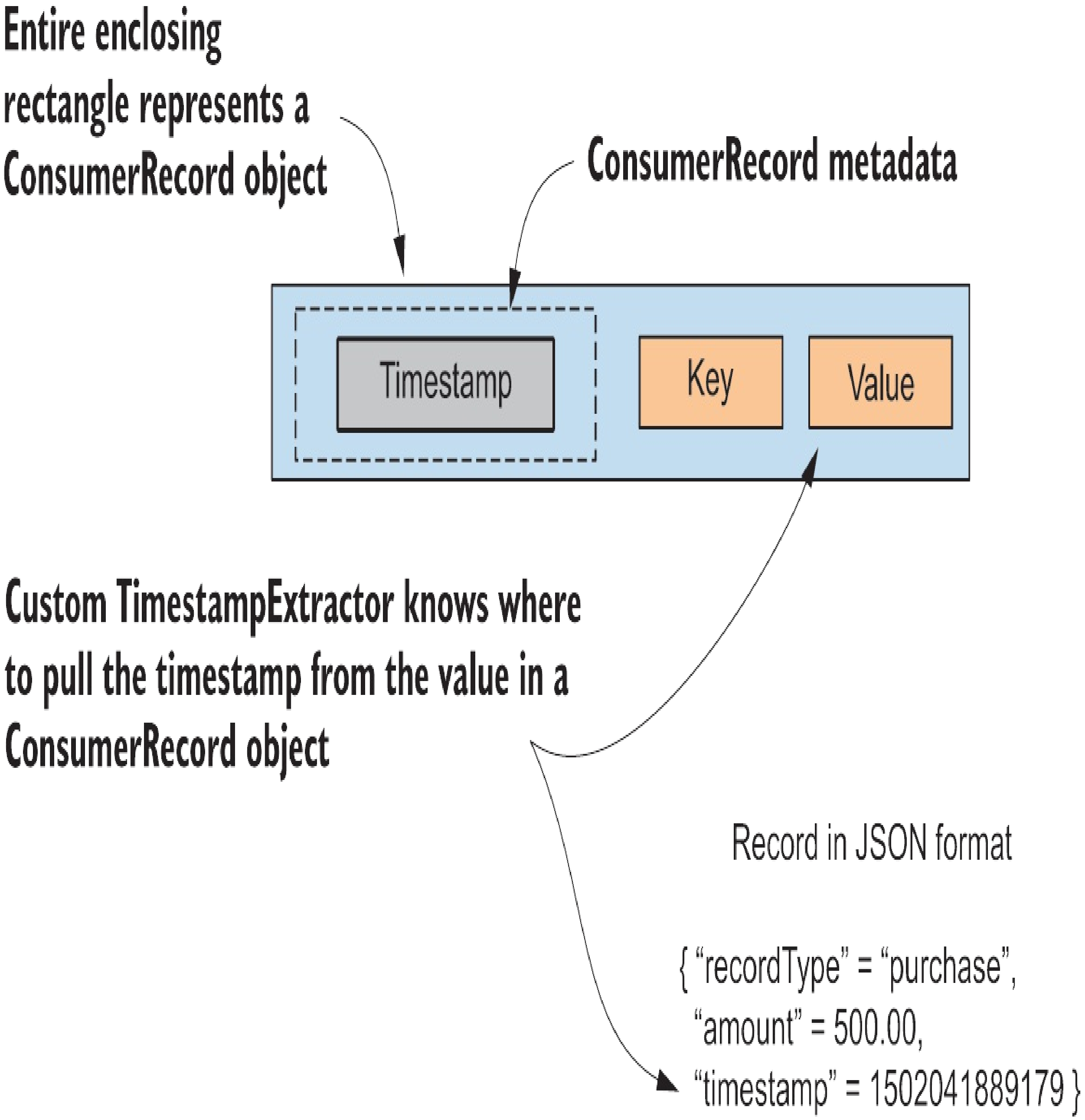
That’s it for the provided timestamp extractors. Next, we’ll look at how you can create a custom version.

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.


Entire enclosing rectangle represents a ConsumerRecord object
ConsumerRecord metadata
Timestamp
Key
Value
Custom TimestampExtractor knows where to pull the timestamp from the value in a ConsumerRecord object
Record in JSON format
{ "recordType" = "purchase",
"amount" = 500.00,
"timestamp" = 1502041889179 }

Here's an example of a `TimestampExtractor` implementation (found in src/main/ java/bbejeck/chapter_4/timestamp_extractor/TransactionTimestampExtractor.

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 TimestampEx

    @Override
    public long extract(ConsumerRecord<Object, Object> record,
         long previousTimestamp) {
        Purchase purchaseTransaction = (Purchase) record.value();
        return purchaseTransaction.getPurchaseDate().getTime();#2
    }
}
```

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 TransactionTimestampExtracto
```

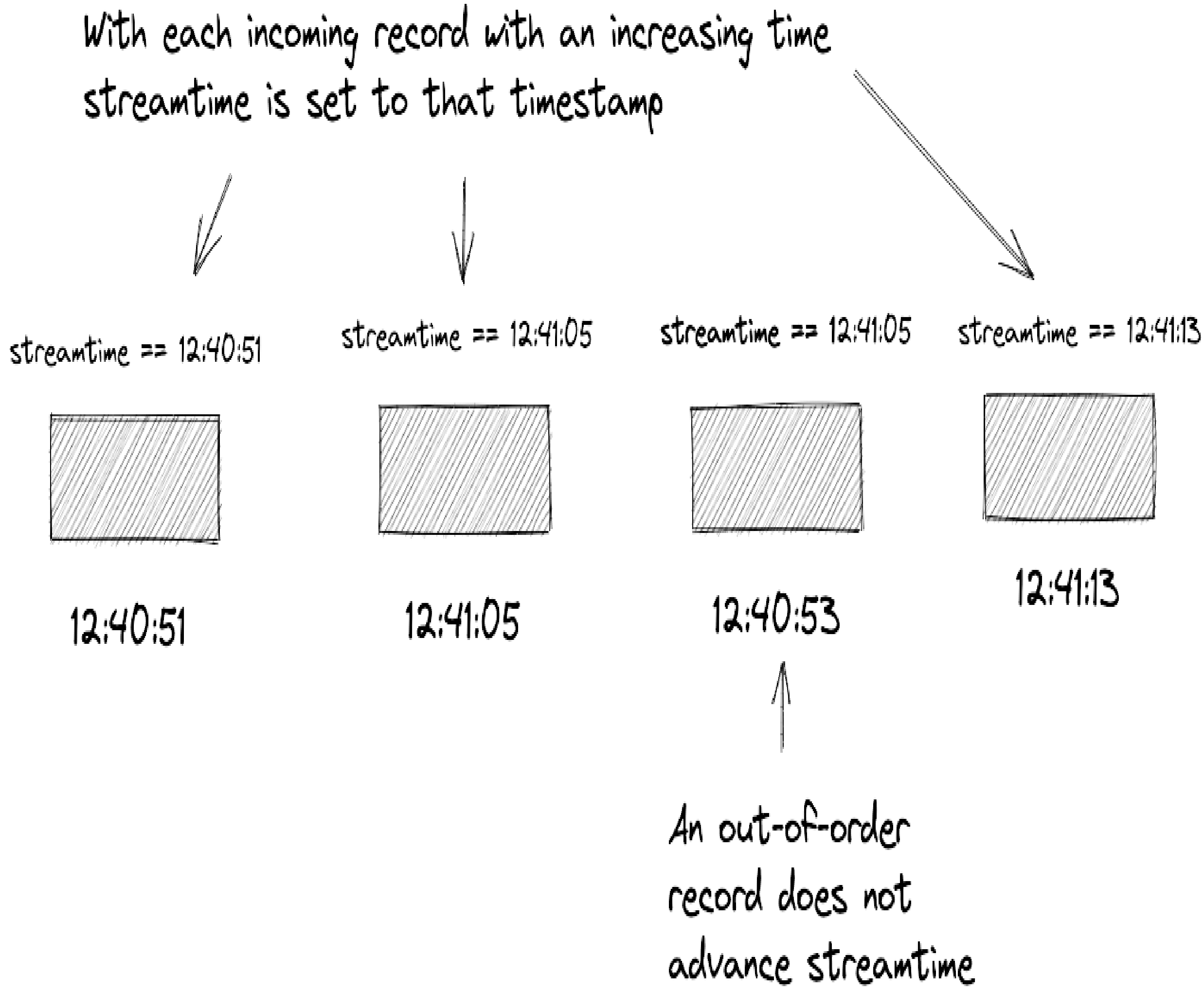
The advantage of doing this is that you have one `TimestampExtractor` per input source, whereas the other option provides a `TimestampExtractor` instance used application wide.

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


With each incoming record with an increasing time streamtime is set to that timestamp
streamtime == 12:40:51
streamtime == 12:41:05
streamtime == 12:41:05
streamtime == 12:41:13
12:40:51
12:41:05
12:40:53
12:41:13
An out-of-order record does not advance streamtime

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 `KTables` and you can also do a foreign key join between two `KTables`
- 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.

[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).

9 The Processor API

This chapter covers

- Evaluating higher-level abstractions versus more control
- Working with sources, processors, and sinks to create a topology
- Digging deeper into the Processor API with a stock analysis processor
- Creating a co-grouping processor
- Integrating the Processor API and the Kafka Streams API

Up to this point in the book, we’ve been working with the high-level Kafka Streams API. It’s a DSL that allows developers to create robust applications with minimal code. The ability to quickly put together processing topologies is an important feature of the Kafka Streams DSL. It allows you to iterate quickly to flesh out ideas for working on your data without getting bogged down in the intricate setup details that some other frameworks may need.

But at some point, even when working with the best of tools, you’ll come up against one of those one-off situations: a problem that requires you to deviate from the traditional path. Whatever the particular case may be, you need a way to dig down and write some code that just isn’t possible with a higher-level abstraction.

9.1 The trade-offs of higher-level abstractions vs. more control

A classic example of trading off higher-level abstractions versus gaining more control is using object-relational mapping (ORM) frameworks. A good ORM framework maps your domain objects to database tables and creates the correct SQL queries for you at runtime. When you have simple-to-moderate SQL operations (simple `SELECT` or `JOIN` statements), using the ORM framework saves you a lot of time. But no matter how good the ORM framework is, there will inevitably be those few queries (very complex joins, `SELECT` statements with nested subselect statements) that just don’t work the way you want. You need to write raw SQL to get the information from the database in the format you need. You can see the trade-off between a higher-level abstraction versus more programmatic control here. Often, you'll be able to mix the raw SQL with the higher-level mappings provided with the framework.

This chapter is about those times when you want to do stream processing in a way that the Kafka Streams DSL doesn't make easy. For example, you've seen from working with the `KTable` API that the framework controls the timing of forwarding records downstream. You may find yourself in a situation where you want explicit control over when a record is sent. You might be tracking trades on Wall Street, and you only want to forward records when a stock crosses a particular price threshold. To gain this type of control, you can use the Processor API. What the Processor API lacks in ease of development, it makes up for in power. You can write custom processors to do almost anything you want.

In this chapter, you'll learn how to use the Processor API to handle situations like these:

- Schedule actions to occur at regular intervals (either based on timestamps in the records or wall-clock time).
- Gain full control over when records are sent downstream.
- Forward records to specific child nodes.
- Create functionality that doesn't exist in the Kafka Streams API (you'll see an example of this when we build a co-grouping processor).

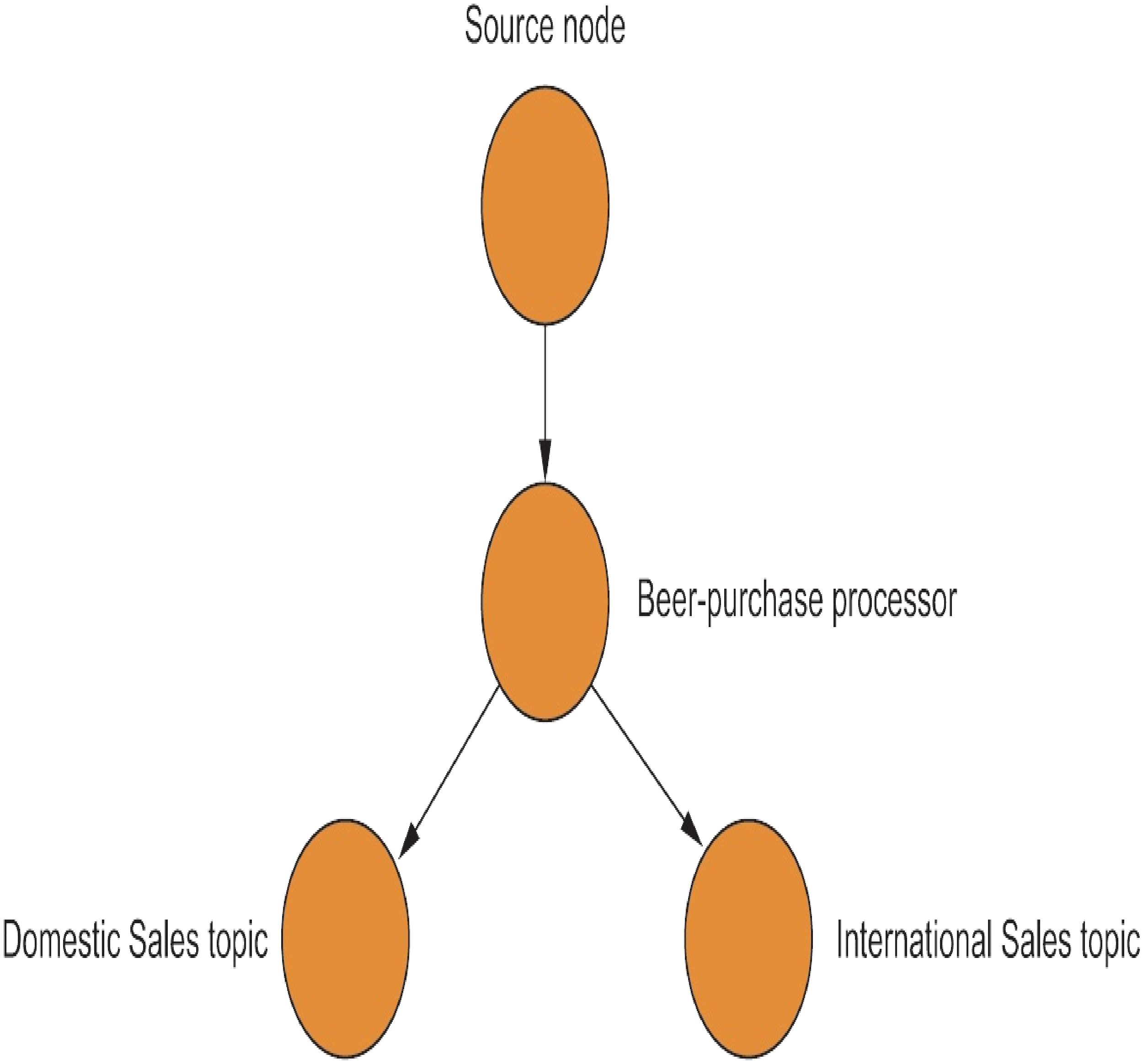
First, let's look at how to use the Processor API by developing a topology step by step.

9.2 Working with sources, processors, and sinks to create a topology

Let's say you're the owner of a successful brewery (Pops Hops) with several locations. You've recently expanded your business to accept online orders from distributors, including international sales to Europe. You want to route orders within the company based on whether the order is domestic or international, converting any European sales from British pounds or euros to US dollars.

If you were to sketch out the flow of operation, it would look something like figure 9.1. In building this example, you'll see how the Processor API gives you the flexibility to select specific child nodes when forwarding records. Let's start by creating a source node

Figure 9.1. Beer sales distribution pipeline



9.2.1 Adding a source node

The first step in constructing a topology is establishing the source nodes. The following listing (found in src/main/java/bbejeck/chapter_9/PopsHopsApplication.java) sets the data source for the new topology.

Listing 9.1. Creating the beer application source node

```
topology.addSource(LATEST,                                 #1
                   purchaseSourceNodeName,                         #2
                   new UsePartitionTimeOnInvalidTimestamp(),
                   stringDeserializer,                             #4
                   beerPurchaseDeserializer,                       #5
                   INPUT_TOPIC) #6
```

In the `Topology.addSource()` method, there are some parameters you didn't use in the DSL. First, you name the source node. When you used the Kafka Streams DSL, you didn't need to pass in a name because the `KStream` instance generated a name for the node. But when you use the Processor API, you need to provide the names of the nodes in the topology. The node name is used to wire up a child node to a parent node.

Next, you specify the timestamp extractor to use with this source. Here, you're using the `UsePartitionTimeOnInvalidTimestamp` class which if the incoming timestamp is invalid (a negative number), it will set it to be the highest value so far (stream time) for the partition of the incoming record.

Next, you provide a key deserializer and a value deserializer, which represents another departure from the Kafka Streams DSL. In the DSL, you supplied `Serde` instances when creating source or sink nodes. The `Serde` itself contains a serializer and deserializer, and the Kafka Streams DSL uses the appropriate one, depending on whether you're going from object to byte array, or from byte array to object. Because the Processor API is a lower-level abstraction, you directly provide a deserializer when creating a source node and a serializer when creating a sink node. Finally, you provide the name of the source topic.

Let's next look at how you'll add processor nodes that do something with the incoming records.

9.2.2 Adding a processor node

Now, you’ll add a processor to work with the records coming in from the source node (found in src/main/java/bbejeck/chapter_9/PopsHopsApplication.java). Let’s first discuss how to wire up the processors, then we’ll cover the functionality of the added processor. .Adding a processor node

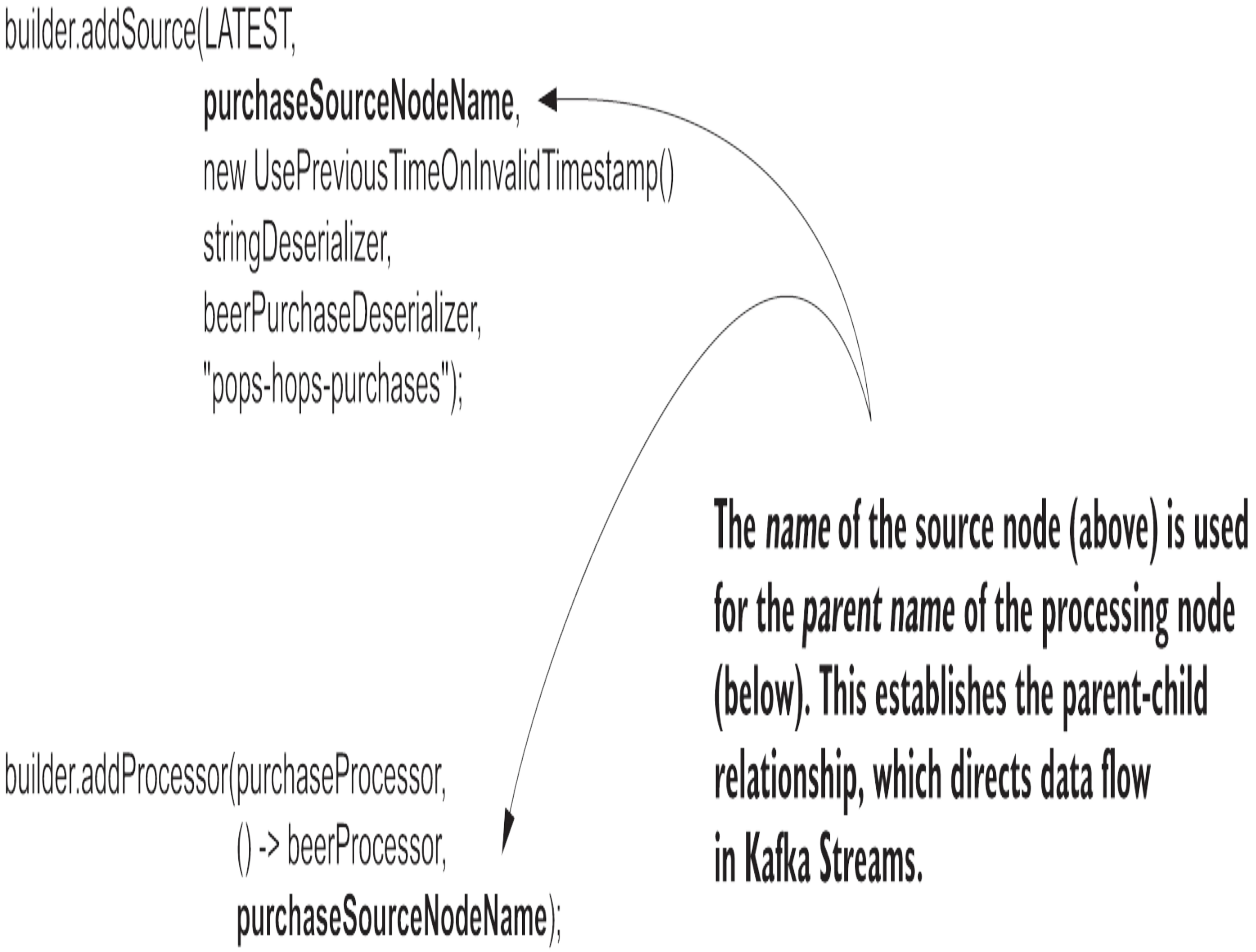
```
//Some details omitted for clarity
 Map<String, Double> conversionRates = Map.of("EURO", 1.1, "POUND

topology.addSource(LATEST,
                  purchaseSourceNodeName,
                  new UsePreviousTimeOnInvalidTimestamp(),
                  stringDeserializer,
                  beerPurchaseDeserializer,
                  INPUT_TOPIC)
       .addProcessor(purchaseProcessor, #1
                     () -> new BeerPurchaseProcessor( #2
                                            domesticSalesSink, #3
                                            internationalSalesSin
                                            conversionRates),
                    purchaseSourceNodeName) #4
```

This code uses the fluent interface pattern for constructing the topology. The difference from the Kafka Streams API lies in the return type. With the Kafka Streams API, every call on a `KStream` operator returns a new `KStream` or `KTable` instance. In the Processor API, each call to `Topology` returns the same `Topology` instance.

In the second annotation, you pass in a `ProcessorSupplier`. The `Topology.addProcessor` method takes an instance of a `ProcessorSupplier` interface for the second parameter, but because the `ProcessorSupplier` is a single-method interface, you can replace it with a lambda expression. The third annotation highlights

The key point in this section is that the third parameter, `purchaseSourceNodeName`, of the `addProcessor()` method is the same as the second parameter of the `addSource()` method, as illustrated in figure 9.2. This establishes the parent-child relationship between nodes. The parent-child relationship, in turn, determines how records move from one processor to the next in a Kafka Streams application. Figure 9.3 reviews what you’ve built so far.

Figure 9.2. Wiring up parent and child nodes in the Processor API

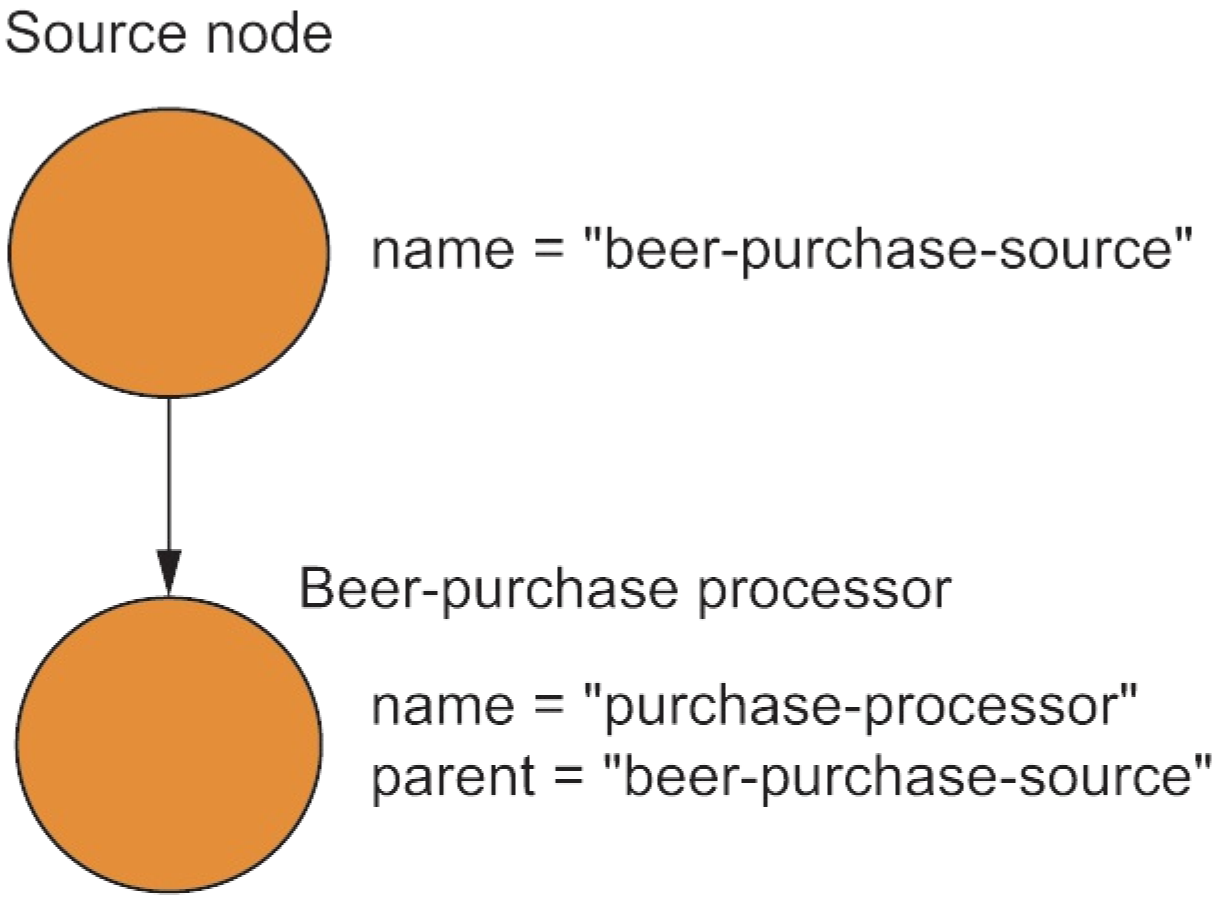


```
builder.addSource(LATEST,
                  purchaseSourceNodeName,
                  new UsePreviousTimeOnInvalidTimestamp()
                  stringDeserializer,
                  beerPurchaseDeserializer,
                  "pops-hops-purchases");
```

The *name* of the source node (above) is used for the *parent name* of the processing node (below). This establishes the parent-child relationship, which directs data flow in Kafka Streams.

```
builder.addProcessor(purchaseProcessor,
                     () -> beerProcessor,
                     purchaseSourceNodeName);
```

Figure 9.3. The Processor API topology so far, including node names and parent names


Source node
name = "beer-purchase-source"
Beer-purchase processor
name = "purchase-processor"
parent = "beer-purchase-source"

Let's take a second to discuss the `BeerPurchaseProcessor`, created in listing 9.1, functions. The processor has two responsibilities:

- Convert international sales amounts (in euros) to US dollars.
- Based on the origin of the sale (domestic or international), route the record to the appropriate sink node.

All of this takes place in the `process()` method. To quickly summarize, here's what the `process()` method does:

1. Check the currency type. If it's not in dollars, convert it to dollars.
2. If it's a non-domestic sale, forward the updated record to the `international-sales` topic.
3. Otherwise, forward the record directly to the `domestic-sales` topic.

Here's the code for this processor (found in src/main/java/bbejeck/chapter_9/processor/BearPurchaseProcessor.java).

Listing 9.2. BeerPurchaseProcessor

```
public class BeerPurchaseProcessor extends
   ContextualProcessor<String, BeerPurchase, String, BeerPurchase

    private final String domesticSalesNode;
    private final String internationalSalesNode;
    private final Map<String, Double> conversionRates;

    public BeerPurchaseProcessor(String domesticSalesNode,
                                 String internationalSalesNode,
                                 Map<String,Double> conversionRat

        this.domesticSalesNode = domesticSalesNode;     #1
        this.internationalSalesNode = internationalSalesNode; #1
        this.conversionRates = conversionRates;
    }

    @Override
    public void process(Record<String, BeerPurchase> beerPurchase

        BeerPurchase beerPurchase = beerPurchaseRecord.value();
        String key  = beerPurchaseRecord.key();
        BeerPurchase.Currency transactionCurrency =
              beerPurchase.getCurrency();

        if (transactionCurrency != BeerPurchase.Currency.DOLLAR)
            BeerPurchase.Builder builder =
                   BeerPurchase.newBuilder(beerPurchase);
            double internationalSaleAmount = beerPurchase.getTota
            String pattern = "###.##";
            DecimalFormat decimalFormat = new DecimalFormat(patte
            builder.setCurrency(BeerPurchase.Currency.DOLLAR);
            builder.setTotalSale(Double.parseDouble(decimalFormat
                   convertToDollars(transactionCurrency.name(),
                   internationalSaleAmount))));         #3
            Record<String, BeerPurchase> convertedBeerPurchaseRec
                       new Record<>(key,builder.build(),
                                    beerPurchaseRecord.timestamp(
            context().forward(convertedBeerPurchaseRecord,   #4
                              internationalSalesNode);
        } else {
            context().forward(beerPurchaseRecord, domesticSalesNo
        }
}
```

This example extends `ContextualProcessor`, a class with overrides for

`Processor` interface methods, except for the `process()` method. The `Processor.process()` method is where you perform actions on the records flowing through the topology.

Note

The `Processor` interface provides the `init()`, `process()`, and `close()` methods. The `Processor` is the main driver of any application logic that works with records in your streaming application. In the examples, you'll usually extend the `ContextualProcessor` class, so you'll only override the methods you want. The `ContextualProcessor` class initializes the `ProcessorContext` for you, so if you don't need to do any setup in your class, you don't need to override the `init()` method.

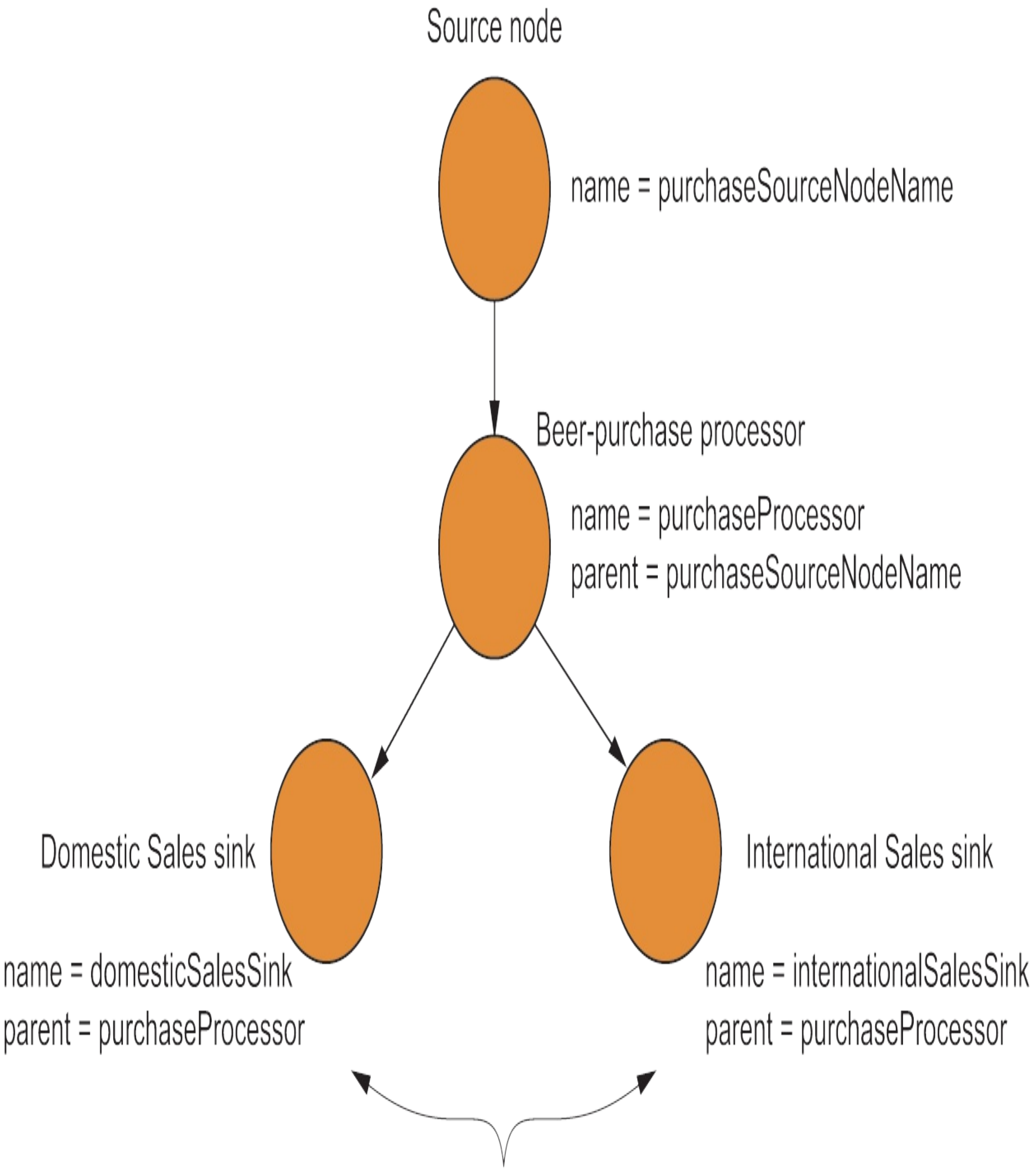
The last few lines of listing 9.3 demonstrate the main point of this example—the ability to forward records to specific child nodes. The `context()` method in these lines retrieves a reference to the `ProcessorContext` object for this processor. All processors in a topology receive a reference to the `ProcessorContext` via the `init()` method, which is executed by the `StreamTask` when initializing the topology.

Now that you've seen how you can process records, the next step is to connect a sink node (topic) so you can write records back to Kafka.

9.2.3 Adding a sink node

By now, you probably have a good feel for the flow of using the Processor API. To add a source, you used `addSource`, and for adding a processor, you used `addProcessor`. As you might imagine, you'll use the `addSink()` method to wire up a sink node (topic) to a processor node. Figure 9.4 shows the updated topology.

Figure 9.4. Completing the topology by adding sink nodes


Source node
name = purchaseSourceNodeName
Beer-purchase processor
name = purchaseProcessor
parent = purchaseSourceNodeName
Domestic Sales sink
name = domesticSalesSink
parent = purchaseProcessor
International Sales sink
name = internationalSalesSink
parent = purchaseProcessor

Note that the two sink nodes here have the same parent.

You can update the topology you're building by adding sink nodes in the code now (found in src/main/java/bbejeck/chapter_9/PopsHopsApplication.java).

Listing 9.3. Adding sink nodes

```
topology.addSource(LATEST,
                   purchaseSourceNodeName,
                   new UsePreviousTimeOnInvalidTimestamp(),
                   stringDeserializer,
                   beerPurchaseDeserializer,
                   INPUT_TOPIC)
       .addProcessor(purchaseProcessor,
                     () -> new BeerPurchaseProcessor(
                                            domesticSalesSink,
                                            internationalSalesSin
                                            conversionRates),
                    purchaseSourceNodeName)

       .addSink(internationalSalesSink,   #1
                "international-sales",    #2
                stringSerializer,         #3
                beerPurchaseSerializer,  #3
                purchaseProcessor)       #4

       .addSink(domesticSalesSink,      #5
                "domestic-sales",      #6
                stringSerializer,        #7
                beerPurchaseSerializer,  #8
                purchaseProcessor);     #9
```

In this listing, you add two sink nodes, one for domestic sales and another for international. Depending on the currency of the transaction, you’ll write the records out to the appropriate topic.

The key point to notice when adding two sink nodes here is that both have the same parent name. By supplying the same parent name to both sink nodes, you’ve wired both of them to your processor (as shown in figure 6.4).

You’ve seen in this first example how you can wire topologies together and forward records to specific child nodes. Although the Processor API is a little more verbose than the Kafka Streams API, it’s still easy to construct topologies. The next example will explore more of the flexibility the Processor API provides.

9.3 Digging deeper into the Processor API with a stock analysis processor

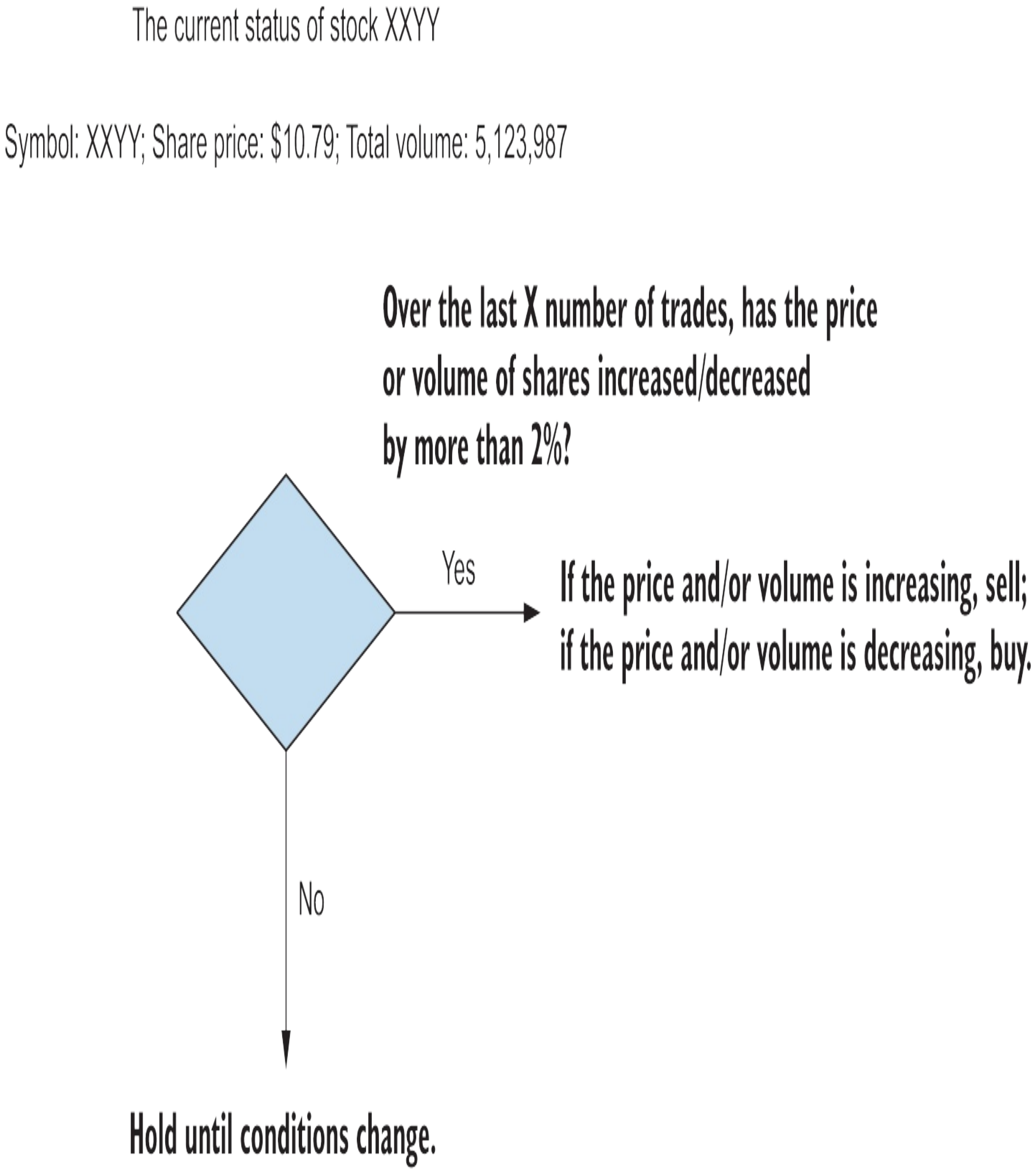
You’ll now return to the world of finance and put on your day trading hat. As a day trader, you want to analyze how stock prices are changing with the intent of picking the best time to buy and sell. The goal is to take advantage of market fluctuations and make a quick profit. We’ll consider a few key indicators, hoping they’ll indicate when you should make a move.

This is the list of requirements:

- Show the current value of the stock.
- Indicate whether the price per share is trending up or down.
- Include the total share volume so far, and whether the volume is trending up or down.
- Only send records downstream for stocks displaying 2% trending (up or down).
- Collect a minimum of 20 samples for a given stock before performing any calculations.

Let’s walk through how you might handle this analysis manually. Figure 6.5 shows the sort of decision tree you’ll want to create to help make decisions.

Figure 9.5. Stock trend updates


The current status of stock XXYY
Symbol: XXYY; Share price: $10.79; Total volume: 5,123,987
Over the last X number of trades, has the price or volume of shares increased/decreased by more than 2%?
Yes
If the price and/or volume is increasing, sell; if the price and/or volume is decreasing, buy.
No
Hold until conditions change.

There are a handful of calculations you'll need to perform for your analysis. Additionally, you'll use these calculation results to determine if and when you should forward records downstream.

This restriction on sending records means you can't rely on the standard mechanisms of commit time or cache flushes to handle the flow for you, which rules out using the Kafka Streams API. It goes without saying that you'll also require state, so you can keep track of changes over time. What you need here is the ability to write a custom processor. Let's look at the solution to the problem.

For demo purposes only

I'm pretty sure it goes without saying, but I'll state the obvious anyway: these stock price evaluations are for demonstration purposes only. Please don't infer any real market-forecasting ability from this example. This model bears no similarity to a real-life approach and is presented only to demonstrate a more complex processing situation. I'm certainly not a day trader!

9.3.1 The stock-performance processor application

Here's the topology for the stock-performance application (found in src/main/java/ bbejeck/chapter_6/StockPerformanceApplication.java).

Listing 9.4. Stock-performance application with custom processor

```
Topology topology = new Topology();
 String stocksStateStore = "stock-performance-store";
 double differentialThreshold = 0.02;  #1

KeyValueBytesStoreSupplier storeSupplier =
 Stores.inMemoryKeyValueStore(stocksStateStore); #2
StoreBuilder<KeyValueStore<String, StockPerformance>> storeBuilde
 = Stores.keyValueStoreBuilder(
 storeSupplier, Serdes.String(), stockPerformanceSerde);  #3


  topology.addSource("stocks-source",
                    stringDeserializer,
                    stockTransactionDeserializer,
                    "stock-transactions")
          .addProcessor("stocks-processor",  #4
                    new StockPerformanceProcessorSupplier(storeBu
                    "stocks-source")
          .addSink("stocks-sink",
                   "stock-performance",
                   stringSerializer,
                   stockPerformanceSerializer,
                        "stocks-processor");  #5
```

You should take note that in this example we’ve used a concrete implementation of the `ProcessorSupplier` instead of a lambda expression at annotation four. This is because the `ProcessorSupplier` interface provides a `stores` method that will automatically wire up the processor with any `StoreBuilder` instances you provide. Here’s a look at the `StockPerformanceProcessorSupplier` source code:

Listing 9.5. ProcessorSupplier impementation

```
public class StockPerformanceProcessorSupplier
       implements ProcessorSupplier<String, Transaction,
                                    String, StockPerformance> {
    StoreBuilder<?> storeBuilder;

    public StockPerformanceProcessorSupplier(StoreBuilder<?> stor
        this.storeBuilder = storeBuilder;
    }

    @Override
    public Processor<String, Transaction, String, StockPerformanc
        return new StockPerformanceProcessor(storeBuilder.name())
    }

    @Override
    public Set<StoreBuilder<?>> stores() {
        return Collections.singleton(storeBuilder); #2
    }
}
```

With the `ProcessorSupplier.stores` method, you have a way to automatically wire up `StateStore` instances to processors, which makes building topology a bit simpler as you don’t need to call `Topology.addStateStore` with the names of the processors having access to the store.

Since this stock performance topology has the same flow as the previous example, we’ll focus on the new features in the processor. In the previous example, you don’t have any setup to do, so you rely on the `ContextualProcessor.init` method to initialize the `ProcessorContext` object. In this example, however, you need to use a state store, and you also want to schedule when you emit records, instead of forwarding records each time you receive them.

Let’s look first at the `init()` method in the processor (found in src/main/java/bbejeck/chapter_6/processor/StockPerformanceProcessor.java).

`init()` method tasks

```
@Override
public void init(ProcessorContext<String, StockPerformance> conte
  super.init(processorContext);                          #1
  keyValueStore = context().getStateStore(stateStoreName);   #2
  StockPerformancePunctuator punctuator =
   new StockPerformancePunctuator(differentialThreshold,
                               context(),
                               keyValueStore); #3
  context().schedule(10000,
                      PunctuationType.STREAM_TIME,
                      punctuator);   #4
 }
}
```

First, you need to initialize the `ContextualProcessor` with the `ProcessorContext`, so you call the `init()` method on the superclass. Next, you grab a reference to the state store you created in the topology. All you need to do here is set the state store to a variable for use later in the processor. Listing 9.5 also introduces a `Punctuator`, an interface that’s a callback to handle scheduled execution of processor logic but encapsulated in the `Punctuator.punctuate` method.

Tip

The `ProcessorContext.schedule(Duration, PunctuationType, Punctuator)` method returns a type of `Cancellable`, allowing you to cancel a punctuation and manage more-advanced scenarios, like those found in the “Punctuate Use Cases” discussion (mng.bz/YSKF). I don’t have examples or a discussion here, but I present some examples in src/main/java/bbejeck/chapter_9/cancellation.

In the last line of listing 9.5, you use the `ProcessorContext` to schedule the

`Punctuator` to execute every 10 seconds. The second parameter, `PunctuationType.WALL_CLOCK_TIME` , specifies that you want to call `Punctuator.punctuate` every 10 seconds based on the timestamps of the data. Your other option is to specify `PunctuationType.WALL_CLOCK_TIME`, which means the execution of `Punctuator.punctuate` is scheduled every 10 seconds but driven by the system time of the Kafka Streams environment. Let's take a moment to discuss the difference between these two `Punctuation-Type` settings.

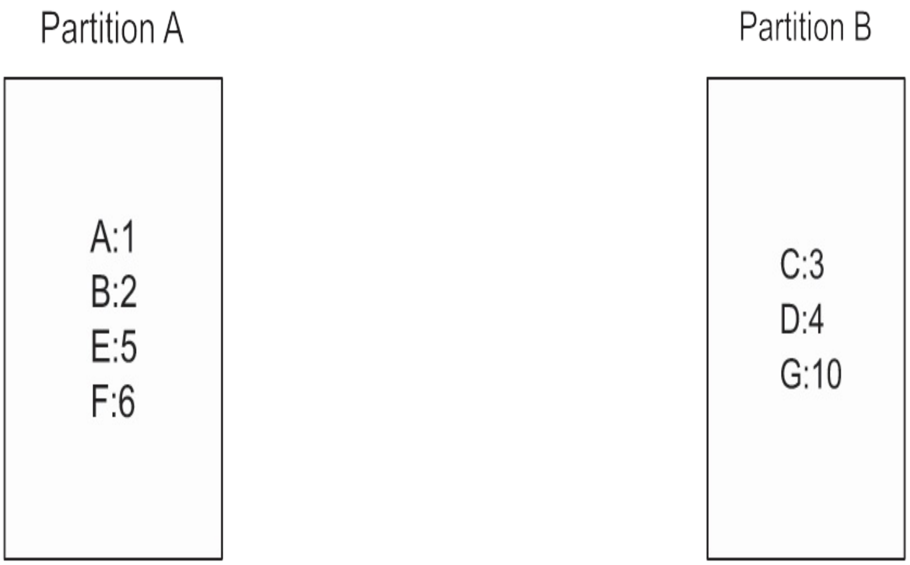
Punctuation semantics

Let's start our conversation on punctuation semantics with `STREAM_TIME`, because it requires a little more explanation. Figure 6.6 illustrates the concept of stream time. Let's walk through some details to gain a deeper understanding of how the schedule is determined (note that some of the Kafka Stream internals are not shown):

1. The `StreamTask` extracts the *smallest* timestamp from the `PartitionGroup`. The `PartitionGroup` is a set of partitions for a given `StreamThread`, and it contains all timestamp information for all partitions in the group.
2. During the processing of records, the `StreamThread` iterates over its `StreamTask` object, and each task will end up calling `punctuate` for each of its processors that are eligible for punctuation. Recall that you collect a minimum of 20 trades before you examine an individual stock's performance.
3. If the timestamp from the last execution of `punctuate` (plus the scheduled time) is less than or equal to the extracted timestamp from the `Partition-Group`, then Kafka Streams calls that processor's `punctuate()` method.

Figure 9.6. Punctuation scheduling using `STREAM_TIME`

In the two partitions below, the letter represents the record, and the number is the timestamp. For this example, we'll assume that `punctuate` is scheduled to run every five seconds.



Partition A

A:1
B:2
E:5
F:6

Partition B

C:3
D:4
G:10

Because partition A has the smallest timestamp, it's chosen first:
1) Process called with record A
2) Process called with record B

Now partition B has the smallest timestamp:
3) Process called with record C
4) Process called with record D

Switch back to partition A, which has the smallest timestamp again:
5) Process called with record E
6) punctuate called because time elapsed from timestamps is 5 seconds
7) Process called with record F

Finally, switch back to partition B:
8) Process called with record G
9) punctuate called again as 5 more seconds have elapsed, according to the timestamps

The key point here is that the application advances timestamps via the `TimestampExtractor`, so `punctuate()` calls are consistent only if data arrives at a constant rate. If your flow of data is sporadic, the `punctuate()` method won’t get executed at the regularly scheduled intervals.

With `PunctuationType.WALL_CLOCK_TIME`, on the other hand, the execution of `Punctuator.punctuate` is more predictable, as it uses wall-clock time. Note that system-time semantics is best effort—wall-clock time is advanced in the polling interval, and the granularity is dependent on how long it takes to complete a polling cycle. So, with the example in listing 9.6, you can expect the punctuation activity to be executed closer to every 10 seconds, regardless of data activity.

Which approach you choose to use is entirely dependent on your needs. If you need some activity performed on a regular basis, regardless of data flow, using system time is probably the best bet. On the other hand, if you only need calculations performed on incoming data, and some lag time between executions is acceptable, try stream-time semantics.

Note

Before Kafka 0.11.0, punctuation involved the `ProcessorContext.schedule(long time)` method, which in turn called the `Processor.punctuate` method at the scheduled interval. This approach only worked on stream-time semantics, and both methods are now deprecated. I mention deprecated methods in this book, but I only use the latest punctuation methods in the examples.

Now that we’ve covered scheduling and punctuation, let’s move on to handling incoming records.

9.3.2 The process() method

The `process()` method is where you’ll perform all of your calculations to evaluate stock performance. There are several steps to take when you receive a record:

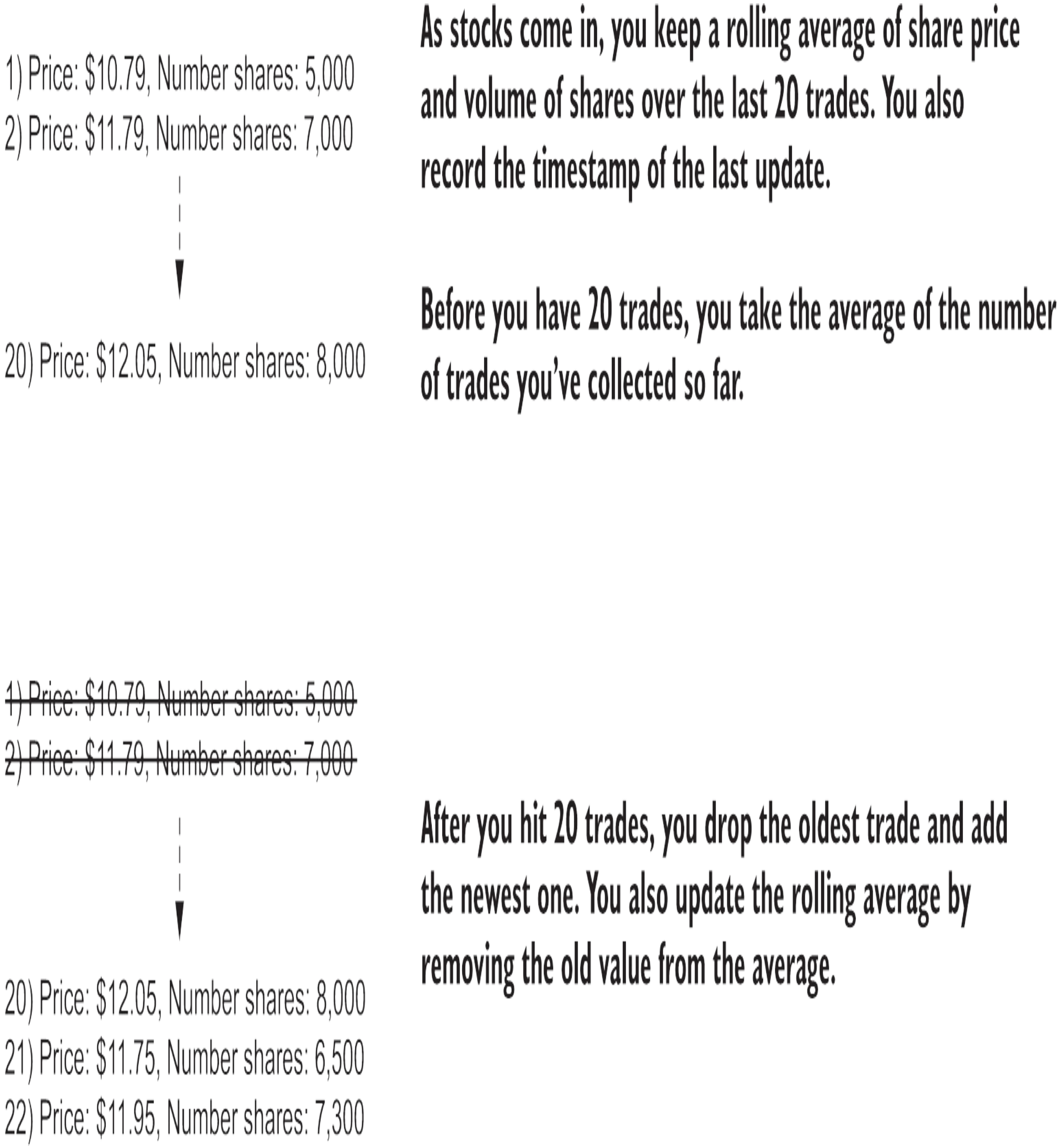
1. Check the state store to see if you have a corresponding `StockPerformance` object for the record’s stock ticker symbol.
2. If the store doesn’t contain the `StockPerformance` object, one is created. Then, the `StockPerfomance` instance adds the current share price and share volume and updates your calculations.
3. Start performing calculations once you hit 20 transactions for any given stock.

Although financial analysis is beyond the scope of this book, we should take a minute to look at the calculations. For both the share price and volume, you’re going to perform a simple moving average (SMA). In the financial-trading world, SMAs are used to calculate the average for datasets of size *N*.

For this example, you’ll set *N* to 20. Setting a maximum size means that as new trades come in, you collect the share price and number of shares traded for the first 20 transactions. Once you hit that threshold, you remove the oldest value and add the latest one. Using the SMA, you get a rolling average of stock price and volume over the last 20 trades. It’s important to note you won’t have to recalculate the entire amount as new values come in.

Figure 9.7 provides a high-level walk-through of the `process()` method, illustrating what you’d do if you were to perform these steps manually. The `process()` method is where you’ll perform all the calculations.

Figure 9.7. Stock analysis `process()` method walk-through


1) Price: $10.79, Number shares: 5,000
2) Price: $11.79, Number shares: 7,000
20) Price: $12.05, Number shares: 8,000
~~1) Price: $10.79, Number shares: 5,000~~
~~2) Price: $11.79, Number shares: 7,000~~
20) Price: $12.05, Number shares: 8,000
21) Price: $11.75, Number shares: 6,500
22) Price: $11.95, Number shares: 7,300

As stocks come in, you keep a rolling average of share price and volume of shares over the last 20 trades. You also record the timestamp of the last update.

Before you have 20 trades, you take the average of the number of trades you've collected so far.

After you hit 20 trades, you drop the oldest trade and add the newest one. You also update the rolling average by removing the old value from the average.

Now, let's look at the code that makes up the `process()` method (found in src/main/ java/bbejeck/chapter_6/processor/StockPerformanceProcessor.java).

`process()` implementation

```
@Override
public void process(String symbol, StockTransaction transaction)
```

```
 StockPerformance stockPerformance = keyValueStore.get(symbol);

 if (stockPerformance == null) {
  stockPerformance = new StockPerformance();  #2
 }

 stockPerformance.updatePriceStats(transaction.getSharePrice());
 stockPerformance.updateVolumeStats(transaction.getShares());  #4
 stockPerformance.setLastUpdateSent(Instant.now()); #5

 keyValueStore.put(symbol, stockPerformance);  #6
}
```

In the `process()` method, you take the latest share price and the number of shares involved in the transaction and add them to the `StockPerformance` object. Notice that all details of how you perform the updates are abstracted inside the `StockPerformance` object. Keeping most of the business logic out of the processor is a good idea—we’ll come back to that point when we cover testing in chapter 8.

There are two key calculations: determining the moving average, and calculating the differential of stock price/volume from the current average. You don’t want to calculate an average until you’ve collected data from 20 transactions, so you defer doing anything until the processor receives 20 trades. When you have data from 20 trades for an individual stock, you calculate your first average. Then, you take the current value of the stock price or a number of shares and divide by the moving average, converting the result to a percentage.

Note

If you want to see the calculations, the `StockPerformance` code can be found in src/main/java/bejeck/model/StockPerformance.java.

In the `Processor` example in listing 6.3, once you worked your way through the `process()` method, you forwarded the records downstream. In this case, you store the final results in the state store and leave the forwarding of records to the `Punctuator-.punctuate` method.

9.3.3 The punctuator execution

We’ve already discussed the punctuation semantics and scheduling, so let’s jump straight into the code for the `Punctuator.punctuate` method (found in src/main/
java/bejeck/chapter_9/processor/punctuator/StockPerformancePunctuator.java

Listing 9.6. Punctuation code

```
@Override
public void punctuate(long timestamp) {
    try (KeyValueIterator<String, StockPerformance> performanceIt
                           = keyValueStore.all()) { #1

        while (performanceIterator.hasNext()) {
            KeyValue<String, StockPerformance> keyValue =
                            performanceIterator.next();
            String key = keyValue.key;
            StockPerformance stockPerformance = keyValue.value;

            if (stockPerformance != null) {
                if (stockPerformance.getPriceDifferential() >=
                                  differentialThreshold ||
                        stockPerformance.getShareDifferential() >
                                    differentialThreshold) { #2
                    context.forward(new Record<>(key,           #
                                                 stockPerformance
                                                 timestamp));
                }
            }
        }
    }
}
```

The procedure in the `Punctuator.punctuate` method is simple. You iterate over the key/value pairs in the state store, and if the value has crossed over the predefined threshold, you forward the record downstream.

An important concept to remember here is that, whereas before you relied on a combination of committing or cache flushing to forward records, now you define the terms for when records get forwarded. Additionally, even though you expect to execute this code every 10 seconds, that doesn’t guarantee you’ll emit records. They must meet the differential threshold. Also note that the `Processor.process` and `Punctuator .punctuate` methods aren’t called concurrently.

ⓘ **Note**

Although we’re demonstrating access to a state store, it’s a good time to review Kafka Streams’ architecture and go over a few key points. Each `StreamTask` has its *own* copy of a *local* state store, and `StreamThread` objects don’t share tasks or data. As records make their way through the topology, each node is visited in a depth-first manner, meaning there’s never concurrent access to state stores from any given processor.

This example has given you an excellent introduction to writing a custom processor. You can also take writing custom processors a bit further by creating an entirely new way of aggregating data that doesn’t currently exist in the API. With this in mind, we’ll move on to adding data-driven aggregation.

9.4 Data Driven Aggregation

We discussed aggregation in chapter 7 when we covered stateful operations in Kafka Streams. But imagine you have some different requirements for creating an aggregation, specifically instead of having windows based on time, you want to have a "window" based on certain aspects of the incoming events. You’ll want to only include events that meet a given criteria in an aggregation, and forward the results once an incoming record doesn’t meet that criteria any longer.

For example, let’s say you are responsible for a production line at a major manufacturer, Waldo Widgets. In an effort to control costs and improve efficiency, you installed several sensors that continually send information about different important indicators in the manufacturing process. All the sensors send their information into Kafka. You’ve determined over time that the temperature sensors are one of the best leading indicator for trouble in the manufacturing process. It seems that prolonged temparature spikes are almost always followed by a costly shutdown of the production line, sometimes for hours until a machine can be serviced or in the worst case, replaced.

So you’ve determined that you’ll need to create a Kafka Streams application to help with the monitoring of the temperature sensor readings. From your experience you’ve developed some requirements of the exact information you really need. What you’ve been able to put together over the years is that when a machine is about to have trouble, it will start with smaller spikes in temperature reading leading to progressively longer periods of increased temperature over time.

So what you need is an aggregation of only the increased temperatures, getting a constant flow of all temperature readings is actually conterproductive, it’s simply too much information to be useful otherwise. You spend some time coming up the precise requirements you want from your new application:

- If the temperature is below a given threshold, simply ignore it
- Once it rises above the threshold, start an aggregation window
- Continue the aggregation as long as the readings exceed the threshold
- When the readings reach a given number emit the aggregation
- Once a reading drops below the threshold cease the aggregation and immediately emit the result
- Since the sensors have a known history of spotty network connections, if more than 10 seconds elapse without either a closing reading or an additonal high reading go ahead and emit the aggregation

By using this approach, the dashboard appliction will display the information your team will need to take action. You start looking at the Kafka Streams DSL API, and you’re naturally drawn to the windowed aggregations. From reading it seems what you’ll like to use is an aggregation with `SessionWindow` as it’s driven by behavior instead of time. But you really need the fine grained control of what to include in the aggregation as well as when to emit the results. So you turn your attention to the Processor API and decide to write your own "DataDrivenWinowing" behavior.

You’ve built Kafka Streams Processor API applications before, so you dive right in and start on the `Processor` implementation you’ll need to complete the task. Here’s the start of the `ProcessorSupplier` you’ve come up with:

Listing 9.7. ProcessorSupplier

```
public class DataDrivenAggregate implements
        ProcessorSupplier<String, Sensor, String, SensorAggregati

    private final StoreBuilder<?> storeBuilder; #1
    private final Predicate<Sensor> shouldAggregate; #2
    private final Predicate<Sensor> stopAggregation; #3

    public DataDrivenAggregate(final StoreBuilder<?> storeBuilder
                               final Predicate<Sensor> shouldAggregat
                               final Predicate<Sensor> stopAggregatio
      this.storeBuilder = storeBuilder;
      this.shouldAggregate = shouldAggregate;
      this.stopAggregation = stopAggregation;
    }
```

So when creating the `ProcessorSupplier` you provide two `Predicate` instances used to determine when to start an aggregation or include a record in the current aggregation. The second one determines when to reject a record from the aggregation which in turn shuts the aggregation off. Let’s now take a look at the `Processor` implementation that the supplier returns.

Listing 9.8. Processor Implementation

```
private class DataDrivenAggregateProcessor extends
        ContextualProcessor<String, Sensor, String, SensorAggrega

    KeyValueStore<String, SensorAggregation> store;
    long lastObservedStreamTime = Long.MIN_VALUE;

  @Override
  public void init(ProcessorContext<String, SensorAggregation> co
        super.init(context);
        store = context().getStateStore(storeBuilder.name()); #1
        context().schedule(Duration.ofSeconds(10),            #2
                           PunctuationType.WALL_CLOCK_TIME,
                           this::cleanOutDanglingAggregations);
    }
```

Initializing the `Processor` is straight forward and it looks very similar to the other processor implementations you’ve built before. The interesting logic occurs however in the `process` method:

Listing 9.9. DataDrivenAggregateProcessor process method implementation

```
@Override
public void process(Record<String, Sensor> sensorRecord) {
    lastObservedStreamTime =
         Math.max(lastObservedStreamTime, sensorRecord.timestamp(
    SensorAggregation sensorAgg = store.get(sensorRecord.key());
    SensorAggregation.Builder builder;
    boolean shouldForward = false;

    if (shouldAggregate.test(sensorRecord.value())) {  #2
        if (sensorAgg == null) {
            builder = SensorAggregation.newBuilder();
            builder.setStartTime(sensorRecord.timestamp());
            builder.setSensorId(sensorRecord.value().getId());
        } else {
            builder = sensorAgg.toBuilder();
        }
        builder.setEndTime(sensorRecord.timestamp());
        builder.addReadings(sensorRecord.value().getReading());
        builder.setAverageTemp(builder.getReadingsList()
                                      .stream()
                                      .mapToDouble(num -> num)
                                      .average()
                                      .getAsDouble());
        sensorAgg = builder.build();
        shouldForward  =
              sensorAgg.getReadingsList().size() % emitCounter ==
        store.put(sensorRecord.key(), sensorAgg);

    } else if (stopAggregation.test(sensorRecord.value()) #4
                          && sensorAgg != null) {
        store.delete(sensorRecord.key());
        shouldForward = true;
    }

    if (shouldForward) {    #5
        context().forward(new Record<>(sensorRecord.key(),
                                       sensorAgg,
                                       lastObservedStreamTime));
    }
}
```

Since the logic for the aggegation is straight forward I’m only going to discuss the main points of this method indicated by the annotations. At the first annotation we’ll update the `lastObservedStreamTime` variable. Stream time only moves forward, so we don’t blindly accept the timestamp from the incoming record, instead we’ll reuse the existing stream time value should an out of order record arrive. You’ll see the importance of the `lastObservedStreamTime` variable in the next section when we cover the method punctuation executes.

At the second annotation gets to the crux of this processor; does the record belong in an aggregation? If yes then you’ll either create a new aggregation object or add the record to an existing one. Additionally, a few lines of code down you’ll see the third annotation where the `shouldForward` variable is set to `true` or `false` indicating if there’s enough readings to require a forwarding of the aggregation before it closes. Since we can’t count on always having a small number of readings, we’ll want to make sure to emit intermediate aggregation results. The final line in this section stores the aggregation in the state store.

If the first `if` block doesn’t evaluate to `true` we’ll end up at annotation four. Here if the record matches the `stopAggregation` condition, meaning the temperature reading dropped below the threshold. If there is an existing aggregate it’s deleted from the state store and the `shouldForward` gets set to true. If the incoming record doesn’t meet the threshold and there’s no current aggregation the record is simply ignored.

Finally at annotation five if the `shouldForward` variable evalutes to `true` the processor forwards the aggregation to any downstream processors.

So at this point we’ve covered the main requirements of this processor, creating a "windowed" (setting a start and ending timestamp) aggregation that is determined by some aspects of the record data, in this case the reading of a temperature sensor. We have one last requirement to take care of, that is if we don’t receive any updates for a given period of time, we’ll want to clear the aggregation out and forward the results. This final requirement is handled by the method executed by the scheduled punctuation. Let’s take a look at the punctuation code now:

Listing 9.10. Punctuation Code

```
void cleanOutDanglingAggregations(final long timestamp) {
 List<KeyValue<String, SensorAggregation>> toRemove = new ArrayLi
  try (KeyValueIterator<String, SensorAggregation> storeIterator
                                                     = store.all(
        while (storeIterator.hasNext()) {
         KeyValue<String, SensorAggregation> entry = storeIterato
          if (entry.value.getEndTime() <
                                 (lastObservedStreamTime - 10_000
                toRemove.add(entry);
            }
        }
    }
    toRemove.forEach(entry -> {  #3
        store.delete(entry.key);
        context().forward(new Record<>(entry.key,
                                       entry.value,
                                       lastObservedStreamTime));
    });
}
```

The scheduled punctuation will execute every ten seconds (wallclock time) and examine all the records in the state store. If a timestamp is more than ten seconds behind the current stream time (meaning it hasn’t had an update in that amount of time) it’s added to a list. After iterating over the records in the store, the resulting list iterates over its contents removing the record from the store an forwarding the aggregation.

This concludes our coverage of using the Processor API for providing functionality that’s not available to you in the DSL. While it’s a made up example, the main point is that when you want finer grained control over emitting records from an aggregation and custom calculations, the Processor API is the key to acheiving that objective.

9.5 Integrating the Processor API and the Kafka Streams API

So far, our coverage of the Kafka Streams and the Processor APIs has been separate, but that’s not to say that you can’t combine approaches. Why would you want to mix the two approaches?

Let’s say you’ve used both the KStream and Processor APIs for a while.

You’ve come to prefer the KStream approach, but you want to include some of your previously defined processors in a KStream application, because they provide some of the lower-level control you need.

The Kafka Streams API offers three methods that allow you to plug in functionality built using the Processor API: `KStream.process`, `KStream.transform`, and `KStream.transformValues`. You already have some experience with this approach because you worked with the `ValueTransformer` in section 4.2.2.

The `KStream.process` method creates a terminal node, whereas the `KStream.transform` (or `KStream.transformValues`) method returns a new `KStream` instance allowing you to continue adding processors to that node. Note also that the transform methods are stateful, so you also provide a state store name when using them. Because `KStream.process` results in a terminal node, you’ll usually want to use either `KStream.transform` or `KStream.transformValues`.

From there, you can replace your `Processor` with a `Transformer` instance. The main difference between the two interfaces is that the `Processor`’s main action method is `process()`, which has a `void` return, whereas the `Transformer` uses `transform()` and expects a return type of `R`. Both offer the same punctuation semantics.

In most cases, replacing a `Processor` is a matter of taking the logic from the `Processor.process` method and placing it in the `Transformer.transform` method. You’ll need to account for returning a value, but returning null and forwarding results with `ProcessorContext.forward` is an option.

💡 **Tip**

The transformer returns a value: in this case, it returns a null, which is filtered out, and you use the `ProcessorContext.forward` method to send multiple values downstream. If you wanted to return multiple values instead, you’d return a `List<KeyValue<K,V>>` and then attach a `flatMap` or `flatMapValues` to send individual records downstream. An example of this can be found in src/main/java/bbejeck/chapter_6/StockPerformanceStreamsAndProcessorMul

To complete the replacement of a `Processor` instance, you'd plug in the `Transformer` (or `ValueTransformer`) instance using the `KStream.transform` or `KStream.transformValues` method.

A great example of combining the KStream and Processor APIs can be found in src/main/java/bbejeck/chapter_6/StockPerformanceStreamsAndProcessorApp.java. I didn't present that example here because the logic is, for the most part, identical to the `StockPerformanceApplication` example from section 6.3.1. You can look it up if you're interested. Additionally, you'll find a Processor API version of the original ZMart application in src/main/java/bbejeck/chapter_6/ZMartProcessorApp.java.

9.6 Summary

- The Processor API gives you more flexibility at the cost of more code.
- Although the Processor API is more verbose than the Kafka Streams API, it's still easy to use, and the Processor API is what the Kafka Streams API, itself, uses under the covers.
- When faced with a decision about which API to use, consider using the Kafka Streams API and integrating lower-level methods (`process()`, `transform()`, `transformValues()`) when needed.

At this point in the book, we've covered how you can *build* applications with Kafka Streams. Our next step is to look at how you can optimally configure these applications, monitor them for maximum performance, and spot potential issues.

10 Further up the Platform: ksqlDB

This chapter covers

- An introduction to ksqlDB
- More about streaming queries
- Building streaming applications with SQL statements
- Creating materialized views over streams
- Using ksqlDB advanced features

At this point in the book we’ve learned about several components of the Kafka event streaming platform - Kafka Connect for integrating external systems and Kafka Streams that you can use for building an event streaming application. These two components together form the bedrock of building event streaming applications. In this chapter you’re going to learn about ksqlDB which allows you to leverage those components by writing SQL to build an event streaming application. ksqlDB is a "Streaming database purpose built for streaming applications", it allows you to build powerful streaming applications with a few SQL statements.

So why would you use ksqlDB? For starters it vastly simplifies the application development process. With ksqlDB you’re not working with code or configuration files. You write your SQL queries and execute them which launches a continually running application where you can get instant notification of events.

Imagine you’re working with business analysts at Fintech company, Big Short Equity. They’ve seen the applications you’ve built using Kafka Streams. They want the ability to build some near-real-time financial analysis applications, but their primary skill set is not writing Java code. While you can try to support the analysts needs and build the Kafka Streams apps required, it would be far more efficient if the analysts could build them on their own. The analysts are experts on using SQL, as most of their work at this point involves writing queries on relational databases. So you have the idea of introducing ksqlDB to them, after all it provides scalable, distributed stream processing including aggregations, joins, and windowing. But unlike running a SQL query against a typical relational database, where the query will return results and stop, the results of a ksqlDB query are ***continuous***.

In a nutshell, you’ll want to use ksqlDB because you can quickly build a powerful event-streaming application in the same time it takes you write a SQL query!

So what you’re going to learn in this chapter is how to apply what you’ve learned so far about building event-streaming applications, but by using the familiar syntax of a SQL query. ksqlDB uses Kafka Streams under the covers, so all the concepts from the previous chapters apply here as well. Additionally, the ksqlDB server provides direct integration with Kafka Connect, so you can build entire end-to-end solution with out any code. We’ll start out the basic concepts of a stream and a table and how ksqlDB handles different data formats, including JSON, Avro, and Protobuf. Finally, we’ll explore more advanced options involving aggregations, joins and windowed operations.

10.7 Understanding what ksqlDB is

Earlier in the book, we discussed the concepts of an event-stream and an update-stream. To refresh your memory an event-stream is infinite sequence of ***independent*** events. Records in an event stream with the same key aren’t related to each other, each one stands alone as an event. But an update stream is little different. An update stream is infinite as well, but events with the same key as a previous event are considered an update to that event. ksqlDB has the same concept; you can query from a `Stream` or a `Table`, additionally you can perform a point-in-time query against a materialized view or a stream.

So let’s get started with your first ksqlDB query. To make the comparison to a Kafka Streams application easier, we’ll repurpose the `Yelling Application`.

ℹ **Note**

You can deploy ksqlDB in one of three ways, in standalone mode, as cluster on premises, or in Confluent Cloud. In this book you’ll work with ksqlDB in standalone mode via Docker. Later in the chapter we’ll cover the ksqlDB architecture in more detail.

First we need to create a `Stream` from a Kafka topic:

Listing 10.1. Creating a Stream in ksqlDB

```
ksql> CREATE STREAM input_stream (phrase VARCHAR) WITH
  (kafka_topic='src-topic', partitions=1, value_format='KAFKA');
```

So the first step is to create a ksqlDB `STREAM`. In this case you’ve created a `STREAM` named `input_stream` and it’s based on the topic `src-topic`.

Note

For all the ksqlDB examples, I’m going to assume your using one of the `docker compose` files (arm64_ksqldb-docker-compose.yml, x86_ksqldb-docker-compose.yml depending on the architecture of your laptop) included in the source code. Once you’ve run `docker compose -f <file name> up` command you’ll want to open a new terminal window and run `docker exec -it ksqldb-cli ksql http://ksqldb-server:8088`. This command will start a ksqlDB CLI session. Consult the README for chapter 10 in the source code for full instructions. Also, these commands should work if you choose to use ksqlDB on Confluent Cloud.

After you execute the `CREATE STREAM…` command you should see something like this on the ksqlDB CLI screen

Listing 10.2. Result of the CREATE STREAM statement

```
Message
----------------
 Stream created
----------------
```

Tip

You can confirm the STREAM objects you’ve created by running the command `show streams;` from the CLI

So now that you have a `STREAM`, the next step is to populate the underlying topic with some data you can start yelling! You could use a `KafkaProducer` to produce records to the topic, but for now let’s stick with SQL commands. Run a few insert like the following:

Listing 10.3. Insert statements for loading data into a stream

```
INSERT INTO input_stream (phrase) VALUES (
  'Chuck Norris finished World of Warcraft');
INSERT INTO input_stream (phrase) VALUES (
  'Chuck Norris first program was kill -9');
INSERT INTO input_stream (phrase) VALUES (
  'generate bricks-and-clicks content');
INSERT INTO input_stream (phrase) VALUES (
  'brand best-of-breed intermediaries');
.....
```

Using `INSERT INTO..` statements are a great way to get going quickly with ksqlDB. But in practice, after you’ve completed some quick prototyping, you’ll want to get data into the topic using a more efficient manner like a `KafkaProducer` or the results of another ksqlDB query.

The next step is to create a continuous or push query. Before we do that, let’s go over some background information. In ksqlDB since the queries are based on data coming into a Kafka topic, the run continuously, as an event stream never really stops. So once you start a query it will continue evaluating the incoming data until you explicitly stop it. These are called "push" queries because the results are "pushed" to the client that issued the query. We’ll cover the different clients you have to issue a ksqlDB query later in the chapter. You’ll also learn about another query type called a "pull" query that yields "point-in-time" results from a materialized store. If these terms seem unfamiliar to you, don’t worry, we’ll make them clear later in chapter when we cover them.

With our background information complete, let’s get back to writing the query that will be your streaming application:

Listing 10.4. A continuous query that becomes your streaming application

```
CREATE STREAM yelling AS  #1
  SELECT UCASE(phrase) AS SHOUT #2
  FROM input_stream
  EMIT CHANGES; #3
```

Congratulation, with just a simple query statement you have a streaming application! Notice that at annotation two you used a ksqlDB built-in function to perform the upper-case of the phrase, there are several built-in functions available. We’ll cover functions as we go along in the chapter.

Before we go on let’s review what you just did and compare it to your first Kafka Streams application, the Yelling App. By comparing the two you’ll get good sense if what ksqlDB is and what it can do for you when developing a streaming application. Let’s start by looking at the Kafka Streams version:

Listing 10.5. Kafka Streams version of the Yelling App

```
Serde<String> stringSerde = Serdes.String();
StreamsBuilder builder = new StreamsBuilder();

builder.stream("src-topic",
                Consumed.with(stringSerde, stringSerde))
        .peek(sysout)
        .mapValues(value -> value.toUpperCase())
        .peek(sysout)
        .to("out-topic",
                Produced.with(stringSerde, stringSerde));
```

As far as Kafka Streams applications go this one is very simple. In the chapter I introduced Kafka Streams, one of the benefits of using the Kafka Streams DSL is that it’s mostly declarative vs. imperative. This means you’re specifying mostly what you **want** to do (declarative) instead of specifying **how** to do it. What’s the significance of this difference? As you can probably guess the declarative approach is simpler to express and understand.

For example, let’s say you’ve invited a friend over for dinner and they ask if you need something and you reply to pick up a bottle of red wine. That would be a declarative statement, you left the decision of where to pick it up and what time to your friend. Now contrast that with having to give directions to a beer and wine store and where to find it in the store and what to do if your favorite brand isn’t available etc, that level of instruction is imperative. I could go on, but you get the point I’m trying to make here.

Using Kafka Streams vs raw or plain `KafkaConsumer` and `KafkaProducer` instances greatly simplifies your application development effort, but ksqlDB takes that to another level. Here’s what you used to create the same stream application with ksqlDB:

Listing 10.6. ksqlDB version of the Yelling App

```
CREATE STREAM yelling AS SELECT UCASE(phrase) AS SHOUT FROM input
    EMIT CHANGES;
```

Just one line of ***text***, a SQL statement, is all it took for creating the Yelling application in ksqlDB.

While the Kafka Streams version is still very easy to build, you still need to provide `Serde` instances, create the `StreamBuilder` instance, compile and run the code etc. This ease of developing a streaming application only gets larger as you start to develop more complex applications. Another benefit, and arguably the most impactful, to using ksqlDB is that since it uses SQL, it opens the door to non-developers for creating streaming applications.

Note that under the covers, a ksqlDB SQL statement compiles down to a Kafka Streams application. So while it’s using SQL it’s very helpful to have an understanding of Kafka Streams when using ksqlDB.

Does this mean that ksqlDB will solve all your problems and you don’t need Kafka Streams any more? Certainly not, no one tool can do it all and Kafka Streams will always have a place for building powerful event streaming applications. ksqlDB is simply another, although powerful, tool for you to use at your disposal.

Next let’s dive into a more complex and realistic example and at the same time learn about using the `Table` abstraction, which is important to learn about as it tracks the latest record for a given key.

10.1 Learning more about streaming queries

Earlier in the book you learned about the `KStream` and `KTable` concepts in Kafka Streams. The `KStream` is an event stream, where records (key-value pairs) with the same key are independent events, but with the `KTable`, an update stream, events with the same key are **updates** to a previous event with the same key. In ksqlDB the `STREAM` and `TABLE` concept follow the same rules. Let’s dig a little deeper with a more realistic example than the yelling application. Along the way we’ll dig into more details of working with ksqlDB.

Let’s say you’ve started a fitness website that encourages its members to get as many "steps" per day. But members can do other activities that count towards steps, not just running or walking. After some influences endorsed your application, your traffic has grown significantly and you’d like to encourage the growth by offering the ability to display the "step" leaders in near-real time and have it update on a regular basis.

You have a mobile application that updates user results to a backend server you have running in the cloud. The server in takes the information from the mobile applications and produces user activity updates to a Kafka topic with the name `user_activity`. This topic is the starting point enabling you publish updates to the website and your first step is to create a `STREAM` based on it -

Listing 10.7. Creating the stream

```
CREATE STREAM user_activity (first_name VARCHAR, #1
                             last_name VARCHAR,
                             activity VARCHAR,
                             event_time VARCHAR,
                             steps INT

     ) WITH (kafka_topic='user_activity',
            partitions=4,
            value_format='JSON',
            timestamp = 'event_time',    #2
            timestamp_format = 'yyyy-MM-dd HH:mm:ss' #3
     );
```

You have a SQL statement here creating the `user_activity` stream. The underlying value in the topic is in JSON and you've specified the structure of the JSON as the column names for the stream. The `WITH` statement contains properties ksqlDB uses for handling the stream and this time we've added a couple of new items; properties named `timestamp` and `timestamp_format`.

You're using these properties to instruct ksqlDB to use a field embedded in the record itself as timestamp for the record. If you recall from an earlier chapter, a `KafkaProducer` embeds a timestamp into a record when it produces to Kafka. The producer added timestamp becomes the event-time for the record. Usually the producer timestamp is close enough to serve as the official time of an event. But in some cases you may not want to use the timestamp set by the producer, but one set on the value of the record itself.

Why would you want to use a value embedded timestamp instead? While there could be many reasons, the chief reason would be when the timestamp on the record represents a more accurate reflection of when the event took place. In our case here, the timestamp on the record is when the user made the entry on your fitness application. The timestamp from the producer reflects when it received the record from your application server. While in practice these two should be very close, when it comes to awarding points or prizes it's more important to track when the user made the entry on their mobile device.

So to enable using the embedded timestamp you provide the `timestamp` property in the `WITH` clause and it specifies the column to use for the event time vs the one Kafka provides. Since you have the `event_time` field as a string, you need to tell ksqlDB the format of it which you've done with the `timestamp_format` property.

In practice it's very common to a `long` primitive for the timestamp in event objects since it represents the number of milliseconds in the Unix epoch time (time in milliseconds since January 1, 1970). In those cases where you have a `long` for the timestamp you would use a type of `BIGINT` for the timestamp column and in the `WITH` clause you'd only need to specify the column name since ksqlDB can work with the field directly. So in that case you'd update the query to create the stream like this:

Listing 10.8. Creating the stream with event time as a long

```
CREATE STREAM user_activity (first_name VARCHAR,
                             last_name VARCHAR,
                             activity VARCHAR,
                             event_time BIGINT, #1
                             steps INT

     ) WITH (kafka_topic='user_activity',
            partitions=4,
            value_format='JSON',
            timestamp = 'event_time' #2
     );
```

You’re using a type of `BIGINT` as that is the numeric type for an 8 byte number in ksqlDB, the backing type in Java is a `long`. But that raises the question what if you want to use the producer provided timestamp? ksqlDB has a system-column for each record called `ROWTIME` and it’s populated with the timestamp from the underlying Kafka record. The term system-column means it’s provided automatically by ksqlDB, you don’t need to do anything for it. We’ll see an example of using `ROWTIME` later in this chapter. Before we move on with our example of the fitness steps application there’s some additional information about the `WITH` clause we should cover.

For review here’s the `WITH` clause from the `user_activity` stream:

Listing 10.9. WITH clause from user_activity stream

```
WITH (kafka_topic = 'user_activity', #1
      partitions = 4,     #2
      value_format = 'JSON',  #3
      timestamp = 'event_time',    #4
      timestamp_format = 'yyyy-MM-dd HH:mm:ss' #5
     );
```

Note

We’ve already discussed the timestamp related properties in the previous section, so I won’t go over them again.

You’ll notice at annotation one we specify the topic name. This topic will be the source of data for the stream of user activity records. Remember, ksqlDB continuously evaluates the query over the incoming records of a topic, and not a relational database table. The `partitions` entry at annotation number two specifies the number of partitions of the topic. Take note that the `kafka_topic` property is always required.

Since ksqlDB uses Kafka Streams under the covers, why are you telling it the number of partitions of the input topic? After all you simply provide the name of the topic and Kafka Streams takes care of everything needed to properly consume from the topic. If the underlying topic doesn't exist when you create the stream, ksqlDB will attempt to create it.

But if the topic does exist, ksqlDB will not overwrite it. An important point to note is that when you provide the `partitions` property for the `STREAM` and the topic exists, the number of partitions you specify ***must*** match the actual number of partitions, otherwise you'll get an error. If the topic exists, you can safely omit the `partitions` property.

When doing quick development or prototyping in a local environment, having ksqlDB create the topics can be a time saver. But in a production setting it's best to always create the topics you need ahead of time to avoid any confusion or miscommunication about the structure of your applications.

The last part of setting stream properties we'll cover here is the `value_format` setting. As you might have guessed, this tells ksqlDB the format of the value in the key-value pair. There is a setting for the key as well - `key_format`. The acceptable entries for either the `key_format` and `value_format` properties include:

- 'JSON'
- 'JSON_SR'
- 'AVRO'
- 'PROTOBUF'
- 'PROTOBUF_NOSR'
- 'NONE'
- 'KAFKA'
- 'DELIMITED'

While I won’t go into a full discussion here on the different types, (I’ll refer you to the ksqlDB documentation instead - https://docs.ksqldb.io/en/latest/reference/serialization/#serialization-formats) it will be worthwhile to point out a few important points. The `JSON_SR`, `AVRO`, and `PROTOBUF` formats use Schema Registry. To use Schema Registry with ksqlDB, you must provide the HTTP endpoint (as you did with either the Kafka clients or Kafka Streams) via the `ksql.schema.registry.url` property.

We’ll cover configuration options for ksqlDB in a later section. If you’re using Schema Registry, ksqlDB will automatically register a schema as needed. But this raises an important point, if you have an existing schema for a given value or key, you can provide the schema id for the key or value in the `WITH` statement when setting the properties for your stream. For example, let’s say you used Avro for the values in the fitness step application. In that case you would the stream in this way:

Listing 10.10. Specifying the schema id when constructing the stream

```
CREATE STREAM user_activity
WITH ( kafka_topic = 'user_activity',
       value_format = AVRO, #1
       value_schema_id = 1 #2
     )
```

At annotation one you’re specifying that the value uses AVRO for the serialization, but you also provide the schema id at annotation two, so it can retrieve the precise version of the schema from Schema Registry. This ability to use the exact version of a schema is important when ksqlDB either consumes from or writes to a topic that other applications work with as it’s essential that all clients use the same schema otherwise you’re application is likely to experience errors due to data formatting errors. If you don’t provide the schema id, ksqlDB will retrieve the latest version of the schema. ksqlDB will assume the subject name for the schema is <topic-name-key> or <topic-name-value>. We covered Schema Registry in chapter 3, so you can refer back to it if you need to refresh your understanding of Schema Registry concepts.

You’ll also notice that you didn’t define any columns for the stream and that’s because when using serialization format supported by Schema Registry, ksqlDB will infer the column names and field types from the schema. In our current example, we specified the column names and types because we’re using plain JSON for the data format there’s no inferencing available. We’ll cover using ksqlDB with Schema Registry in more detail in a later section, but for now let’s get back to building the fitness step application.

For review, here’s the initial query you have:

Listing 10.11. Initial query for building the fitness step application

```
CREATE STREAM user_activity (first_name VARCHAR,
                             last_name VARCHAR,
                             activity VARCHAR,
                             event_time VARCHAR,
                             steps INT

     ) WITH (kafka_topic='user_activity',
            partitions=4,
            value_format='JSON',
            timestamp = 'event_time',
            timestamp_format = 'yyyy-MM-dd HH:mm:ss'
     );
```

With this stream in place you’re now primed to start building the often requested new feature, a leader board. You decide you’re going to add up the steps for a given activity and present the results in sorted order from highest to lowest. To create the first leader board stat you are going to need to perform an aggregation, in this case just a simple sum of the `steps` column.

Listing 10.12. Adding a sum aggregation for calculating leaders in a category

```
CREATE TABLE activity_leaders AS    #1
  SELECT
     last_name,
     SUM(steps)                     #2
FROM user_activity
GROUP BY last_name #3
EMIT CHANGES;
```

When you use an aggregation in ksqlDB, the result of the query is a `TABLE`, so we’ll need to explicitly create one with our query. Also note that just like when you query a relational database you must include the columns in the SELECT statement in the GROUP BY clause. So what you’ve created here a table or materialized view of an aggregation, a SUM in this case and it will continue to provide results since it’s selecting from the underlying stream user_activity - so whenever new results end up on the user_activity stream, your table here updates with new results. Additionally, the ksqlDB creates a changelog topic as well that it uses to make sure the records in the aggregation are backed up should the ksqlDB server running the table query experience any issues.

You should also take note that under the covers, creating this table results in creating a new topic in Kafka, named activity_leaders by default since that’s the name of the table. If you want to use a different name for the underlying topic supporting the table, you can provide the topic name in a WITH statement and we’ll see just how to do that next.

Initially, this query works for you and the customers you have in your demo preview respond well to the leader dashboard. But in the solicited feedback there are several comments for you to update the dashboard with more details. First of all, the results combine the steps score across all activities and are only displayed with the users' last name making it difficult to disambiguate the differences between members of your app with the same last name.

So you decide to put the release on hold and pursue the required changes based on the user provided feedback. Since you’re in a development environment for the preview you’d like to clean up the table query. Since you plan on complete revamp of the activity_leaders table it would be a good idea to remove the backing topic as well. So your next step to clean things up is to run this command:

Listing 10.13. Command to remove the table and the underlying topic

```
DROP TABLE activity_leaders DELETE TOPIC;
```

This command will delete the table from ksqlDB and the backing topic. This command marks the topic for deletion, the actual removal of the topic is asynchronous and occurs eventually when the broker cleans up resources.

NOTE: The `DELETE TOPIC` clause at the end is optional. If you want to keep the records contained in the topic then you can consider leaving off the delete part off the command. Also can add `IF EXISTS` to your command so it won't error out should a table you're trying to delete doesn't exist. So the command would look like `DROP TABLE IF EXISTS activity_leaders DELETE TOPIC`

Now that you've cleaned up the existing table you go about redefining you table query and you decide that adding the first name and the activity type should satisfy the user comments - you'll get to distinguish between users and see the leaders for each activity type. Since you're proficient in SQL it doesn't take you long to come up with the query:

Listing 10.14. Updating the sum aggregation for leaders per activity and full name

```
CREATE TABLE activity_leaders AS
  SELECT
     first_name,
     last_name,
     activity,
     SUM(steps)
FROM user_activity
GROUP BY first_name, last_name, activity
EMIT CHANGES;
```

Your updated query is essentially the same but now you're selecting additional columns, `first_name` and `activity`. Since you've added two columns in the select portion of the SQL, you'll need to add those columns as well in the `GROUP BY` clause. This is required because with an aggregation we'll need to group the records by the selected fields to form unique results, the grouping acts as a "key" for the aggregation. You execute the new table statement in the ksqlDB CLI and you see an unexpected error:

Listing 10.15. Error creating a Group By with multiple columns

```
Key format does not support schema.
format: KAFKA
schema: Persistence{columns=[`FIRST_NAME` STRING KEY, `LAST_NAME`
  STRING KEY, `ACTIVITY` STRING KEY], features=[]}
reason: The 'KAFKA' format only supports a single field. Got:
  [`FIRST_NAME` STRING KEY, `LAST_NAME` STRING KEY, `ACTIVITY` ST
```

This detailed error message explains things pretty well, but let’s add a little more information. When ksqlDB attempts to create the underlying topic for the table, the primary key type will default to `KAFKA` meaning it must be a scalar or schema-less type supported by Kafka - a string or integer for example. But here you’ve provided three columns that will make up a composite key. This is required because you’re grouping by these three columns to ensure the results are unique per row, so three fields won’t work with a schemaless key.

Fortunately, there is a simple solution to getting your updated query to work. You’ll just need to specify the key format for the new table to one that supports a schema, and this case we’ll use `JSON`. To do this, we’ll make a slight change to query and add a `WITH` statement like so:

Listing 10.16. Update the CREATE TABLE to specify a key format that supports a schema

```
CREATE TABLE activity_leaders WITH (KEY_FORMAT = 'JSON') AS
  SELECT
     first_name,
     last_name,
     activity,
     SUM(steps)
FROM user_activity
GROUP BY first_name, last_name, activity
EMIT CHANGES;
```

So by adding `WITH (KEY_FORMAT = 'JSON')` your new table will now use JSON and create a composite key containing the `first_name`, `last_name`, and `activity` columns. But you’re not quite done yet, as this change will make the query successful in performing the aggregation, but the group-by columns are in the key and won’t show up in the final results.

So you’ll need to instruct ksqlDB that you want those columns in value portion of the results (remember Kafka works in key-value pairs). Again this is easily achieved by using `AS_VALUE` function which instructs ksqlDB to copy a row’s key into its value as well. You’ll need to keep the original columns in the select statement and add an `AS_VALUE` function for each key you want copied into the value:

Listing 10.17. Specifying key format and copying keys to value as well

```
CREATE TABLE activity_leaders WITH (KEY_FORMAT = 'JSON') AS
  SELECT
     first_name as first_name_key, #1
     last_name as last_name_key,  #1
     activity as activity_key,   #1
     AS_VALUE(first_name) as first_name, #2
     AS_VALUE(last_name) as last_name, #2
     AS_VALUE(activity) as activity,  #2
     SUM(steps) as total_steps
FROM user_activity
GROUP BY first_name, last_name, activity
EMIT CHANGES;
```

So now you have a working aggregate query with multiple `GROUP BY` items which is essentially a composite key for each row. Note that this is not something you need to do with all aggregations in ksqlDB. But having multiple columns you are grouping by is common enough that it's worth us taking the time to explain how to handle the situation.

What you've seen so far is just scratching the surface with what you can do with ksqlDB, more complex queries are possible and there are several built-in functions to learn about. But before we continue down the path of exploring the capabilities of ksqlDB, let's take a quick pause to discuss some of the conceptual types of queries that are possible.

10.2 Persistent vs. Push vs. Pull queries

So far we've built a streaming application that show updates as the `user_activity` stream continues receiving user input. Once a client executes the query, the results are continually ***pushed*** to that client unless the query is specifically terminated. But there could be a situation where you want to issue a single query to retrieve specific result at that point in time vs. a constant stream of updates. Additionally, you may need continuous query that doesn't serve a particular client, but can be used by any client issuing a query, something more **permanent*. We'll discuss how to do implement both of these approaches next.

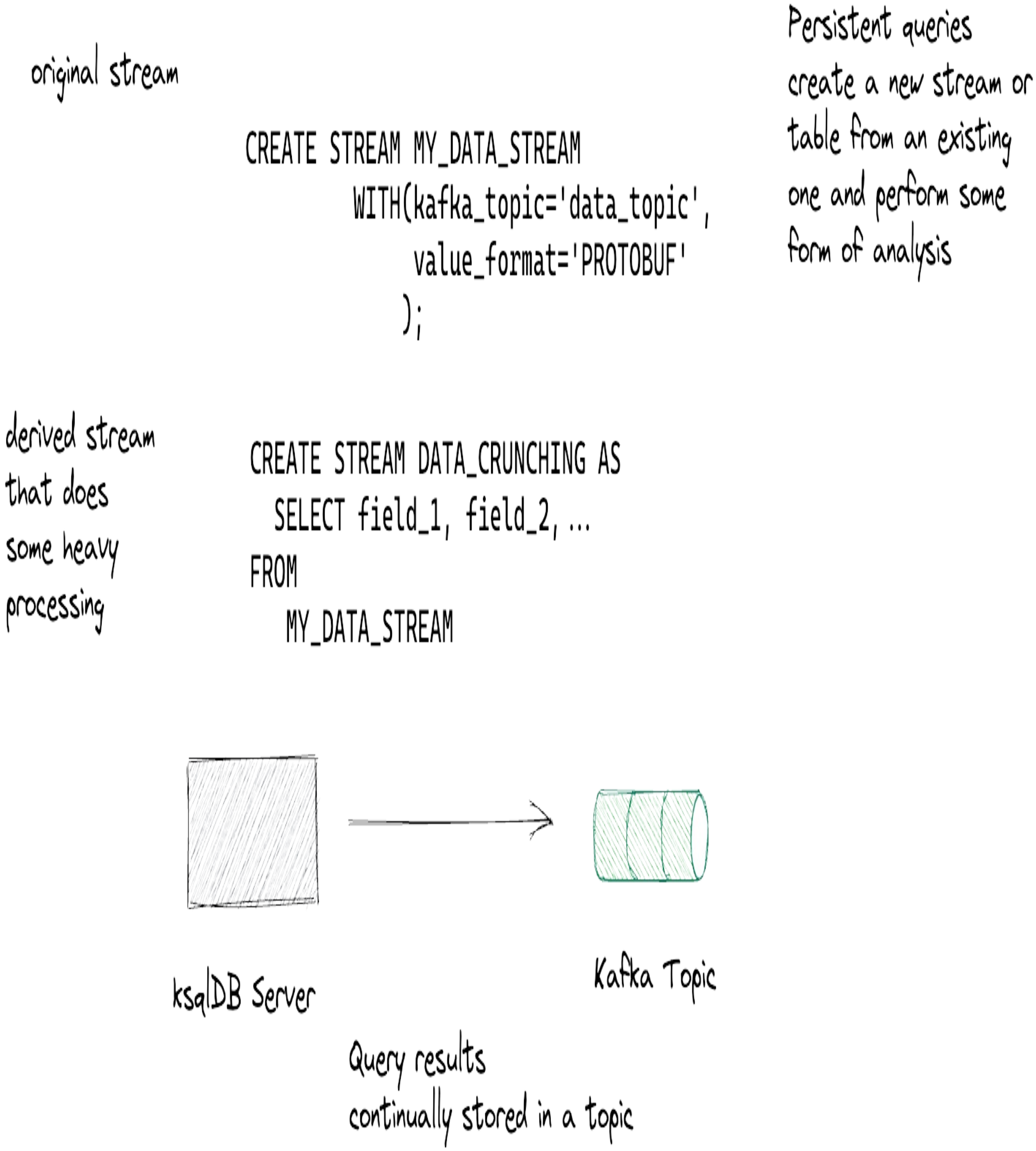
ksqlDB has three categories of query types. A push or continuous query where the stream or table constantly executes against the incoming records to the underlying topic returning the results to the original client issuing the query. The `activity_leaders` table you created in the previous section is an example of a persistent query.

Push queries are an excellent choice for an asynchronous work flow, that is where you issue a command or request, but you don't expect an answer immediately, it executes in the background and the answer will come later. A concrete example of an asynchronous work flow is sending an email, you write your text and then send it, knowing that you'll get a response at some point, but you don't need an immediate answer.

For more of a synchronous, request-response workflow, ksqlDB offers a ***pull*** query. An example of synchronous type of workflow is you have contractors at your house and they need to get your permission to take out a section of the wall, all work stops until they receive an answer. You can issue a pull query against a stream or a table, and there are some restrictions which we'll get into in a moment.

So the question is which one to use? To answer that question let's compare all three query types (persistent, push, and pull) against each other.

Figure 10.1. Persistent queries run on the server and persist results to a topic



```
CREATE STREAM MY_DATA_STREAM
        WITH(kafka_topic='data_topic',
             value_format='PROTOBUF'
            );
```

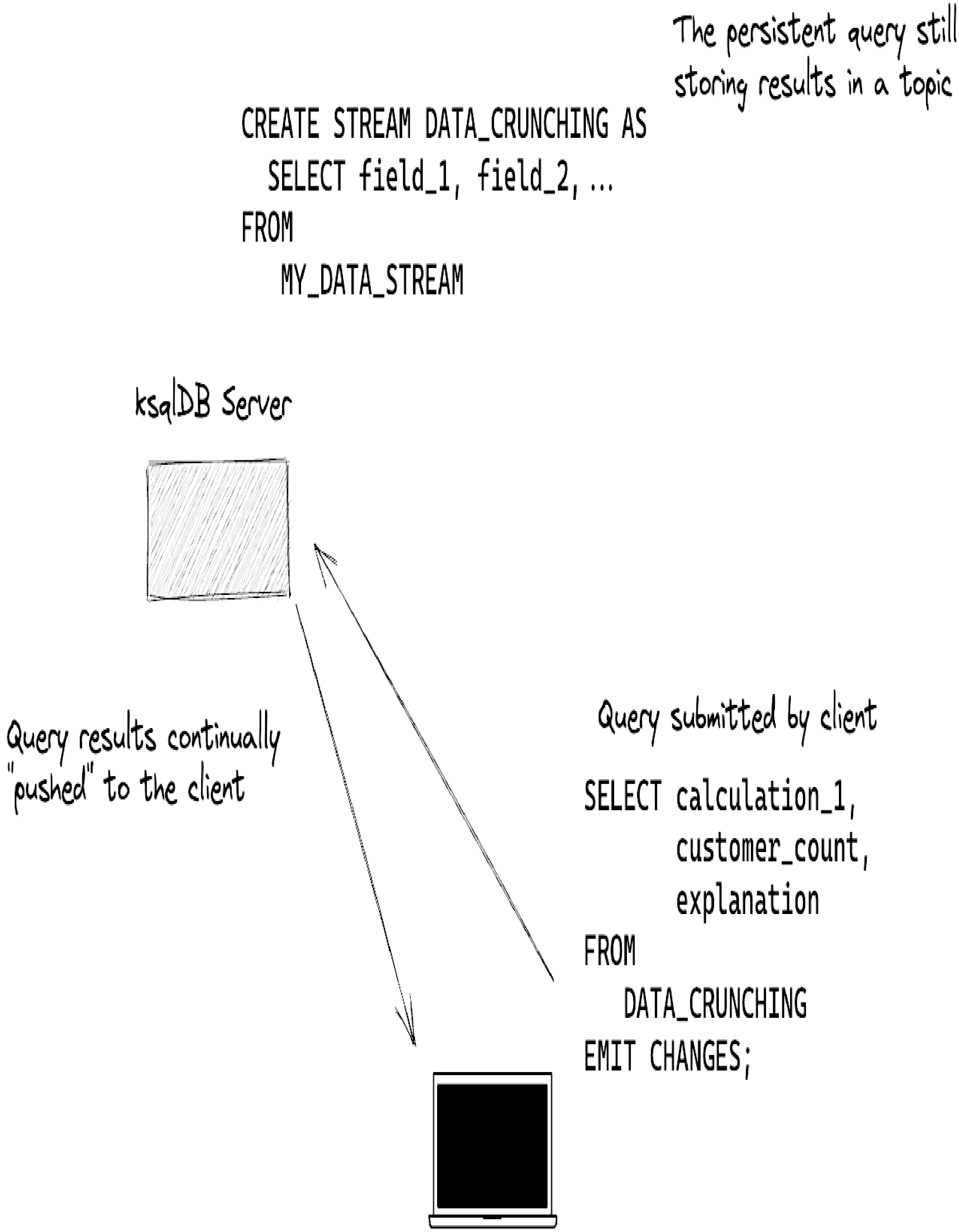
Persistent queries create a new stream or table from an existing one and perform some form of analysis

```
CREATE STREAM DATA_CRUNCHING AS
  SELECT field_1, field_2, ...
FROM
   MY_DATA_STREAM
```

Query results continually stored in a topic

A persistent query runs on the server it stores the results of the query in a Kafka topic - so the results persist for duration of the topic’s configured retention time. Also, once you have a persistent query running, you can easy share outcome of the query by virtue of the fact that any Kafka consumer client can read the records from the topic. You can think of a persistent query as the workhorse or backbone, it carries the full load of performing the analysis of your streaming application. With a persistent query you can use the full range of ksqlDB SQL syntax. Persistent queries take the form of `CREATE TABLE|STREAM AS SELECT...`

Figure 10.2. Push queries don’t store results but return them to the client that submitted the query


The persistent query still storing results in a topic
ksqlDB Server
Query results continually "pushed" to the client
Query submitted by client

```
CREATE STREAM DATA_CRUNCHING AS
  SELECT field_1, field_2, ...
FROM
   MY_DATA_STREAM
```

```
SELECT calculation_1,
       customer_count,
       explanation
FROM
   DATA_CRUNCHING
EMIT CHANGES;
```

A push query on the other hand does not persist its results to a topic. A push query returns it’s results to the client issuing the query. But the results are continually ***pushed*** to the client - you can think of a push query as a subscription for changes to the persistent query. For example a push query against the `activity_leaders` could look like this:

Listing 10.18. Push Query example on Activity Leaders table

```
SELECT
    last_name, activity, total_steps
FROM activity_leaders
EMIT CHANGES;
```

With this query you’re going to receive updates for each user as they update their activity. But these changes aren’t stored anywhere, the results of this query return to the original client executing the query, whether it’s from the ksqlDB CLI, the REST API or the Java client that’s available. We’ll cover the client options available for ksqlDB later in this chapter. This query returns all updates to the table, but you could further refine the results further with a `WHERE` clause:

Listing 10.19. Push Query with WHERE clause

```
SELECT,
  last_name, activity, total_steps
FROM activity_leaders
WHERE total_steps > 1000
EMIT CHANGES;
```

Now you’ll only receive updates where the total number of steps is greater than one-thousand. The conditions you put in the `WHERE` clause can refer to any column the stream or table defines including the pseudo columns `ROWTIME`, `ROWPARTITION`, and `ROWOFFSET` defined by ksqlDB. These columns are injected or attached to each row for an incoming record into a Kafka topic backing a stream or table. Let’s take a moment to define each of these:

- ROWTIME - Is the timestamp associated with the Kafka record which either the producer or the broker sets, depending on your configuration.
- ROWPARTITION - The rowpartition value is records' partition it belonged to from the backing topic
- ROWOFFSET - Each record in a Kafka topic has an offset which

represents its logical position in the topic.

When would you use any of these pseudo columns in a query? Sometimes you may find that it’s helpful to filter results by external factors about a record - for example you may only want to view events that occurred within a given time-frame. Even if the record doesn’t define a timestamp as one of its columns, you can still filter results by the values contained in the `ROWTIME` for each one.

You could specify more conditions to refine the results, I won’t enumerate them all here, but I’m sure you’re getting the picture to what you do. At this point you should be able to see the relationship between persistent and push queries - the persistent query carries the full-load and allows you to issue a push query to receive a subset of the information you’re interested in. Since the push query doesn’t persist its results, it makes sense to use push queries as an alerting source until a given state is reached.

Then once the given state is reached, say where you observe a given user reaching ten-thousand steps, you can terminate your query. There’s a way you can place a hard cap on the number of results, using the `LIMIT` clause. For example let’s say you are testing out a query from the ksqlDB CLI and you only want to see of the query works properly then exit you can change the push query above to this:

Listing 10.20. Push Query with a LIMIT

```
SELECT
  last_name, activity, total_steps
FROM activity_leaders
WHERE total_steps > 1000
EMIT CHANGES
LIMIT 10;
```

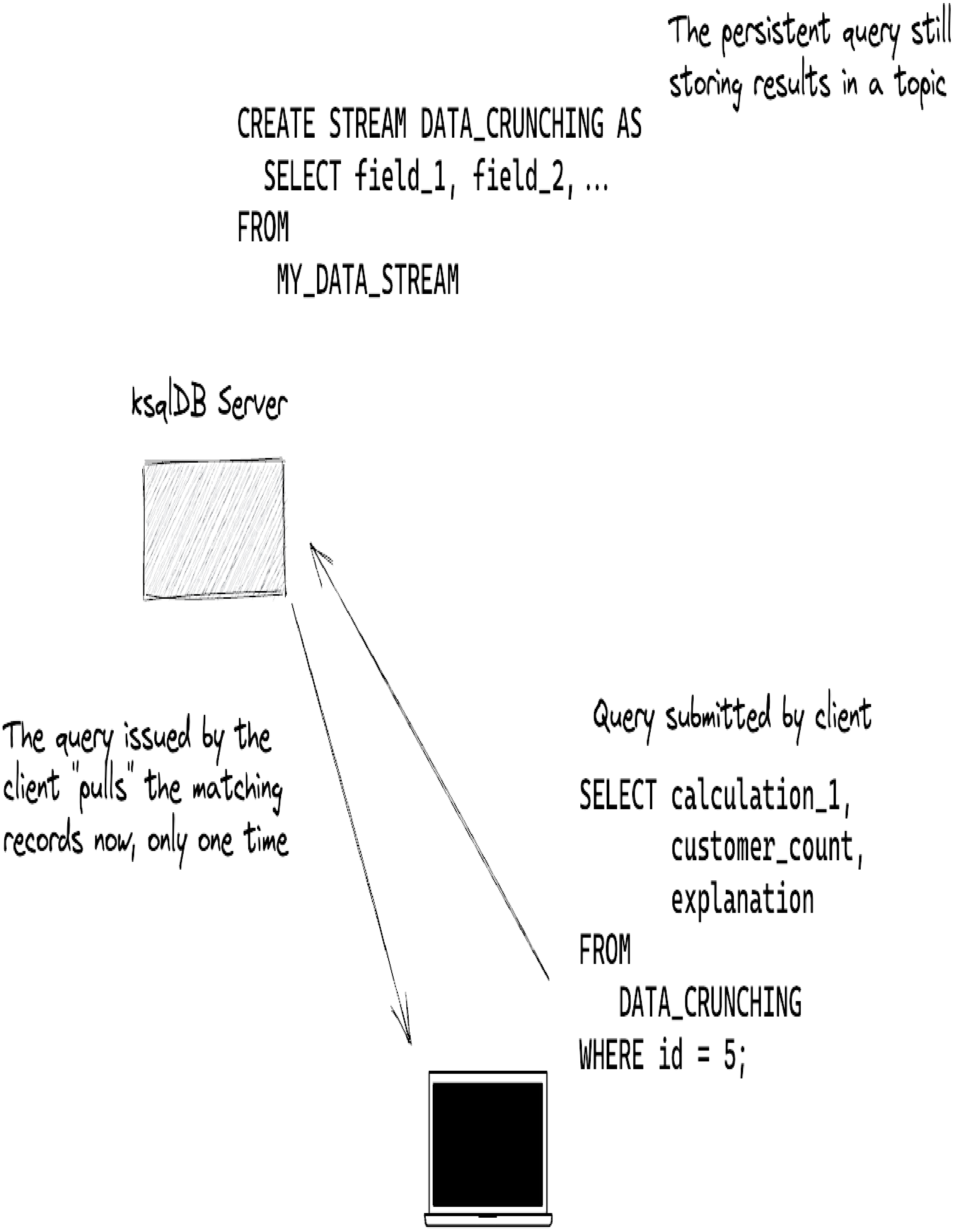
Now the query will terminate once it emits 10 result records. I should not at this point that push queries can use the full range of SQL commands ksqlDB provides. There are two main differences between persistent and push queries:

1. Push queries don’t persist results, they are emitted to the console or back

to the client executing the query

2. Push queries aren’t shared. With a persistent query, it evaluates against the incoming records ***once*** and stores the results in a Kafka topic. But with push queries, if separate clients issue the same queries ksqlDB evaluates each one independently even if they provide the same output.

Figure 10.3. Pull queries return point in time results once to get updates you need to re-submit the query



```
CREATE STREAM DATA_CRUNCHING AS
  SELECT field_1, field_2, ...
FROM
   MY_DATA_STREAM
```

```
SELECT calculation_1,
       customer_count,
       explanation
FROM
   DATA_CRUNCHING
WHERE id = 5;
```

The persistent query still storing results in a topic
ksqlDB Server
Query submitted by client
The query issued by the client "pulls" the matching records now, only one time

Now let’s move on to the final query type, the pull query. Where the persistent and push queries provided constant evaluation of the incoming records, a pull query evaluates its statement ***once*** and terminates afterwards. Like the push query however, the pull query does not persist its results in a topic, the outcome of the query is returned to the client. One can think of a pull query as reaching out an pulling down a result at a particular point in time. The role of a pull query is best used in a situation where you need an immediate answer, as you would in a request - response workflow.

Pull queries only support a subset of the ksqlDB SQL statements. You can use a pull query with any stream and tables created with `CREATE TABLE as SELECT`, but currently not tables created directly against a backing topic. Additionally, pull queries don’t support the use of `JOIN`, `GROUP BY`, `PARTITION BY`, and `WINDOW` clauses.

Out of the box, there are limitations with what you can do in a `WHERE` clause as well. The restrictions in the `WHERE` clause are that you must use a key column and the comparison needs to be against a literal value. For example here’s a pull query against the `activity_leaders` table you created earlier:

Listing 10.21. Example of pull query WHERE clause with a key lookup

```
SELECT last_name, activity, total_steps
FROM activity_leaders
WHERE key_1 = 'Smith'
```

So now you can execute a query against the `activity_leaders` table and get results for where the last name is equal to "Smith" and the query terminates. To get any additional updates for this query, you’d have to execute it again.

Note

There is a configuration you can set allowing for more liberal use of the `WHERE` clause. If you set `ksql.query.pull.table.scan.enabled` to true in either a CLI session or a ksqlDB server and it allows for several additional enhancements like using non-key columns or comparing columns to other columns. See the ksqlDB documentation for more information on table scans - https://docs.ksqldb.io/en/latest/developer-guide/ksqldb-reference/select-pull-query/#where-clause-guidelines

We’ve covered a lot of ground on the different query types available so let’s wrap things up with a table to compare each situation where persistent, push and pull queries are most effective.

Type	Syntax	Best Use	Running Mode	Results Stored in Topic
Persistent	CREATE [STREAM TABLE] AS SELECT.. EMIT CHANGES	Asynchronous response, heavy work on server	Continual updates	Yes
Push	SELECT [items] FROM [STREAM TABLE]… EMIT CHANGES	Asynchronous response, more refined queries	Continual updates	No
Pull	SELECT [items] FROM [STREAM Materialized TABLE]… WHERE	Evaluate query and terminate, point in time query	No updates must re-issue for additional results	No

To summarize our table here `persistent` queries take the form of `CREATE STREAM|TABLE AS SELECT.. EMIT CHANGES` and the query results are persisted in a Kafka topic. Since different clients can share the results in a topic, persistent queries are best suited for doing heavy or more complex queries. Changes are continually emitted as new records arrive in the stream or table.

A `push` query starts with `SELECT [items] FROM.. EMIT CHANGES` but the results are not persisted in a topic, they are returned continually to the client. A `push` query can use the full range of SQL available in ksqlDB. Since the results are not persisted, the same query issued from different clients is always evaluated. The `push` query is optimal for subscribing to changes in a stream or table as in an event driven architecture, but usually the query is much simpler.

Finally the `pull` query retrieves a distinct result and terminates, there are no updates as new records arrive. A `pull` query takes the form of `SELECT [items] FROM..` The results are not persisted as well and are returned to the issuing client. The `pull` query is best used for obtaining a single result in a request-response format. A `pull` query has limitations on the SQL statements it can use most notably in the `WHERE` clause.

The general pattern then is to have persistent queries running on the ksqlDB server and use a combination of push and pull queries to extract a subset of information from them in your applications.

This wraps up our coverage on query types, but before we go on let’s formalize the ways you can create a stream or a table with the different query types.

10.3 Creating Streams and Tables

So far we’ve established that you can create streams and tables in ksqlDB, but we’ve done it without categorizing the different ways that you can do so, but we’ll take care of that in this section. This will also be a good time to talk about integrating with Schema Registry because when you define a stream or table where the backing topic has a schema, defining the columns in the create statement is optional, ksqlDB will infer the names and types based on the schema. We’ll discuss creating streams and tables without schemas first, then well move on to integration with Schema Registry later in this section.

The first category of a stream or table is you can create could be considered a "base" stream or table. You create these base streams or tables directly against a backing Kafka topic. We’ve seen creating a base stream earlier with the `user_activity` stream, but we’ll repeat it here:

Listing 10.22. Creating the user_activity stream

```
CREATE STREAM user_activity (first_name VARCHAR,
                             last_name VARCHAR,
                             activity VARCHAR,
                             event_time VARCHAR,
                             steps INT

     ) WITH (kafka_topic='user_activity',
            partitions=4,
            value_format='JSON',
            timestamp = 'event_time',
            timestamp_format = 'yyyy-MM-dd HH:mm:ss'
     );
```

This statement creates a stream with the backing topic of `user_activity`. If you recall a Kafka topic stores key-value pair records, but as defined here, this stream contains only values, the keys for each record are null. Defining a key on a stream is optional in ksqlDB and in this case there are no keys for the `user_activity` topic. But what if a topic does have keys, how would you change the `CREATE STREAM` statement here? Let’s say that the `user_activity` topic does have populated keys - an integer that represents the user-id you would update the create statement like this:

Listing 10.23. Creating the user_activity stream

```
CREATE STREAM user_activity ( user_id INT KEY, #1
                              first_name VARCHAR,
                              last_name VARCHAR,
                              activity VARCHAR,
                              event_time VARCHAR,
                              steps INT

     ) WITH (kafka_topic='user_activity',
            partitions4,
            key_format='KAFKA' #2
            value_format='JSON',
            timestamp='event_time',
            timestamp_format='yyyy-MM-dd HH:mm:ss'
     );
```

So for adding a key the only change you needed to make was adding a column declaring its type and adding the KEY reserved word telling ksqlDB this is the key of the key-value pair. You also need to tell ksqlDB how the key is formatted, which you’ve done at annotation two, specifying a format of KAFKA indicating it’s one of the basic types supported by Kafka - String, Long, Integer for example. Just like we’ve seen with Kafka clients and Kafka Streams, having a key will drive the partitioning for the incoming records and without a key records are pretty much evenly distributed across partitions.

You can also create a table directly with a backing topic. Let’s take our user_activity stream and create it as a table named user_activity_table instead:

Listing 10.24. Creating the user_activity_table table

```
CREATE TABLE user_activity_table (user_id INT PRIMARY KEY, #1
                                 first_name VARCHAR,
                                 last_name VARCHAR,
                                 activity VARCHAR,
                                 event_time VARCHAR,
                                 steps INT

     ) WITH (kafka_topic='user_activity',
            partitions4,
            key_format='KAFKA' #2
            value_format='JSON',
            timestamp='event_time',
            timestamp_format='yyyy-MM-dd HH:mm:ss'
     );
```

Creating a table is very similar to creating a stream, but with one significant difference. The KEY column on the stream is optional, but with a table it’s required. In annotation two, you’re specifying the format if the key which follows the exact same rules as the key format specification of the stream above. Key format values can also be AVRO, PROTOBUF, and JSON_SR (for

JSON Schema), but for our purposes, as we’ve done throughout the book, the examples you’ll use will only use keys of type `KAFKA`.

But there are also several differences between a stream and a table with the semantics of keys and values. In a stream, we’ve established a valid record can have a null key, but a table will drop any incoming record with a null key. Another difference, we’ve covered before, is that in a stream records with the same key don’t have an impact on each other, the remain independent of each other.

But in a table, just like in a relational database table, you can only have one primary key, so an incoming record with the same key as a previous record will be an update replacing it. There’s also a difference in semantics with null values between a stream and a table. A null value in a stream holds no special meaning, but a null value in a table is considered a ***tombstone***. The significance of a tombstone record marks that row for deletion from the table (which under the covers means it’s deleted from the backing topic). Let’s summarize these differences in a table for quick reference:

Stream	Table	Key optional
Key required, otherwise the record is dropped	Keys don’t have to be unique, records with the same key aren’t related	Keys must unique, records with the same key are updates
Null values have no impact or meaning	Null values are tombstones, marking the row for deletion from the table.	Type KEY

As side note it’s important to understand that a row with null value does not get immediately deleted, rather it’s marked for deletion. Under the covers a table uses a backing topic that is a ***compacted*** topic and only when the log cleaner runs will the record get removed. The log cleaner runs at regular intervals as configured.

When you create either a stream or table with a `CREATE` statement directly against a Kafka topic, you don’t directly query them. These streams and tables are how you bring a Kafka topic into ksqlDB. It’s the subsequent streams and tables you create by selecting from these base streams and tables that provide results either returned to a client in the case of push or pull queries (remember push queries run indefinitely until terminated and pull queries evaluate once and terminate) or persisting their results to a topic.

So to summarize you have three basic types of streams and tables:

1. The "base" streams or tables that expose a topic to ksqlDB for further queries
2. Persistent queries that publish results to a Kafka topic
3. Push and Pull queries that you can run against either the base ones or persistent queries

Now that we’ve wrapped up our coverage on query types, let’s move on to the formats of keys and values and Schema Registry integration.

10.4 Schema Registry Integration

Schema Registry integrates fairly seamless with ksqlDB and it offers a sizable advantage when defining a stream or a table where there is a schema. Since the schema contains the field names and the types, you can omit the column definitions when you create a stream or table. For example let’s take another look at the `user_activity_table` definition and assume the value used Protobuf:

Listing 10.25. User Activity Table with Schema Registry - Protobuf schema

```
CREATE TABLE user_activity_table (user_id INT PRIMARY KEY #1

     ) WITH (kafka_topic='user_activity_proto',
            partitions4,
            key_format='KAFKA' #2
            value_format='PROTOBUF', #3
            timestamp='event_time',
            timestamp_format='yyyy-MM-dd HH:mm:ss'
     );
```

We still need to provide the primary key definition since it’s a basic `KAFKA` type, but for the columns we omit providing the definitions and ksqlDB infers the names and types from the schema. A situation like this where you use a `KAFKA` type and a type supported by Schema Registry is known as partial schema reference. If the key were also of type `Protobuf`, then the table definition would be simplified further to this:

Listing 10.26. User Activity Table with Schema Registry - Protobuf schema for key and value

```
CREATE TABLE user_activity_table
     WITH (kafka_topic='user_activity_proto',
           partitions4,
           key_format='PROTOBUF' #1
           value_format='PROTOBUF', #2
           timestamp='event_time',
           timestamp_format='yyyy-MM-dd HH:mm:ss'
     );
```

I’m only showing the updated table definition for the `user_activity_table` because the `user_activity` stream would have the identical changes for creating the stream with schema enabled values and/or keys. There is an exception to this rule of omitting column definitions with schemas and that happens when you only want to use subset of the columns. For example let’s revisit the `user_activity_table` but now let’s say you only want to use three columns - `last_name`, `activity`, and `steps` your table definition would look like this:

Listing 10.27. User Activity Table with Schema Registry - Protobuf schema subset of columns

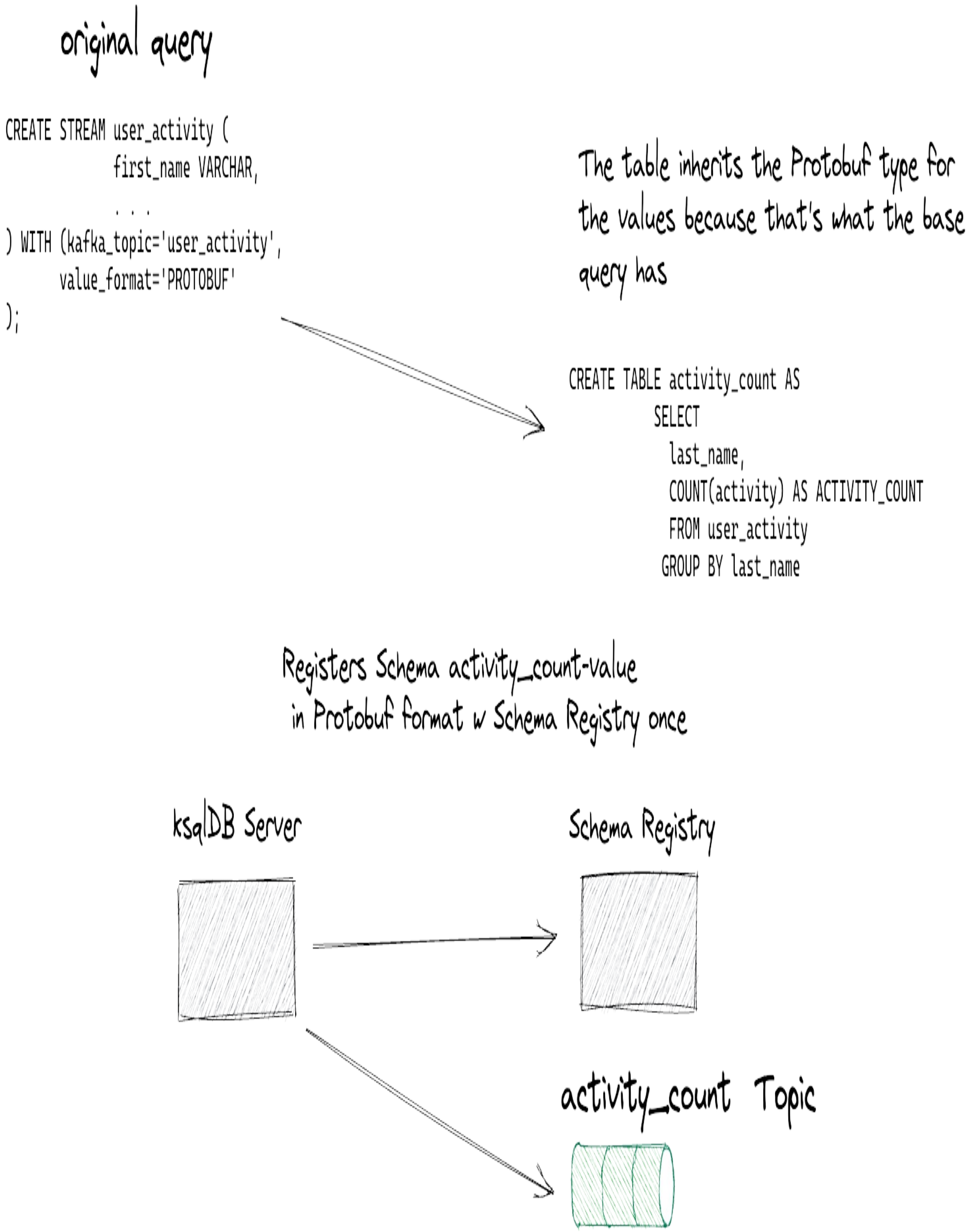
```
CREATE TABLE user_activity_table (user_id INT PRIMARY KEY, #1
                                  last_name VARCHAR, #2
                                  activity VARCHAR,
                                  steps INT

     ) WITH (kafka_topic='user_activity_proto',
            partitions4,
            key_format='KAFKA',
            value_format='PROTOBUF'
     );
```

So here even though the value is in Protobuf format, we need to declare the names and types of columns since we’re only selecting a subset of them. Although the inferencing done by ksqlDB reduces the amount of writing you need to do for a stream or table definition, it also reduces the clarity as now you’ll have to either issues a `DESCRIBE` statement or view the physical schema to understand the column names and types. This could be considered a minor point, but there’s something to be said from reading the sql statements and gaining a full understanding of the underlying data types.

We have two final points to consider in our Schema Registry section, when ksqlDB infers data types and writes a schema, and the conversion of datatypes.

When you create a persistent query, for recall that means using syntax of `CREATE STREAM AS SELECT…` , the persistent query will inherit the key and value format of the base stream or table. To refresh your memory persistent queries end up storing their results in a topic with the name of the stream or table.

Figure 10.4. When creating a new query from a persistent one it will inherit the data format by default



```
CREATE STREAM user_activity (
         first_name VARCHAR,
         . . .
) WITH (kafka_topic='user_activity',
     value_format='PROTOBUF'
);
```

The table inherits the Protobuf type for the values because that's what the base query has

```
CREATE TABLE activity_count AS
         SELECT
           last_name,
           COUNT(activity) AS ACTIVITY_COUNT
           FROM user_activity
          GROUP BY last_name
```

Registers Schema activity_count-value in Protobuf format w Schema Registry once

So this means if the value is in `Protobuf` format ksqlDB will register a new schema with Schema Registry using a subject of the stream or table followed by `value`. For example let’s revisit the `user_activity` stream, but this time we’ll say the value is uses `Protobuf` :

Listing 10.28. The user_activity stream in Protobuf

```
CREATE STREAM user_activity (first_name VARCHAR,
                             last_name VARCHAR,
                             activity VARCHAR,
                             event_time VARCHAR,
                             steps INT

     ) WITH (kafka_topic='user_activity',
            partitions=4,
            value_format='PROTOBUF',
            timestamp = 'event_time',
            timestamp_format = 'yyyy-MM-dd HH:mm:ss'
     );
```

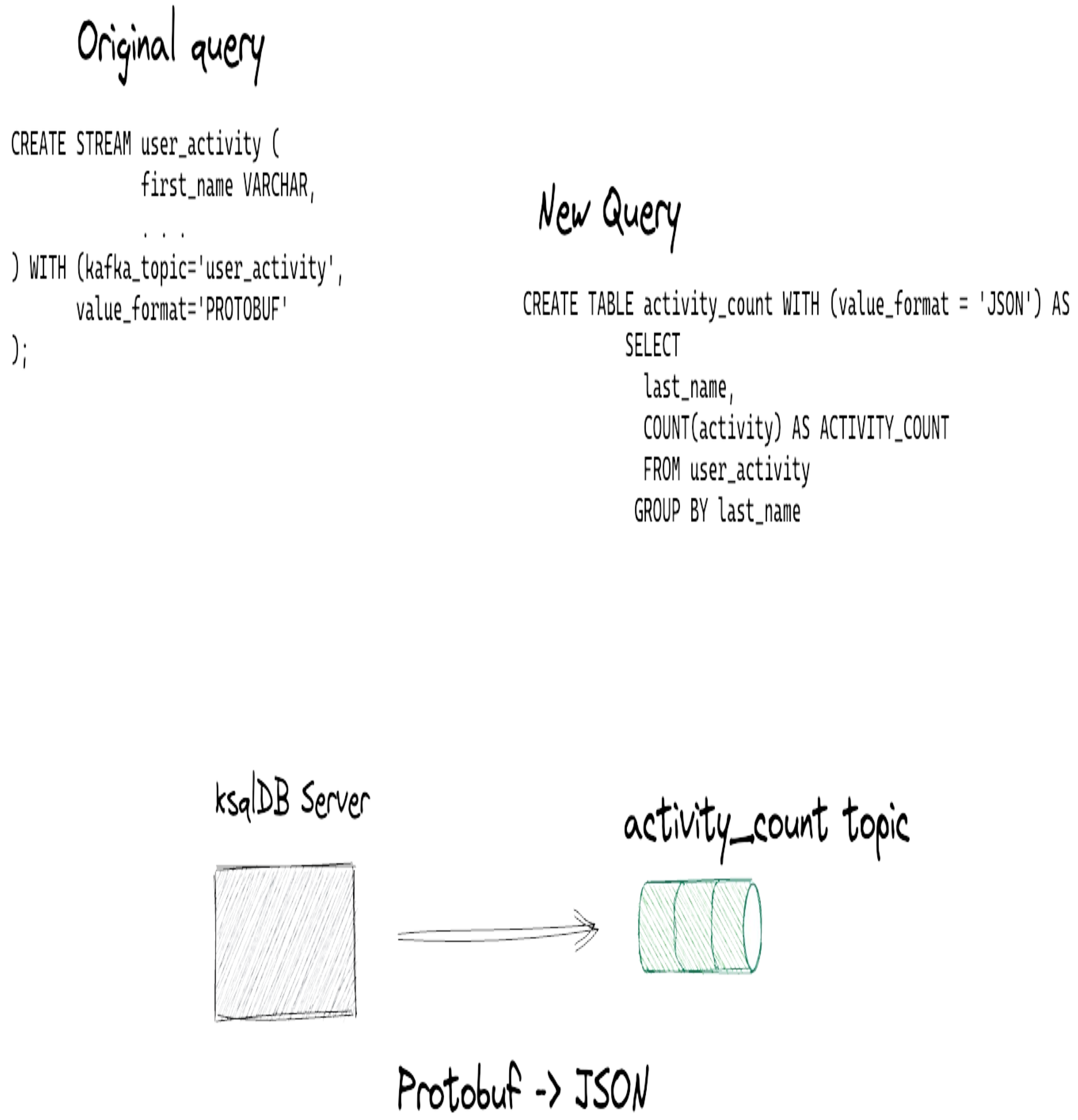
Even though we don’t have to have the column definitions there, we’ll have them there as it should help with the clarity of the example. Now let’s say you want a persistent query to count the number of activities by the user’s last name you’d end up with a persistent query looking like this:

Listing 10.29. Persistent query inheriting value format

```
CREATE TABLE activity_count AS
  SELECT
    last_name,
    COUNT(activity) AS ACTIVITY_COUNT
  FROM user_activity
  GROUP BY last_name
 EMIT CHANGES;
```

From creating this table, ksqlDB registers a schema named `activity_count-value` and it’s format is `Protobuf` since the source stream is in that format. But let’s now say that the materialized topic from the query needs to be in `JSON` format, as some of the downstream clients aren’t able to support another data format right now.

Figure 10.5. ksqlDB can change the datatype on the fly when creating a new persistent query



```
CREATE STREAM user_activity (
            first_name VARCHAR,
            . . .
) WITH (kafka_topic='user_activity',
      value_format='PROTOBUF'
);
```

```
CREATE TABLE activity_count WITH (value_format = 'JSON') AS
         SELECT
           last_name,
           COUNT(activity) AS ACTIVITY_COUNT
           FROM user_activity
          GROUP BY last_name
```

That’s not going to be a problem for ksqlDB, as you can seamlessly change the data format from the underlying persistent query by overriding the value data format like this:

Listing 10.30. Overriding the value format of a source stream

```
CREATE TABLE activity_count WITH (value_format = 'JSON') AS
```

```
  SELECT
    last_name,
    COUNT(activity) AS ACTIVITY_COUNT
  FROM user_activity
  GROUP BY last_name
 EMIT CHANGES;
```

Now the resulting topic `activity_count`, will contain records with values in `JSON`. You can use this ability to transform the record format in ksqlDB with source streams/tables and persistent queries. Since push and pull queries output their results directly back to the client in a deserialized format, there’s no option or need to convert the resulting format.

As we conclude this chapter I’d like to give a final example of converting the serialization format. We’ll cover a practical example of why you’d want to covert the format in the next chapter. Let’s say you have a stream of IoT data in Avro format, and you need to have all records in the stream converted to Protobuf to support more downstream clients. Here’s the stream you have in AVRO:

Listing 10.31. Stream defined in AVRO

```
CREATE STREAM IoT_TEMP_AVRO (device_id INT KEY, temp DOUBLE)
 WITH (kafka_topic = "iot_temp", value_format 'AVRO');
```

You want to create an identical stream, but in Protobuf format. To do this you can simply create a new stream by selecting everything from from the `IoT_TEMP_AVRO` stream like this:

Listing 10.32. Create a new stream in different serialization format

```
CREATE STREAM IoT_TEMP_PROTOBUF WITH (value_format 'PROTOBUF') AS
    SELECT * FROM IoT_TEMP_AVRO;
```

So now you have created an identical stream in a different serialization format with single line of SQL! Now that we’ve covered the core of ksqlDB, let’s move on to more advanced features including joins and aggregations.

10.5 ksqlDB advanced features

So far in this chapter you’ve learned how to use ksqlDB to create streams and tables, but usually you’ll need to use more advanced features to solve complex issues. Consider our scenario from a previous chapter where we have two streams of different purchases, one for coffee bought at the internal store cafe and the other representing all other purchases made in the store. To refresh your memory, we wanted to join purchases made within thirty minutes of each other to create a promotion for the customer. Let’s get started by first creating a stream for each category:

Listing 10.33. Creating ksqlDB streams for each purchase category

```
CREATE STREAM coffee_purchase_stream (custId VARCHAR KEY,
                                      drink VARCHAR,
                                      drinkSize VARCHAR,
                                      price DOUBLE,
                                      purchaseDate BIGINT)
     WITH (kafka_topic = 'coffee-purchase',
           partitions = 1,
           value_format = 'PROTOBUF',
           timestamp = 'purchaseDate'
     );


CREATE STREAM store_purchase_stream(custId VARCHAR KEY,
                                    credit_card VARCHAR,
                                    purchaseDate BIGINT,
                                    storeId VARCHAR,
                                    total DOUBLE)
     WITH (kafka_topic = 'store-purchase',
           partitions = 1,
           value_format = 'PROTOBUF',
           timestamp = 'purchaseDate'
     );
```

Now that you have your two streams, the next step is to setup the join itself, but before we do that let’s take a moment to discuss requirements. Just like in Kafka Streams, in order to perform a join both streams need be ***co-partitioned*** meaning that the underlying topics must have the same number of partitions and they are keyed the same (keys are the same field and type). In our case here, both topics have four partitions and the key for both streams is the customer id, so they are all set for joining. Now let’s take a look the SQL for the join:

Listing 10.34. Creating a stream-stream join for potential customer rewards

```
CREATE STREAM customer-rewards-stream AS
  SELECT c.custId AS customerId, #1
       s.total as amount, #2
       CASE       #3
         WHEN s.total < 25.00 THEN 15
         WHEN s.total < 50.00 THEN 50
         ELSE 75
        END AS reward_points
  FROM coffee-purchase-stream c
    INNER JOIN store-purchase-stream s
    WITHIN 30 MINUTES GRACE PERIOD 2 MINUTES  #4
    ON c.custId = s.custId #5
```

So what you've done here is to select one field from each stream, the customer id and the total amount of the store purchase. To determine the amount of reward points you using a `CASE` statement which ends up assigning different point levels depending on the total amount the customer spent in the store. `CASE` statements are elegant way of evaluating different conditions based on the value of a field that's part of the query.

Note

For stream-stream joins in ksqlDB other join types of LEFT OUTER, RIGHT OUTER and FULL OUTER they follow the same semantics you learned about with KStream joins.

But you're not limited to joining streams in ksqlDB, you can also perform stream-table joins. When using a stream-table join in ksqlDB, only new records on the stream side will trigger a result (just like Kafka Streams), so typically you'll use a stream-table join when you want to enrich the stream side by performing a lookup in the table.

For example consider the results of the stream-stream join you just implemented. One of the columns you projected into the join was the customer id, but you'd like to have more complete information about the customer. So to accomplish adding additional information would be to join the customer-rewards-stream with a fact table, members, which contains full details of all the shoppers who participate in rewards program.

Let’s say you have a sink connector that is exporting all the records from a "members" table into a Kafka topic named "rewards-members", so the first thing you’ll need to do is create a table in ksqlDB:

Listing 10.35. Create a lookup table in ksqlDB from an existing topic

```
CREATE TABLE rewards_members (member_id VARCHAR PRIMARY KEY,
                              first_name VARCHAR,
                              last_name VARCHAR,
                              address VARCHAR,
                              year_joined INT)
     WITH (kafka_topic = 'rewards-members',
           partitions = 1,
           value_format = 'PROTOBUF'
    );
```

Now that you have your table created you can now go about setting up a join with the customer-rewards-stream. But in this case not all customers are members of the rewards program, so you’re going to set up a LEFT OUTER join which will enable you to filter records later on that don’t contain customer information:

Listing 10.36. Stream - table left outer join for adding customer information when present

```
CREATE STREAM enriched-rewards-stream  #1
   WITH (kafka_topic='customer-rewards-stream', #2
          value_format='PROTOBUF') AS
   SELECT crs.custID as customer_id,  #3
          rm.first_name + ' ' + rm.last_name as name,
          rm.year_joined as member_since
          crs.amount as total_purchase,
          crs.reward_points as points
    FROM customer-rewards-stream crs
    LEFT OUTER JOIN rewards-members rm
              on crs.customerId = rm.member_id #4
```

Here you’ve created an enriched stream by joining the rewards stream with a customer information table and you’ve specified this as a `LEFT OUTER JOIN` meaning if the customer-rewards-stream does not find a correspond record in the rewards-members table, you’ll still get a join result but any of the fields representing the table will null.

This has been an example of a stream-table join and ksqlDB also supports table-table joins. What’s unique about table-table joins in ksqlDB is that in addition to the expected primary key joins, it also supports foreign key joins. You’ll need a foreign key join when the primary key of one table matches a non-primary-key column on another table. For example, let’s take the activity-count table you created earlier in the chapter and join it against the rewards-members table from the stream-table join example. I’ll repeat the definition of both tables here:

Listing 10.37. Two tables for performing a foreign key join

```
CREATE TABLE activity_count WITH (value_format = 'JSON') AS
  SELECT
    last_name,
    COUNT(activity) AS ACTIVITY_COUNT
  FROM user_activity
  GROUP BY last_name
 EMIT CHANGES;


CREATE TABLE rewards_members (member_id VARCHAR PRIMARY KEY,
                              first_name VARCHAR,
                              last_name VARCHAR,
                              address VARCHAR,
                              year_joined INT)
     WITH (kafka_topic = 'rewards_members',
           partitions = 3,
           value_format = 'PROTOBUF'
);
```

Let’s say that your fitness app was purchased by the same retail store we built the stream-stream and stream-table joins for and they’d like to award members of the rewards club points for store use based on their participation in the fitness app. Since you already have the table reward-members you should be able to join it against the activity-counts table.

Now the activity-counts table has a primary-key of `last_name` and the rewards-members primary key is the member’s id, but it does have a `last_name` column so we can join on the `rewards-members.last_name` as a foreign key. Since we’re joining on a column that’s part of the value, we won’t have the restrictions of co-partitioning. This is because we are joining against a value, and there’s no way to deterministically know which partition it belongs to since we put records on a partition by the key.

Tip

In you need to change a key for a stream or table in ksqlDB you’d use `SELECT FROM` and a `partitionBy=<column>` in the `with` clause to get the correct key.

So let’s create the join between these two tables:

Listing 10.38. Foreign Key join between activity_count and rewards-members

```
CREATE TABLE rewards-members-fitness-count AS
SELECT * FROM
activity_count ac JOIN rewards-members rm
   ON ac.last_name = rewards-members.last_name
EMIT CHANGES;
```

So without much effort at all you’ve joined the activity_count table with members information by using a non-primary key on another table.

Before we conclude this chapter on ksqlDB we should discuss one of the more powerful features of ksqlDB. We’ve discussed the different data formats ksqlDB supports, Avro, Protobuf and JSON. So far all the objects we’ve worked with have been flat, meaning there is one top level object and all the fields we want to access are attributes of that object. But what would you do when the data has nested structures? Consider the following JSON schema:

Listing 10.39. JSON with nested structures

```
"event_id": 1234,
  "school_event": {
       "type": "registration",
       "date": "2023-02-18",
       "student": {
             "first_name": "Rocky",
             "last_name": "Squirrel",
             "id": 1234567,
             "email": "rsquirl@gmail.com"

       },
       "class": {
             "name": "Geology-100",
             "room": "23RF",
             "professor": {
                  "first_name" : "Bullwinkle",
                  "last_name"  : "Moose"
                  "other_classes" : ["Geology-200", "Rocks-400",
                }

       }
  }
```

So this is a deeply nested JSON structure. As you can see from the schema here, that information is available, but how to model and access it? Fortunately ksqlDB makes accessing nested data easy.

To access nested data ksqlDB uses a data type of a `STRUCT`, which maps string (VARCHAR) keys to arbitrary values. You will use `STRUCT`s to describe the schema of the nested data when defining a stream. For example, using the JSON schema we just looked at, you'd define a stream for it this way:

Listing 10.40. Using the STRUCT datatype to represent nested data

```
CREATE STREAM school_event_stream (
  event_id INT,
  event STRUCT<type VARCHAR,  #1
               date VARCHAR,
               student STRUCT<first_name VARCHAR, #2
                              last_name VARCHAR,
                              id BIGINT,
                              email VARCHAR
                              >,
                class STRUCT<name VARCHAR,
                             room VARCHAR,
                             professor STRUCT<first_name VARCHAR,
                                              last_name VARCHAR,
                                              other_classes ARRAY
                                             >
                            >
                >
)
WITH (kafka_topic='school_events',
      partitions=1,
     value_format='JSON'
)
```

As can see from this code listing each time you define a nested object in the same manner you would for any column. First you provide the name followed by the type which is `STRUCT<` followed by the names and types of the fields on the object, when you reach the last field for the object you’d close it with a > character. You repeat this process each time you encounter an object in the schema.

To query the nested fields you provide the name of the outermost key and use a → to dereference from the object, again repeating as necessary. To see this in action, lets say you want to write a query that would make course suggestions for each given student based on the other courses taught by a professor of a class the student is currently attending. You’d write such a query in this way:

Listing 10.41. Query on nested data for course recommendations

```
SELECT
       event->student->id as student_id,
       event->student->email as student_email,
       event->class->professor->other_classes as suggested
FROM
   school_event_stream

EMIT CHANGES
```

So to access the nested data you use the `name→` pattern until getting to fields you’d like to retrieve in your query. You can follow the same pattern to access individual elements of an array or map. For example, let’s say you only want to offer one suggestion, so just access the first entry of `other_classes`, you’d update the query to this:

Listing 10.42. Query nested data and access a specific array entry

```
SELECT
       event->student->id as student_id,
       event->student->email as student_email,
       event->class->professor->other_classes[1] as suggested
FROM
   school_event_stream

EMIT CHANGES
```

Now your query will only offer one suggested course to take

Tip

To access individual elements from a nested `Map` you’d use `name→map_name['key']` and if the map entry was another `STRUCT` you could drill down using the same dereferencing syntax.

The book’s source code contains SQL files, the required docker compose files, and instructions for running examples shown in this chapter, found in streams/src/main/java/bbejeck/chapter_10.

10.6 Summary

1. ksqlDB is an event streaming database where you can build event streaming applications using the familiar syntax of SQL. The queries you write will continually evaluate events coming into a Kafka topic and may persist the results to a topic or return them directly to a client application.
2. You can create a `STREAM` or a `TABLE` in ksqlDB and they have the same semantics as the corresponding streams and tables in Kafka Streams. A `STREAM` is an unbounded stream of independent events and the `TABLE` is an update stream where event key-value pairs with the same key are an update to a previous one with the same key.
3. There different query types in ksqlDB - source queries where you create a `STREAM` or `TABLE` with a backing Kafka topic, persistent queries that select some or all of the columns from a source query, they persist their results to a Kafka topic. Persistent query results can be shared with multiple clients. A push query is one that selects a subset of columns from a persistent query but the results are streamed directly back to the

client. Push queries will run indefinitely until the client terminates the connections. Pull query results aren’t shared; ksqlDB will execute identical queries coming from different clients. A pull query executes once and terminates. A pull query has some limitations on the SQL statements it supports.

4. ksqlDB seamlessly supports Schema Registry serialization formats. Streams and tables inherit the key and value format of their backing topics or source streams and tables. You can easily change the format of stream or table by using a `WITH` clause and providing a different key and/or value format. ksqlDB will automatically register a schema when creating stream or table that is based off a persistent query. When defining a stream or table and you’re using Avro, Protobuf or JSON Schema format for your events, you don’t have to declare the column names and types in the definition.
5. ksqlDB offers a rich library aggregation functions like `COUNT`, `SUM`, and `AVE` out of the box. For the most part it probably has most math functions, but if there’s not something provided you can use a user-defined-function (UDF) to create your own.
6. You can create new streams by using stream-stream joins or enrich a stream with a stream-table join. Stream-Stream and Stream-Table joins require that both sides of the join are co-partitioned (same key type and number of partitions). You can also perform Table-Table joins with primary keys, but you can also perform a foreign key joins for cases where you want to join on a field that’s in the value of one of the tables.
7. You can query arbitrarily nested data using the `STRUCT` datatype to model the schema for a stream or table and then use the → operator to deference objects and drill down to the desired field.

11 Spring Kafka

This chapter covers

- Learning about Spring and why you’ll use it with Kafka
- Using Spring Kafka for building Kafka and Kafka Streams application
- Learning about Kafka Streams Interactive Queries
- Building an Ineractive Queries app with Spring Boot

In this chapter you’re going to learn about using an other open source library, Spring, to build Kafka and Kafka Streams applications. But before we get into that, let’s give some quick background on what Spring is and why you’ll use it. Spring has it origins as IoC (Inversion of Control) container originally developed by Rod Johnson. The inversion of control principal essentially means that the main program or appliction does not controll where its dependencies are cominging from. If you are familiar with Spring, you can skip this section and go directly to the next one where we dive into a Kafka producer and consumer appliction with Spring Kafka.

A non-IoC application, say a payment processing system, would directly instantiate all of the collaboratig components in the application, essentially it’s in control of the other components used. This control includes being aware of the concrete types of the components vs the interface. But with IoC the main application only has references to the interface types of the collaborators and instead of directly instantiatign them, they are injected into the application. This makes for much more flexible and testible applications, as you can change the implementation as needed and testing is far easier, since you control what’s injected, you can supply mock instances into the applicaiton.

Dependency injection is one way of achieving IoC, there’s an external mechanism that injects dependencies, either via constructors, getter methods or directly at the field level. That’s exacty what Spring provides, a mechanism for wiring up applications where each component only uses interfaces. Spring has grown over the years to be a large thriving project, providing much more than just a basic IoC container. I'm not going to cover the details of Spring, just what you need to get started using it with Kafka and Kafka Streams. For more complete coverage of Spring I would suggest starting with the following titles from Manning Books:

1. Spring in Action, Craig Walls
2. Spring Boot in Practice, Somnath Musib
3. Spring Security in Action, Laurențiu Spilcă
4. Java Persistence with Spring Data and Hibernate, Cătălin Tudose

As for why you'd want to use Spring, let's look at a concrete exmple of a payment system mentioned before. Without a doubt you payment application will require a network connection for receiving and sending payment information. Additionally, you'll not want to use raw network sockets for this network communication. Instead you'll want wrap up in a software component so that your payment processor doesn't need to know the details of connecting to the network and communicating with it.

So a basic skeleton of your payment processor class could look like this:

Listing 11.1. Payment Processor Class

```
public class PaymentProcessor implements Processor {

  private NetworkClient networkClient;


  public PaymentProcessor(NetworkClient networkClient) {
     this.networkClient = networkClient;
  }


 payment = networkClient.receive();
 // do some work
 networkClient.send(processedPayment);

}
```

From looking at the `PaymentProcessor` class you see that it has a dependency on the `NetworkClient` to complete its job. But notice that the code knows nothing of the `NetworkClient` other than the exposed methods on the interface. This is a very benefitial situation, as you don’t want the `PaymentProcessor` to have any knowledge beyond the contract specified by the interface. Why is this important? As time goes on with your project you’ll make changes and it might include changing the implementation of the `NetworkClient`, but from the way you’ve written the code here, that won’t matter, the `PaymentProcessor` only need something that implements the expected interface. Otherwise you’d have find all uses of the specific implementation and update the useage.

You reap an additional benefit when it comes to testing you want the test to run fast and you only want to validate the logic of the `PaymentProcessor`, so you can inject a "mock" `NetworkClient` that implements the interface but doesn’t actually connect to the internet, it will simply provide the canned information you’ve provided. We’ll go into more detail about mocks and different approaches for testing in another chapter.

So now, hopfully you can see the benefit of using and dependency injection approach to composing software applications, but the question remains how do I actually inject the different required classes? Enter the Spring container. Spring provides different annotations that you use to "annotate" the different relationship different classes have with each other let’s take a look at an example:

Listing 11.2. Annotations on classes specifying the relationship between them

```
@Component #1
public class PaymentProcessor implements Processor {

  private NetworkClient networkClient;

  @Autowired #2
  public PaymentProcessor(NetworkClient networkClient) {
     this.networkClient = networkClient;
  }
 ...
}

...

@Component#3
public class SecureNetworkClient implements NetworkClient {
  public SecureNetworkClient(...) {
    ...
  }

 ....
}
```

By providing these annotations when you start up the application, Spring will scan all the classes and wire-up the classes based on the annotations on them. So Spring takes care of putting all the pieces together for you. This concludes our brief introduction to the Spring framework. In the next section we’ll dive into using Spring to create Kafka enabled applications.

Note

Spring Kafka is a very comprehensive and it would difficult to cover everything in one chapter. We’re going to cover the basics to get you started using Spring to build Kakfa enabled applications. For more details see the Spring books from Manning mentioned above and also consult Spring’s excellent documentation at https://docs.spring.io/spring-kafka/docs/current/reference/html/

11.1 Building Kafka Enabled Applications

When building Kafka applications with Spring, you have two choices; you can build a more traditional application with Java configuration or you can use Spring Boot. Spring Boot provides the ability to quickly build standalone Spring enabled applications. We’ll show some examples of using both approaches for building a basic producer consumer applications.

The difference between the two is that Spring Boot is opininated on how to structure things and therefore can provide a higher level of abstraction, taking care of many of the details required to use Kafka with Spring. I’m covering the two approaches for completeness, but for the most part I’d reccomend using Spring Boot as it makes creating an application easier and it also provides great default options like a built in web-server which we’ll cover how this is useful a little later in the chapter.

Let’s say you are working for a startup that’s specializing in on-line loan applications for mortgages, car and business loans. The company, called Dime, is taking an approach of offering substantially lower interest rates and plans to become profitable by having a large volume of loans to make up for the reduced interest income. The plan is to provide quick turnaround on loan applications and operate in a very agile manner, automating the loan application process as much as possible. To achieve this, loan applications completed on the company website are forwarded to a Kafka topic, and there is a sophisticated underwriting application that will process the loans. The underwriting application sends the results to potentiall three topics, one each for accepted applications, rejected applications, and a qa department that will audit loans selected at random to ensure the rigor of the loan process.

We’re going to start with the Java configuration approach first.

11.1.1 Java config Kafka application

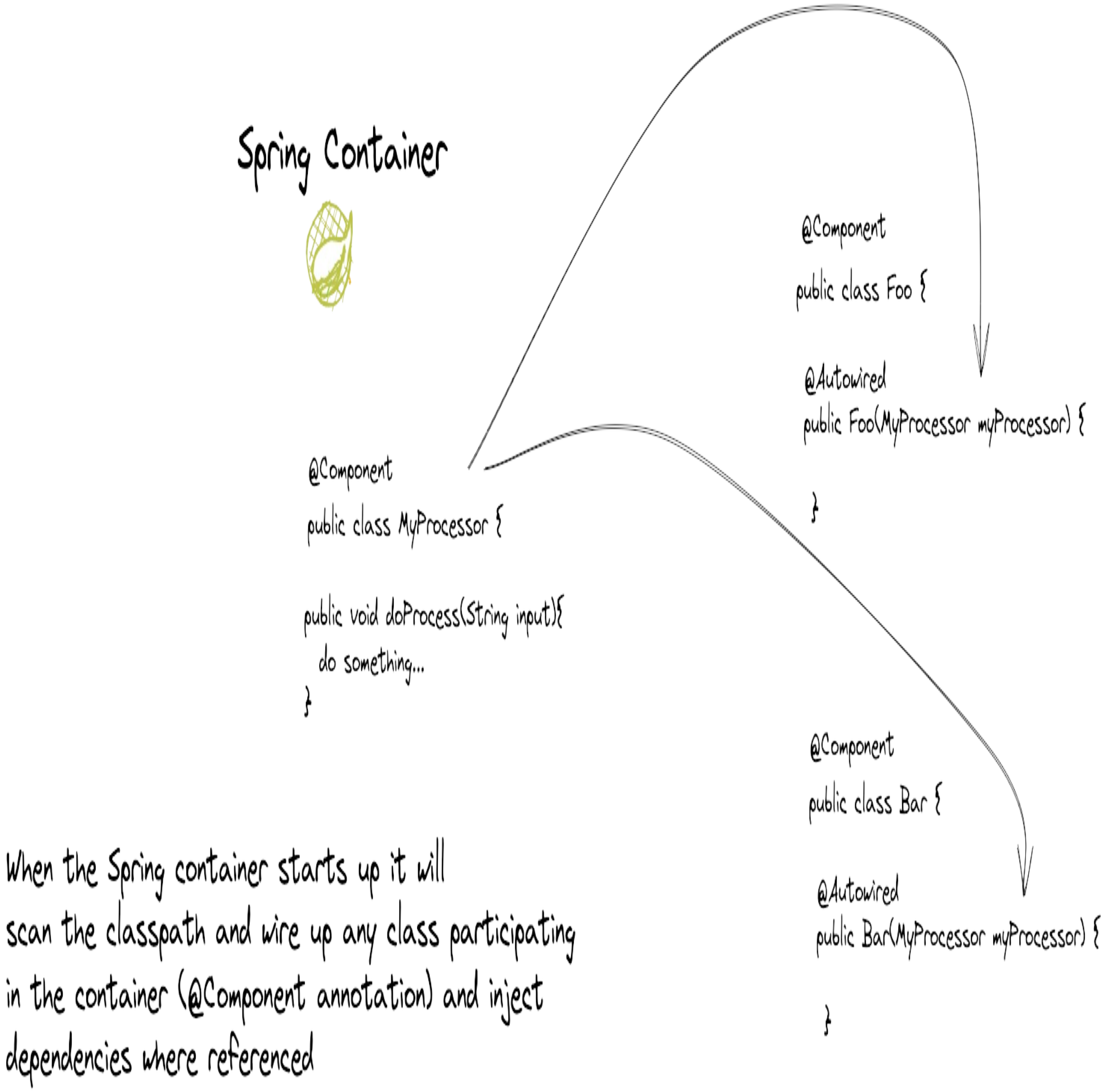
You’re going to build your application taking the following steps:

1. Create a class that contains configuration information. This class will contain properties used by different components, but since it’s centralized it will allow for quick changes with different settings.
2. Define the behavior you want for you consumer and producer components
3. Build a class that pulls all to components together and runs the functionality you’ve defined

Let’s create the configuration component first. The configuration class contains all of the information needed to connect to the Kafka cluster plus the topic names and other information that may change. By putting all of this information in a single class it makes it simple and less error prone since all configuration items are in one place.

When you start your Spring application, Spring scans the classpath and finding the classes with the `@Component` annotations and injects them where referended with an `@Autowired` annotation. Let’s look at another illustration depicting this process:

Figure 11.1. Classes with @Component annotations get injected into classes with a reference to it with @Autowired



```
@Component
public class MyProcessor {

public void doProcess(String input){
  do something...
}
```

```
@Component
public class Foo {

 @Autowired
 public Foo(MyProcessor myProcessor) {

 }
```

```
@Component
public class Bar {

 @Autowired
 public Bar(MyProcessor myProcessor) {

 }
```

When the Spring container starts up it will scan the classpath and wire up any class participating in the container (@Component annotation) and inject dependencies where referenced

What’s important to know here is that by default, any class with the `@Component` annotation is a singleton, Spring injects the same instance into ***all*** classes that have a reference to it. This is an important point to remember if you have any mutable state in your object as you’ll need to ensure serial access to the state with synchronization.

With that background information completed let’s go ahead jump into the application code. First let’s take a look at how the project is structured.

Listing 11.3. Structure of the Spring Java configuration application

```
bbejeck
  spring
    application
      |_ CompletedLoanApplicationProcessor #1
      |_ NewLoanApplicationProcessor    #2
    java
      |_ LoanApplicationAppJavaConfiguration #3
      |_ LoanApplicationProcessingApplication #4
```

Note

Some details have been left out of the directory structure for clarity

These classes are contained in two different packages, but with Spring annotations, we’ll have no troubles getting them all together for one complete application. Let’s review the code from a Spring perspective, that is how things are going to be wired up. The first place we’ll look is the `LoanApplicationAppJavaConfiguration` as this drives most of the configuration process, starting with the class declaration.

Listing 11.4. Class declaration for the LoanApplicationAppJavaConfiguration

```
@Configuration #1
@EnableKafka #2
@PropertySource("classpath:application-java-config.properties") #
public class LoanApplicationAppJavaConfiguration {
...
}
```

Let’s quickly describe each annoation and it’s role in the Spring application. A class marked with `@Configuration` means it provides one or more methods annotated `@Bean` which means the Spring container will handle those at runtime to create objects to satisfy dependency needs. If this is not entirely clear to you, we’ll fill in the details in the next section.

The `@EnableKafka` annotation makes it possible to use message-driven-POJO (a POJO stands for Plain Old Java Object) with a `@KafkaListener` annotation defined as a component in a Spring container. A message driven POJO meands you can define a method in a class to recieve Kafka messages, but the class or method itself is not necessarily tied to Kafka. It accomplishes this by taking an expected `ConcurrentKafkaListenerContainerFactory` defined in the configuration class and creating a `KafkaListenerContainer` for each `@KafkaListener` found in the container.

Finally the configuration class uses the properties file defined in the `@PropertySource` source annotation to fulfill property placeholders. A propery placeholder any place in your code where you don't want a hard-coded value - meaning you want set its value at runtime from a properties file for the corresponding variable.

The advantage of using properties in the manner is it makes your applications flexible. For example setting the endpoint for your producer to send messages to a Kafka broker can change, by using a properties file to define that endpoint means if you need to change it, all you do is update the property in the file and it's seamlessly transferred to the code, you don't need to update the application itself. Using different environments (development, staging, production) are an excellent example of for using propeties, to deploy your application in a production environment shouldn't require a code change, just different properties for connections to various services.

If you found this section a little abstract, we're going to make it clear in the next section where we'll show how this annotations at the class declaration work.

Listing 11.5. A deeper look into the configuration class

```
@Configuration
@EnableKafka
@PropertySource("classpath:application-java-config.properties")
public class LoanApplicationAppJavaConfiguration {

  @Value("${bootstrap.servers}") #1
  private String bootstrapServers;
  @Value("${loan.app.input.topic}") #2
  private String loanAppInputTopic;

  // Other properties omitted for clarity

 @Bean
 public ConcurrentKafkaListenerContainerFactory<String, LoanAppli
    ConcurrentKafkaListenerContainerFactory<String, LoanApplicati
                new ConcurrentKafkaListenerContainerFactory<>();
    kafkaListenerContainerFactory.setConsumerFactory(consumerFact
    return kafkaListenerContainerFactory;
 }


@Bean
public ConsumerFactory<String, LoanApplication> consumerFactory()
  return new DefaultKafkaConsumerFactory<>(consumerConfigs(),
                new StringDeserializer(),
                new JsonDeserializer<>(LoanApplication.class)); #
 }

// Other beans left out for clarity

}
```

Here we’re taking a bit of a deeper look into the configuration class. At the top you can see how we inject property values into the class, by placing a `@Value` annotation directly above the field we want to set the value into. By using this approach, we make our application code more flexible, because we are free to change the property values and our code doesn’t need to change at all since none of the values are hard-coded into it.

Note

You can also use this approach for injecting propery values into any class that is part of the Spring container.

There are several other properties we define here, but I’m not showing them here as it’s repetitive but you can consult the source code at bbejeck.spring.java.LoanApplicationAppJavaConfiguration.java class in the `spring-kafka` gradle module.

Now let’s turn our attention to the methods marked with the `@Bean` annotations. The container uses these methods to construct the infrastructure code needed to perform the Kafka consumer activity. You can also create methods here that supply objects for dependency injection used by other components in the Spring container.

When using a `@Bean` annotation the name of the bean in the Spring context is the method name. This is important to know if you needed to access a Spring bean once you start the application. To provide a different name for the bean you’d supply it with the annotation. For example if we wanted to provide a custom name for the container listener factory we would do something like this `@Bean("loanApplicationContainerFactory")`. We’ll see examples of when you’ll want to provide the name for a Spring bean and how to access them later in the chapter.

So at annotation three you see the `kafkaListenerContainerFactory` method returning a `ConcurrentKafkaListenerContainerFactory` instance. The Spring container uses the factory to create a `KafkaListener` instance ***for each method*** that you’ve annotated with `@KafkaListener`. The `KafkaListener` is an abstraction over a `KafkaConsumer`. But by using annotations in this manner, Spring allows to you enable any class as a "message driven bean" capable of recieving Kafka records. We’ll get into more details of what a message driven bean is and using listener container factories later in the chapter.

The next bean at annotation four is a `KafkaConsumerFactory` which the Spring container will use to create instances of a consumer, or more precisely the listener factory will use it to create consumer instances. If you look into the details of the method, you’ll see the constructor parameters passed to the consumer factory. The first parameter is Map containing the properties (the method not shown here) for constructing the consumer, a key deserializer and the value deserializer. So what you’ll notice here is that the container listener factory is tied to a specific type, which means when you won’t be able to reuse the container listener across the application for all topics, you’ll need to create a factory for each topic with different types. We’ll cover using multiple container factories later in the chapter.

So far we’ve shown injecting the configuration properties and setting up the infrastructure for consuming Kafka records. But you’ll want to produce messages as well from your application, so we’ll cover that next.

Listing 11.6. Configuration required for producing messages

```
private Map<String, Object> producerConfigs() {  #1
  Map<String, Object> configs = new HashMap<>();
  configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapS
  configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                             StringSerializer.class);
  configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                             JsonSerializer.class);
  return configs;
}

@Bean
public ProducerFactory<String, LoanApplication> producerFactory()
    return new DefaultKafkaProducerFactory<>(producerConfigs());
}

@Bean
public KafkaTemplate<String, LoanApplication> kafkaTemplate() { #
    return new KafkaTemplate<>(producerFactory());
}
```

Here you see the configuration required for producing records to Kafka in your application. But you’ll notice that there’s a `KafkaTemplate` bean and it uses the `ProducerFactory`, in a Spring application you won’t interact directly with a `KafkaProducer`. Instead all operations for producing to a Kafka topic are handled by the `KafkaTemplate`, but under the covers it’s still using a producer. The `KafkaTemplate` provides some convenience methods over working directly with the producer, but the abstraction won’t change too much how you produce messages with a plain `KafkaProducer`, there’s a few changes and we’ll cover those a bit later on in the chapter.

What is the same as working with a plain producer is that the `KafkaTemplate` is bound to a particular class for producing to Kafka. In the producer configuration Map at annotation one, you’ll notice you provide specific serializing classes for the key and value of the Kafka you’ll send to the producer. So if you require to produce different types to more than one topic you’ll have to provide a separate `KafakTemplate` for each class type. We’ll cover sending different record types later on in the same section where we take a deeper look at the `ConcurrentKafkaListenerContainerFactory`.

There’s one last part of the configuration we should cover before moving on. In a production Kafka environment it’s typical that individual developers wont' have the the responsibility for creating topics, that’s handled usually handled by someone in the cluster operations role. But for local development, that is another matter.

The Spring-Kafka library provides for automatically creating topics for you when starting the Spring container, you simply need to provide some minor infrastucture in the form of Spring Beans for a Kafka `AdminClient` and `NewTopic` instances for each topic you want created. Let’s take a look at the code you need to provide in your configuration class:

Listing 11.7. Cofiguration code required for auto topic creation

```
@Value("${num.partitions}")
private int partitions;

@Value("${replication.factor}")
private short replicationFactor;

@Value("${loan.app.input.topic}")
private String loanAppInputTopic;

@Bean
public KafkaAdmin kafkaAdmin() { #1
    return new KafkaAdmin(Map.of(
                         AdminClientConfig.BOOTSTRAP_SERVERS_CONF
                         bootstrapServers));
}

@Bean
public NewTopic loanAppInputTopic() { #2
    return new NewTopic(loanAppInputTopic,
                        partitions,
                        replicationFactor);
}
```

So by providing the `KafkaAdmin` and `NewTopic` Spring Beans, when you start the Spring container (when you start the application), any of the `NewTopic` instances in the configuration class result in the creation of the topic it respresents, if it already exists, that’s fine the original topic remains. Also take note here of the use of injected properties for the topic name, partitions and the replication factor. Should these values differ from environment to environment, your configuration code is blissfully unware of these changes, and will work with whatever values are provided to them.

At this point we’ve fully covered how you would use basic Spring-Kafka (non Spring Boot) configuration class for getting a Kafka application up and running. Before we move on to how you wrap all this up and start the application, let’s take a look at how you would do provide the same configuration in a Spring Boot application, I promise it will worth your time as it will greatly simplify things for you in most cases.

Listing 11.8. Class declaration for configuration class with Spring Boot

```
@Configuration  #1
@EnableKafka    #2
public class LoanApplicationAppBootConfiguration {

...
}
```

So with the class declaration we can see one simplification right away. You don’t need to provide a `@PropertySource` annotation, when using Spring Boot, it will automatically look in the `src/main/resources` directory for a file named `application.properties`. If you give the file a different name or place it in a different location, then you’ll need to tell Spring where to find it like we did with our previous non-Spring Boot application previously. But savings in configuration don’t stop there, lets take another look at the configuration class:

Listing 11.9. Basic configuration is substantially less with Spring Boot

```
@Configuration
@EnableKafka
public class LoanApplicationAppBootConfiguration {
    @Value("${application.group}")
    private String groupId;

    @Value("${loan.app.input.topic}") #1
    private String loanAppInputTopic;
    //Other configurations left out for clarity

   @Bean
   public NewTopic loanAppInputTopic() { #2
        return new NewTopic(loanAppInputTopic,
                            partitions,
                            replicationFactor);
   }

   //Other NewTopic beans left out for clarity
```

Other than some additional fields for injected property values and `NewTopic` beans for creating the required topics, this all all there is to the configuration class for the Spring Boot application! As you can see, using Spring Boot removes most of the infrastructure configuration required for a Spring Kafka application. This simplification isn’t always the case, as we’ll see later in the chapter when we change the applciation requirements. But for the cases where you just need the Kafka infrastructure classes as they come straight out of the box, using Spring Boot is a faster path for development.

For the most part, I’d reccomend using Spring Boot most of the time when building a Spring Kafka application, as the savings in configuration effort are worth it.

Before we move on to our detailed look into the specific Spring Kafka components, let’s take a look at how you’d start the Spring Boot application.

Listing 11.10. Main class for starting a Spring Boot application

```
@SpringBootApplication(scanBasePackages = "bbejeck.spring.applica
public class LoanApplicationProcessingApplication {

public static void main(String[] args) {

 SpringApplicationBuilder applicationBuilder =
  new SpringApplicationBuilder(LoanApplicationProcessingApplicati
            .web(WebApplicationType.NONE);        #3
    ConfigurableApplicationContext context = applicationBuilder.r
   // Some details left out for clarity
 }
}
```

So as you’ve come to expect at this point, to create a Spring Boot application, you just need to provide a `@SpringBootApplication` annotation at the top of the class with `main` method for starting it. We also provide the base packages containing the different components that we want the Spring container to pick up and include.

At annotation two, you see the `SpringApplicationBuilder` class which we use for building the applciation, this also creates an `ApplicationContext` which is the main interface for configuring the different components specifed for the Spring Container. What you see at annotation three is specifying that you don’t want a web application started.

By default, Spring Boot starts a web server (Apache Tomcat is the default), but for our purposes here, we won’t need it, so we set the type to `NONE`. Later on in the chapter, you’ll build another Spring Boot application where you’ll need the embedded web server. At annotation four your starting application and storing the returned `ConfigurableApplicationContext` which as mentioned before, you can use to retrieve application components. Let’s take a look at how you can use the `ApplicationContext` to retrieve a Spring component once you’ve started built the application:

Listing 11.11. Retrieving a Spring bean from the ApplicationContext

```
ConfigurableApplicationContext context = applicationBuilder.run(a
MockLoanApplicationDataGenerator producer =
     context.getBean(MockLoanApplicationDataGenerator.class); #2
producer.sendData();
```

So once you’ve started the application, we need to produce records to Kafka. So to do that we’ve created a class to generate some mock loan application records and once we’ve started the application we retrieve the `MockLoanApplicationDataGenerator` and start producing records. You could also retrieve it by using a String with a name provided with the `@Bean` annotation.

So far we’ve covered how to configure a Spring-Kafka application and how you would start it. What’s next for us to go over is the different components of the application and how it all ties together. We won’t focus to much on the business logic of the application itself, as that’s not what really important here, in the applications you’ll develop you will provide the specific logic needed.

11.1.2 Spring Kafka application components

For your loan processing application there are two main components, one to recieve the application information and apply the approval algorithm to each incoming record which contains all the information required to process it. Once this first processor applies the algorithm to a given application, meaning it’s either approved or not, the result gets produced to a Kafka topic, one for approvals, another for rejections, and a third topic for quality assurance. A certain number of processed loan applications are chosen at random for review (by a human!) to make sure the alogorithm is performing as expected.

With that background in mind, let’s take a look at the `NewLoanApplicationProcessor` starting with the class declartion and constructor first:

Listing 11.12. NewLoanApplicationProcess declaration and constructor

```
@Component  #1
public class NewLoanApplicationProcessor {

  @Value("${accepted.loans.topic}")  #2
  private String acceptedLoansTopic;

  @Value("${rejected.loans.topic}")
  private String rejectedLoansTopic;

  @Value("${qa.application.topic}")
  private String qaLoansTopic;

  private final KafkaTemplate<String, LoanApplication> kafkaTempl

  @Autowired #4
  public NewLoanApplicationProcessor(
           KafkaTemplate<String, LoanApplication> kafkaTemplate)
      this.kafkaTemplate = kafkaTemplate;
  }
```

At the top where you declare the class name there is a `@Component` annotation. When starting the application, the Spring container scans for any classes with this annotation and includes them in the application context. Having a `@Component` at the top of the class allows for having dependencies injected into it or for it to be used as a dependency where other classes have a reference to it.

Next you see the injection of propties for the variables containing the name of the different output topics, and this should look familiar as you saw the same thing in the configuration class. Finally take a look at annotation three and four. Annotation three is the variable declared for a `KafkaTemplate` instance and we've decorated the constructor with an `@Autowired` annotation, which instructs the Spring container we want it to provide any of the parameters found there. In our case we'll get the `KafkaTemplate` created at container startup.

Note

You can also have autowired dependencies at the field level.

Now let's take a look where the "rubber meets the road" so to speak in the method where you handle the loan application processing:

Listing 11.13. Loan processing handling in the method

```
@KafkaListener(topics = "${loan.app.input.topic}", #1
              groupId = "${application.group}") #2
public void doProcessLoanApplication(LoanApplication loanApplicat

  // Most details left out for clarity
  boolean loanApproved = debtRatio <= 0.33 && loanApplication.get
  String topicToSend = loanApproved ? acceptedLoansTopic : reject

  LoanApplication processedLoan = LoanApplication.Builder.newBuil
  kafkaTemplate.send(topicToSend, processedLoan.getCustomerId(),
  if (random.nextInt(100) > 75) {
      kafkaTemplate.send(qaLoansTopic, processedLoan.getCustomerI
  }
}
```

By placing the `@KafkaListener` annotation at the top of this method, the Spring container uses the `ConcurrentKafkaListenerContainerFactor` to create a `KafkaListener` instance that wraps a `KafkaConsumer` that will consume records from the topic(s) specified in the `@KafkaListener` declaration. We'll explore the relationship between listeners, consumers, and topics soon, but let's continue for the moment with specifying listener itself.

At annotation one you're specifying the topic name for the listener, which under the covers, a `KafkaConsumer` subscribes to, then at annotation two, you're setting the consumer group id for the underlying consumer. For both attributes you'll notice you're not supplying hard-coded values, but instead you're utilizing property replacements, which again gives you more flexible code. Should the topic(s) or group id need to change, you only need update the properties file, versus changing and re-compiling your code.

You also see here the idea of a message-driven-pojo; there's nothing specific to Kafka about this class. But by applying one line of code, you've taken a basic Java class and converted into one that can handle receiving messages from a Kafka broker.

After you've processed the loan application (I've intentionally left the code out here as it's not importing to learning how to use Spring), you configure the destination topic for the loan based on its approval status. Then you use the `KafkaTemplate` to produce the processed loan application record back to a Kafka topic.

Here, you'll notice some of the convenience the `KafkaTemplate` provides, you're only providing a topic name, key, and a value. If you recall from our chapter on Kafka clients, when sending a record with the `KafkaProducer`, you first need to create a `ProducerRecord` instance to send to Kafka. The `KafkaTemplate#send` method returns a `ListenableFuture<SendResult<K, V>>` object and to get the result of the send you'd need to wait for the future to complete by executing the `ListenableFuture#get` method, but that would block the application until the `get` method returned a result. You can provide the `ListenableFuture` a callback that will process the result of the send asynchronously.

In our example here, you're not capturing the returned `ListenableFuture` but we’re going to revisit how you’re using the `KafkaTemplate` among other things and update the loan processing application. But before we do that, we have one more piece of the application to consider, the post loan processing.

Listing 11.14. Post loan processing class

```
@Component  #1
public class CompletedLoanApplicationProcessor {

@KafkaListener(topics = "${accepted.loans.topic}", #2
              groupId = "${accepted.group}")
public void handleAcceptedLoans(LoanApplication acceptedLoan) {
    ....
}

@KafkaListener(topics = "${rejected.loans.topic}", #2
              groupId = "${rejected.group}")
public void handleRejectedLoans(LoanApplication rejectedLoan) {
   ....
}

@KafkaListener(topics = "${qa.application.topic}", #2
               groupId = "${qa.group}")
public void handleQALoans(LoanApplication qaLoan) {
   ....
}

}
```

We won’t spend much time here as you’ve already learned about the `@Component` and `@KafkaListener` annotations, but there are is some additional information we’ll get into here about the `@KafkaListener`. I said before that the `@KafkaListner` annotation means the Spring container creates `KafkaListenerContainer` for each annotation it encounters when starting the application. Even if you provide another method with `@KafkaListener` (and a different group id) it will create a separate listner container.

This means since you have four methods decorated with `@KafkaListener`, there will be four `KafkaListenerContainer` instances running for your application. Each listener container will wrap one `KafkaConsumer` subscribed to the topic(s) specified in its declaration. The consumer created by the container factory will consume from all paritions of the topic. Before we move on to the next section, there’s one final subject to cover with the `@KafkaListener`.

In all the examples seen so far, you’ve placed the annotation at the method level. You can place it at the class level as well and that involves some additional work on your part. Let’s look at simple example:

Listing 11.15. KafkaListener at the class level requires handler annotations on the method

```
@Component
@KafkaListener(topics = "${loan.app.input.topic}", groupId = "${a
public class NewLoanApplicationProcessorListenerClassLevel {

  @KafkaHandler      #2
  public void doProcessLoanApplication(LoanApplication loanApplic
      // Handle the loan application
  }

  @KafkaHandler(isDefault = true)  #3
  public void handleUnknownObject(Object unknown) {
     // Handle the unknown object
  }
```

As you can see here, setting the class as the `KafkaListener` is simple as placing the `@KafkaListener` at the class declaration along with the `@Component` annotation. The extra steps you need to take are to have ***at least*** one method annotated with `@KafkaHandler`. Here you can see we added `@KafkaHandler` to the `doProcessLoanApplication` and another `handleUnknownObject` marked as the default method with the `isDefault` attribute. When using the entire class as a `KafkaListener` Spring determines the method to use based on the parameter type of its signature.

This selection process means that all `@Handler` methods ***must have a single parameter*** and the different methods can’t have any ambigutity between their parameter types. In this class level `@KafkaListener` example we’ve added a handler for a Kafka record that contains a value with a type other than the `LoanApplication`, but it’s not representive as the best use of using the `KafkaListener` at the class level, I’m including this here for completeness. I’m going to assert that the canonical use of a listener at the class level would be when you are consuming mulitple types from a single topic - for example:

Listing 11.16. KafkaListener at the class level for consuming multiple types

```
@Component
@KafkaListener(topics = "${multi.topic.input}", groupId = "${mult
public class NewLoanApplicationProcessorListenerClassLevel {

  @KafkaHandler      #2
  public void doSomethingWithLong(String someString) {
      // Do something with a String type
  }

  @KafkaHandler      #3
  public void doSomething(Long longNumber) {
      // Do something with the Long object
  }

  @KafkaHandler      #4
  public void doSomething(Double doubleNumber) {
      // Do something with the double
  }

  @KafkaHandler(isDefault = true)  #5
  public void handleUnknownObject(Object unknown) {
     // Handle the unknown object
  }
```

Here each method has a distinct type for Spring to make the selection for handling the records. So the question is, when would you choose to place the `@KafkaListener` at the class level vs. the method level? While there isn’t a hard and fast answer, I’m going to take the opininated approach that one should always favor using a listen at the method level. Topics should for the most part represent a single event type and have an appropriate name that’s easy to reason why that topic exists. Of course there are always exceptional situations that you’ll need to account for, and if one of them is multiple types in a single topic then using a listener at the class level is one way to handle it.

So that wraps up our coverage of building your first Spring-Kafka application, but we’re not going to stop here. There’s more advanced functionality we can cover that should be more broadly applicapable to you for creating an event stream application. For single partition topics or topics with low traffic levels a single consumer for all partitions may be sufficient. But what if we want a faster way to consume, i.e. not on single thread?

Additionally, what about getting the key of the Kafka record and other metadata (timestamp, partition) or sending records of different types? That’s exactly what we’er going to cover next.

11.1.3 Enhanced application requirements

You’ve gotten your online loan operation off the ground and things are going well. But there’s some changes that will make the application better. Those changes you’ll need to make are:

1. Add more partitions to the input topic
2. With more partitions, parallelize the application so that you have a consumer per partition
3. Capture the key and timestamp of the incoming record
4. Track the offset and timestamp of records produced

While that seems like quite a list, fortunately the changes needed to accomodate them are easily achievable. We’re only going to focus on the changes you’ll need to make for the Spring-Kafka application, I’m going to assume changes like partition number and domain objects you already know how do (the updates will be in the source code for you to examine).

Let’s take on increasing the partition count and how you’re going to increase the concurrency of your application to improve throughput. Chapter four covered the Kafka clients and we discussed the unit of parallelization of a Kafka topic is the partition. Generally speaking to increase the througput of a Kafka application, you add more partitions (or over partition at the beginning with growth in mind) with the aim of assigning a single consumer for each partition. By using a dedicated consumer for each partition you can maximize the throughput of the application (I’m making a generalization here, of course there’s always exceptions and different considerations that can occur).

You’ve done some analysis and determined that the input topic for the loan application should have three partitions for optimal throughput. This number takes into account the current level of applications and the increase you expect in the near future. You’re also going to increase the partitions of the post-processing topics, but to a lesser degree, with an increase to two partitions each. The thought process behind the smaller partition increase is that each loan application will only exist in one of two states - approved or rejected, so with an approval rating hovering 50%, each path in the post processing will only need to accomodate half of the expected max loan application traffic.

Earlier in the chapter you saw that by using Spring Boot and the `@EnableKafka` annotation, the Spring container automatically created a `ConcurrentKafkaListenerContainerFactory` for you. To refresh your memory the listener container factory creates a `KafkaListenerContainer` for each method decorated with `@KafkaListener`. So if you increase the number of partitions in your input topic, as things stand now you’ll have a single consumer for *all* partitions, but you want to have a consumer for *each* partition. The good news is that while Spring Boot provides a lot of functionaliy out of the box, you’re always free to provide customized configurations, which we’ll do now.

The first step is to create a Spring Bean for a `ConcurrentKafkaListenerContainerFactory`, but we’re going to set a specific property, `concurrencyLevel` :

Listing 11.17. Increasing the concurrency level for a container listener factory

```
@Bean
public ConcurrentKafkaListenerContainerFactory<String, LoanApplic
                    kafkaListenerContainerFactory() {  #1
    ConcurrentKafkaListenerContainerFactory<String, LoanApplicati
           kafkaListenerContainerFactory =
            new ConcurrentKafkaListenerContainerFactory<>();
    kafkaListenerContainerFactory.setConsumerFactory(consumerFact
    kafkaListenerContainerFactory.setConcurrency(partitions);  #2
    return kafkaListenerContainerFactory;
}
```

This configuration here is very similar to the one you created in the non-Spring Boot application earlier with one distinct difference. We’ve used the `setConcurrency` method, setting it to the number of configured partitions. The impact of setting the concurrency level to the same number of partitions means you get a `KafkaConsumer` per partition, which is what you’ll need for maximum throughput.

Another point to consider is that we’ve used the same name for method as the default expected by the container - why is that? By using `kafkaListenerContainerFactory` for the method name (remember without providing a `name` attribute the method name becomes the bean name), the Spring container will pick up your custom container factory instead of creating the default factory with the same name.

So using this "shadow" bean naming process, your processing class will automatically use the updated container factory with no code changes, which keeps your code flexible as it will pick up and use either container.

That’s not to say you should always use the same name when overriding a Spring Boot default. When using a custom factory can give it any name you want, you’ll just need to explicitly tell the `KafkaListener` which container factory to use by adding an additional attribute, `containerFactory` to the `@KafkaListener` annotation:

Listing 11.18. Adding the custom container factor for the listener

```
// In the configuration class
@Bean("custom-container")  #1
public ConcurrentKafkaListenerContainerFactory<K, V>
                                       customContainerFactory()


// In the @Componet class
@KafkaListener(groupId = "${application.group}",
                containerFactory = "custom-container", #2
```

Above is the alternative to providing a custom container. In the configuration class you create the custom container and you supply a name to the `@Bean` annotation and in the `@Component` class you use the same name for the `containerFactory` attribute.

Note

For increasing the concurrency of the `ConcurrentKafkaListenerContainerFactory` you can also provide a configuration `spring.kafka.listener.concurrency` and set it equal to the desired number. In our case it could be something like `spring.kafka.listener.concurrency=${num.partitions}` assuming the `num.partitions` comes earlier in the `application.properties` file. I’ve chosen to create a custom container factory to serve as an example of providing an override to the defaults provided by Spring Boot. Another approach would be to provide a `concurrency` attribute directly with `@KafkaListener` annotation.

Now that you’ve achieved the desired consumer-per-partition there’s an additional consideration you’ll have to take into account, you now have a multi-threaded processing application.

The Spring container creates a separate thread equal to the number you set for the level of concurrency. Creating new threads is expected and is essential for increased throughput. But this means each thread will call the listener method _*concurrently*_ as well. The concurrent calls are fine as long as the method is _*thread-safe*_ i.e. there’s no shared mutable state.

Tip

It’s easy to find the threads associated with particular Kafka listener by adding adding an `id` attribute to the `@KafkaListener` annotation. The Spring will use the id you provide as part of the thread name making it easy to identify which threads are consuming from different topics.

In the example code there is a single `KafkaTemplate` instance that is shared across all threads, but since the underlying `KafkaProducer` is thread-safe itself, the template is as well. It would be wise to keep the Kafka listener methods stateless otherwise you’ll need to add synchronization to your code to make sure you get deterministic results.

Note

I’m not going to cover Java threading or synchronization here but a quick Google search on the topic will yield plenty of resouces for you explore the subject.

Before we move on we have two more applciation improvements you still need to implement from our list above. To refresh your memory the remaing requirements are:

1. Capture and log the key and timestamp of the incoming loan application
2. When producing the processed loan result, log the offset and timestamp of the produced record. Also if there’s an error producing the record, you won’t have an offset and timestamp, so you’d like to log out the error as well.

For getting the key and timestamp, as well as other metadata associated with the incoming Kafka record you’ll apply `@Header` annotations to addtional parameters to your listener method. So the changes you’ll make to your loan processing class will look like this:

Listing 11.19. Getting the message key and timestamp from the incoming Kafka record

```
public void doProcessLoanApplication(LoanApplication loanApplicat
          @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long timestamp
          @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) String key)
```

So by adding the `@Header` anntation with the desired specific header you can now retrieve the original timestamp and message key. There are other header values available such as the topic, partition, and offset.

Tip

When extracting header information for consumed records make sure to specify `KafkaHeaders.RECEIVED_X` the `KafkaHeaders` class constants for both producer and consumer records and the ones for the consumer records start with `RECEIEVED`.

Now that you’ve retrieved the message key and the timestamp of the incoming record your next task is to log the offset and timestamp of the *produced* record after loan processing.

Earlier in the chapter we demonstarted thatthe loan application processing class, `NewLoanApplicationProcessor` after processing a loan, produces a record back to Kafka with the approval status of the loan determining the topic. To refresh your memory here’s the line of code responsible for producing the record:

Listing 11.20. Producing a processed loan

```
 kafkaTemplate.send(topicToSend, processedLoan.getCustomerId(), p
```

We also mentioned that the `KafkaTemplate.send` method returns a `ListenableFuture<SendResult<K, V>>` and you could extract the timestamp and offset directly at that point. But that as the drawback of needing to call the `ListenableFuture.get` method which will block your application’s main thread until the get method returns, meaning waiting until the produce request completes. You’d rather not take that approach as that would impact performance. So what you can do in this case is supply the returned `ListenableFuture` a callback that it will execute asynchronously when the produce request completes whether it’s a sucess or failure.

Listing 11.21. Creating a callback for getting the correct information after completed produce request

```
private final ListenableFutureCallback<SendResult<String, LoanApp
           produceCallback = new ListenableFutureCallback<>() {
  @Override
  public void onFailure(Throwable ex) {  #2
       LOGGER.error("Problem producing a record", ex);
  }

  @Override
  public void onSuccess(SendResult<String, LoanApplication> resul
      RecordMetadata metadata = result.getRecordMetadata();
       LOGGER.info("Produced a record to topic {} at offset{} at
               metadata.topic(), metadata.offset(), metadata.time
  }
};
```

Now that you have your callback created, it’s simply a matter of adding it to the resulting `CompletableFuture` after executing a `KafkaTemplate.send` action:

Listing 11.22. Adding a callback for notification of completed produce requests

```
ListenableFuture<SendResult<String, LoanApplication>> produceResu
    kafkaTemplate.send(topicToSend,
                       processedLoan.getCustomerId(),
                       processedLoan);

produceResult.addCallback(produceCallback); #2
```

Now you’ll get notification of compeleted produce requests even in the case of a failure. This is a very similar process to what you saw in the chapter on Kafka clients, but when working directly with a `KafkaProducer` you add the callback directly as a parameter to the `KafkaProducer.send` method.

This wraps up our coverage of building a Kafka application using Spring Kafka and now we’ll move on to using Spring Kafka with Kafka Streams applications.

11.2 Spring Kafka Streams

Just as we saw how Spring Kafka with Spring Boot can simplify building Kafka based applications, there is also support for building Kafka Streams applications. It’s a little different since Kafka Streams already abstracts away a lot of the details of working with Kafka. As with using any tool or framework there’s tradeoffs to consider. In this case the tradeoff I’m referring to is simplified application development vs. more control over the application buidling process.

When using Spring Boot with Kafka Streams, there are some advantages with having to deal with less infrastructure code, but there’s some loss of controll over how to build the application, vs. if you were to just use Spring for it’s dependency injection capability. We’ll take a look at both approaches so you’ll be able to make an informed decision on which approach is best for you. We’ll also cover one of the great features of using Spring Boot with Kafka Streams; Spring Boot applications will by default launch a web-server when you start them.

Having a web-server automatically with our Kafka Streams application is a real advantage when it comes to using the Interactive Queries. Interactive Queries (IQ from now on) give the ability to directly query the results of stateful operations in Kafka Streams. So within your Spring - Kafka Streams application you can include web based classes to handle incoming requests for serving up IQ queries. This has the potential for simplifying your application architecture, we’ll explain how that works a litte later on.

For now let’s get started with a simple Kafka Streams application, first using all the Spring Boot utilites and then just using Spring for wiring up the application.

Let’s say you’ve decided to re-do your Kafka loan application to use Kafka Streams. It will still use the same logic for loan approval, but instead of using listeners and the `KafkaTemplate`, it’s all handled in Kafka Streams. This will allow you do perform aggregations on loan approavals and rejections directly in one place.

Your first step towards using Spring Kafka with Kafka Streams is that you’ll add an additional annotation to the configuration class:

Listing 11.23. Adding an additional annotation to the configuration class

```
@SpringBootApplication(scanBasePackages = {"bbejeck.spring.datage
                       "bbejeck.spring.streams.boot"}) #1
@EnableKafka        #2
@EnableKafkaStreams  #3
@Configuration #4
public class KafkaStreamsLoanApplicationApplication {

  //Configuration items left out for clarity

}
```

Of the four annotations listed above, you’ve seen three of them already, but the `@EnableKafkaStreams` is new and it’s required to activate Spring Boot’s support for running Kafka Streams applications.

Tip

You’ll notice at annotation one we’ve specified the packages to scan for components to include in the container. You won’t need to do this if you have everything needed for the application in the same package as the annotated configuration class.

By applying the `@EnableKafkaStreams` annotation Spring will create a wrapper around the `KafkaStreams` instance and it will contol the lifecycle (starting and stopping) of the streams application. Before we go on to building the Kafka Streams application itself, there’s one more bit of configuration you’ll need to provide.

Listing 11.24. Configurations required for Kafka Streams with @EnableKafkaStreams

```
@Bean(name =
     KafkaStreamsDefaultConfiguration.DEFAULT_STREAMS_CONFIG_BEAN
KafkaStreamsConfiguration kafkaStreamsConfiguration() {
  Map<String, Object> streamsConfigMap = new HashMap<>(); #2
  streamsConfigMap.put(StreamsConfig.APPLICATION_ID_CONFIG,
                       "loan-processing-app");
  streamsConfigMap.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
                       bootstrapServers);
  return new KafkaStreamsConfiguration(streamsConfigMap); #3
}
```

When starting a Spring Boot application with the `@EnableKafkaStreams` annotation, it will expect to find a Spring Bean named `defaultKafkaStreamsConfig` providing the configurations used to create the `StreamBuilderFactoryBean` which creates the `StreamsBuilder` instance and will also controll the starting and stopping of the Kafka Streams application. If you don’t require any modifications on the `KafkaStreams` instance itself, then at this point you’ve done everything needed to run a streams application, all that is left is to create the application itself.

But before we move on to building the application, let’s quickly discuss what steps you can take when you ***do need access*** to the underlying `KafkaStreams` instance. For those times you require access Spring provides the `StreamsBuilderFactoryBeanCustomizer` interface, which is a functional or single abstract method interface. The functional interface is ideal to work with as you can use a Java lambda to represent it vs. a concrete object instance. So when would you need to access the `KafkaStreams` object? Consider the case where you’d like to have a `StateListener` to notify you of when `KafkaStreams` transitions to a running state so you can log the active tasks which contains topic-partition assignment information.

So to set the `StateListener` you’d create two new bean definitions in your configuration class. The first is the `KafkaStreamsCustomizer` which gives you access to `KafkaStreams` ***before*** it starts. The second bean is the `StreamsBuilderFactoryBeanCustomizer` which accepts the `KafkaStreamsCustomizer` for applying your desired changes.

Listing 11.25. Kafka Streams customizer for setting a StateListener

```
@Bean
KafkaStreamsCustomizer getKafkaStreamsCustomizer() {
 return  kafkaStreams ->
    kafkaStreams.setStateListener((newState, oldState) -> { #1
    if (newState == KafkaStreams.State.RUNNING) {
        LOG.info("Streams now in running state");
      kafkaStreams.metadataForLocalThreads()
        .forEach(tm -> LOG.info("{} active task info: {}",
                         tm.threadName(), tm.activeTasks())); #2
    }
    });
}
```

So with the the `KafkaStreamsCustomizer` we can access the `KafkaStreams` instance in this case to set the `StateListener`. Now to get this bean into the `StreamsBuilderFactoryBean` you create the second bean definition like this:

Listing 11.26. Creating the StreamsBuilderFactoryBeanCustomizer to apply KafkaStreams object settings

```
@Bean
StreamsBuilderFactoryBeanCustomizer kafkaStreamsCustomizer() {#1
  return  streamsFactoryBean ->
 streamsFactoryBean
     .setKafkaStreamsCustomizer(getKafkaStreamsCustomizer()); #2
}
```

With these two bean definitions added to the configuration class, you now have access to the `KafkaStreams` object.

Now that we’ve covered how to access the `KafkaStreams` object in the Spring Boot application, let’s move on to building the application itself.

When you build the Kafka Streams application with the `@EnableKafkaStreams` annotation it’s different from what we’ve seen before. To show the differences, let’s dive right into an example. Let’s say you’ve taken your loan application and converted it from using Kafka producer and consumer clients to a Kafka Streams application:

Listing 11.27. Converted Kafka producer and consumer app to Kafka Streams

```
@Component #1
public class LoanApplicationProcessor {

  @Value("${loan.app.input.topic}")
  private String loanAppInputTopic;


@Autowired
public void loanProcessingTopology(StreamsBuilder builder) { #2

KStream<String, LoanApplication> processedLoanStream = #3
   builder.stream(loanAppInputTopic,
                Consumed.with(stringSerde,
                        loanApplicationSerde))
                .mapValues(loanApp -> {
double monthlyIncome = loanApp.getReportedIncome() / 12;
double monthlyDebt = loanApp.getReportedDebt() / 12;
double monthlyLoanPayment =
        loanApp.getAmountRequested() / (loanApp.getTerm() * 12);
double debtRatio =
        (monthlyDebt + monthlyLoanPayment) / monthlyIncome;

boolean loanApproved =
       debtRatio <= 0.33 && loanApp.getCreditRating() > 650;

return LoanApplication.Builder.newBuilder(loanApp)
                              .withApproved(loanApproved)
                              .build();
  });
}
```

From looking at this code listing, it’s standard Spring configuration; declaring a class as a `@Component` and inject the desired objects with the `@Autowired` annotation, this time on a method not a constructor. I mentioned previously there are some differences and if you look closely at annotation two and notice the `void` return type for the `loanProcessingTopology` method. The method doesn’t return the `StreamsBuilder` instance since it’s managed by the Spring container.

When starting the container to build a streams application Spring will pass around a singleton `StreamsBuilder` to all methods that reference it an have the `@Autowired` annotation on it. Then when `StreamsBuilderFactoryBean` begins the process of starting the `KafkaStreams` instance it will execute the `StreamsBuilder.build` method.

The impact of having Spring control the `StreamsBuilder` instance in this way means you can potentially have the classes making up your topology spread out among different classes. While it’s possible to take this approach, I’d still reccomend having the entire topology in a single class, as when it comes time to debug any potential issues it will be much easier to track down what’s wrong by viewing the entire topology in one place.

You’ve just seen how to build a Kafka Streams application with Spring Boot and the `@EnableKafkaStreams` annotation. Taking this approach does make things easier as Spring takes care of most of the configuration details for you. But everything is a trade-off - and here what you gain in ease of starting up and managing Kafka Streams you also lose some visibility into what’s going on with the application.

But there’s another trade-off you can make; giving up some of the convenience in exchange for more control and visibility into how Kafka Streams is put together and we’re going to cover that next. You’re still going to use Spring to wire the application together, but instead of the Spring container managing the lifecycle of the `KafkaStreams` object and how the topology gets put together, you’ll take on that responsibility.

We’re going to take the same Kafka Streams application and make some small changes, starting with the class that builds the topology.

Listing 11.28. First change is to rename the class to reflect its role

```
@Component
public class LoanApplicationTopology { #1
 // Details left out  for clarity
}
```

The first change is to rename the `LoanApplicationProcessor` to `LoanApplicationTopology`. By changing the name you're declaring the intent that the class will contain entire topology for the Kafka Streams application, not just a single processor or section of it. Next you'll update the signature of the `loanProcessingTopology` method which itself will require an additional change:

Listing 11.29. Updating the signature of the loanProcessingTopology method and explicity creating a StreamsBuilder

```
private final KafkaStreamsConfiguration streamsConfigs;

@Autowired
public LoanApplicationTopology(  #1
     KafkaStreamsConfiguration streamsConfigs) {
    this.streamsConfigs = streamsConfigs;
}

public Topology topology() { #2
  StreamsBuilder builder = new StreamsBuilder(); #3

// Build topology

return builder.build(streamsConfigs.asProperties()); #4
```

So the first change to support the method change is to update the constrcutor to autowire the `KafkaStreamsConfiguration`, you'll see next how this is related to refactoring the method. The next change at annotation two, you've changed the name of the method to `topology` to reflect its role and you've removed the `StreamsBuilder` parameter, creating it directly instead at annotation three. You've also updated the return type from `void` to `Topology` reflecting the change at last line of the method where you execute `StreamBuilder.build` returning the `Topology` instance.

The next change is actually an addition - creating a class to support creating the `KafkaStreams` instance and controlling the lifecycle of the streams application.

Listing 11.30. Creating a class to support Kafka Streams

```
@Component
public class KafkaStreamsContainer {

//Details left out for clarity

@Autowired
public KafkaStreamsContainer(
       final LoanApplicationTopology loanApplicationTopology, #1
       final KafkaStreamsConfiguration appConfiguration) {  #2
    this.loanApplicationStream = loanApplicationTopology;
    this.appConfiguration = appConfiguration;
}

//Details left out for clarity
```

Here you've created the `KafakStreamsContainer` class that will handle the tasks required for building and running the Kafka Streams application. Notice the constructor has the `@Autowired` annotation and two parameters the `LoanApplicationTopology` and the `KafkaStreamsConfiguration` class that Spring will inject for us. Next let's look at how you'll use these two objects to get Kafka Streams up and running.

Listing 11.31. Creating and running Kafka Streams

```
@PostConstruct #1
public void init() {
    Properties properties = appConfiguration.asProperties();
    Topology topology = loanApplicationStream.topology(); #2
    kafkaStreams = new KafkaStreams(topology, properties); #3

    kafkaStreams.setStateListener((newState, oldState) -> {
      if (newState == KafkaStreams.State.RUNNING) {
        LOG.info("Streams now in running state");
        kafkaStreams.metadataForLocalThreads().forEach(tm ->
          LOG.info("{} assignments {}", tm.threadName(), tm.activ
      }
        });
    kafkaStreams.start(); #4
}
```

Next you add the `init` method that creates the `KafkaStreams` instance and starts it running. But the question is when to call the `init` menthod? This part is handled by Spring for you via the `PostConstruct` you've decorated the method with at annotation one. When you add a `PostConstruct` to a container compoent class, the Spring container will execute the method when the object is ***fully constructed*** , so in this case after the required dependencies are injected, Spring will call the `init` method automaticall creating the `KafkaStreams` instance and starting it up. But we did say you would handle the full life-cycle so what about stopping a `KafkaStreams` application? For that you'll add one more method:

Listing 11.32. Stopping the Kafka Streams application

```
@PreDestroy  #1
public void tearDown(){
    kafkaStreams.close(Duration.ofSeconds(10)); #2
}
```

To stop the application you'll use a similar approach, placing the `PreDestroy` annotation on a method that the Spring container will call before it tears down the managed component when the Spring container is stopping.

Now the changes are complete for giving you a bit more visibility into the construction of the Kafka Streams application and how it's started and shutdown. There isn't a right or wrong decision here with the approach you can take, it comes down to personal preference and the different requirements you are faced with.

As we wrap up this chapter on Spring Kafka/Kafka Streams lets talk about the great support Spring Boot provides for a powerful feature of Kafka Streams - Interactive Querries.

11.3 Learning about Interactive Queries

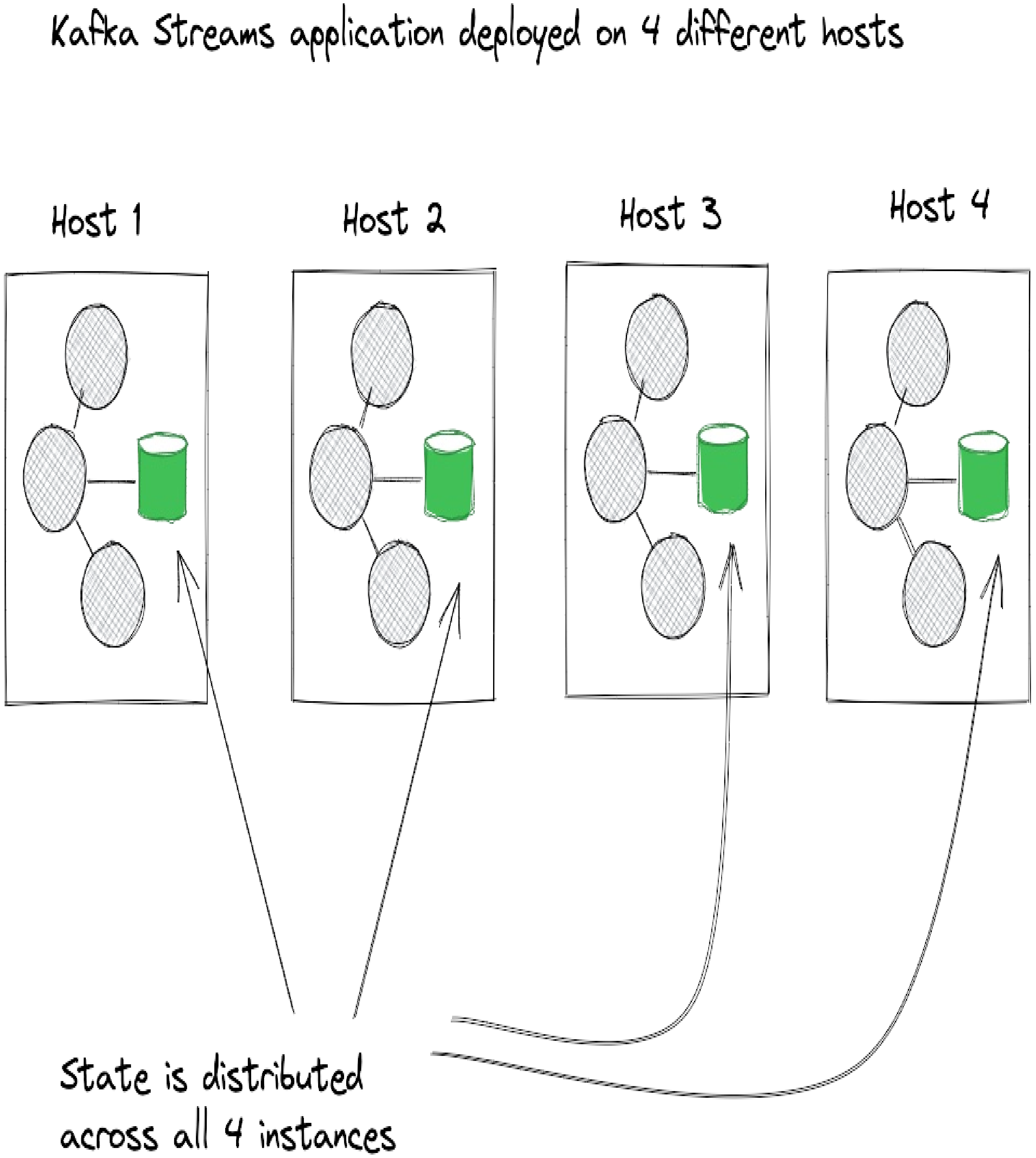
Before we get into how to use Spring for Interactive Queries (IQ), let's have a dicussion on what IQ is. When you build a stateful Kafka Streams application, which is one with at least one aggregation, reduce etc. if you provide a name to the underlying state store, you can query against that store to view the contents, essentially you can view the status of the stateful operation in real-time.

To enable IQ, you only need to provide one configuration, `application.server` which contains a host and port that you’ll use to connect to a Kafka Streams application instance to run a query. If you have more than one Kafka Streams instance (all with the same application id), then each instance ***must*** provide a unique host:port combination.

To refresh your memory, Kafka Streams ends up with a state store per task, and since each task represents a single partition, you end up with a store per partition. What make this important to our discussion here is that when you deploy multiple Kafka Streams application instances, each instance is only responsible for a subset of the total number of tasks (partitions), as Kafka Streams spreads the processing load across other the other applications. For example, if the source topic has six partitions, and you’ve started three application instances, each one will be responsible for two partitions each.

Since the state stores for Kafka Streams are local for each instnace, this means the impact of spreading the processing load is that your state is distributed across muliple machines. Here’s an illustration that should help make this concept clear to you:

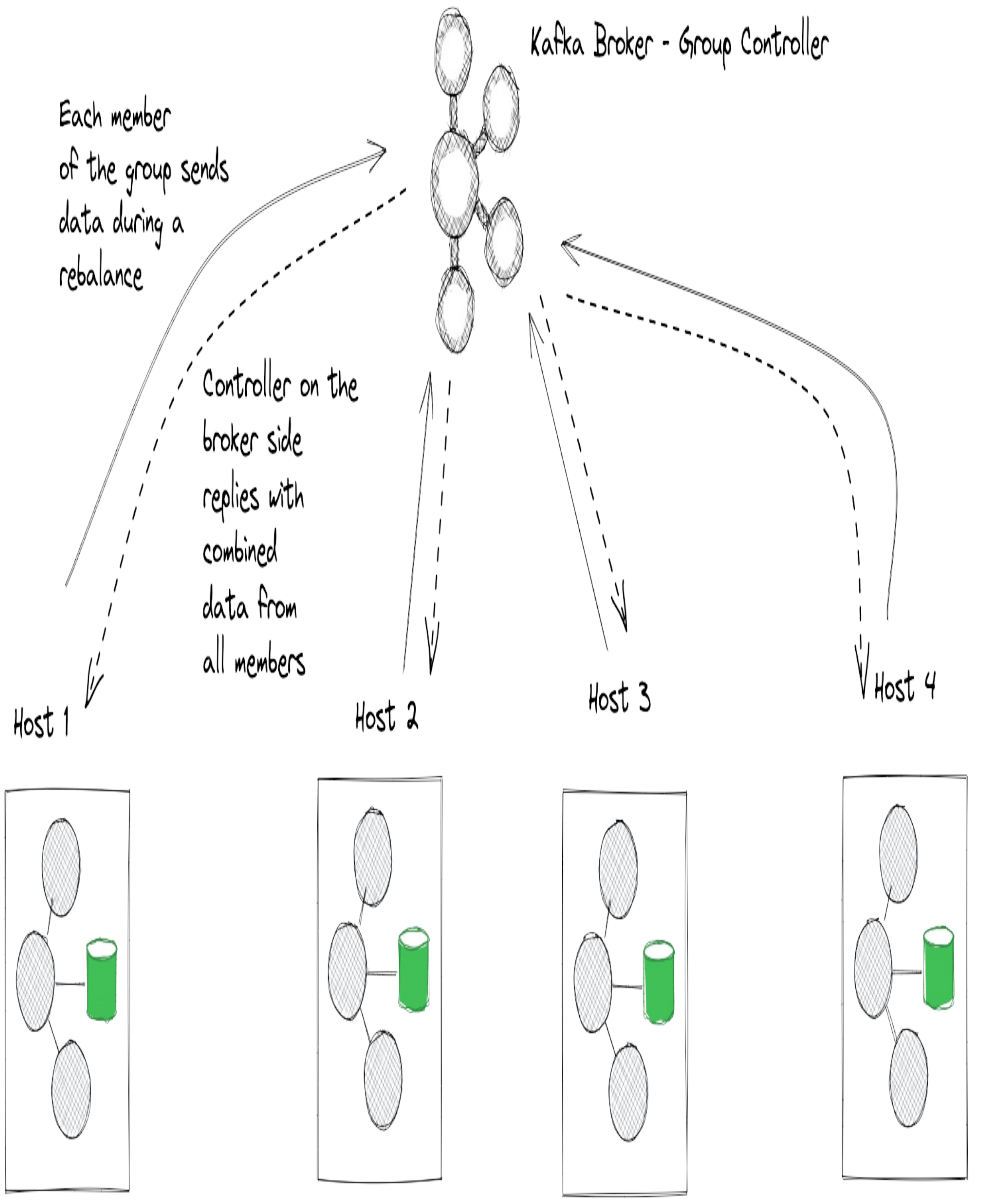
Figure 11.2. State in Kafka Streams ends up distributed across multiple machines


Kafka Streams application deployed on 4 different hosts
Host 1
Host 2
Host 3
Host 4
State is distributed across all 4 instances

So given that your state is distributed and the state stores in Kafka Streams are key-value pairs how does one know which instance to query? The good news Kafka Streams provides the infrastructure so you don’t have to know which instance is responsible for the partition where a given key is - you pick one instance to query and if its state doesn’t have the key in question, it knows which one does and it will forward the query on your behalf and return the results to you.

This is possible because the rebalance protocol allows for encoding arbitrary data in the payload. So when Kafka Streams application instances rebalance, in addition to providing all the partitions each one is responsible for, they also include the individual `application.server` configurations. Let’s look at a graphic illustrating this process:

Figure 11.3. Rebalance distributes metadata to all instances with the same application.id

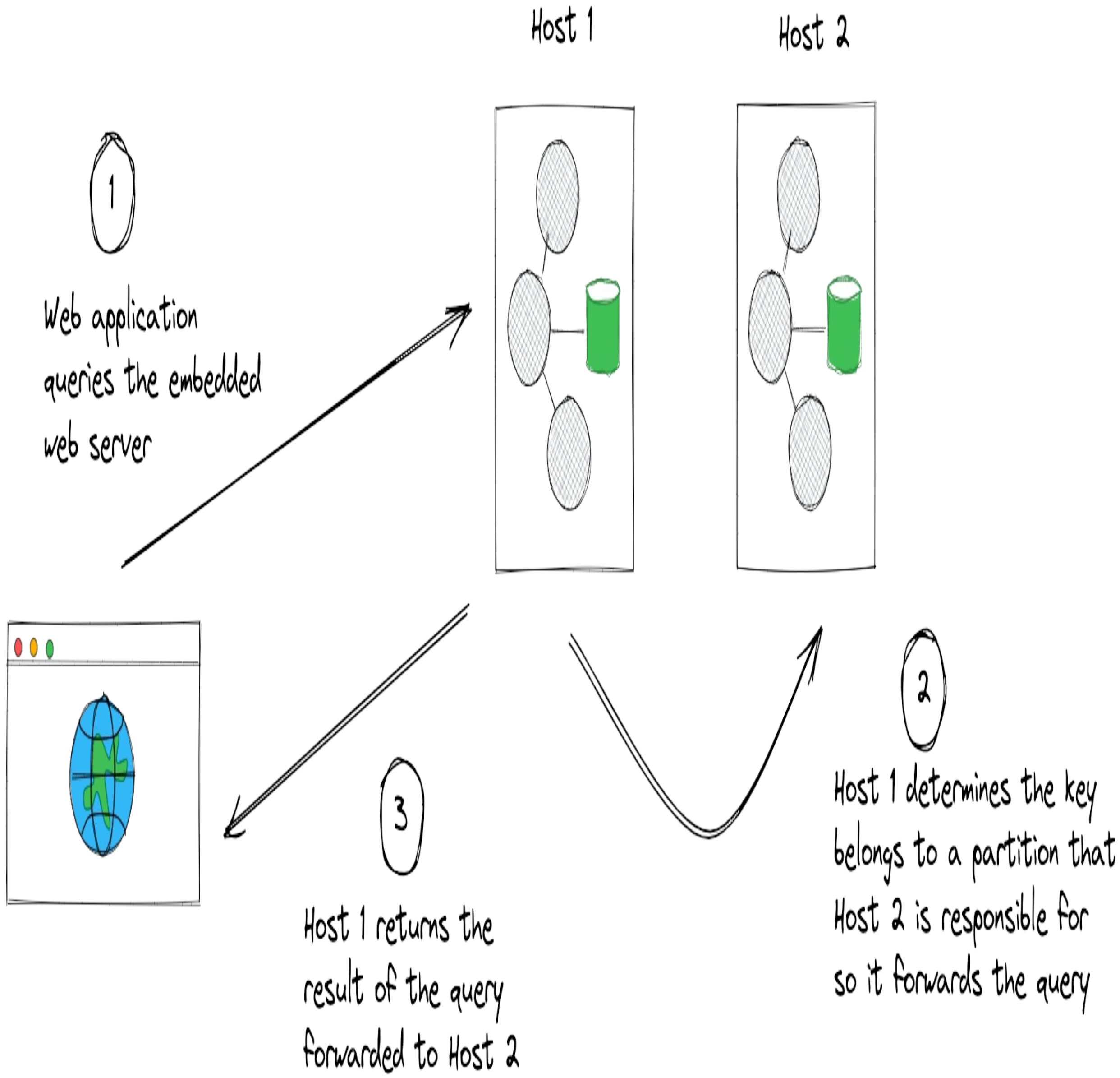

Kafka Broker - Group Controller
Each member of the group sends data during a rebalance
Controller on the broker side replies with combined data from all members
Host 1
Host 2
Host 3
Host 4

So each application ends up with metadata knowing all the partitions each other instnace (again with the same application id) is responsible for.

Additionally the application server information is known for each one as well, so when you query a Kafka Streams application with a given key, it determines the partition the key would fall into by taking the hash of the key modulo the number of partitions (assuming hash partitioning). If the current Kafka Steams app doesn’t have that partition, it knows exactly which one does and it will forward the request since it knows the host and port for the queries on that machine as well.

This is a somewhat complex process, so here’s an illustrationg showing this process in action:

Figure 11.4. Determining the correct host for a given query


Host 1
Host 2
1
Web application queries the embedded web server
2
Host 1 determines the key belongs to a partition that Host 2 is responsible for so it forwards the query
3
Host 1 returns the result of the query forwarded to Host 2

While Kafka Streams provides the internal plumbing for application instances with the same application id to share metadata, the communication layer between them all is not provided, you need to implement that part yourself and that’s what you’re going to do in the next section.

11.4 Building an Ineractive Queries app with Spring Boot

We're going to take a step by step process to building the Interactive Query layer in your Kafka Streams application. The good news is that you've already laid the groundwork with the last example application, `bbejeck.spring.streams.container.KafkaStreamsLoanApplicationAppli` Here are the steps you're going to take next:

1. Enable the Spring Boot application to have an embedded web-server when starting up
2. Add a `RestControler` to handle incoming query requests either directly from users or sibling applications
3. Inside the controller class add logic to process queries
4. Build an HTML index page to make REST API calls and continually displaying loan application results in single page

Along the way with building this you'll also learn about the latest version of Interactive Queries (IQv2) which is a big improvement over the first version.

So let's tackle each item on our list in order, starting with enabling a web-server on start-up. With our previous Spring boot application in the main method we explicity disabled the web-server like this:

Listing 11.33. Disabling the web-server

```
SpringApplicationBuilder applicationBuilder =
new SpringApplicationBuilder(LoanApplicationProcessingApplication
                .web(WebApplicationType.NONE); #1
```

You turned off the web-server as you had no need of it, this application ran with no need to accept external requests. But now you want to expose your app to receive requests over HTTP to query the state of your loan processing. To do this you change the `WebApplicationType` enum parameter to the `SpringApplicationBuilder` :

Listing 11.34. Enabling a web-server for a Spring Boot application

```
SpringApplicationBuilder applicationBuilder =
new SpringApplicationBuilder(KafkaStreamsLoanApplicationApplicati
                .web(WebApplicationType.SERVLET); #1
```

So the only change you need to make is switch the type from `NONE` to `SERVLET` and this will enable the web-server when starting your application.

Note

Servlets are a Java technology that you can deploy on web-servers for building web-based applications. We won’t go into details on it here but you can find more information by going to https://www.oracle.com/java/technologies/servlet-technology.html

Now the nest step is for you to create a class that will handle the incoming query requests.

Listing 11.35. Creating the controller for responding to HTTP requests

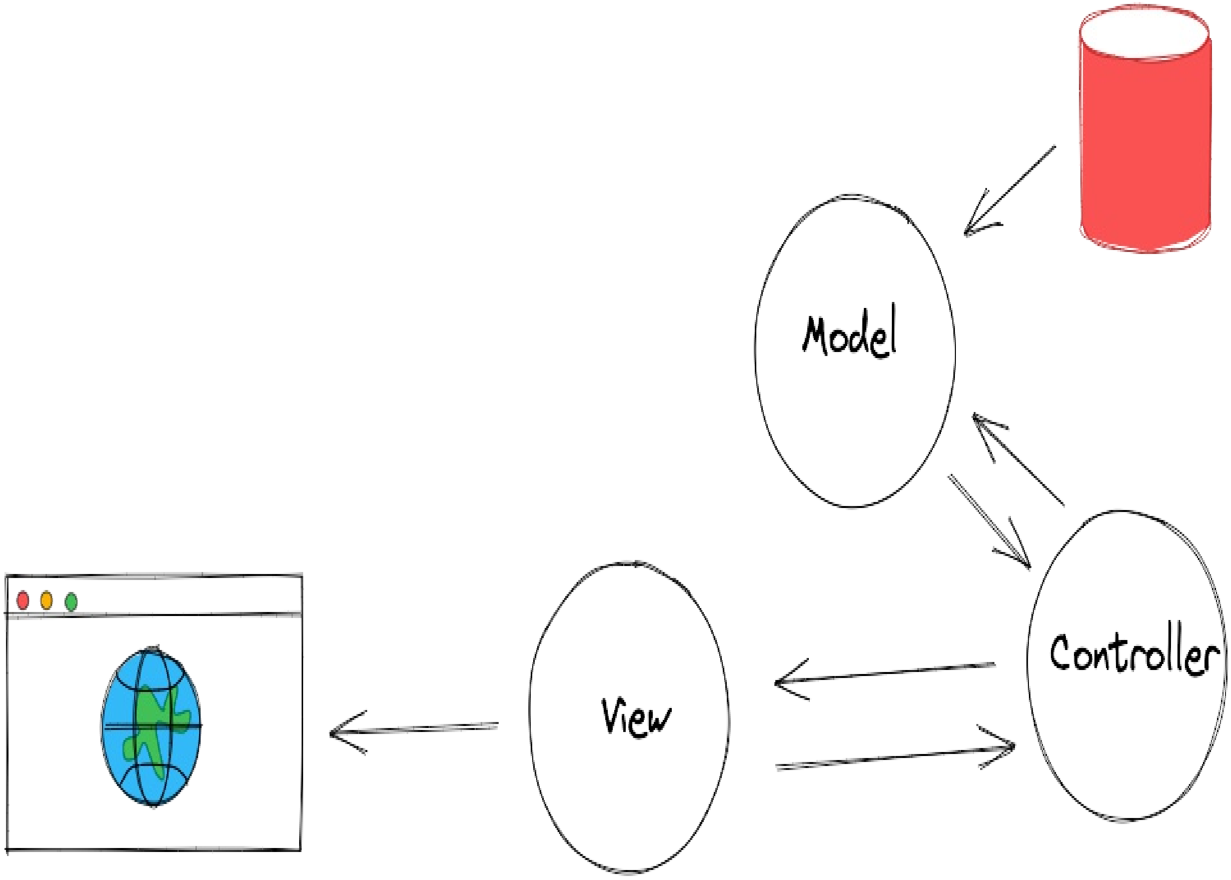
```
@RestController  #1
@RequestMapping("/loan-app-iq") #2
public class LoanApplicationController {

  //Details left out for clarity

}
```

The first step is to create the class and add two new annotations at the class declaration level. The `@RestController` annotation does to main things for us - marks this class as a web-controller and also enables the automatic conversion to JSON of the object you return in repsonse to a request. The `@RequestMapping` annotation maps incoming HTTP requests to request-handling classes and methods. The annotation at the class level specifies that this controller handles all requests with the specified base URL.

Before we go any futher, let’s take a minute to define what the term controller means. A web controller is part of a MVC (Model-View-Controller) design patterns for web-applications. What your going to build in this section is a Spring MVC application. The model represents the data, the view is responsible for presenting visual component, and the controller is repsonsible for accepting and procesing incoming requests and providing a response to them displayed via the view component.

Here’s an illustartion depicting how the MVC components fit together in a web application:

Figure 11.5. MVC components in a web application



Now with the brief description of the MVC pattern for web applications, lets get back to building out the controller. So far we’ve covered declaring the class and providing the required annotations for to controller to recieve HTTP requests, now let’s look further into how it will work:

Listing 11.36. Dependencies required by the controller

```
@RestController
@RequestMapping("/loan-app-iq")
public class LoanApplicationController {

  @Value("${store.name}")  #1
  private String storeName;
  @Value("${application.server}")
  private String applicationServer; #2

  private final KafkaStreams kafkaStreams; #3
  private final RestTemplate restTemplate; #4

  @Autowired
  public LoanApplicationController(KafkaStreams kafkaStreams,
                                   RestTemplate restTemplate) {
      this.kafkaStreams = kafkaStreams;
      this.restTemplate = restTemplate;
   }
}
```

Here you see the injected configuration fields that you are accustomed to seeing by now and the `@Autowired` annotation on the contstructor injecting the `KafkaStreams` instance built in the `KafkaStreamsContainer` and the `RestTemplate` from Spring which you'll use to communciate the other Kafka Steams applications when handling requests with keys the current application instance is not responsible for.

To inject the `KafkaStreams` instance we will need to make a quick modification to the `KafkaStreamsContainer` class:

Listing 11.37. Exposing a KafkaStreams object as a Spring bean in the KafkaStreamsContainer class

```
@Bean  #1
public KafkaStreams kafkaStreams() {
    return kafkaStreams;
}
```

By adding this method to the `KafkaStreamsContainer` class with the `@Bean` annotation, when any other class in the container has a reference to a `KafkaStreams` object as dependency, the Spring container executes this `kafkaStreams` method to inject into the class. In our case here when the Spring container executes the `LoanApplicationController` contstructor, it follows this exact process. You'll see how the controller uses the `KafakStreams` object next when we look at a request handling method.

Listing 11.38. Method annotated to handle incoming requests for a given loan category

```
 @GetMapping(value = "/loantype/{category}") #1
 public QueryResponse<LoanAppRollup> getCategoryRollup(@PathVaria
                            String category) {  #2
    KeyQueryMetadata keyMetadata =   #3
            getKeyMetadata(symbol,
                Serdes.String().serializer());
  if (keyMetadata == null) {
        return QueryResponse.withError(String.format(
    "ERROR: Unable to get key metadata after %d retries", MAX_RET
  }

}
```

The `@GetMapping` annotation on the method means web requests with this URL: `http://loan-app-iq/loantype/<category>` are routed to the `LoanApplicationController.getCategoryRollup` method. The `KeyQueryMetadata` is an important object as it will contain the information required to determine if the current instance queried is responsible for the partition of the key involved in the query. The `getKeyMetadata` is simply an internal method and it’s worth us takeing a quick look at what’s going there:

Listing 11.39. Implementation of the getKeyMetadata method retrieving KeyQueryMetadata

```
private <K> KeyQueryMetadata getKeyMetadata(K key,
                     Serializer<K> keySerializer) {
  int currentRetries = 0;
  KeyQueryMetadata keyMetadata =
      kafkaStreams.queryMetadataForKey(storeName,
                                       key,
                                       keySerializer); #1
  // Details left out for clarity

  return keyMetadata;
```

In this code block you can now see one of the reasons we need a reference to the `KafkaStreams` object, the ability to extract essential metadata for the key involved in the query. One of the key fields of the `KeyQueryMetadata` is the `HostInfo` field, which contains the host and port of the server instance with the Kafka Streams instance containing the data in question. So to enable this comparison of host information, when building the controller you’ll use the `application.server` configuration to build a `HostInfo` object for comparision:

Listing 11.40. Building a HostInfo object from the application server configuration

```
@PostConstruct
public void init() {
    String[] parts = applicationServer.split(":");
    thisHostInfo = new HostInfo(parts[0],
                    Integer.parseInt(parts[1])); #1
}
```

Here in the `init` method (executed by the Spring container after the controller is fully constructed), you build a `HostInfo` object. You'll use this to compare it to the one returned in the metadata as shown here:

Listing 11.41. Comparing the HostInfo from key metadata with the HostInfo for the current host

```
// Details left out for clarity

 if (targetHostInfo.equals(thisHostInfo)) { #1

  Set<Integer> partitionSet =
        Collections.singleton(keyMetadata.partition());

  StateQueryResult<ValueAndTimestamp<LoanAppRollup>> keyQueryResu
      kafkaStreams.query(StateQueryRequest.inStore(storeName) #2
          .withQuery(query)
          .withPartitions(partitionSet));

  QueryResult<ValueAndTimestamp<LoanAppRollup>> queryResult =
         keyQueryResult.getOnlyPartitionResult(); #3


} else {     #4
  String path = "/loantype/" + type;
  String host = targetHostInfo.host();
  int port = targetHostInfo.port();
  queryResponse = doRemoteRequest(host, port, path); #5
}
```

To determine if you've queried the correct host, you compare the `HostInfo` object retrieved from the metadata with the one created by the controller. If they are equal, then you run the query with the `KafkaStreams.query` method and extract the results and return the results. Otherwise you extract the host and port information and execute the search remotely on the correct host with a REST API call. There's several details in this code snipet that I'm skipping over, so let’s take a minute to go over those details now.

Note

With these code examples there are a few internal objects created for the support of running examples and are not part of Spring or Kafka Streams. I won’t describe those as they don’t add anything to what you’re learning, but you can view them by looking at the source code for the book.

The key component of Interactive Queries is the `Query<R>` interface that all queries must implement. There are currently 4 implementations:

1. `KeyQuery`
2. `RangeQuery`
3. `WindowKeyQuery`
4. `WindowRangeQuery`

You’ll use a `KeyQuery` when your looking for results of an individual key. The `RangeQuery` is useful when you have a need to see results from a range of keys. You’d use a `WindowKeyQuery` with windowed statestores as it allows you to specify a time from and to for the query, and the `WindowRangeQuery` is also for windowed stores and it allows you to specify a time range for querying. Other than the specific parameters you provide to the individual `Query` object, you’ll end up using them in the same manner when executing the query.

You create a `KeyQuery` by using the static factory method provided by the class:

Listing 11.42. Creating a KeyQuery

```
KeyQuery<String, ValueAndTimestamp<LoanAppRollup>>
        keyQuery = KeyQuery.withKey(loanType);
```

So with the `KeyQuery.withKey` method you provide the key and the method returns a `KeyQuery` object. The types on the `KeyQuery` are the expected query results; key and the value which is wrapped in a `ValueAndTimestamp` abstraction, which provides the timestamp of the record stored in the statestore as well.

Then next step is to create a `StateQueryRequest` object that you'll pass to the `KafkaStreams.query` method. It uses the builder pattern as you'll see here:

Listing 11.43. Building the StateQueryRequest

```
StateQueryRequest.inStore(storeName) #1
          .withQuery(query) #2
          .withPartitions(partitionSet)) #3
```

The `StateQueryRequest` object contains the essential information like the name of the statestore, the query object, and the partition for the key (you got this earlier from the `KeyMetadata` object). Other than the required parameters of the store name and the query itself, the other parameters of the `StateQueryRequest` are optional and include the partitions, requiring active tasks, and the amount of acceptable lag when querying standby tasks. We'll cover queries with standby tasks a little later in this chapter.

After executing the `KafkaStreams.query` method you're set capture the resulting `StateQueryResult` in a variable:

Listing 11.44. Setting the query result in a variable

```
StateQueryResult<ValueAndTimestamp<LoanAppRollup>> stateQueryResu
         kafkaStreams.query(StateQueryRequest.inStore(storeName)
                            .withQuery(query)
                            .withPartitions(partitionSet));
```

Once you have the `StateQueryResult` you can extract the underlying result from the state store. The overall query result(s) are stored in a `Map` with the partition as the key and a `QueryResult` object as the value. In this case we know the result only relates to single partition since it's a key query so we can use the convenience method `StateQueryResult.getOnlyPartitionResult`:

Listing 11.45. Extracting the query result

```
 QueryResult<ValueAndTimestamp<LoanAppRollup>> queryResult =
     stateQueryResult.getOnlyPartitionResult(); #1

 ValueAndTimestamp valueAndTimestamp = queryResult.getResult() #2
 valueAndTimestamp.getTimestamp(); #3
 valueAndTimestamp.getValue(); #4
```

Now that you've extracted the result you're ready to return it for rendering in the web application:

Listing 11.46. Returning the result to the web view

```
// Details left out for clarity

queryResponse =
    QueryResponse.withResult(queryResult.getResult().value()); #1
// possibly add some metadata from query
return queryResponse; #2
```

After getting the raw result you store it a custom object capable of storing some metadata for display in the view as well. If you recall from a little earlier in this section, everything is automatically converted to JSON for rendinging in the web application. For the multi-partition results you'll extract the `Map` and iterate over the contents:

Listing 11.47. Extracting multiple partition results

```
//Details left out for clarity
Map<Integer, QueryResult<KeyValueIterator<String,
  ValueAndTimestamp<LoanAppRollup>>>> allPartitionsResult =
       result.getPartitionResults(); #1

        allPartitionsResult.forEach((key, queryResult) -> { #2
            // Do somethign with the results
        });
```

There are a couple of important notes here to keep in mind. First is that when retrieving the results of a query if you use the `getOnlyPartition` method, but there are actually mutiple partition results, the `QueryResult` object will throw an exception. But the rule is pretty clear in this case, with a query involving a single key i.e `KeyQuery` you can safely use the single partition extraction approach and for all others you'll want to iterate over the result map.

So far you’ve seen running the query on the current host, but how would you execute it on a remote one when the current does not have the key? A bit earlier in this section we saw an internal method `doRemoteQuery` used by the controller for that case. Here is the implementation of that method:

Listing 11.48. Executing a query on the remote host

```
//Details left out for clarity
 private <V> QueryResponse<V> doRemoteRequest(String host,
                                              int port,
                                              String path) {
  QueryResponse<V> remoteResponse;
  try {
      remoteResponse = restTemplate.getForObject(BASE_IQ_URL + pa
                                             QueryResponse.class,
                                             host,
                                             port); #1

  } catch (RestClientException exception) {  #2
      remoteResponse = QueryResponse.withError(exception.getMessa
  }
  return remoteResponse;
```

For the case where another Kafka Streams application instance is responsible for the key, you’ll use the `RestTemplate` to issue a REST API call to the remote host, where it will follow the exact execution path and return the result to the current host which will return it to view for rendering.

Before we wrap up this chapter, we have a couple of additional items to discuss. First is the different approach you’ll take with a range query. Unlike the key query where you can specifically query a single host, the range query needs to execute against all Kafka Streams instances, due to the distributed parition assignment. To execute the query across all instances, you’ll want to know which ones contain the state store in question and its partition assignment:

Listing 11.49. Setting up to query all instances for a range query

```
Collection<StreamsMetadata> streamsMetadata =
                  kafkaStreams.streamsMetadataForStore(storeName)
List<LoanAppRollup> aggregations = new ArrayList<>();
```

```
streamsMetadata.forEach(streamsClient -> {  #2
 Set<Integer> partitions =
            getPartitions(streamsClient.topicPartitions());
  QueryResponse<List<LoanAppRollup>> queryResponse =
                    doRangeQuery(streamsClient.hostInfo(), #3
            Optional.of(partitions),
            Optional.empty(),
            lower,
            upper);
```

Since you need to execute a range query across all hosts, you’ll need to use the `KafkaStreams.streamsMetadataForStore` which returns metadata from each Kafka Streams application that contains the store in its topology. You can then iterate over the metadata for each one and execute the query for the current instance and remotely for the rest.

The second item to discuss is the ability to query a standby task. When a Kafka Streams instance goes offline, there will be a rebalallance to reassign its partitions to the remaining active members. But this can take some time to complete, so Kafka Streams allows to make the trade-off of availability over consistency by allowing for querying a standby task. The trade-off is the standby will not be fully caught up to the active before it went down but you can still serve query results.

To query a standby, you’ll look for any error when executing a query on the active host by catching a `RestClientException`. Then in your query processing if you find an error, you’ll issue the query to a standby as shown here:

Listing 11.50. Issuing a standby query in the event of an error querying the active host

```
if (queryResponse.hasError() && !standbyHosts.isEmpty()) {  #1
  Optional<QueryResponse<StockTransactionAggregation>> standbyRes
           standbyHosts.stream()     #2
          .map(standbyHost -> doKeyQuery(standbyHost,
                                         keyQuery,
                                         keyMetadata,
                                         symbol,
                                         HostStatus.STANDBY))
          .filter(resp -> resp != null && !resp.hasError())
          .findFirst();  #3
  if (standbyResponse.isPresent()) {
        queryResponse = standbyResponse.get();
    }
}

 return queryResponse; #4
```

So when detecting an error and standby tasks are enabled, you can elect to query the standby task when you receive an error querying the active task. It goes without saying that you must have standby tasks enabled via configuration to use this feature.

11.5 Summary

- Spring Kafka provides abstractions making it easier to work with Kafka. You can use the standard approach of using standard Spring but you'll have to add the required configuration for the `KafkaTemplate` and `KafkaListener` supporting classes. By contrast by using Spring Boot, the level of configuration you're required to provide is greatly reduced by simply adding `@SpringBoot` and `@EnableKafka` annotations at the class declaration level. Typically you'd want to place this on a configuration class denoted with a `@Configuration` annotation as well.
- When setting the concurrency level for a `KafkaListener` the Spring Container will create a corresponding number of threads running a `KafkaConsumer` for the listener. You must take care to ensure the method decorated with the `@KafkaListener` annotation is thread-safe when setting the concurrency level greater than one.
- Spring Boot also provides an `@EnableKafkaStreams` annotation that takes care of the life-cycle of working with Kafka Streams. But this is not required and if you prefer to have more control over your stream application you can use Spring just for the dependency injection capabilities only.
- Spring Boot by default starts a web server automatically with the application, but you can opt to turn this off by using the `WebApplicationType.NONE`. This embedded web server is great for Kafka Streams applications as you'll automatically have the serving layer for Interactive Queries available.
- By using the `@RestController` annotation on a controller class methods handling web requests will automatically have the return objects

converted to JSON for the web response.

12 Testing

This chapter covers

- Learning about the difference between unit and integration testing
- Testing Kafka Producers and Consumers
- Creating tests for Kafka Streams operators
- Writing tests for a Kafka Streams topology
- Developing effective integration tests

So far we’ve covered the component for building an event streaming application: Kafka Producers and Consumers, Kafka Connect and Kafka Streams. But there' an additional crucial part of this development I’ve left out until now: how to test your application. One of the critical concepts we’ll focus on is placing your business logic in standalone classes that are entirely independent of you event streaming application, because that makes your logic much more accessible to test. I expect you’re aware of the importance of testing but I’d like cover what I consider the top two reasons for why testing is just as necessary as the development process itself.

First as you develop your code, you’re creating an implicit contract of what you and others can expect about how the code performs. The only way to prove that the application works is by testing thoroughly, so you’ll use testing to provide a good breadth of possible inputs and scenarios to make sure everything work appropriately under reasonable circumstances. The second reason you need comprehensive testing is helps you deal with the inevitable changes that occur with software. Have a rigorous set of tests gives you immediate feedback when the new code breaks the expected set of behaviors.

Additionally, when you do a major refactor of the application, having your tests pass gives you level of confidence about releasing the updated software. Even once you understand the importance of testing, writing tests for distributed applications like Kafka Streams isn’t always straighforward. You still have the option of running these applications with simple inputs and observing the results, but there’s a serious drawback to using this type of approach. You wan't a suite of ***repeatable*** tests that you can run at any time and that's also part of your CI build. Another component of testing is you want them to run as fast as possible so that means for a large segment of your tests you'll want them to run without a Kafka broker. Testing without a Kafka broker is one of the most important points we'll cover in this chapter. But there will be times when you need a live broker to effictively test the application. This tension between using a broker or not in testing is the boundary I like to call between unit and integration testing.

12.1 Learning the difference between unit and integration testing

In this section, I'm going to provide an opinionated definition here of unit and integeration testing. I'll define unit testing as a type of test that exercises a specific sub-component of an application, a specific point of the logic. For example, let's say you have an international sales application and when you receive an order you immediately convert the transaction amount into US dollars. A unit test would validate only the currency coversion part, with separate tests or a parametarized test for each type of expected currency.

These types of tests are usually at the method level of a class and run very fast. The issue with unit testing is that often there's external dependencies, a Kafka broker for example, that's required to run the application. So going with our immediate example here, does that mean you need to run a Kafka broker to feed the different currency types? The answer to that is no and I'll answer the question as to why you wouldn't by way of an analogy.

Let's say you're in a play at your local theatre and you need to learn your part. You don't need any of the other actors on stage with you to do this, just someone, a stand-in, to feed you the required lines for you to speak your parts. Additionally, the person helping can tell you if you got everything correct or not.

The same is true with a unit test, you don't need the whole system to test a smaller part, just a mechanism to provide the input and the ability to validate the results. This mechanism is called a mock object. A mock is an object that has the same interface as the remote connecting component, but no real behavior. You explicity tell the mock object what it should do, such as supplying specic values then you validate the results.

Going back to our analogy, there will definitely be times where need to get everyone together to rehearse to make sure all the actors not only know their lines correctly, but interact with each other as expected. In code this type of test is an integration test, not the live production but there’s no mocking, all external components are the real thing.

Integration tests are essential as well, but you won’t have the same number of them as you will have unit tests. One of the big reasons you need more unit tests vs. integration tests is speed of execution. A typical unit test will run in a quarter to a half a second but an integration test can run for up to several seconds as high as 30 seconds to a couple of minutes. Just those numbers for a single test isn’t bad, but once you get test numbering in the hundreds or thousands you can see the need to have your test suite run a quickly as possible.

So what’s a good indicator for a writing a unit test vs an integration test? As I stated earlier, you’ll want to use unit tests to validate the behavior of individual methods. An indicator of needed an integration test is when you need the actual behavior of a remote component. For example you want to see how Kafka client application behaves when there is a rebalance, you’ll want an itegration test where you can trigger an actual rebalance and validate the behavior. But an integration test does not always equate to a production-like environment, you’ll still want them contained to the code repository. Continuing with our Kafka application example, you’ll run the integration test with a Kafka broker in a docker image, so you still have a live broker, but it’s contained completely in your local (i.e on your laptop) development environment.

Let’s wrap up this section with a table sumarrizing the difference between unit and integration testing:

Table 12.1. Unit testing comparted to integration testing

Type	Purpose	Speed	Percentage of use

Unit	Method level, finer-grained logic, objects in isolation	Subsecond	Majority
Integration	Holistic, integrated components, course grained	Seconds to minutes	Minority

12.2 Testing Kafka Producers, Consumers and Connect

Let’s say you have an application that runs a simple currency exchange operation. The results of non-US currency transactions are produced to a Kafka topic named `exchange-input`. Your application consumes from the `exchange-input` topic, converts the currency amount to US currency, then produces the converted amount to another Kafka topic `exchange-output`. Here’s an abbreviated look at the code:

Listing 12.1. Currency exchange class CurrencyExchangeClient

```
// Details left out for clarity

public void runExchange() {
  while (keepRunningExchange) {
        ConsumerRecords<String, CurrencyExchangeTransaction> cons
               exchangeConsumer.poll(Duration.ofSeconds(5));
        consumerRecords.forEach(exchangeTxn -> {
            CurrencyExchangeTransaction tx = exchangeTxn.value();
            double convertedAmount =
            tx.currency().exchangeToDollars(tx.amount());
         CurrencyExchangeTransaction converted =
            new CurrencyExchangeTransaction(convertedAmount,
                      CurrencyExchangeTransaction.Currency.USD);
         ProducerRecord<String, CurrencyExchangeTransaction> prod
                  new ProducerRecord<>(outputTopic, converted);
         exchangeProducer.send(producerRecord...)
  }
 }
```

It’s a straightfoward application, and you’d like to have a test for this currency exchange process. You’d like to write a test for this applicaiton, but you don’t want to want to do it as a unit test, meaning no need for a live Kafka broker. You’ve designed the class in such a way that only the `Consumer` and `Producer` interface are the types expected:

Listing 12.2. Specifying interfaces in the constructor

```
public CurrencyExchangeClient(
   final Consumer<String, CurrencyExchangeTransaction> exchangeCo
   final Producer<String, CurrencyExchangeTransaction> exchangePr
   final String inputTopic,
   final String outputTopic) {

        this.exchangeConsumer = exchangeConsumer;
        this.exchangeProducer = exchangeProducer;
        this.inputTopic = inputTopic;
        this.outputTopic = outputTopic;
    }
```

By only specifying the interface used by your application (a proper design decision at all times) you have set yourself up for easily testing the `CurrencyExchangeClient` class by using the `MockConsumer` and `MockProducer` classes. The `MockConsumer` implements the `Consumer` interface and the `MockProducer` does likewise with the `Producer` interface so we can substitue these in a test allowing you to fully execute the application without the need of a live broker. Additionally you can verify the interactions of the consumer and producer as well as the final output.

The tradeoff for using mock objects is that you have to specify all the interactions and steps it needs to take. Let’s start with the consumer. There’s a couple of things to consider when we construct the test. The currency exchange client runs in a loop that will run indefinitely until the `CurrencyExchangeClient.close` method is called. But since the method runs in a loop, once we execute `CurrencyExchnageClient.runExchange` in the test, controll won’t return to the test until unitl the loop terminates, giving us a chicken-and-egg situation.

So what we need is a clean way to stop the loop without resorting to starting an additional thread in the test. Dealing with a loop is a common situation when dealing with code using a `KafkaConsumer` - idealy you want the consumer to run indefinately as event streams never stop. Fortunately the `MockConsumer` provides this capability with the `schedulePollTask` method which allows you to provide a task that is added to a queue the consumer will execute with a `poll(Duration)` call. Let’s take a look at this in action:

Listing 12.3. Using the schedule poll task

```
@Test
void runExchangeApplicationTest()
CurrencyExchangeClient exchangeClient = new CurrencyExchangeClien
                mockConsumer,
                mockProducer,
                "input",
                "output");

mockConsumer.schedulePollTask(() -> {  #2
    final Map<TopicPartition, Long> beginningOffsets = new HashMa
    TopicPartition topicPartition = new TopicPartition("input", 0
    beginningOffsets.put(topicPartition, 0L);
    mockConsumer.rebalance(Collections.singletonList(topicPartiti
    mockConsumer.updateBeginningOffsets(beginningOffsets);
});
```

Here you create an instance of the class under test, next you add the first task to the queue for the consumer, in this case it’s all the required setup for getting the `MockConsumer` in an initial state. Note that the tasks you supply here don’t have to provide records the consumer will return from the `poll` call, they are arbitrary code you need executed. However, at some point we want to provide records so the consumer can return them, exercising the code in the loop, which is what you’ll do next:

Listing 12.4. Adding a task for returning records

```
mockConsumer.schedulePollTask(() -> {
  mockConsumer.addRecord(new ConsumerRecord<>("input", 0, 0, null
                                                    euroTransaction
  mockConsumer.addRecord(new ConsumerRecord<>("input", 0, 1, null
                                                    gbpTransaction)
  mockConsumer.addRecord(new ConsumerRecord<>("input", 0, 2, null
                                                  jpyTransaction)
});
```

With this task, you’re supplying three records the consumer will return from the next `poll` call and all of these records should be processed **in this order** and passed to the producer. Finally you’ll add another task that will shut down the exchange client application:

Listing 12.5. Adding a final task for shutting down the exchange loop

```
mockConsumer.schedulePollTask(exchangeClient::close);  #1
exchangeClient.runExchange();  #2
```

So the final task you provide here executes the `CurrencyExchangeClient.close` method which will shutdown the application. The next command is where you start the `CurrencyExchangeClient` in the test, the loop will run three times, because you’ve provided three tasks, and then the application shuts down cleanly. But we have one more part to test and that is has the producer received the expected records with the currency exchanged to US Dollars?

Listing 12.6. Validating the producer in the test

```
List<CurrencyExchangeTransaction> actualTransactionList = mockPro
                                        .history()   #1
                                        .stream()
                                        .map((ProducerRecord::val
                                        .toList();

assertThat(actualTransactionList.get(0), equalTo(expectedEUROToUS
assertThat(actualTransactionList.get(1), equalTo(expectedGBDToUS)
assertThat(actualTransactionList.get(2), equalTo(expectedJPYToUS)
```

To validate that the producer receieved records with the correct currency translation and in the proper order you use the `MockProducer.history` method. You map the resulting `ProducerRecord` list to a list of the value objects and then compare them one at time to the expected value constructed earlier in the test.

You have tested the `CurrencyExchangeClient` application in a unit test that runs very fast. You can and should add additional tests for different error conditions that occur, but I won't cover those here. Now let's continue the discussion of testing with mocks for testing Kafka Streams operators

12.3 Creating tests for Kafka Streams operators

A Kafka Streams topology will contain one or more operations that take a Single Abstract Method (SAM) interface. This style of development allows for quickly creating an application without the need for any concreate classes for these operations, you can supply a lambda to satisify the behavior requirements. But the trade-off by doing so makes it next to impossible to provide a test for just that operation as your providing the implementation in-line. The other option is to create a concrete class implmenting the expected SAM interface, but some of these interfaces may need to work with other Kafka Streams objects that are internal to the application, making testing a challenge.

For example consider the `Punctuator` interface. Since it only has one method, `punctuate` it qualifies as a SAM interface, but typically the punctuator doesn't work by itself and requires some collaboration with other objects.

In this section I'm going to show you how to mock arbitrary interfaces and objects with Mockito(https://site.mockito.org/) to make testing with external collaborating objects a breeze.

For this example you're going to create a test for the `bbejeck.chapter_9.punctuator.StockPerformancePunctuator` class. When executed, this punctorator instance will examine the contents of a state store and foward any records that meet a certain criteria. I'm not going to discuss how Kafka Streams use punctuators here, instead I'll focus exclusively on the test. To get an idea of the collaborators for the punctuator, let's take a look at the constructor:

Listing 12.7. The constructor for the punctuator shows us the required collaborators

```
public StockPerformancePunctuator(double differentialThreshold,
        ProcessorContext<String, StockPerformance> context,
        KeyValueStore<String, StockPerformance> keyValueStore) {
```

```
  this.differentialThreshold = differentialThreshold;
  this.context = context;
  this.keyValueStore = keyValueStore;
 }
```

As you can see, the first constructor parameter is a Java primitive type which poses no issue with testing, but the other two are Kafka Streams interfaces and have expected behavior during the punctuation call. Namely you’ll iterate over everthing from the state store and forward records matching a performance metric. By using mock objects however, writing this test will be straight forward. First let’s look at creating the mock objects for the test:

Listing 12.8. Creating the required mock objects

```
 @BeforeEach
 public void setUp() {
   context = mock(ProcessorContext.class);  #1
   keyValueStore = mock(KeyValueStore.class);  #2
   stockPerformancePunctuator =
     new StockPerformancePunctuator(differentialThreshold,  #3
                                    context,
                                    keyValueStore);
}
```

Creating a mock object is as simple as calling the `Mockit.mock` (shown here as a static import) passing the class of the object or interface to mock. In our case, Mocktio returns an object implmenting the expected interface so you pass the returned objects to the `StockPerformancePunctuator` constructor to satisfy the parameter requirements.

While mock objects satisfy the interface requirements, they lack any behavior so the next step is to give the mock objects instructions on how they are going to behave which you’ll do in the test method:

Listing 12.9. The test for the StockPerformancePunctuator

```
//Some details left out for clarity
@Test
void shouldPunctuateRecordsTest() {
    StockPerformanceProto.StockPerformance stockPerformance =
         getStockPerformance(); #1
```

```
Iterator<KeyValue<String, StockPerformanceProto.StockPerforma
       List.of(KeyValue.pair("CLFT", stockPerformance)).itera
long timestamp = Instant.now().toEpochMilli();

Record<String, StockPerformanceProto.StockPerformance> record
       new Record<>("CFLT", stockPerformance, timestamp); #3

when(keyValueStore.all()) #4
      .thenReturn(
       TestUtils.kvIterator(storeKeyValues.iterator()));
context.forward(record); #5

stockPerformancePunctuator.punctuate(timestamp); #6
verify(context, times(1)).forward(record);#7
```

Some of these steps are creating the necessary objects to work with. At annotation one, you get a `StockPerformance` object created the correct properties to pass the performance criteria. Then at annotation two you create an `Iterator` by first building an `ArrayList` containing the `KeyValue` object we want the store to return. Next we create a `Record` instance to give to the `ProcessorContext` to forward.

At annotation four, your telling the mock `KeyValueStore` that when the `all` method gets called return this stubbed out instance of a `KeyValueIterator` (created by a testing utility method in the source code) which will use the real iterator you created before. Then at annotation five you set the behavior for the mock `ProcessorContext`, specifically it should expect to execute the `forward` method with the `Record` object created before.

You run the `Punctuator.punctuate` method at annotation six, which will exercise all the code inside the method including the mock objects. Finally, you validate that the mock `ProcessorContext` did exactly what it was expected to do.

In this section you’ve learned how to unit test Kafka Streams operators by using mock objects to stand in for the collaborating objects. Next we’ll move on to testing a Kafka Streams application without the need of mock objects or a live broker.

12.4 Writing tests for a Kafka Streams topology

When it comes to testing a Kafka Streams application there should be two levels of testing. It’s usually a best practice to write the different operations, filter, mapping, and aggregations etc as concrete classes so you can test them individually. But the Kafka Streams DSL takes most of these as lambda functions, making it very easy to write a complete application without much effort. But even with individual unit tests for these various functions, it’s important to have tests that exercise the entire topology to ensure everything works together as expected.

But since a Kafka Streams application is designed to work against a Kafka broker, what can we do to develop fast tests? Enter the `TopologyTestDriver` class, designed to allow you to fully test a Kafka Streams topology (complete with state stores) but without the need for a live broker, a unit test for your entire Kafka Streams topology. The best way to learn the `TopologyTestDriver` is to dive in with some examples. Let’s start with our first example of a Kafka Streams application - the Yelling Application.

Since this is our first time using the `TopologyTestDriver` we’ll step through each part of setting the test up, but for future examples we’ll only show the main part of what’s being covered.

Listing 12.10. Setting up to test Yelling Kafka Streams application

```
@Test
@DisplayName("Should Yell At Everyone")
void yellingTopologyTest() {
 KafkaStreamsYellingApp yellingApp = new KafkaStreamsYellingApp()
 Topology yellingTopology = yellingApp.topology(new Properties())
 Serializer<String> stringSerializer = Serdes.String().serializer
 Deserializer<String> stringDeserializer =
          Serdes.String().deserializer();#4
```

So you start of like you normally would for any test method by decorating the method with the `@Test` and `@DisplayName` annotations. Inside the method you first create an instance of the `KafkaStreamsYellingApp` which you’ll need to extract the topology on the next line with the `KafkaStreamsYellingApp.topology` method. This way we get the extact same topology used when running the application. You also create a serializer and deserializer which you'll see in action in the next step when we get into the crucial parts of building the test:

Listing 12.11. Building the TopologyTestDriver and input and output topics

```
 try (TopologyTestDriver driver =
                new TopologyTestDriver(yellingTopology)) { #1

    TestInputTopic<String, String> inputTopic = #2
            driver.createInputTopic("src-topic",
                                    stringSerializer,
                                    stringSerializer);

    TestOutputTopic<String, String> outputTopic = #3
            driver.createOutputTopic("out-topic",
                                    stringDeserializer,
                                    stringDeserializer);
```

In this next step you construct the `TopologyTestDriver` instance which will be the harness for running the topology under test which you pass as a constructor parameter as seen at annotation one. There are a few overloaded constructors for the `TopologyTestDriver` and we'll cover them a bit later with some of the other examples.

Next you need to create a `TestInputTopic` that you'll use to pipe records into the topology for running the test. When you create the `TestInputTopic` the name for the topic is the first parameter, and this name ***must*** match the topic name you use when building the topology. The other parameters here are the serializers for the keys and the values that you'll provide for running the test. The `TestInputTopic` will serialize each key and value you provide so that your Kafka Streams application will recieve the expected byte arrays as it would when running for real.

You'll also need an output topic for the topology to write results to as well, which is what you're doing when you create the `TestOutputTopic`. It has the same requirement for the topic name parameter that it must match the name in the real application. For the `TestOutputTopic` you supply key and value ***deserializers*** since the topology serializes the output, so you'll need a way to convert them back to concrete objects so you can validate the output of the test.

The next step in our test is to send input records through the topology then validate the output.

Listing 12.12. Sending records through the topology and validating results

```
List<String> inputValues = List.of("if you don't eat your meat",#
                    "you can't have any pudding!",
                    "How can you have any pudding",
                    "if you don't eat your meat!" );

inputTopic.pipeValueList(inputValues); #2
List<String> expectedOutput = inputValues.stream()
                                        .map(String::toUpperCase)
                                        .toList();     #3

List<String> actualOutput = outputTopic.readValuesToList();#4

assertThat(actualOutput, equalTo(expectedOutput));#5
```

In this final section of the example, you first create some sample input (Bonus points to name the artist and song lyrics we're using). You'll then take the sample input and send through the topology by using the `TestInputTopic.pipeValueList` method. In this case, since our Kafka Streams application only works on the values of the key-value pair, it's perfectly acceptable to provide all the values in a list. You're not limited to sending in a list of values though, the `TestInputTopic` provides additional methods for piping input into it for testing. I'll list a few of them here:

Listing 12.13. Other methods for piping input from the TestInputTopic

```
pipeInput(K key, V value)  #1
pipeInput(K key, V value, Instant)  #2
pipeInput(TestRecord<K, V> testRecord) #3
```

I'd like to note that there are also overloaded versions of the methods listed here accepting a `List` of the parameters, the case of the `key, value` variants you'd use a `List<KeyValue>` . So the question is, when do you decide to use the different method types? While there are no hard rules, I'll provide a table here with some general guidance:

Method Parameters	Reason to use
single value or list of values	Simple topology, input topic has no keys, no stateful operations or stateful ones where you extract the key entirely from the value
single key-value or list of key-value	Stateful operations, input topic does have keys
single TestRecord or list of TestRecord	Topologies using timestamps and headers, the test will advance stream time based on the timestamps in the records
Instant and Duration	Using an Instant provides the timestamp for that record. Overloads with an Instant and Duration will use the Instant for the starting timestamp and the Duration for the advance of each record

As you can see from the table here, the method you use for piping input into a Kafka Streams test is not arbitrary, it’s highly dependant on what actions the topology performs. There’s also a similar variety of methods for reading output from your Kafka Streams application with the `TestOutputTopic`

Listing 12.14. Methods for reading output from the test

```
readKeyValue()
readKeyValuesToList()
readRecord()
readRecordsToList()
```

The choice of method you'll use to verify the output in general, could mirror how you feed the records into the test. While not a strict rule, it's something that I do myself, dumping an entire list of input, then you can read a list out and compare what's expected versus what the acutal output you've received. Other times you my want to pipe in a record assert the outcome, then pipe another record assert the outcome and so on.

So far in this section I've introduced you setting up a `TopologyTestDriver` for a basic Kafka Streams application, but it's not limited to testsing simple topologies. You can use the `TopologyTestDriver` to test highly complex topologies and we'll see some examples of that in the next section.

But before we move on to more advanced examples, I'd like to point out another use for the `TopologyTestDriver` beyond testing your application for correctness. You can also use the TTD for building quick prototypes of a Kafka Streams application. For the most part using the TTD will provide ***most*** of the functionality you'll need for observing the behavior of a Kafka Streaams application. There are some obvious parts that the TTD can't provide, task assignemts, rebalancing, repartititions ect, but otherwise you can speed up your development by building a Kafka Streams application without the immediate need for a Kafka broker. Of course you'll always need tests with a live broker and I'll get to those a bit later in the chapter.

12.5 Testing more complex Kafka Streams applications

In this section we'll explore using the `TopologyTestDriver` for more advanced Kafka Streams applications. Since we covered creating a test from start to finish in the previous section, I'm only going to show the specific sections of the test you'll need to know for testing more advanced Kafka Streams applications. For our first venture into advanced testing we'll look into a stateful application which performs a reduce operation. I should mention here that for stateful topologies the `TopologyTestDriver` does not buffer any records, each input generates an output record.

In chapter 7, we discussed building stateful Kafka Streams applications and one of the operations covered is a `reduce` operation, so let's look at a test for one of the examples (bbejeck.chapter_7.StreamsPokerGameInMemoryStoreReducer) now:

Listing 12.15. Setting up the test for a reduce operation

```
try (TopologyTestDriver driver = new TopologyTestDriver(topology)
  TestInputTopic<String, Double> inputTopic = driver.createInputT
  TestOutputTopic<String, Double> outputTopic = driver.createOutp

 inputTopic.pipeInput("Anna", 65.75);  #1
 inputTopic.pipeInput("Matthias", 55.8);
 inputTopic.pipeInput("Neil", 47.43);
```

To refresh your memory the poker game Kafka Streams application has input of players of an online poker game where the key is the username and the value is their current score in the game. So here to start the test you input 3 scores, and you first want to verify the reduce happens in-order so you’ll want to read the output for next 3 records and assert the order matches the input order:

Listing 12.16. Asserting the order of records in a reduce

```
KeyValue<String, Double> actualKeyValue = outputTopic.readKeyValu
assertThat(actualKeyValue, equalTo(KeyValue.pair("Anna", 65.75)))

actualKeyValue = outputTopic.readKeyValue();
assertThat(actualKeyValue, equalTo(KeyValue.pair("Matthias", 55.8

actualKeyValue = outputTopic.readKeyValue();<
assertThat(actualKeyValue, equalTo(KeyValue.pair("Neil", 47.43)))
```

In this block, you read three records and assert the order of them matches exactly to the order that you piped them into the application. So far so good, but there’s another level of testing you can perform. This current application is a stateful one, so there’s a state store involved keeping the state of the latest reduce operation. While you just verified the output you can also inspect the state store and validate what’s there matches the latest output:

Listing 12.17. Validating the state store contents matches latest output

```
KeyValueStore<String, Double> kvStore =
     driver.getKeyValueStore("memory-poker-score-store"); #1

assertThat(kvStore.get("Anna"), is(65.75));#2
assertThat(kvStore.get("Matthias"), is(55.8));
assertThat(kvStore.get("Neil"), is(47.43));
```

To validate the contents of a state store, the `TopologyTestDriver` provides a `getKeyValueStore` method allowing you to retrieve a `KeyValueStore` by name, which you do here and validate the contents match the latest records output.

Tip

The `TopologyTestDriver` provides several methods for retreiving store types, session, window, and timestamped stores. For the cases where you haven’t named the store, so it’s name is generated by Kafka Streams, there is the `getAllStateStores` method returning a `Map` of the stores, from there you can iterate over the entries and extract the store from there.

To conclude the test, you pipe in a bunch of records in random order. Since you’ve already validated everything gets processed in order, you just want to verify the total final output matches what you expect it to be. But since you just inputed a bunch of random records you’ll need a way to get the final output for each key:

Listing 12.18. Retrieving the last output for inputs

```
Map<String, Double> allOutput = outputTopic.readKeyValuesToMap();
        assertThat(allOutput.get("Neil"), is (252.43));
        assertThat(allOutput.get("Anna"), is (185.75));
        assertThat(allOutput.get("Matthias"), is (180.8));

        assertThat(kvStore.get("Anna"), is(185.75)); #2
        assertThat(kvStore.get("Matthias"), is(180.8));
        assertThat(kvStore.get("Neil"), is(252.43));
```

To get the last output per key you use the `TestOutputTopic.readKeyValuesToMap` which presents a final table view of the results where more recent entries update and replace previous ones. If you want to separately inspect each result you’d use one of the

`TestOutputTopic.readXXXToList` methods.

Note

With the built in Kafka Steams aggregations don’t feel compelled to test the contents of a store. But for Kafka Streams applications involving a state store with the Processor API, I’d highly reccomend validating the contents of the store. I’ve provided this example to show how it’s done.

You just learned how to test a stateful application. Let’s move on to testing an application where the timestamps of the records drive the behavior with a test for the `StockPerformanceApplication` from chapter 9. For a quick review of the `StockPerformanceApplication` only emits records after a `punctuation` call scheduled to run every 10 seconds ***based on stream-time*** which means the punctuation only executes when stream-time advances due to timestamps of the records. To drive this punctuation behavior, you’ll need to manually provide the corresponding timestamps like so:

Listing 12.19. Providing timestamps to drive stream-time behavior from test bbejeck.chapter_9.StockPerformanceApplicationTest

```
try (TopologyTestDriver driver = new TopologyTestDriver(topology)
 //Details left out for clarity

inputTopic.pipeInput("ABC", transactionOne, instant); #1
inputTopic.pipeInput("ABC", transactionTwo,
                            instant.plus(15, ChronoUnit.SECONDS))
inputTopic.pipeInput("ABC", transactionThree,
                            instant.plus(25, ChronoUnit.SECONDS))

//Punctuation should fire 3 times
assertThat(outputTopic.getQueueSize(), is(3L));
```

So to drive the timestamp behavior you use the `TestInputTopic.pipeInput` method that accepts the following parameters: a key, value, and a timestamp for the record represented as a `java.time.Instant`. By providing these timestamps with forward time setting, it will move the stream time appropriately for Kafka Streams to perform the expected correct number of punctuations. Here we validate the corrent number of times Kafka Streams executed a `punctuate` call by verifying the number records contained in the internal queue of the `TestOutputTopic`.

Tip

For Kafka Streams applications with punctuation based on ***wallclock time*** you’ll have to explicity move the internal wallclock time of the `TopologyTestDriver` with the `advanceWallClockTime(Duration advanceAmount)` method.

You should note that using the `pipeInput` method accepting a timestamp parameter only advances stream-time tracked by the `TopologyTestDriver` and does not advance the internal event time of the `TestInputTopic`. To understand what this means is that if you provide another record to the `TestInputTopic` without explictly providing a timestamp, that record’s event time will be the intial timestamp of the `TestInputTopic` when you created it in this case, as when you provide a record without an explicit timestamp it uses the current event time of the topic. To advance the event time of the input topic you’d take an approach like this:

Listing 12.20. Providing timestamps by advancing the event time of the input topic

```
inputTopic.pipeInput("ABC", transactionOne);
inputTopic.advanceTime(Duration.ofSeconds(15));#1
inputTopic.pipeInput("ABC", transactionTwo);
inputTopic.advanceTime(Duration.ofSeconds(25));#2
inputTopic.pipeInput("ABC", transactionThree);
```

This example provides the same behavior for the test, it validates Kafka Streams performing 3 punctuations. Which approach should you use? The answer to that depends on how you structure your test. If you provide a small number of records, manually setting each timestamp is ideal as you can clearly see the timestamp for each record. However if you have a test where you want generate a large number of inputs, it could be cummbersome to set a timestamp for each record; so advancting the event time of the `TestInputTopic` at various intervals would be more effective in that case.

Before we wrap up our coverage of the `TopologyTestDriver` we have one more example to cover regarding timestamps on records and that is when you have windowed Kafka Streams applications. For this example you’ll look at a test for the `bbejeck.chapter_8.window.StreamsCountTumblingWindowSuppressedEage` application, which does a windowed count of incoming records where the window size is one minute with no grace period.

When you have a windowed operation with suppression, Kafka Streams will not emit a result until the window closes, which means that you’ll need to input records with timestamps moving stream-time forward to validate the results. To do window stream-time advancement you’ll take a similar approach to what you did with the test for punctuation found in the `bbejeck.chapter_8.window.StreamsCountTumblingWindowSuppressedEage`

Listing 12.21. Setting timestamps to move stream time ahead to emit suppressed windowed results in

```
 Stream.generate(() -> "Foo").limit(10)
            .forEach(item -> inputTopic.pipeInput(item, item));
assertThat(outputTopic.getQueueSize(), is(0L));  #2
inputTopic.pipeInput("Foo", "Foo",
    instant.plus(75, ChronoUnit.SECONDS));  #3
assertThat(outputTopic.readValue(), is(10L));  #4
```

For this test you first generate 10 records and input them into the topology and assert that Kafka Streams has not emitted any records by validating the queue of the output topic is empty. You then add another record and explicity set it’s timestamp to one minute fifteen seconds in the future. Since the window of the application is set to one minute, you should observe Kafka Streams forward the count of all events by key from the previous window with a count of ten.

Note

This approach of setting forward timestamps to move stream time is the way to test any windowed Kafka Streams application

So far you’ve learned about unit testing for Kafka producer and consumer clients and testing a Kafka Streams application without the need of a live broker. But unit testing and not using a broker is only part of the picture, some of your tests really should include integration of a Kafka broker and we’ll cover that in our final section next.

12.6 Developing effective integration tests

For developing an integration test for Kafka Streams the main difference you’ll notice is the extra code needed to interact with a live broker. Additionally you’ll have to account for how Kafka Streams works as live application which is especially true when you have a stateful operation due to the caching behavior. You’ll still write a JUnit 5 test, but it will operatate a little bit differently mainly due to time it takes to start the broker, which where we’ll start.

You’re going to use Testcontainers for providing a Kafka broker for the integration tests. Testcontainers (https://www.testcontainers.org/) is a Java library which provides access to external components running in Docker containers directly in your JUnit tests. I’m going to assume you are familiar with Docker, but for more information can go to the Docker website - https://www.docker.com/ for more information.

Note

Testcontainers for Kafka in JUnit 5 tests requires the following dependencies: org.testcontainers:junit-jupiter:1.17.1 and org.testcontainers:kafka:1.17.1. The source code for the book already does this but I’m adding this here for your information.

Let’s get started with your integration testing by creating a test for the `bbejeck.chapter_8.window.StreamsCountHoppingWindow` application. The Testcontainers library provides annotations that you’ll use to annotate your code that will handle the lifecycle of your Kafka Docker container. Let’s start building the test now:

Listing 12.22. Building the integration tests with the required annotations for Testcontainers

```
@Testcontainers  #1
class StreamsCountHoppingWindowIntegrationTest {


@Container  #2
private static final KafkaContainer kafka =
   new KafkaContainer(
        DockerImageName.parse("confluentinc/cp-kafka:7.3.0.arm64"
```

You first create the test class `StreamsCountHoppingWindowIntegrationTest` and add `@Testcontainers` at the class level. The `@Testcontainers` annotation is a JUnit Jupiter extension and it automatically manages the lifecycle for any containers in the test by finding any fields with a `@Container` annotation. You can see the `@Container` here annotating the `KafkaContainer` field. When you define the field as static the container is shared with all test methods meaning it starts before the first test and shuts down after the last test completes.

Should you define the container field as non-static then the container starts and stops for each test. Unless you have a specific reason for stopping and starting a container for each test, I’d recommend using the static field approach as this will save a few CPU cycles by only starting and stopping the container once.

At annotation 3 you are creating the `KafkaContainer` instance by passing a `DockerImageName` object which you create by passing in a string in the standard Docker format (registry/name:tag).

Note

I’m not going to cover Docker but you can learn more about it in https://www.manning.com/books/docker-in-practice-second-edition and https://www.manning.com/books/docker-in-action-second-edition from Manning books.

As far as setting up the Kafka Docker container for the test you are done! What’s left is for us to discuss the elements you’ll need for running the Kafka Streams application in the test. Since Kafka Streams gets its input from topics and produces the final results to a Kafka topic as well, you’ll need a

`KafkaProducer` to feed a topic for the test and a `KafkaConsumer` to help in validating the test results.

Let’s take a look at the details you need to put together: .Setting the test up with producer and consumer configurations

```
@BeforeEach
public void setUp() { #1
 //Details left out for clarity

   streamsCountHoppingWindow = new StreamsCountHoppingWindow();
   kafkaStreamsProps.put("bootstrap.servers",
             kafka.getBootstrapServers());#2

    kafkaStreamsProps.put(StreamsConfig.APPLICATION_ID_CONFIG,
             "hopping-windows-integration-test");

   //Producer configs
   producerProps.put("bootstrap.servers",
             kafka.getBootstrapServers());  #2
   producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                          StringSerializer.class);
       producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_C
                         StringSerializer.class);


   //Consumer configs
       consumerProps.put("bootstrap.servers",
              kafka.getBootstrapServers());  #2
       consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_C
                                 StringDeserializer.class);
       consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS
                                 LongDeserializer.class);
       consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG,
                                "integration-consumer");
       consumerProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG
                                "earliest");

       Topics.create(kafkaStreamsProps,
                     streamsCountHoppingWindow.inputTopic,
                     streamsCountHoppingWindow.outputTopic);#3
```

In the test `setUp` method you set all the required configurations for Kafka Streams, producer and, consumer clients. You wont' directly create a `KafkaProducer` and `KafkaConsumer` in the test,relying instead on some static helper methods to produce and consume records as needed in the test and I’ll cover those methods soon. The last bit of code you see in the `setUp` method is creating the required topics. There’s a corresponding method `tearDown` executed after each test completes as well:

Listing 12.23. The teardown method run after each test

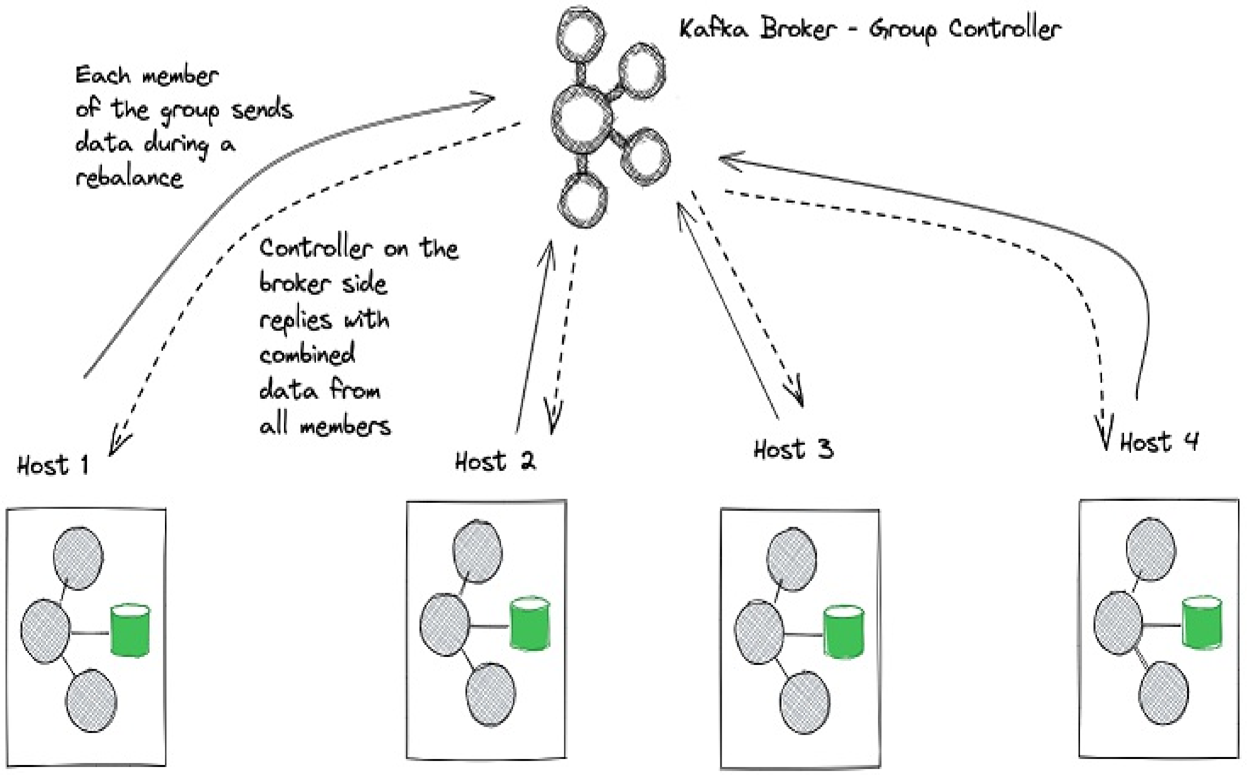
```
@AfterEach
public void tearDown() {
    Topics.delete(kafkaStreamsProps,
                  streamsCountHoppingWindow.inputTopic,
                  streamsCountHoppingWindow.outputTopic);
}
```

The `tearDown` method deletes all the topics created before the test ran. It’s important to follow this practice of creating and deleting topics for each test ensuring we have a clean starting point so the results aren’t affected by the running of a previous test. To do accomplish the creation and deletion of topics we use the utility class `bbejeck.utils.Topics` provided in the book’s source code.

Now let’s get into the heart of the integration test. Since the Kafka Streams application under test, `StreamsCountHoppingWindow` uses hopping windows (windows that have an advance smaller than the window size) with a size of 1 minute with an advance of 10 seconds you’d like to provide values in such a way that you can clearly see the overlap of each window advance, meaning each time the window moves forward by 10 seconds, the count includes previous results until the window size is reached, then you’ll see the count to start to decline.

Here’s an illustration showing what we want our test to validate:

Figure 12.1. Integration test for a hopping window count should increase up to the window size then start declining


Kafka Broker - Group Controller
Each member of the group sends data during a rebalance
Controller on the broker side replies with combined data from all members
Host 1
Host 2
Host 3
Host 4

Note

Since I’ve covered Kafka Streams windowing in a previous chapter, I’m only going to focus on details relevant for the test.

Now that you have all setup details complete let’s get to writing the test itself:

Listing 12.24. Writing the test for a hopping window Kafka Stream application

```
@Test
@DisplayName("Integration test for hopping window")
void shouldHaveHoppingWindowsTest() {
    final Topology topology =
         streamsCountHoppingWindow.topology(kafkaStreamsProps);
    AtomicBoolean streamsStarted = new AtomicBoolean(false);
    try (KafkaStreams kafkaStreams =
            new KafkaStreams(topology, kafkaStreamsProps)) {
        kafkaStreams.cleanUp();
        kafkaStreams.setStateListener((newState, oldState) -> { #
            if (newState == KafkaStreams.State.RUNNING) {
                streamsStarted.set(true);
            }
        });
        kafkaStreams.start(); #3
        while (!streamsStarted.get()) {  #4
            time.sleep(250);
        }
```

In the test method you create the `KafkaStreams` instance and before starting you set a `StateListener` for when Kafka Streams goes into a RUNNING state, that way we can wait to start the test until Kafka Streams is ready for work. While this step is not required, it’s my preference to start the test once that application is in a known state.

Now let’s move on to producing input for Kafka Streams to work with:

Listing 12.25. Producing records for the Kafka Streams application

```
long startTimestamp = Instant.now().toEpochMilli();  #1
for (int i = 1; i <= 6; i++) {
  List<KeyValue<String, String>> list =
  Stream.generate(() ->
           (KeyValue.pair("Foo", "Bar"))).limit(i).toList();  #2
  TestUtils.produceKeyValuesWithTimestamp(  #3
            streamsCountHoppingWindow.inputTopic,
            list,
            producerProps,
            startTimestamp,
            Duration.ofMillis(100L));
  startTimestamp += 10_000;  #4
}
```

Here you’re using the helper method `TestUtils.produceKeyValuesWithTimestamp` from the book’s source code that takes care of all the details of produing message to Kafka for the `StreamsCountHoppingWindow` to process. For each iteration we send the number of records matching the loop index and increment the timestamp by 10 seconds to ensure the next batch of records produced will end up in next the window advance. So what we expect to see is each advance will include the sum of the previous records up to the window size boundary, then we should see the count start declining as records age off.

So test our expectations we add the following code to the test:

Listing 12.26. Setting an expected list of records we expect Kafka Streams to produce

```
List<KeyValue<String, Long>> expectedKeyValues =
    List.of(KeyValue.pair("Foo", 1L),
    KeyValue.pair("Foo", 3L),
    KeyValue.pair("Foo", 6L),
    KeyValue.pair("Foo", 10L),
    KeyValue.pair("Foo", 15L),
    KeyValue.pair("Foo", 21L), #1
    KeyValue.pair("Foo", 20L),
    KeyValue.pair("Foo", 18L),
    KeyValue.pair("Foo", 15L),
    KeyValue.pair("Foo", 11L),
    KeyValue.pair("Foo", 6L));


List<KeyValue<String, Long>> actualConsumed =
     TestUtils.readKeyValues(streamsCountHoppingWindow.outputTopi
                    consumerProps,
                    45_000, #3
                    12);  #4
assertThat(actualConsumed, equalTo(expectedKeyValues));  #5
```

So we create an expected list where the count increments by the number of records sent plus the previous count (1 + 2 + 3 + 4..) then we use another utility method to consume from the topic and return the results to the test to assert the results match our expectations of the application output. When running this test, it should pass every time. The main points of this section are to first create an expected results for comparison, but also to use utility helper methods that faciliate re-use across the different integration tests. I haven’t included examples of integration testing for the producer and consumer classes, but you’d follow the same pattern, create a test class with the `@Testcontainer` annotation and a `@Container` field in the class and use the helper methods for producing and consuming to drive the test. There are examples of these types of integration tests in source code for the book.

12.7 Summary

- Testing is a critical part of development and there should be tests corresponding to each piece of the application to a reasonable level.
- Unit tests should make up the bulk of your testing strategy, but you’ll

still need to include integration tests to make sure everything works together as expected.

- Using the MockProducer and MockConsumer are effective for performing testing of client applications without requiring the use of a live Kafka broker.
- For Kafka Streams applciations you should have unit tests for the various mappers, aggregators and, punctuators to validate their expected behavior, but you can also use the `TopologyTestDriver` to test your entire topology but without the need for a live Kafka broker.
- When performing an integration test you should use Testcontainers to provide the Kafka broker for the test as it takes care of getting the container and managing the container lifecycle for you.