Kafka Streams IN ACTION

Real-time apps and microservices with the Kafka Streaming API

Bill Bejeck

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Kafka Streams in Action
Real-time apps and microservices with the Kafka Streaming API
Version 9

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

welcome

After several years of subscribing to Manning's MEAPs, I'm excited now to welcome you to the MEAP for my own book, *Kafka Streams in Action*!

These days, you can't afford to ignore the hot topics of Big Data, streaming data, and distributed programming. We're at a point where stream processing has become an increasingly important factor for businesses looking to harness the power from data generated in real time. With 13 years in software development, I've spent the last six years working exclusively on the back end, leading ingest teams, and handling large volumes of data daily.

Kafka Streams makes it seamless to implement stream processing on the data flowing into Kafka. So, while Kafka is a de facto standard in the industry for feeding and exporting data, Kafka Streams represents a powerful new feature. In this book, I will teach you Kafka Streams, so you, too, can add stream processing to your toolkit.

This is a particular exciting time, as this is the last MEAP release for Kafka Streams in Action! All chapters and code examples have been updated for the new 1.0 API which came out in October of 2017.

In addition the book has been updated per reviewer comments, so what you read here should be very close the final published version.

While I've made every attempt to make sure the book is clear an accurate, feel free to join me on the *Kafka Streams in Action* Author Forum at manning.com, to ask questions, offer feedback, and participate in the conversation to shape this book.

—Bill Bejeck

brief contents

Part 1: Getting Started with Kafka Streams

In part 1 of this book, we'll discuss the big data era: how it began with the need to process large amounts of data and eventually progressed to stream processing—processing data as it becomes available. We'll also discuss what Kafka Streams is, and I'll show you a mental model of how it works without any code so you can focus on the big picture. We'll also briefly cover Kafka to get you up to speed on how to work with it.

1 Welcome to Kafka Streams

This chapter covers

- Understanding how the big data movement changed the programming landscape
- Getting to know how stream processing works and why we need it
- Introducing Kafka Streams
- Looking at the problems solved by Kafka Streams

In this book, you'll learn how to use Kafka Streams to solve your streaming application needs. From basic extract, transform, and load (ETL) to complex stateful transformations to joining records, we'll cover the components of Kafka Streams so you can solve these kinds of challenges in your streaming applications.

Before we dive into Kafka Streams, we'll briefly explore the history of big data processing. As we identify problems and solutions, you'll clearly see how the need for Kafka, and then Kafka Streams, evolved. Let's look at how the big data era got started and what led to the Kafka Streams solution.

1.1 The big data movement, and how it changed the programming landscape

The modern programming landscape has exploded with big data frameworks and technologies. Sure, client-side development has undergone transformations of its own, and the number of mobile device applications has exploded as well. But no matter how big the mobile device market gets or how client-side technologies evolve, there's one constant: we need to process more and more data every day. As the amount of data grows, the need to analyze and take advantage of the benefits of that data grows at the same rate.

But having the ability to process large quantities of data in bulk (*batch processing*) isn't always enough. Increasingly, organizations are finding that they need to process data as it becomes available (*stream processing*). *Kafka Streams*, a cutting-edge approach to stream processing, is a library that allows you to perform per-event processing of records. Per-event processing means you process each record as soon as it's available—no grouping of data into small batches (*microbatching*) is required.

> **NOTE** When the need to process data as it arrives became more and more apparent, a new strategy was developed: *microbatching*. As the name implies, microbatching is nothing more than batch processing, but with smaller quantities of data. By reducing the size of the batch, microbatching can sometimes produce results more quickly; but microbatching is still batch processing, although at faster intervals. It doesn't give you real per-event processing.

1.1.1 The genesis of big data

The internet started to have a real impact on our daily lives in the mid-1990s. Since then, the connectivity provided by the web has given us unparalleled access to information and the ability to communicate instantly with anyone, anywhere in the world. An unexpected byproduct of all this connectivity emerged: the generation of massive amounts of data.

For our purposes, I'll say that the big data era officially began in 1998, the year Sergey Brin and Larry Page formed Google. Brin and Page developed a new way of ranking web pages for searches: the PageRank algorithm. At a very high level, the PageRank algorithm rates a website by counting the number and quality of links pointing to it. The assumption is that the more important or relevant a web page is, the more sites will refer to it.

Figure 1.1 offers a graphical representation of the PageRank algorithm:

- Site A is the most important, because it has the most references pointing to it.
- Site B is somewhat important. Although it doesn't have as many references, an important site does point to it.
- Site C is less important than A or B. More references are pointing to site C than site B, but the quality of those references is lower.
- The sites at the bottom (D through I) have no references pointing to them. This makes them the least valuable.

The figure is an oversimplification of the PageRank algorithm, but it gives you the basic idea of how the algorithm works.

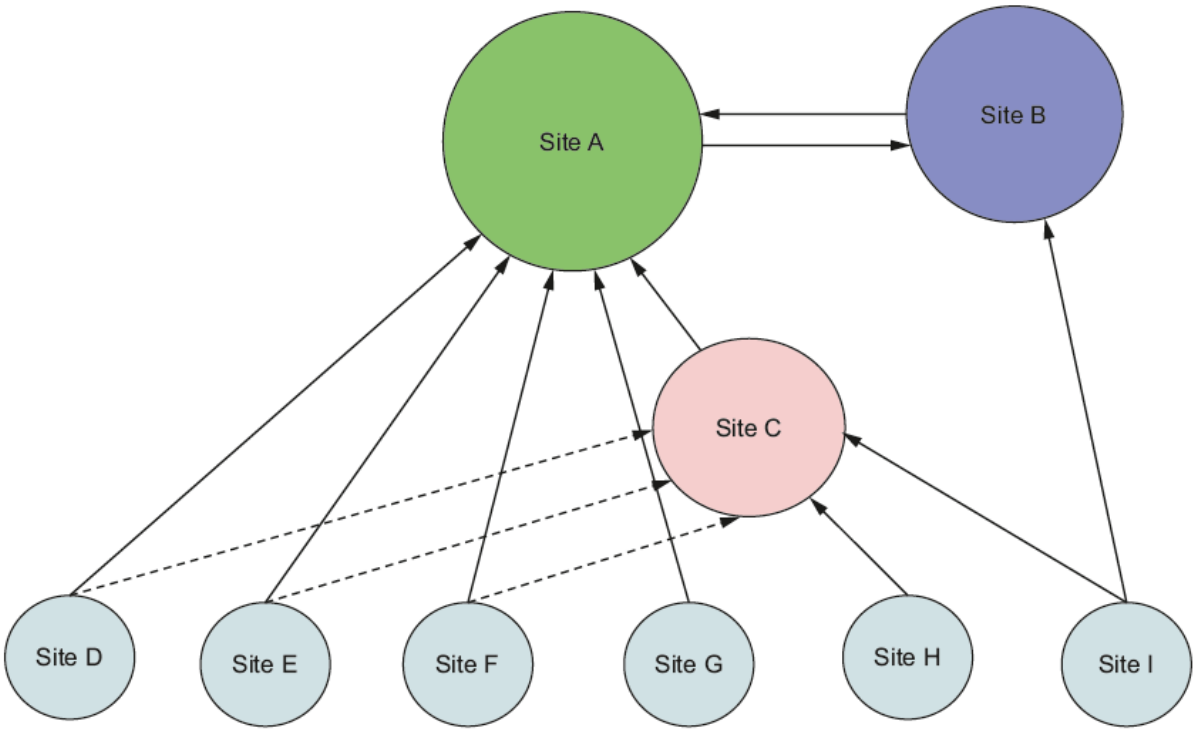


Figure 1.1 The PageRank algorithm in action. The PageRank algorithmcircles represent websites, and the larger ones represent sites with more links pointing to them from other sites.

At the time, PageRank was a revolutionary approach. Previously, searches on the web were more likely to use Boolean logic to return results. If a website contained all or most of the terms you were looking for, that website was in the search results, regardless of the quality of the content. But running the PageRank algorithm on all internet content required a new approach—the traditional approaches to working with data took too long. For Google to survive and grow, it needed to index all that content quickly ("quickly" being a relative term) and present quality results to the public.

Google developed another revolutionary approach for processing all that data: the

MapReduce paradigm. Not only did MapReduce enable Google to do the work it needed to as a company, it inadvertently spawned an entire new industry in computing.

1.1.2 Important concepts from MapReduce

The map and reduce functions weren't new concepts when Google developed MapReduce. What was unique about Google's approach was applying those simple concepts at a massive scale across many machines.

At its heart, MapReduce has roots in functional programming. A map function takes some input and maps that input into something else without changing the original value. Here's a simple example in Java 8, where a `LocalDate` object is mapped into a `String` message, while the original `LocalDate` object is left unmodified:

```
Function<LocalDate, String> addDate =
       (date) -> "The Day of the week is " + date.getDayOfWeek();
```

Although simple, this short example is sufficient for demonstrating what a map function does.

On the other hand, a reduce function takes a number of parameters and reduces them down to a singular, or at least smaller, value. A good example of that is adding together all the values in a collection of numbers.

To perform a reduction on a collection of numbers, you first provide an initial starting value. In this case, we'll use `0` (the identity value for addition). The next step is adding the seed value to the first number in the list. You then add the result of that first addition to the second number in the list. The function repeats this process until it reaches the last value, producing a single number.

Here are the steps to reduce a `List<Integer>` containing the values `1`, `2`, and `3`:

```
0 + 1 = 1   ❶
1 + 2 = 3   ❷
3 + 3 = 6   ❸
```

❶ Adds the seed value to the first number

❷ Takes the result from step 1 and adds it to the second number in the list

❸ Adds the sum of step 2 to the third number

As you can see, a reduce function collapses results together to form smaller results. As in the map function, the original list of numbers is left unchanged.

The following example shows an implementation of a simple reduce function using a Java 8 lambda:

```
List<Integer> numbers = Arrays.asList(1, 2, 3);

int sum = numbers.reduce(0, (i, j) -> i + j );
```

The main topic of this book is not MapReduce, so we'll stop our background discussion here. But some of the key concepts introduced by the MapReduce paradigm (later implemented in Hadoop, the original open source version based on Google's MapReduce white paper) come into play in Kafka Streams:

- How to distribute data across a cluster to achieve scale in processing
- The use of key/value pairs and partitions to group distributed data together
- Instead of avoiding failure, embracing failure by using replication

The following sections look at these concepts in general terms. Pay attention, because you'll see them coming up again and again in the book.

DISTRIBUTING DATA ACROSS A CLUSTER TO ACHIEVE SCALE IN PROCESSING

Working with 5 TB (5,000 GB) of data could be overwhelming for one machine. But if you can split up the data and involve more machines, so each is processing a manageable amount, your problem is minimized. Table 1.1 illustrates this clearly.

Table 1.1 How splitting up 5 TB helps processing

Number of machines	Amount of data processed per server
10	500 GB
100	50 GB
1000	5 GB
5000	1 GB

As you can see from the table, you may start out with an unwieldy amount of data to process, but by spreading the load across more servers, you eliminate the difficulty of processing the data. The 1 GB of data in the last line of the table is something a laptop could easily handle.

This is the first key concept to understand about MapReduce: by spreading the load across a cluster of machines, you can turn an overwhelming amount of data into a

manageable amount.

USING KEY/VALUE PAIRS AND PARTITIONS TO GROUP DISTRIBUTED DATA

The key/value pair is a simple data structure with powerful implications. In the previous section, you saw the value of spreading a massive amount of data over a cluster of machines. Distributing your data solves the processing problem, but now you have the problem of collecting the distributed data back together.

To regroup distributed data, you can use the keys from the key/value pairs to partition the data. The term *partition* implies grouping, but I don't mean grouping by identical keys, but rather by keys that have the same hash code. To split data into partitions by key, you can use the following formula:

```
int partition = key.hashCode() % numberOfPartitions;
```

Figure 1.2 shows how you could apply a hashing function to take results from Olympic events stored on separate servers and group them on partitions for different events. All the data is stored as key/value pairs. In the image below the key is the name of the event, and the value is a result for an individual athlete.

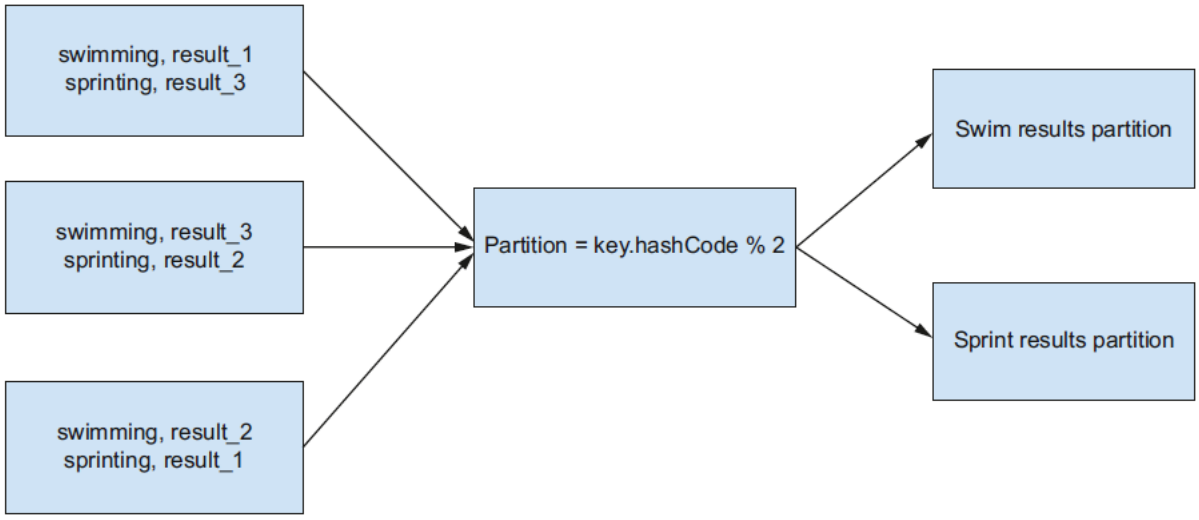


Figure 1.2 Grouping records by key on partitions. Even grouping recordsthough the records start out on separate servers, they end up in the appropriate partitions.

Partitioning is an important concept, and you'll see detailed examples in later chapters.

EMBRACING FAILURE BY USING REPLICATION

Another key component of Google's MapReduce is the Google File System (GFS). Just as Hadoop is the open source implementation of MapReduce, Hadoop File System (HDFS) is the open source implementation of GFS.

At a very high level, both GFS and HDFS split data into blocks and distribute those blocks across a cluster. But the essential part of GFS/HDFS is the approach to server and disk failure. Instead of trying to prevent failure, the framework embraces failure by replicating blocks of data across the cluster (by default, the replication factor is 3).

By replicating data blocks on different servers, you no longer have to worry about disk failures or even complete server failures causing a halt in production. Replication of data is crucial for giving distributed applications fault tolerance, which is essential for a distributed application to be successful. You'll see later how partitions and replication work in Kafka Streams.

1.1.3 Batch processing is not enough

Hadoop caught on with the computing world like wildfire. It allowed people to process vast amounts of data and have fault tolerance while using commodity hardware (cost savings). But Hadoop/MapReduce is a batch-oriented process, which means you collect large amounts of data, process it, and then store the output for later use. Batch processing is a perfect fit for something like PageRank because you can't make determinations of what resources are valuable across the entire internet by watching user clicks in real time.

But business also came under increasing pressure to respond to important questions more quickly, such as these:

- What is trending right now?
- How many invalid login attempts have there been in the last 10 minutes?
- How is our recently released feature being utilized by the user base?

It was apparent that another solution was needed, and that solution was stream processing.

1.2 Introducing stream processing

There are varying definitions of stream processing. In this book, I define *stream processing* as working with data as it's arriving in your system. The definition can be further refined to say that stream processing is the ability to work with an infinite stream of data with continuous computation, as it flows, with no need to collect or store the data to act on it.

Figure 1.3 represents a stream of data, with each circle on the line representing data at a point in time. Data is continuously flowing, as data in stream processing is unbounded.

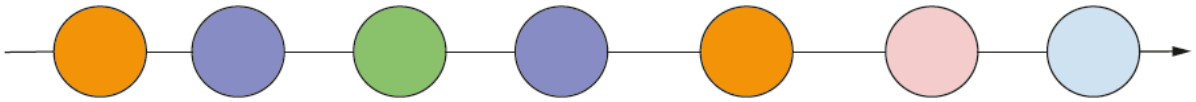


Figure 1.3 This marble diagram is a simple representation of stream processing. Each circle represents some information or an event occurring at a particular point in time. The number of events is unbounded and moves continually from left to right.

Who needs to use stream processing? Anyone who needs quick feedback from an observable event. Let's look at some examples.

1.2.1 When to use stream processing, and when not to use it

Like any technical solution, stream processing isn't a one-size-fits-all solution. The need to quickly respond to or report on incoming data is a good use case for stream processing. Here are a few examples:

- *Credit card fraud*—A credit card owner may not notice a card has been stolen, but 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*—Analyzing application log files after a breach has occurred may be helpful to prevent future attacks or to improve security, but the ability to monitor aberrant behavior in real time is critical.
- *A large race, such as the New York City Marathon*—Almost all runners will have a chip on their shoe, and when runners pass sensors along the course, you can use that information to track the runners' positions. By using the sensor data, you can determine the leaders, spot potential cheating, and detect whether a runner is potentially having problems.
- *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.

On the other hand, stream processing isn't a solution for all problem domains. To effectively make forecasts of future behavior, for example, you need to use a large amount of data over time to eliminate anomalies and identify patterns and trends. Here

the focus is on analyzing data over time, rather than just the most current data:

- *Economic forecasting*—Information is collected on many variables over an extended period of time in an attempt to make an accurate forecast, such as trends in interest rates for the housing market.
- *School curriculum changes*—Only after one or two testing cycles can school administrators measure whether curriculum changes are achieving their goals.

Here are the key points to remember: If you need to report on or take action immediately as data arrives, stream processing is a good approach. If you need to perform in-depth analysis or are compiling a large repository of data for later analysis, a stream-processing approach may not be a good fit. Let's now walk through a concrete example of stream processing.

1.3 Handling a purchase transaction

Let's start by applying a general stream-processing approach to a retail sales example. Then we'll look at how you can use Kafka Streams to implement the stream-processing application.

Suppose Jane Doe is on her way home from work and remembers she needs toothpaste. She stops at a ZMart, goes in to pick up the toothpaste, and heads to the checkout to pay. The cashier asks Jane if she's a member of the ZClub and scans her membership card, so Jane's membership info is now part of the purchase transaction.

When the total is rung up, Jane hands the cashier her debit card. The cashier swipes the card and gives Jane the receipt. As Jane is walking out of the store, she checks her email, and there's a message from ZMart thanking her for her patronage, with various coupons for discounts on Jane's next visit.

This transaction is a normal occurrence that a customer wouldn't give a second thought to, but you'll have recognized it for what it is: a wealth of information that can help ZMart run more efficiently and serve customers better. Let's go back in time a little, to see how this transaction became a reality.

1.3.1 Weighing the stream-processing option

Suppose you're the lead developer for ZMart's streaming-data team. ZMart is a big-box retail store with several locations across the country. ZMart does great business, with total sales for any given year upwards of $1 billion. You'd like to start mining the data from your company's transactions to make the business more efficient. You know you have a tremendous amount of sales data to work with, so whatever technology you implement will need to be able to work fast and scale to handle this volume of data.

You decide to use stream processing because there are business decisions and opportunities that you can take advantage of as each transaction occurs. After data is gathered, there's no reason to wait for hours to make decisions. You get together with management and your team and come up with the following four primary requirements for the stream-processing initiative to succeed:

- *Privacy*—First and foremost, ZMart values its relationship with its customers. With all of today's privacy concerns, your first goal is to protect customers' privacy, and protecting their credit card numbers is the highest priority. However you use the transaction information, customer credit card information should never be at risk of exposure.
- *Customer rewards*—A new customer-rewards program is in place, with customers earning bonus points based on the amount of money they spend on certain items. The goal is to notify customers quickly, once they've received a reward—you want them back in the store! Again, appropriate monitoring of activity is required here. Remember how Jane received an email immediately after leaving the store? That's the kind of exposure you want for the company.
- *Sales data*—ZMart would like to refine its advertising and sales strategy. The company wants to track purchases by region to figure out which items are more popular in certain parts of the country. The goal is to target sales and specials for best-selling items in a given area of the country.
- *Storage*—All purchase records need to be saved in an off-site storage center for historical and ad hoc analysis.

These requirements are straightforward enough on their own, but how would you go about implementing them against a single purchase transaction like Jane Doe's?

1.3.2 Deconstructing the requirements into a graph

Looking at the preceding requirements, you can quickly recast them in a *directed acyclic graph* (DAG). The point where the customer completes the transaction at the register is the source node for the entire graph. ZMart's requirements become the child nodes of the main source node (figure 1.4).

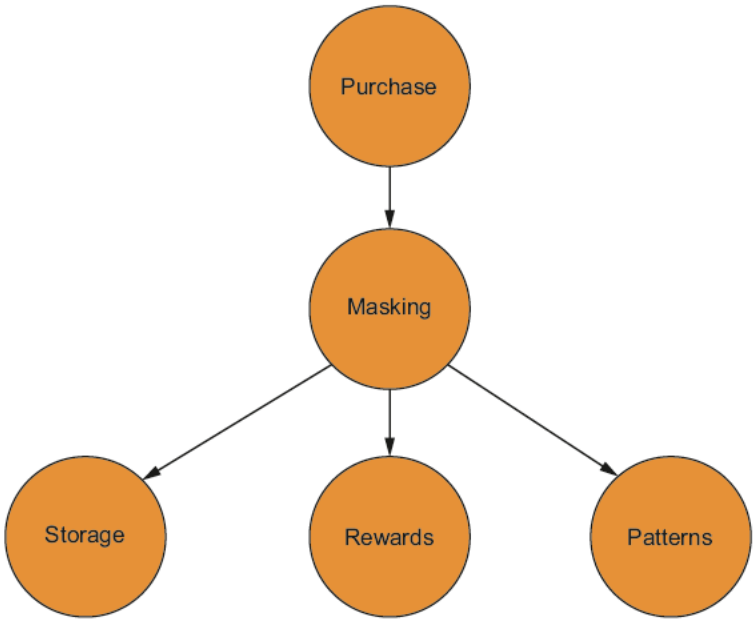


Figure 1.4 The business requirements for the streaming application presented as a directed acyclic graph. Each vertex represents a requirement, and the edges show the flow of data through the graph.

Next, you need to determine how to map a purchase transaction to the requirements graph.

1.4 Changing perspective on a purchase transaction

In this section, we'll walk through the steps of a purchase and see how it relates, at a high level, to the requirements graph from figure 1.4. In the next section, we'll look at how to apply Kafka Streams to this process.

1.4.1 Source node

The graph's source node (figure 1.5) is where the application consumes the purchase transaction. This node is the source of the sales transaction information that will flow through the graph.

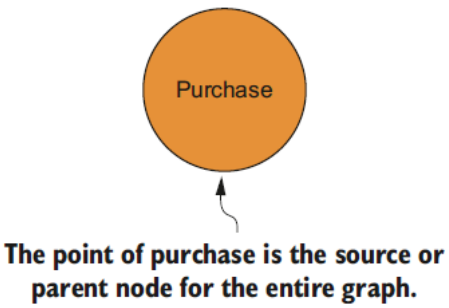


Figure 1.5 The simple start for the sales transaction graph. This node is the source of raw sales transaction information that will flow through the graph.

1.4.2 Credit card masking node

The child node of the graph source is where the credit card masking takes place (figure 1.6). This is the first vertex or node in the graph that represents the business requirements, and it's the only node that receives the raw sales data from the source node, effectively making this node the source for all other nodes connected to it.

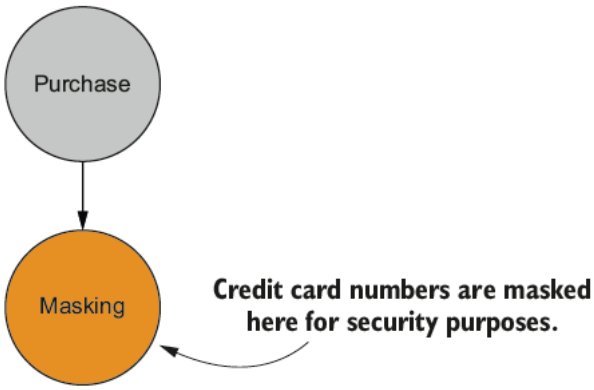


Figure 1.6 The first node in the graph that represents the business requirements. This node is responsible for masking credit card numbers and is the only node that receives the raw sales data from the source node, effectively making it the source for all other nodes connected to it.

For the credit card masking operation, you make a copy of the data and then convert all the digits of the credit card number to an *x*, except the last four digits. The data flowing through the rest of the graph will have the credit card field converted to the *xxxx-xxxx-xxxx-1122* format.

1.4.3 Patterns node

The patterns node (figure 1.7) extracts the relevant information to establish where customers purchase products throughout the country. Instead of making a copy of the data, the patterns node will retrieve the item, date, and ZIP code for the purchase and create a new object containing those fields.

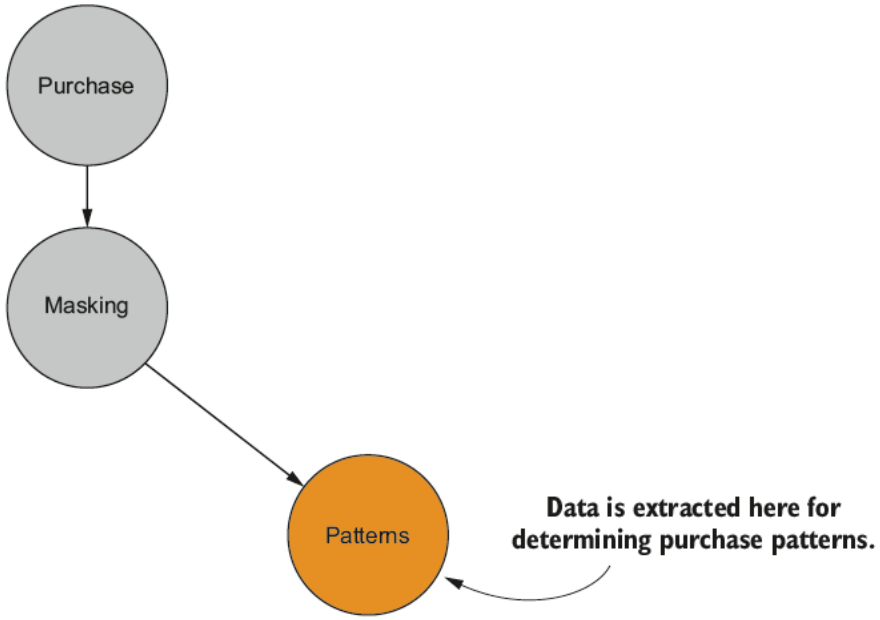


Figure 1.7 The patterns node consumes purchase information from the masking node and converts it into a record showing when a customer purchased an item and the ZIP code where the customer completed the transaction.

1.4.4 Rewards node

The next child node in the process is the rewards accumulator (figure 1.8). ZMart has a customer rewards program that gives customers points for purchases made in the store. This node's role is to extract the dollar amount spent and the client's ID and create a new object containing those two fields.

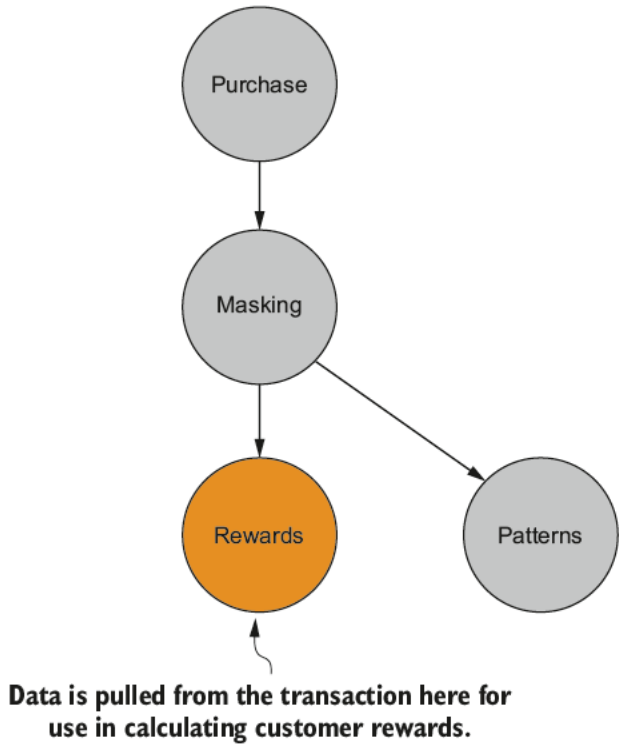


Figure 1.8 The rewards node is responsible for consuming sales records from the masking node and converting them into records containing the total of the purchase and the customer ID.

1.4.5 Storage node

The final child node writes the purchase data out to a NoSQL data store for further analysis (figure 1.9).

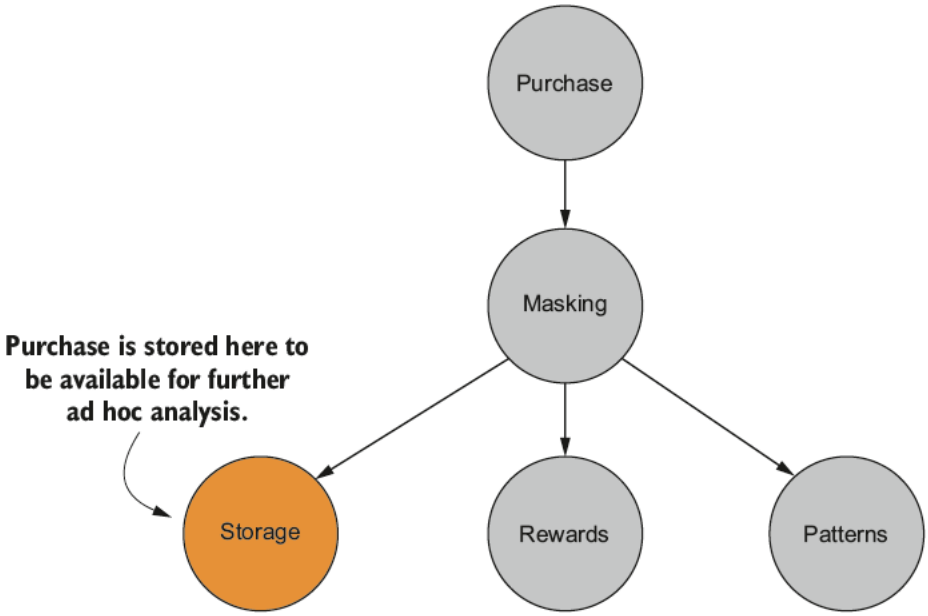


Figure 1.9 The storage node consumes records from the masking node as well. These records aren't converted into any other format but are stored in a NoSQL data store for ad hoc analysis later.

We've now tracked the example purchase transaction through ZMart's graph of requirements. Let's see how you can use Kafka Streams to convert this graph into a functional streaming application.

1.5 Kafka Streams as a graph of processing nodes

Kafka Streams is a library that allows you to perform per-event processing of records. You can use it to work on data as it arrives, without grouping data in microbatches. You process each record as soon as it's available.

Most of ZMart's goals are time sensitive, in that you want to take action as soon as possible. Preferably, you'll be able to collect information as events occur. Additionally, there are several ZMart locations across the country, so you'll need all the transaction

records to funnel into a single flow or *stream* of data for analysis. For these reasons, Kafka Streams is a perfect fit. Kafka Streams allows you to process records as they arrive and gives you the low-latency processing you require.

In Kafka Streams, you define a topology of processing *nodes* (I'll use the terms *processor* and *node* interchangeably). One or more nodes will have as source Kafka topic(s), and you can add additional nodes, which are considered child nodes (if you aren't familiar with what a Kafka topic is, don't worry—I'll explain in detail in chapter 2). Each child node can define other child nodes. Each processing node performs its assigned task and then forwards the record to each of its child nodes. This process of performing work and then forwarding data to any child nodes continues until every child node has executed its function.

Does this process sound familiar? It should, because you similarly transformed ZMart's business requirements into a graph of processing nodes. Traversing a graph is how Kafka Streams works—it's a DAG or topology of processing nodes.

You start with a source or parent node, which has one or more children. Data always flows from the parent to the child nodes, never from child to parent. Each child node, in turn, can define child nodes of its own, and so on.

Records flow through the graph in a depth-first manner. This approach has significant implications: each record (a key/value pair) is processed *in full* by the entire graph before another record is forwarded through the topology. Because each record is processed depth-first through the whole DAG, there's no need to have backpressure built into Kafka Streams.

NOTE

Definition

There are varying definitions of *backpressure*, but here I define it as the need to restrict the flow of data by buffering or using a blocking mechanism. Backpressure is necessary when a *source* is producing data faster than a *sink* can receive and process that data.

By being able to connect or chain together multiple processors, you can quickly build up complex processing logic, while at the same time keeping each component relatively straightforward. It's in this composition of processors that Kafka Streams' power and complexity come into play.

NOTE **Definition**

A *topology* is the way you arrange the parts of an entire system and connect them with each other. When I say Kafka Streams has a topology, I'm referring to transforming data by running through one or more processors.

SIDEBAR **Kafka Streams and Kafka**

As you might have guessed from the name, Kafka Streams runs on top of Kafka. In this introductory chapter, you don't need to know about Kafka, because we're focusing more how Kafka Streams works conceptually. A few Kafka-specific terms may be mentioned, but for the most part, we'll be concentrating on the stream-processing aspects of Kafka Streams.

If you're new to Kafka or are unfamiliar with it, you'll learn what you need to know about Kafka in chapter 2. Knowledge of Kafka is essential for working effectively with Kafka Streams.

1.6 Applying Kafka Streams to the purchase transaction flow

Let's build a processing graph again, but this time we'll create a Kafka Streams program. To refresh your memory, figure 1.4 shows the requirements graph for ZMart's business requirements. Remember, the vertexes are processing nodes that handle data, and the edges show the flow of data.

Although you'll be building a Kafka Streams program as you build your new graph, you'll still be taking a relatively high-level approach. Some details will be left out. We'll go into more detail later in the book when we look at the actual code.

The Kafka Streams program will consume records, and when it does, you'll convert the raw records into `Purchase` objects. These pieces of information will make up a `Purchase` object:

- The ZMart customer ID (scanned from the member card)
- The total dollar amount spent
- The item(s) purchased
- The ZIP code of the store where the purchase took place
- The date and time of the transaction
- The debit or credit card number

1.6.1 Defining the source

The first step in any Kafka Streams program is to establish a source for the stream. The source could be any of the following:

- A single topic
- Multiple topics in a comma-separated list
- A regex that can match one or more topics

In this case, it will be a single topic named `transactions`. If any of these Kafka terms are unfamiliar to you, remember—they'll be explained in chapter 2.

It's important to note that to Kafka, the Kafka Streams program looks like any other combination of consumers and producers. Any number of applications could be reading from the same topic in conjunction with your streaming program. Figure 1.10 represents the source node in the topology.

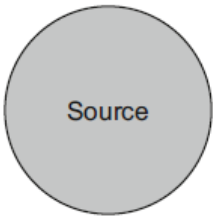


Figure 1.10 The source node: a Kafka topic

1.6.2 The first processor: masking credit card numbers

Now that you have a source defined, you can start creating processors that will work on the data. Your first goal is to mask the credit card numbers recorded in the incoming purchase records. The first processor will convert credit card numbers from something like 1234-5678-9123-2233 to xxxx-xxxx-xxxx-2233.

The `KStream.mapValues` method will perform the masking represented in figure 1.11. It will return a new `KStream` instance with values masked as specified by a `ValueMapper`. This particular `KStream` instance will be the parent processor for any other processors you define.

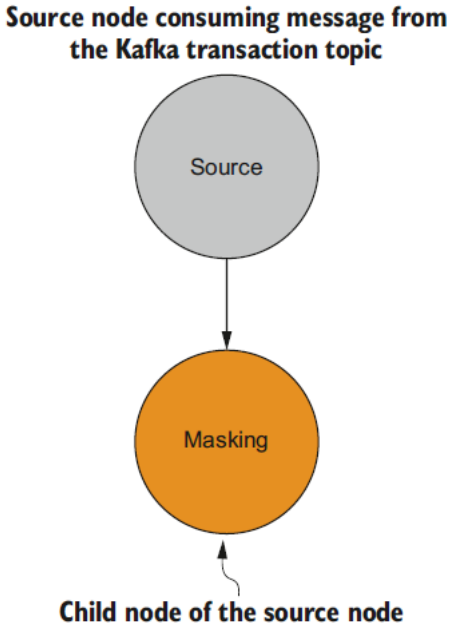


Figure 1.11 The masking processor is a child of the main source node. It receives all the raw sales transactions and emits new records with the credit card number masked.

CREATING PROCESSOR TOPOLOGIES

Each time you create a new `KStream` instance by using a transformation method, you're in essence building a new processor that's connected to the other processors already created. By composing processors, you can use Kafka Streams to create complex data flows elegantly.

It's important to note that calling a method that returns a new `KStream` instance doesn't cause the original instance to stop consuming messages. A transforming method creates a new processor and adds it to the existing processor topology. The updated topology is then used as a parameter to create the next `KStream` instance, which starts receiving messages from the point of its creation.

It's very likely that you'll build new `KStream` instances to perform additional transformations while retaining the original stream for its original purpose. You'll work with an example of this when you define the second and third processors.

It's possible to have a `ValueMapper` convert an incoming value to an entirely new type, but in this case it will return an updated copy of the `Purchase` object. Using a mapper to update an object is a pattern you'll see over and over in `KStreams`.

You should now have a clear image of how you can build up your processor pipeline to transform and output data.

1.6.3 The second processor: purchase patterns

The next processor to create is one that can capture information necessary for determining purchase patterns in different regions of the country (figure 1.12). To do this, you'll add a child-processing node to the first processor (`KStream`) you created. The first processor produces `Purchase` objects with the credit card number masked.

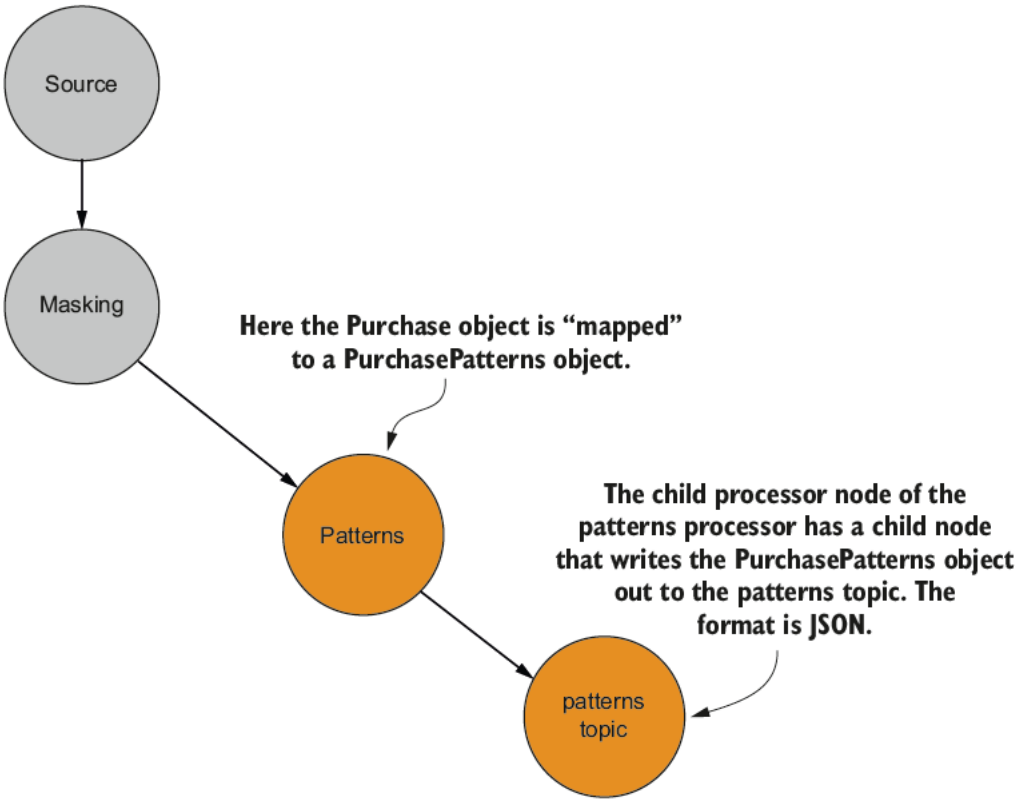


Figure 1.12 The purchase-pattern processor takes Purchase objects and converts them into PurchasePattern objects PurchasePattern objectcontaining the items purchased and the ZIP code where the transaction took place. A new processor takes records from the patterns processor and writes them out to a Kafka topic.

The purchase-patterns processor receives a `Purchase` object from its parent node and maps the object to a new `PurchasePattern` object. The mapping process extracts the item purchased (toothpaste, for example) and the ZIP code it was bought in and uses that information to create the `PurchasePattern` object. We'll go over exactly how this mapping process occurs in chapter 3.

Next, the purchase-patterns processor adds a child processor node that receives the new `PurchasePattern` object and writes it out to a Kafka topic named `patterns`. The `PurchasePattern` object is converted to some form of transferable data when it's written to the topic. Other applications can then consume this information and use it to determine inventory levels as well as purchasing trends in a given area.

1.6.4 The third processor: customer rewards

The third processor will extract information for the customer rewards program (figure 1.13). This processor is also a child node of the original processor. It receives the `Purchase` objects and maps them to another type: the `RewardAccumulator` object.

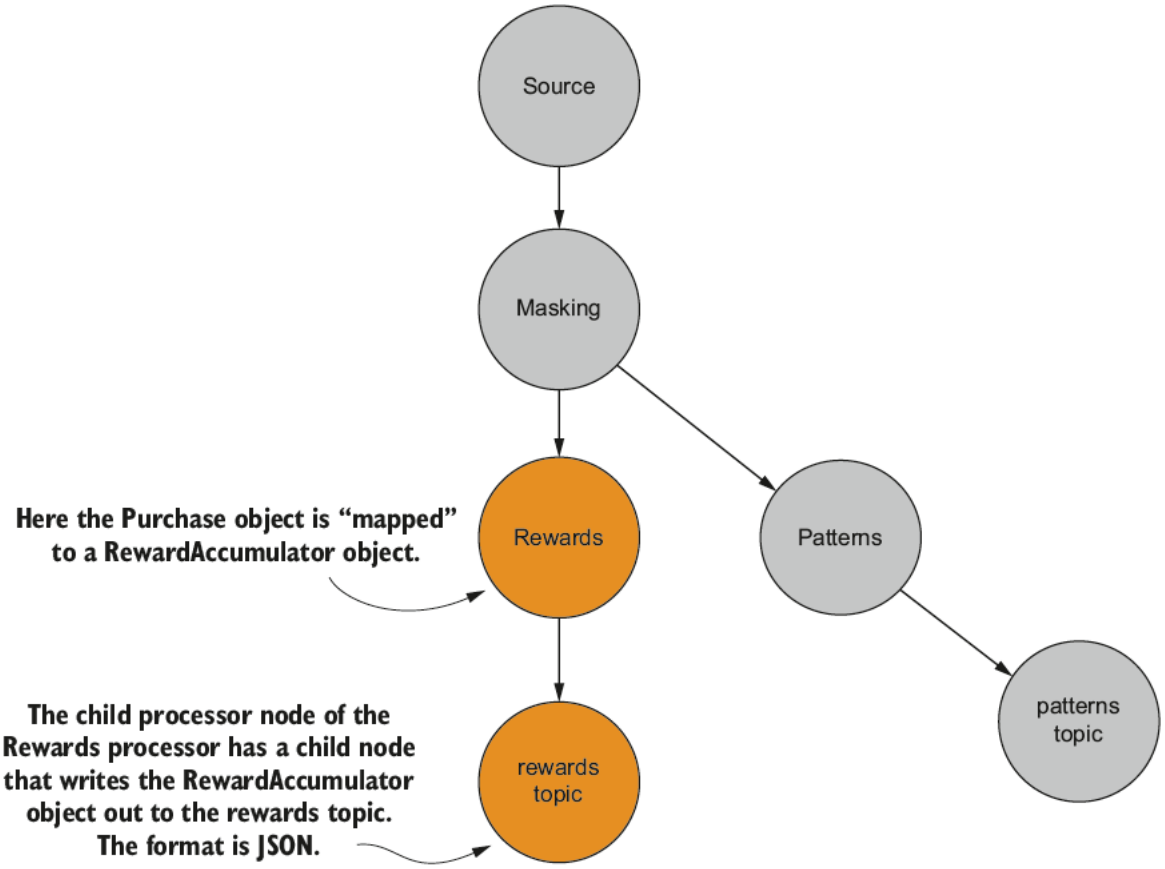


Figure 1.13 The customer rewards processor is responsible for transforming Purchase objects into a RewardAccumulator object containing the customer ID, date, and dollar amount of the transaction. A child processor writes the Rewards objects to another Kafka topic.

The customer rewards processor also adds a child-processing node to write the `RewardAccumulator` object out to a Kafka topic, `rewards`. By consuming records from the `rewards` topic, other applications can determine rewards for ZMart customers and produce, for example, the email that Jane Doe received.

1.6.5 The fourth processor—writing purchase records

The last processor is shown in figure 1.14. This is the third child node of the masking processor node, and it writes the entire masked purchase record out to a topic called `purchases`. This topic will be used to feed a NoSQL storage application that will consume the records as they come in. These records will be used for later analysis.

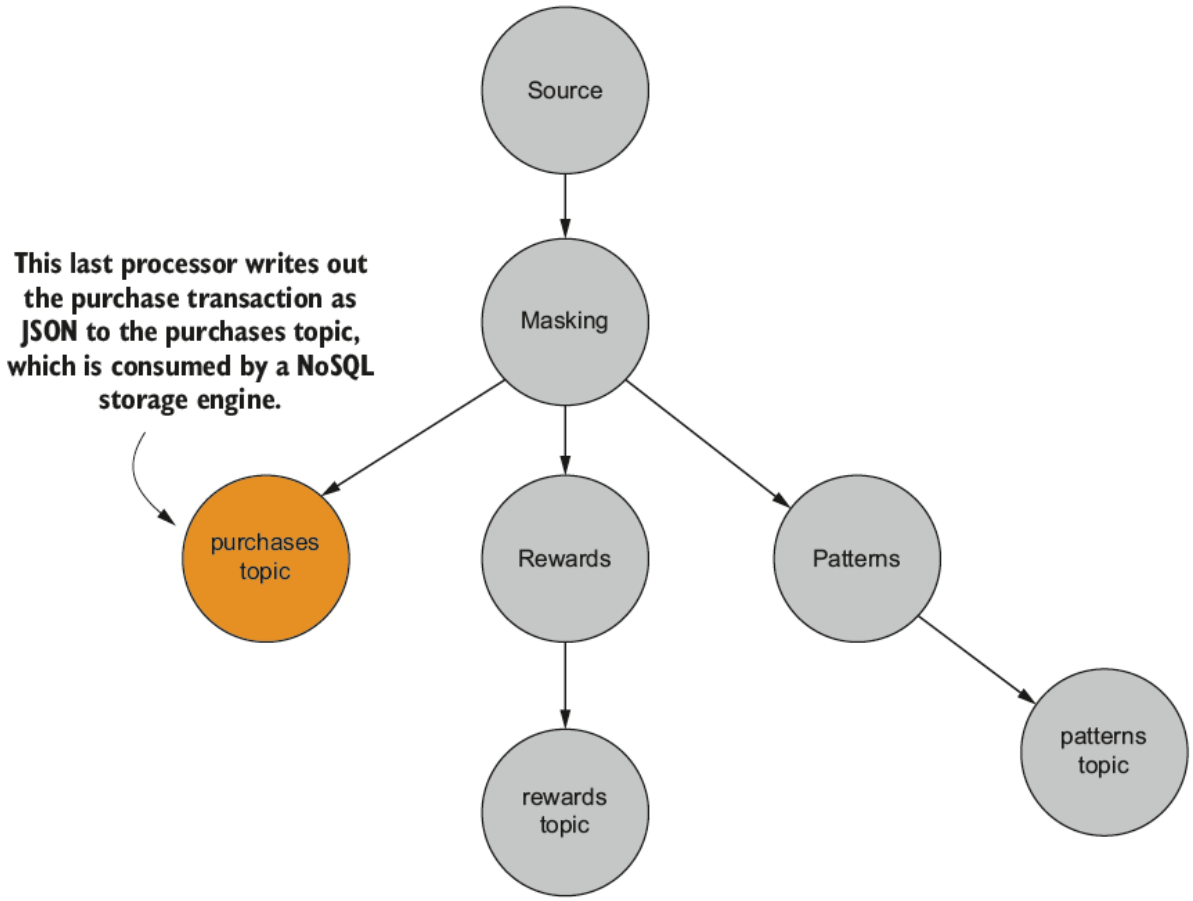


Figure 1.14 The final processor is responsible for writing out the entire Purchase object to another Kafka topic. The consumer for this topic will store the results in a NoSQL store such as MongoDB.

As you can see, the first processor, which masks the credit card number, feeds three other processors: two that further refine or transform the data, and one that writes the masked results to a topic for further use by other consumers. By using Kafka Streams, you can build up a powerful processing graph of connected nodes to perform stream processing on your incoming data.

1.7 Summary

- Kafka Streams is a graph of processing nodes that combine to provide powerful and complex stream processing.
- Batch processing is powerful, but it's not enough to satisfy real-time needs for working with data.
- Distributing data, key/value pairs, partitioning, and data replication are critical for distributed applications.

To understand Kafka Streams, you should know some Kafka. For those who don't know Kafka, we'll cover the essentials in chapter 2:

- Installing Kafka and sending a message
- Exploring Kafka's architecture and what a distributed log is
- Understanding topics and how they're used in Kafka
- Understanding how producers and consumers work and how to write them effectively

If you're already comfortable with Kafka, feel free to go straight to chapter 3, where we'll build a Kafka Streams application based on the example discussed in this chapter.

2 Kafka quickly

This chapter covers

- Examining the Kafka architecture
- Sending messages with producers
- Reading messages with consumers
- Installing and running Kafka

Although this is a book about Kafka Streams, it's impossible to explore Kafka Streams without discussing Kafka. After all, Kafka Streams is a library that runs on Kafka.

Kafka Streams is designed very well, so it's possible to get up and running with little or no Kafka experience, but your progress and ability to fine-tune Kafka will be limited. Having a good fundamental knowledge of Kafka is essential to get the most out of Kafka Streams.

NOTE This chapter is for developers who are interested in getting started with Kafka Streams but have little or no experience with Kafka itself. If you have a good working knowledge of Kafka, feel free to skip this chapter and proceed directly to chapter 3.

Kafka is too large a topic to cover in its entirety in one chapter. I'll cover enough to give you a good understanding how Kafka works and a few of the core configuration settings you'll need to know. For in-depth coverage of Kafka, take a look at *Kafka in Action* by Dylan Scott (Manning, 2018).

2.1 The data problem

Organizations today are swimming in data. Internet companies, financial businesses, and large retailers are better positioned now than ever to use this data, both to serve their customers better and to find more efficient ways of conducting business. (We're going to take a positive outlook on this situation and assume only good intentions when looking at customer data.)

Let's consider the various requirements you'd like to have in the ZMart data-management solution:

- You need a way to send data to a central storage quickly.
- Because machines frequently fail, you also need the ability to have your data replicated, so those inevitable failures don't cause downtime and data loss.
- You need the potential to scale to any number of consumers of data without having to keep track of different applications. You need to make the data available to anyone in an organization, but not have to keep track of who has and has not viewed the data.

2.2 Using Kafka to handle data

In chapter 1, you were introduced to the large retail company ZMart. At that point, ZMart wanted a streaming platform to use the company's sales data in order to offer better customer service and improve sales overall. But six months before that, ZMart was looking to get a handle on its data situation. ZMart had a custom solution that initially worked well but had become unmanageable for reasons you'll soon see.

2.2.1 ZMart's original data platform

Originally, ZMart was a small company that had retail sales data flowing into its system from separate applications. This worked fine initially, but over time it became evident that a new approach would be needed. Data from sales in one department is not of interest only to that department. Several parts of the company are interested, and each part has a different take on what's important and how they want the data structured. Figure 2.1 shows ZMart's original data platform.

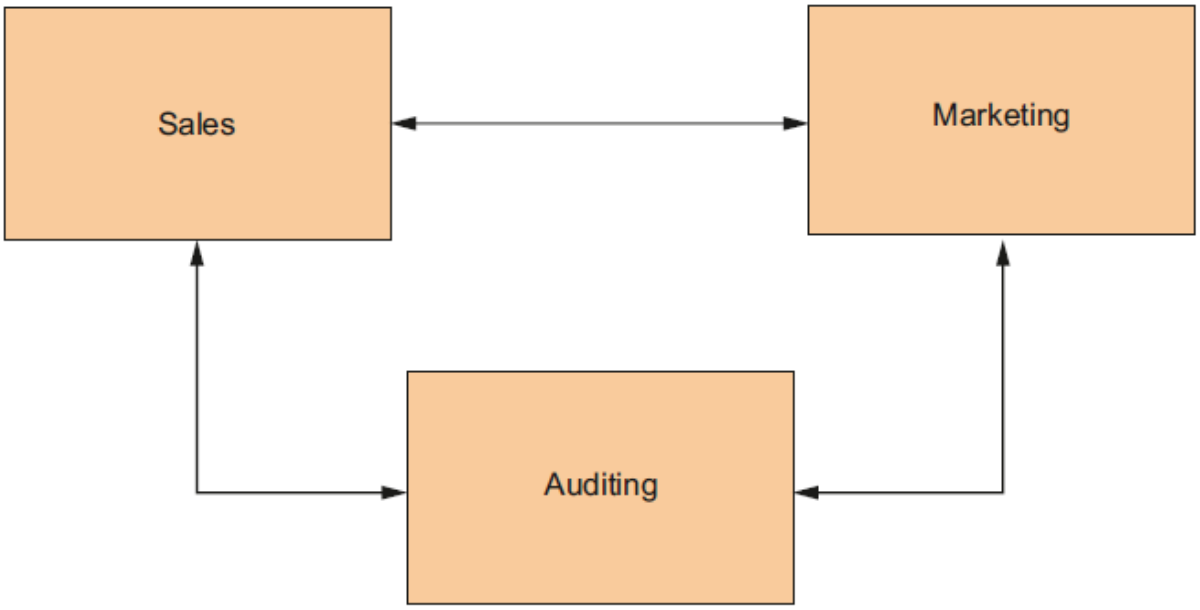


Figure 2.1 The original data architecture for ZMart was simple enough to have information flowing to and from each source of information.

Over time, ZMart continued to grow by acquiring other companies and expanding its offerings in existing stores. With each addition, the connections between applications become more complicated. What started out as a handful of applications communicating with each other turned into a veritable pile of spaghetti. As you can see in figure 2.2, even with just three applications, the number of connections is cumbersome and confusing. You can see how adding new applications will make this data architecture unmanageable over time.

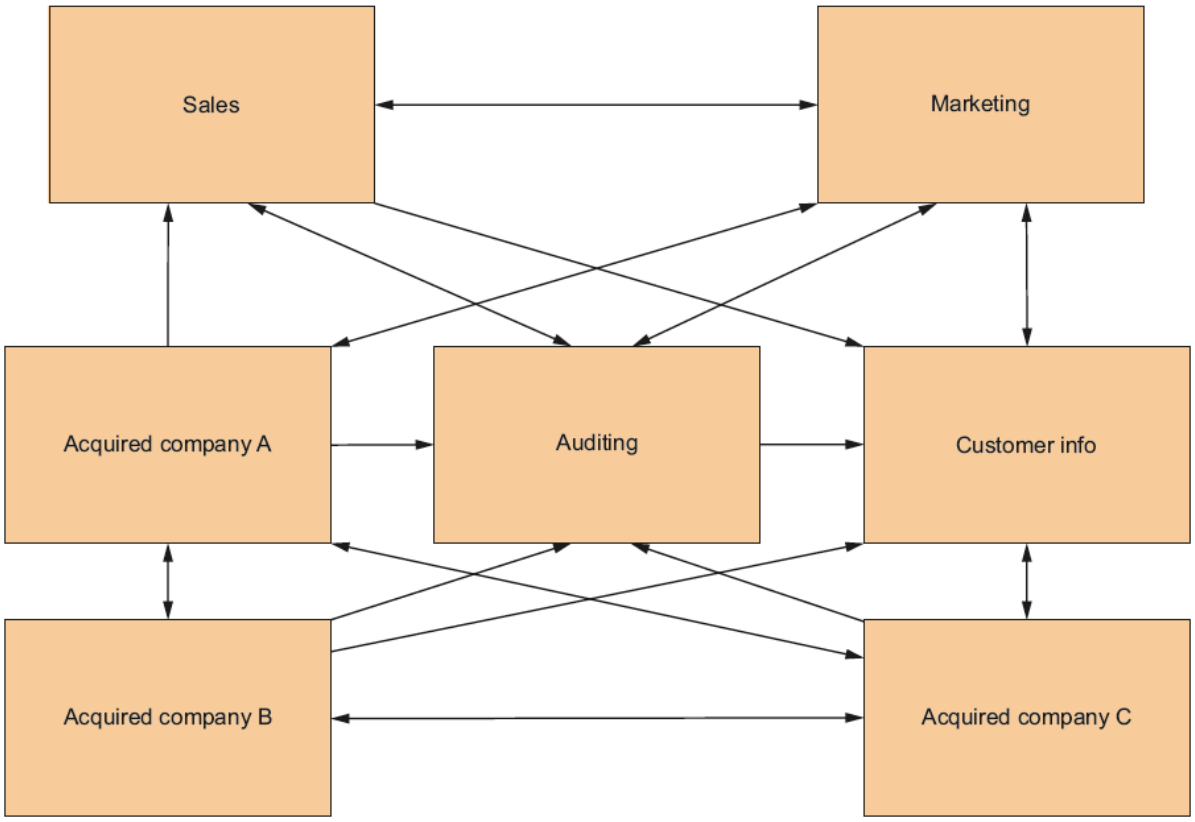


Figure 2.2 With more applications being added over time, connecting all these information sources has become very complex.

2.2.2 A Kafka sales transaction data hub

A solution to ZMart's problem is to create one intake process to hold all transaction data—a transaction data hub. This transaction data hub should be stateless, accepting transaction data and storing it in such a fashion that any consuming application can pull the information it needs. It will be up to the consuming application to keep track of what it's seen. The transaction data hub will only know how long it's been holding any transaction data, and when that data should be rolled off or deleted.

In case you haven't guessed it yet, we have the perfect use case here for Kafka. Kafka is a fault-tolerant, robust publish/subscribe system. A single Kafka node is called a *broker*, and multiple Kafka servers make up a *cluster*. Kafka stores messages written by *producers* in *topics*. *Consumers* subscribe to topics and contact Kafka to see if messages are available in those subscribed topics. Figure 2.3 shows how you can envision Kafka as the sales transaction data hub.

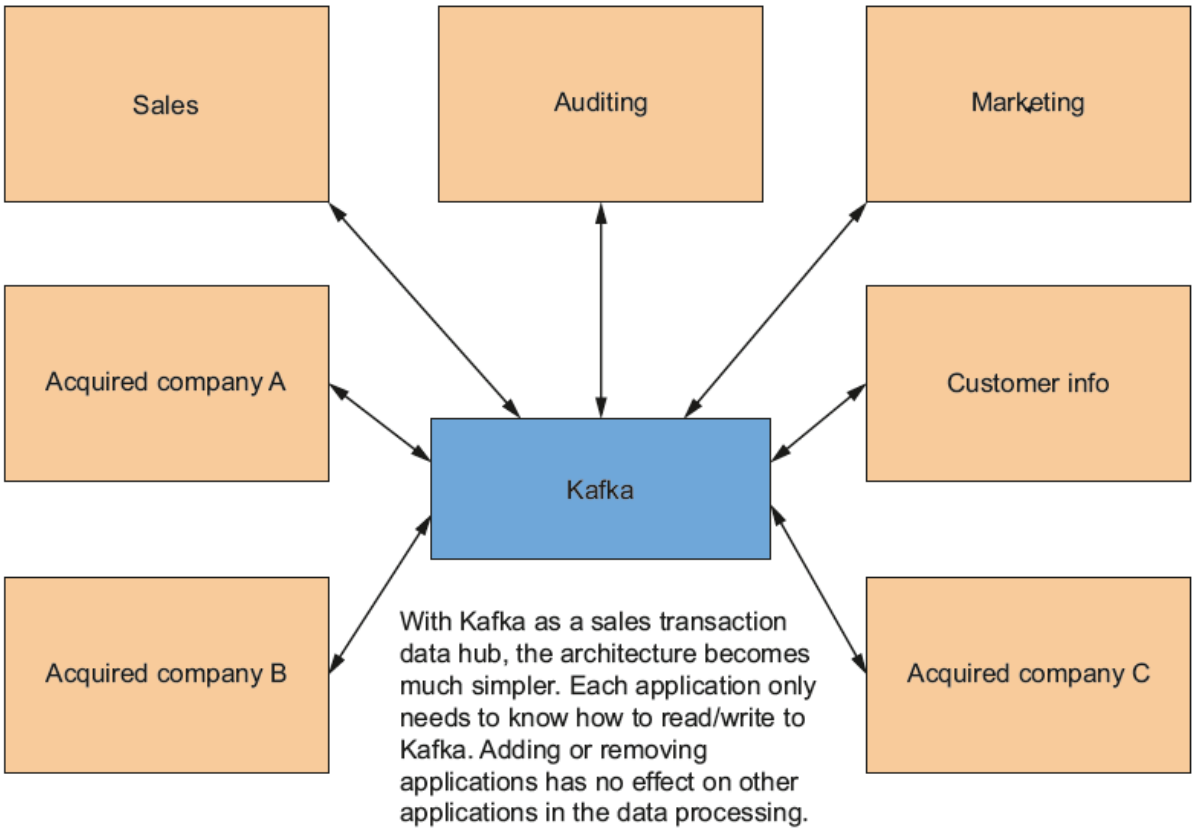


Figure 2.3 Using Kafka as a sales transaction hub simplifies the ZMart data architecture significantly. Now each machine doesn't need to know about every other source of information. All they need to know is how to read from and write to Kafka.

You've seen an overview of Kafka from 50,000 feet. We'll take a closer look in the following sections.

2.3 Kafka architecture

In the next several subsections, we'll look at the key parts of Kafka's architecture and at how Kafka works. If you're interested in kicking the tires on Kafka sooner rather than later, skip ahead to section 2.37, on installing and running Kafka. After you've got it installed, come back here to continue learning about Kafka.

2.3.1 Kafka is a message broker

Earlier, I stated that Kafka is a publish/subscribe system, but it would be more precise to say that Kafka acts as a message broker. A *broker* is an intermediary that brings together two parties that don't necessarily know each other for a mutually beneficial exchange or deal. Figure 2.4 shows the evolution of the ZMart data infrastructure. The producers and consumers have been added to show how the individual parts communicate with Kafka. They don't communicate directly with each other.

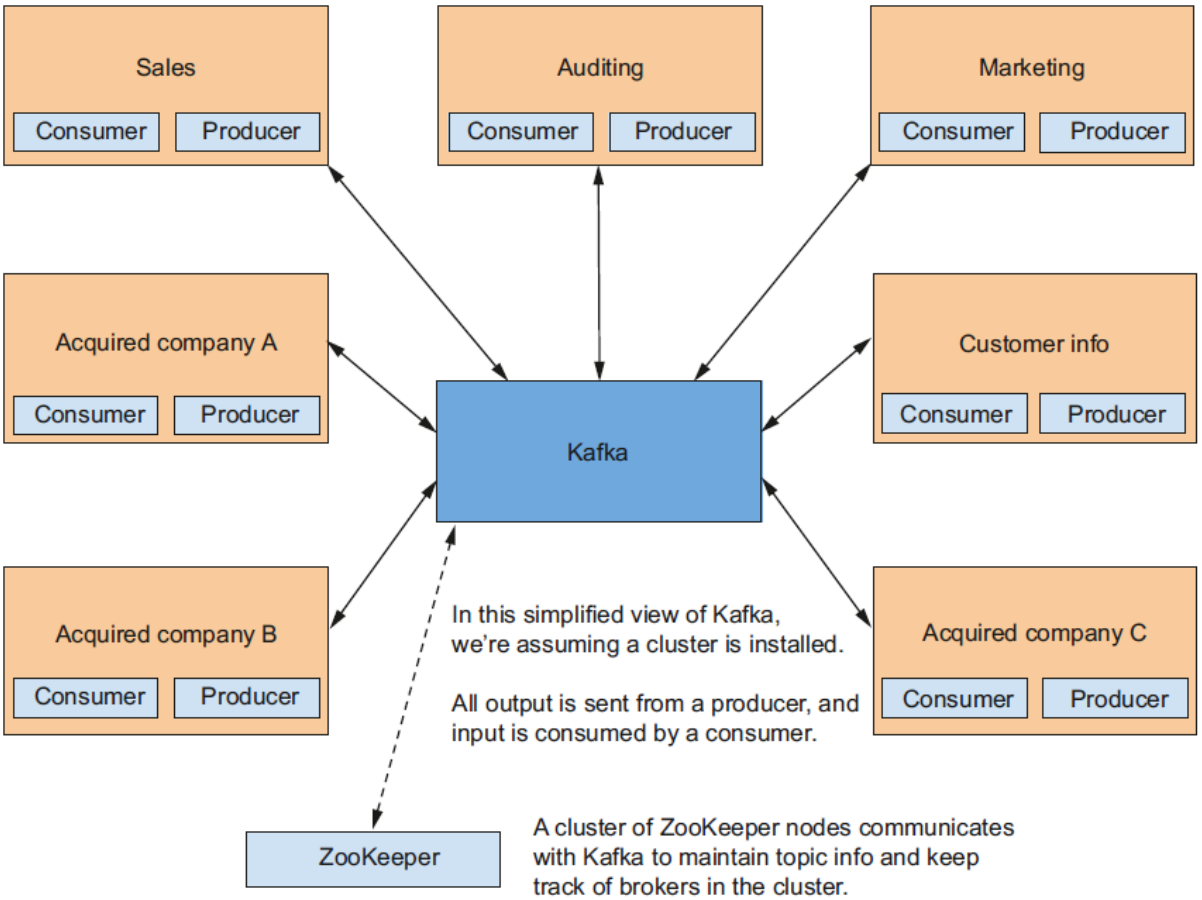


Figure 2.4 Kafka is a message broker. Producers send messages to Kafka, and those messages are stored and made available to consumers via subscriptions to topics.

Kafka stores messages in topics and retrieves messages from topics. There's no direct connection between the producers and the consumers of the messages. Additionally, Kafka doesn't keep any state regarding the producers or consumers. It acts solely as a message clearinghouse.

The underlying technology of a Kafka topic is a *log*, which is a file that Kafka appends incoming records to. To help manage the load of messages coming into a topic, Kafka uses partitions. We discussed partitions in chapter 1, and you may recall that one use of partitions is to bring data located on different machines together on the same server. We'll discuss partitions in detail shortly.

2.3.2 Kafka is a log

The mechanism underlying Kafka is the *log*. Most software engineers are familiar with logs that track what an application's doing. If you're having performance issues or errors in your application, the first place to check is the application logs. But that's a different sort of log. In the context of Kafka (or any other distributed system), a log is "an append-only, totally ordered sequence of records ordered by time."[1]

Footnote 1 Jay Kreps, "The Log: What Every Software Engineer Should Know About Real-time Data's Unifying Abstraction," http://mng.bz/eE3w.

Figure 2.5 shows what a log looks like. An application appends records to the end of the log as they arrive. Records have an implied ordering by time, even though there might not be a timestamp associated with each record, because the earliest records are to the left and the last record to arrive is at the right end.

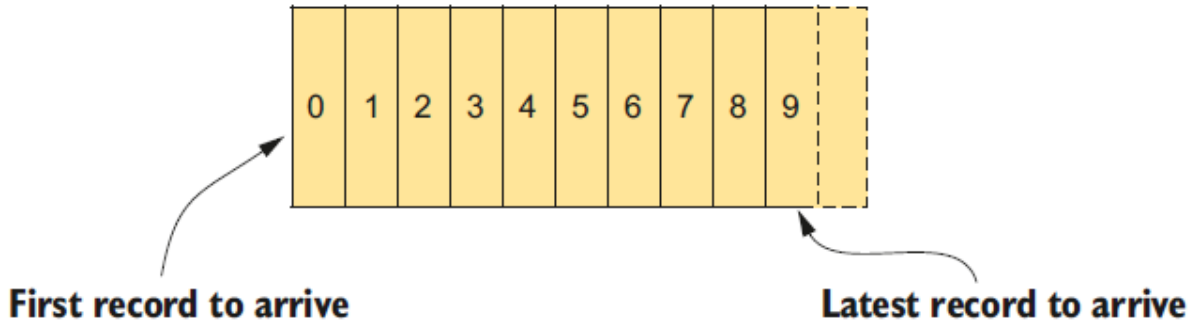


Figure 2.5 A log is a file where incoming records are appended—each newly arrived record is placed immediately after the last record received. This process orders the records in the file by time.

Logs are a simple data abstraction with powerful implications. If you have records in order with respect to time, resolving conflicts or determining which update to apply to different machines becomes straightforward: the latest record wins.

Topics in Kafka are logs that are segregated by topic name. You could almost think of topics as labeled logs.

If the log is replicated among a cluster of machines, and a single machine goes down, it's easy to bring that server back up: just replay the log file. The ability to recover from failure is precisely the role of a distributed commit log.

We've only scratched the surface of a very deep topic when it comes to distributed applications and data consistency, but what you've seen so far should give you a basic understanding of what's going on under the covers with Kafka.

2.3.3 How logs work in Kafka

When you install Kafka, one of the configuration settings is `log.dir`, which specifies where Kafka stores log data. Each topic maps to a subdirectory under the specified log directory. There will be as many subdirectories as there are topic partitions, with a format of *partition-name_partition-number* (I'll cover partitions in the next section). Inside each directory is the log file where incoming messages are appended. Once the log files reach a certain size (either a number of records or size on disk), or when a configured time difference between message timestamps is reached, the log file is "rolled," and Kafka appends incoming messages to a new log (see figure 2.6).

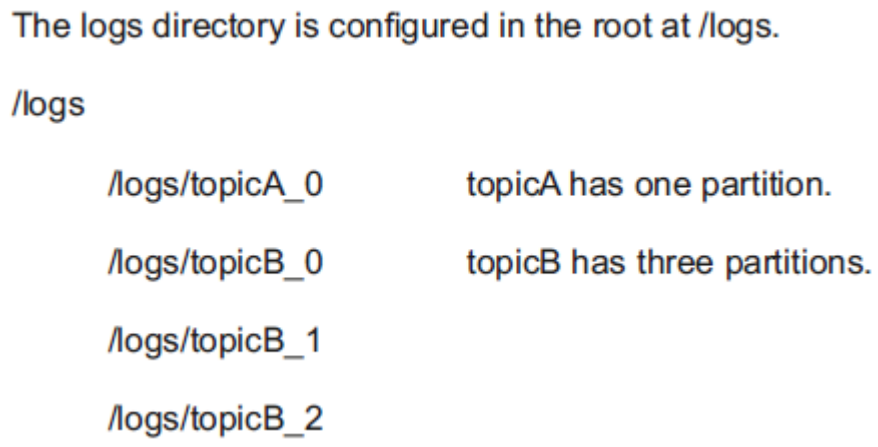


Figure 2.6 The logs directory is the base storage for messages. Each directory under /logs represents a topic partition. Filenames within the directory start with the name of the topic, followed by an underscore, which is followed by a partition number.

You can see that logs and topics are highly connected concepts. You could say that a topic is a log, or that it represents a log. The topic name gives you a good handle on which log the messages sent to Kafka via producers will be stored in. Now that we've covered the concept of logs, let's discuss another fundamental concept in Kafka: partitions.

2.3.4 Kafka and partitions

Partitions are a critical part of Kafka's design. They're essential for performance, and they guarantee that data with the same keys will be sent to the same consumer and in order. Figure 2.7 shows how partitions work.

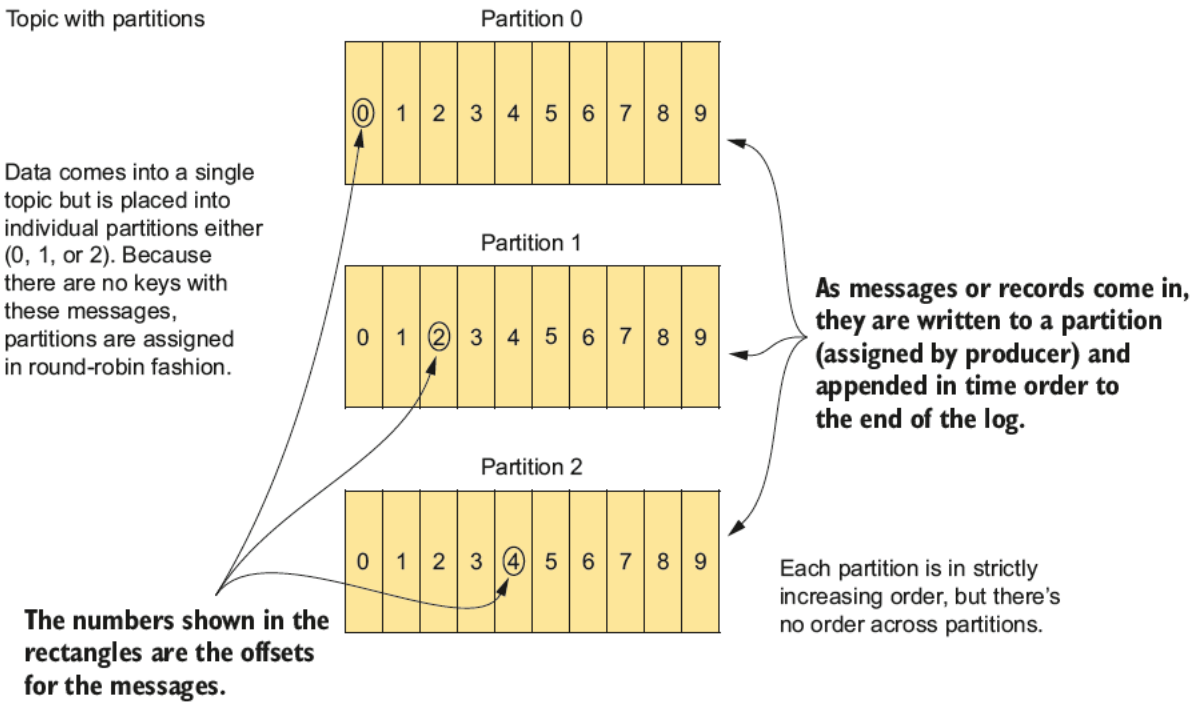


Figure 2.7 Kafka uses partitions to achieve high throughput and spread the messages for an individual topic across several machines in the cluster.

Partitioning a topic essentially splits the data forwarded to a topic across parallel streams, and it's key to how Kafka achieves its tremendous throughput. I explained that a topic is a distributed log; each partition is similarly a log unto itself and follows the same rules. Kafka appends each incoming message to the end of the log, and all messages are strictly time-ordered. Each message has an offset number assigned to it. The order of messages across partitions isn't guaranteed, but the order of messages within each partition is guaranteed.

Partitioning serves another purpose, aside from increasing throughput. It allows topic messages to be spread across several machines so that the capacity of a given topic isn't limited to the available disk space on one server.

Now let's look at another critical role partitions play: ensuring messages with the same keys end up together.

2.3.5 Partitions group data by key

Kafka works with data in key/value pairs. If the keys are null, the Kafka producer will write records to partitions chosen in a round-robin fashion. Figure 2.8 shows how partition assignment operates with non-null keys.

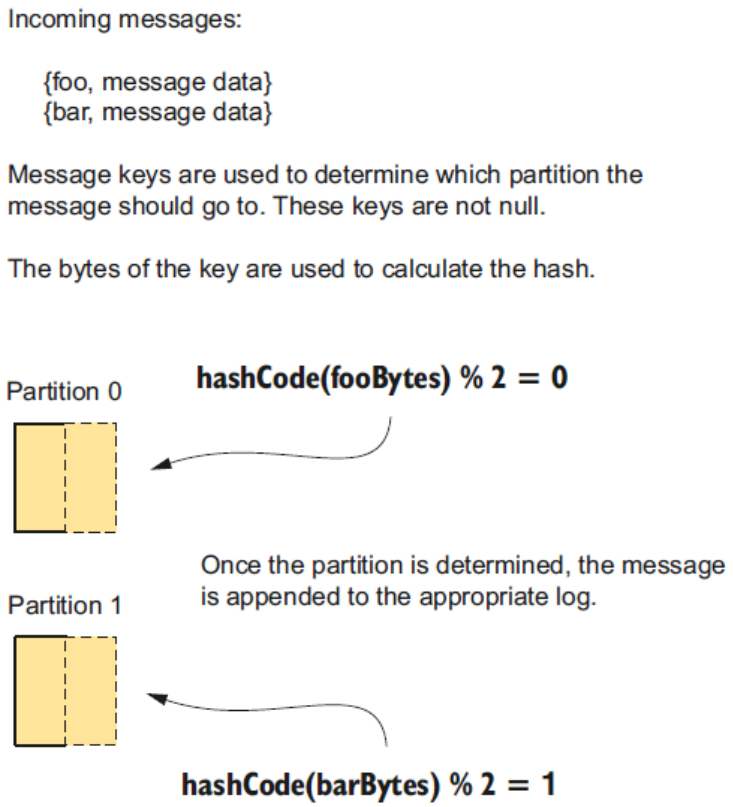


Figure 2.8 "foo" is sent to partition 0, and "bar" is sent to partition 1. You obtain the partition by hashing the bytes of the key, modulus the number of partitions.

If the keys aren't null, Kafka uses the following formula (shown in pseudocode) to determine which partition to send the key/value pair to:

```
HashCode.(key) % number of partitions
```

By using a deterministic approach to select a partition, records with the same key will *always* be sent to the same partition and in order. The default partitioner uses this approach; if you need a different strategy for selecting partitions, you can provide a custom partitioner.

2.3.6 Writing a custom partitioner

Why would you want to write a custom partitioner? Of the several possible reasons, we'll look at one simple case here—the use of composite keys.

Suppose you have purchase data flowing into Kafka, and the keys contain two values: a customer ID and a transaction date. But you need to group values by customer ID, so taking a hash of the customer ID and the purchase date won't work. In this case, you'll need to write a custom partitioner that knows which part of the composite key determines

which partition to use. For example, the composite key found in src/main/java/bbejeck/model/PurchaseKey.java (source code can be found on the book's website here: https://manning.com/books/kafka-streams-in-action) is shown in the following listing.

Listing 2.1 `PurchaseKey` composite key

```
public class PurchaseKey {

    private String customerId;
    private Date transactionDate;

    public PurchaseKey(String customerId, Date transactionDate) {
        this.customerId = customerId;
        this.transactionDate = transactionDate;
    }

    public String getCustomerId() {
        return customerId;
    }

    public Date getTransactionDate() {
        return transactionDate;
    }
}
```

When it comes to partitioning, you need to ensure that all transactions for a particular customer go to the same partition, but using the key in its entirety won't enable this to happen. Because purchases happen on many dates, including the date will result in different key values for a single customer, placing the transactions across random partitions. You need to ensure you send all transactions with the same customer ID to the same partition. The only way to do that is to only use the customer ID when determining the partition.

The following example custom partitioner does what's required. `PurchaseKeyPartitioner` (from src/main/java/bbejeck/chapter_2/partitioner/PurchaseKeyPartitioner.java) extracts the customer ID from the key to determine which partition to use.

Listing 2.2 `PurchaseKeyPartitioner` custom partitioner

```
public class PurchaseKeyPartitioner extends DefaultPartitioner {

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

        if (key != null) {                                    (1)
            PurchaseKey purchaseKey = (PurchaseKey) key;
            key = purchaseKey.getCustomerId();
```

```
            keyBytes = ((String) key).getBytes();          ❷
        }
        return super.partition(topic, key, keyBytes, value, valueBytes, cluster);   ❸
    }
}
```

❶ If the key isn't null, extracts the customer ID

❷ Sets the key bytes to the new value

❸ Returns the partition with the updated key, delegating to the superclass

This custom partitioner extends `DefaultPartitioner`. You could implement the `Partitioner` interface directly, but there's existing logic in `DefaultPartitioner` that we're using in this case.

Keep in mind that when creating a custom partitioner, you aren't limited to using only the key. Using the value alone, or the value in combination with the key, is valid as well.

NOTE The Kafka API provides a `Partitioner` interface that you can use to write a custom partitioner. We won't be covering writing a partitioner from scratch, but the principles are the same as those in the listing 2.2.

You've just seen how to construct a custom partitioner. Next, let's wire up the partitioner with Kafka.

2.3.7 Specifying a custom partitioner

Now that you've written a custom partitioner, you need to tell Kafka you want to use it instead of the default partitioner. Although we haven't covered producers yet, you specify a different partitioner when configuring the Kafka producer:

```
partitioner.class=bbejeck_2.partitioner.PurchaseKeyPartitioner
```

By setting a partitioner per producer instance, you're free to use any partitioner class for any producer. We'll go over producer configuration in detail when we cover using Kafka producers.

WARNING You must exercise some caution when choosing the keys you use and when selecting parts of a key/value pair to partition on. Make sure the key you choose has a fair distribution across all of your data. Otherwise, you'll end up with a data-skew problem, because most of your data will be located on just a few of your partitions.

2.3.8 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 key 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 ultimately the slower processing will hinder performance.[2].

Footnote 2 Jun Rao, "How to Choose the Number of Topics/Partitions in a Kafka Cluster?" http://mng.bz/4C03.

2.3.9 The distributed log

We've discussed the concepts of logs and partitioned topics. Let's take a minute to look at those two concepts together to demonstrate distributed logs.

So far, we've focused on logs and topics on one Kafka server or broker, but typically a Kafka production cluster environment includes several machines. I've intentionally kept the discussion centered on a single node, as it's easier to understand the concepts when considering one node. But in practice, you'll always be working with a cluster of machines in Kafka.

When a topic is partitioned, Kafka doesn't allocate those partitions on one machine—Kafka spreads them across several machines in the cluster. As Kafka appends records to a log, Kafka is distributing those records across several machines by partition. In figure 2.9, you can see this process in action.

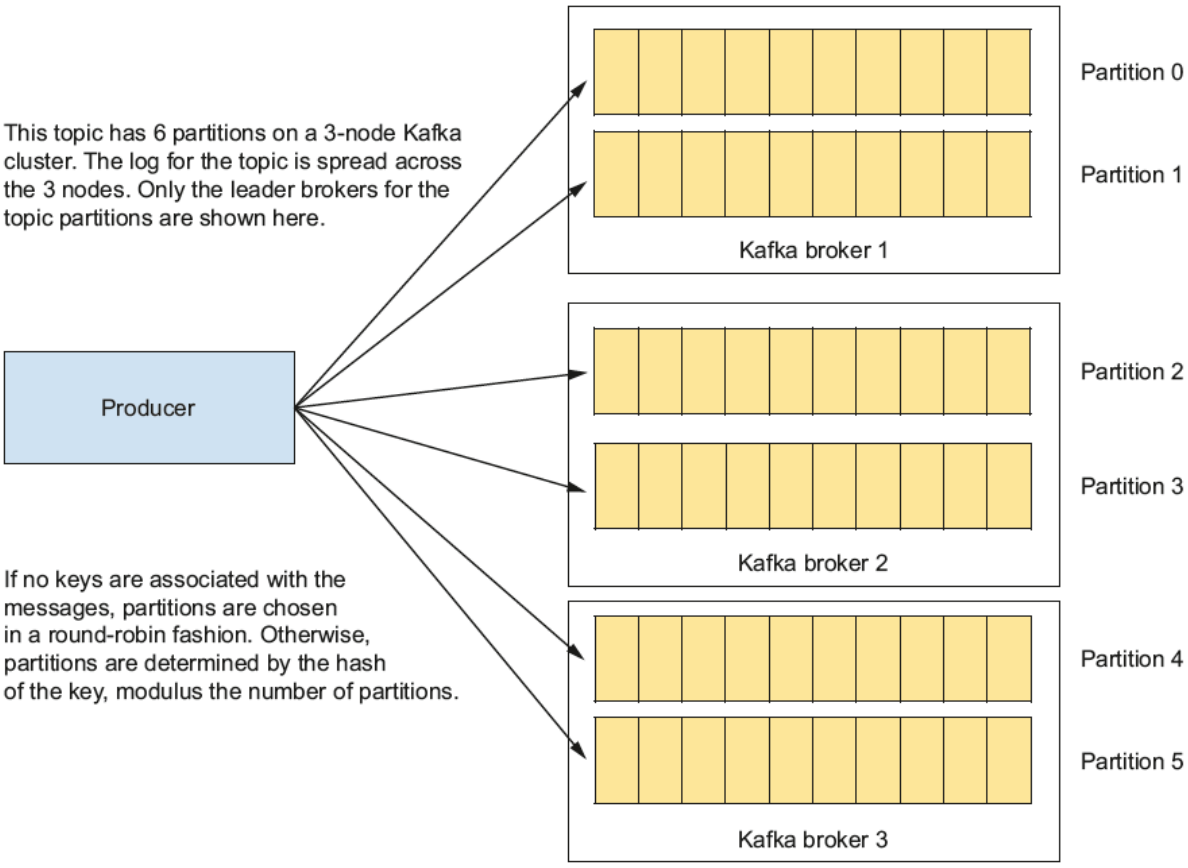


Figure 2.9 A producer writes messages to partitions of a topic. If no key is associated with the message, the producer chooses a partition in a round-robin fashion. Otherwise, the hash of the key, modulus the number of partitions is used.

Let's walk through a quick example using figure 2.9 as a guide. For this example, we'll assume one topic and null keys, so the producer assigns partitions in a round-robin manner.

The producer sends its first message to partition 0 on Kafka broker 1, the second message to partition 1 on Kafka broker 1, and the third message to partition 2 on Kafka broker 2. When the producer sends its sixth message, it goes to partition 5 on Kafka broker 3, and the next message starts over, going to partition 0 on Kafka broker 1. Message distribution continues in this manner, spreading message traffic across all nodes in the Kafka cluster.

Although storing data remotely may sound risky, because a server can go down, Kafka offers data redundancy. Data is replicated to one or more machines in the cluster as you write to one broker in Kafka (we'll cover replication in an upcoming section).

2.3.10 ZooKeeper: leaders, followers, and replication

So far, we've discussed the role topics play in Kafka, and how and why topics are partitioned. 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.

Kafka has the notion of leader and follower brokers. In Kafka, for each topic partition, one broker is chosen as the *leader* for the other brokers (the *followers*). One of the chief duties of the leader is to assign *replication* of topic partitions to the follower brokers. Just as Kafka allocates partitions for a topic across the cluster, Kafka also replicates the partitions across machines. Before we go into the details of how leaders, followers, and replication work, we need to discuss the technology Kafka uses to achieve this.

2.3.11 Apache ZooKeeper

If you're a complete Kafka newbie, you may be asking yourself, "Why are we talking about Apache ZooKeeper in a Kafka book?" Apache ZooKeeper is integral to Kafka's architecture, and it's ZooKeeper that enables Kafka to have leader brokers and to do such things as track the replication of topics (https://zookeeper.apache.org):

> ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.

Given that Kafka is a distributed application, it should start to be clear how ZooKeeper is involved in Kafka's architecture. For this discussion, we'll only consider Kafka installations where there are two or more Kafka servers installed.

In a Kafka cluster, one of the brokers is "elected" as the *controller*. We covered partitions in the previous section and discussed how Kafka spreads partitions across different machines in the cluster. Topic partitions have a leader and follower(s) (the level of replication determines the degree of replication). When producing messages, Kafka sends the record to the broker that is the leader for the record's partition.

2.3.12 Electing a controller

Kafka uses ZooKeeper to elect the controller broker. Discussing the consensus algorithms involved is way beyond the scope of this book, so we'll take the 50,000-foot view and just state that ZooKeeper elects a broker from the cluster to be the controller.

If the controlling broker fails or becomes unavailable for any reason, ZooKeeper elects a new controller from a set of brokers that are considered to be caught up with the leader (an in-sync replica [ISR]). The brokers that make up this set are dynamic, and ZooKeeper recognizes only brokers in this set for election as leader.[3]

Footnote 3 Kafka documentation, "Replicated Logs: Quorums, ISRs, and State Machines (Oh my!)," http://kafka.apache.org/documentation/#design_replicatedlog.

2.3.13 Replication

Kafka replicates records among brokers to ensure data availability, should a broker in the cluster fail. You can set the level of replication for each topic (as you saw in our previous example of publishing and consuming) or for all topics in the cluster. Figure 2.10 demonstrates the replication flow between brokers.

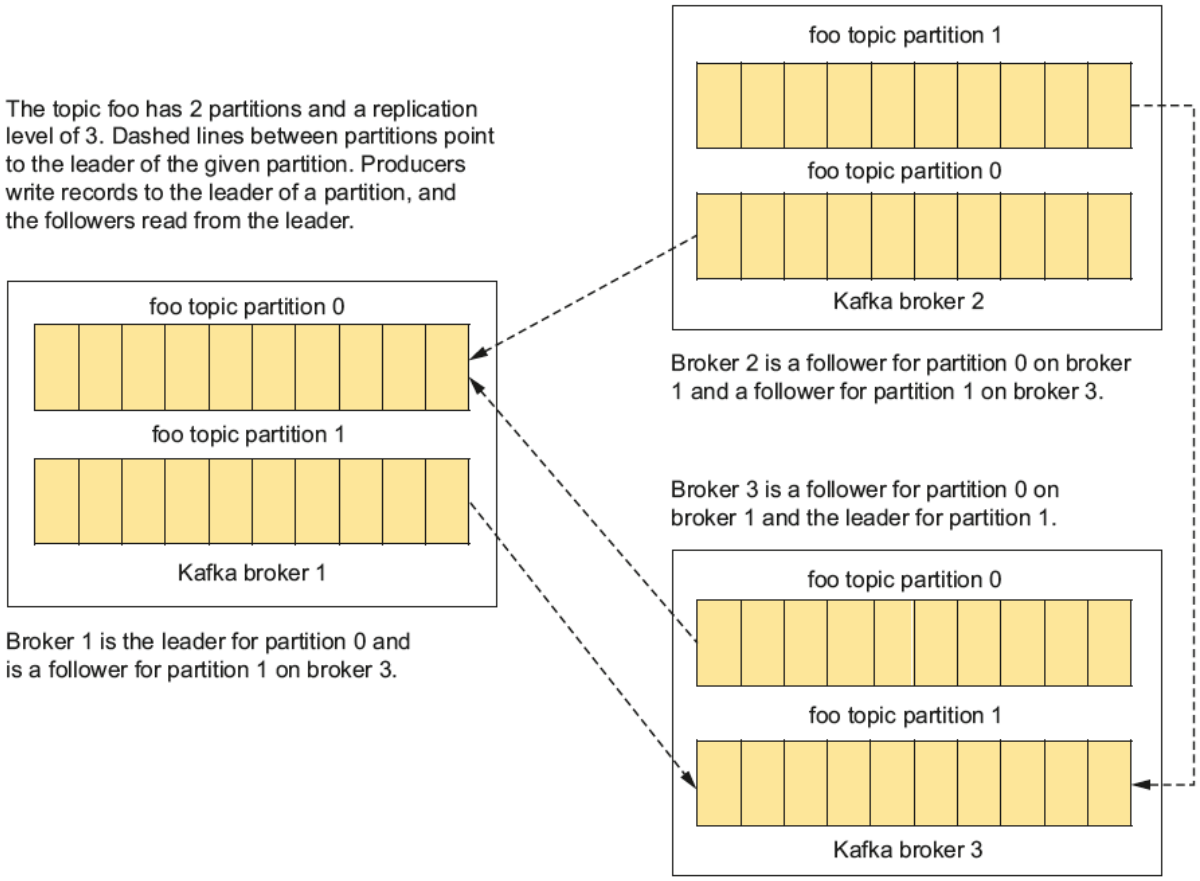


Figure 2.10 Brokers 1 and 3 are leaders for one topic partition and followers for another, whereas broker 2 is a follower only. Follower brokers copy data from the leader broker.

The Kafka replication process is straightforward. Brokers that follow a topic partition consume messages from the topic-partition leader and append those records to their log.

As discussed in the previous section, follower brokers that are caught up with their leader broker are considered to be ISRs. ISR brokers are eligible to be elected leader, should the current leader fail or become unavailable.[4].

Footnote 4 Kafka documentation, “Replication,” http://kafka.apache.org/documentation/#replication.

2.3.14 Controller responsibilities

The controller broker is responsible for setting up leader/follower relationships for all partitions of a topic. If a Kafka node dies or is unresponsive (to ZooKeeper heartbeats), all of its assigned partitions (both leader and follower) are reassigned by the controller broker. Figure 2.11 illustrates a controller broker in action.[5]

Footnote 5 Some of the information in this section came from answers given by Gwen Shapira, “What is the actual role of Zookeeper in Kafka? What benefits will I miss out on if I don’t use Zookeeper and Kafka together?” on Quora at http://mng.bz/25Sy.

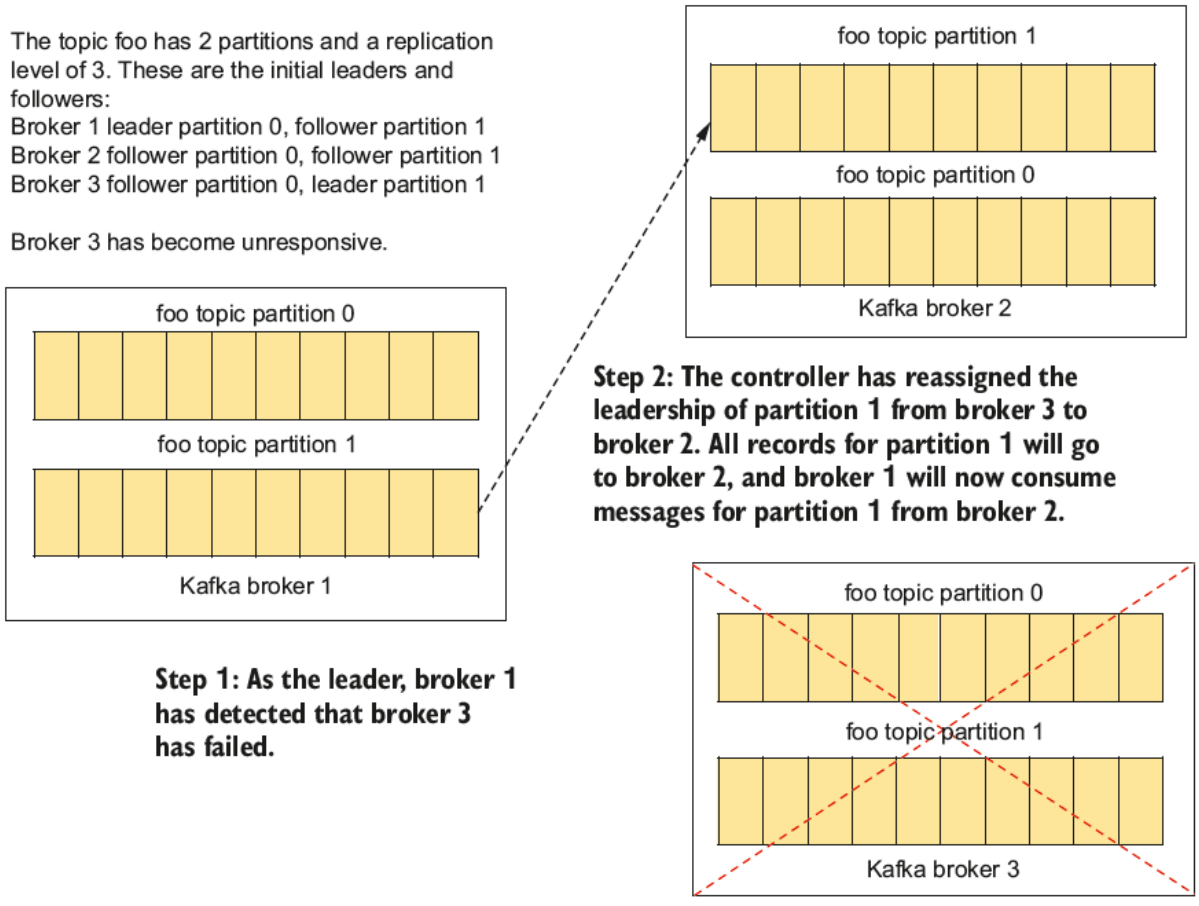


Figure 2.11 The controller broker is responsible for assigning other brokers to be the leader broker for some topics/partitions and followers for other topics/partitions. When a broker becomes unavailable, the controller broker will reassign the failed broker’s assignments to other brokers in the cluster.

The figure shows a simple failure scenario. In step 1, the controller broker detects that

broker 3 isn't available. In step 2, the controller broker reassigns the leadership of the partition on broker 3 to broker 2.

ZooKeeper is also 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 who can read from and write to particular topics.

You've now seen why Kafka has a dependency on Apache ZooKeeper. It's ZooKeeper that enables Kafka to have a leader broker with followers. The head broker has the critical role of assigning topic partitions for replication to the followers, as well as reassigning them when a member broker fails.

2.3.15 Log management

We've covered appending messages, but we haven't talked about how logs are managed as they continue to grow. The amount of space on spinning disks in a cluster is a finite resource, so it's important for Kafka to remove messages over time. When it comes to removing old data in Kafka, there are two approaches: the traditional log-deletion approach, and compaction.

2.3.16 Deleting logs

The log-deletion strategy is a two-phased approach: first, the logs are rolled into segments, and then the oldest segments are deleted. To manage the increasing size of the logs, Kafka *rolls* them into segments. The timing of log rolling is based on timestamps embedded in the messages. Kafka rolls a log when a new message arrives, and its timestamp is greater than the timestamp of the first message in the log plus the `log.roll.ms` configuration value. At that point, the log is rolled and a new segment is created as the new active log. The previous active segment is still used to retrieve messages for consumers.

Log rolling is a configuration setting you can specify when setting up a Kafka broker.[6] There are two options for log rolling:

Footnote 6 Kafka documentation, "Broker Configs," http://kafka.apache.org/documentation/#brokerconfigs.

- `log.roll.ms`—This is the primary configuration, but there's no default value.
- `log.roll.hours`—This is the secondary configuration, which is only used if

`log.role.ms` isn't set. It defaults to 168 hours.

Over time, the number of segments will continue to grow, and older segments will need to be deleted to make room for incoming data. To handle the deletion, you can specify how long to retain the segments. Figure 2.12 illustrates the process of log rolling.

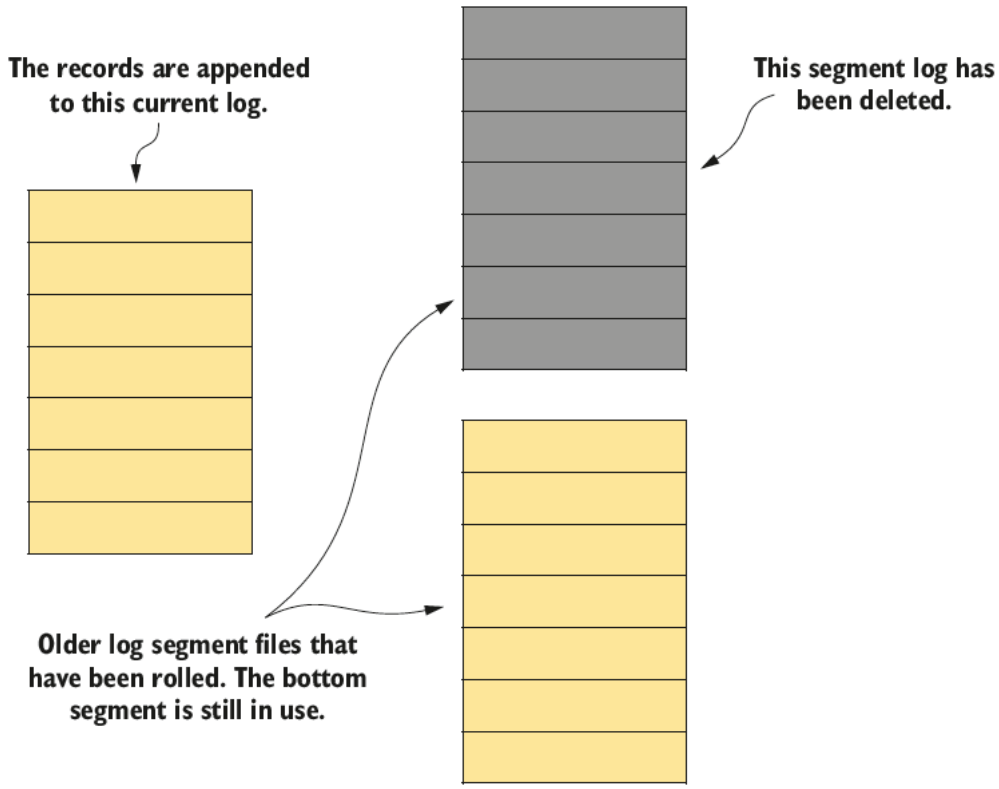


Figure 2.12 On the left are the current log segments. On the upper right is a deleted log segment, and the one below it is a recently rolled segment still in use.

Like log rolling, the removal of segments is based on timestamps in the messages and not just the clock time or time when the file was last modified. Log-segment deletion is based on the largest timestamp in the log. Here are three settings, listed in order of priority, meaning that configurations earlier in the list override the later entries:

- `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`—Log file retention in hours

I present these settings based on the assumption of high-volume topics, where you're guaranteed to reach the maximum file size in a given time period. Another configuration setting, `log.retention.bytes`, could be specified with a longer rolling-time threshold to keep down I/O operations. Finally, to guard against the case of a significant spike in volume when there are relatively large roll settings, the `log.segment.bytes` setting governs how large an individual log segment can be.

The deletion of logs works well for non-keyed records, or records that stand alone. But if you have keyed data and expected updates, there's another method that will suit your needs better.

2.3.17 Compacting logs

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.[7]

Footnote 7 Kafka documentation, "Log Compaction," http://kafka.apache.org/documentation/#compaction.

If you use the deletion policy, a segment could be removed 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 last known value for a given key, treating the next record with the same key as you would 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 very high level, a 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.

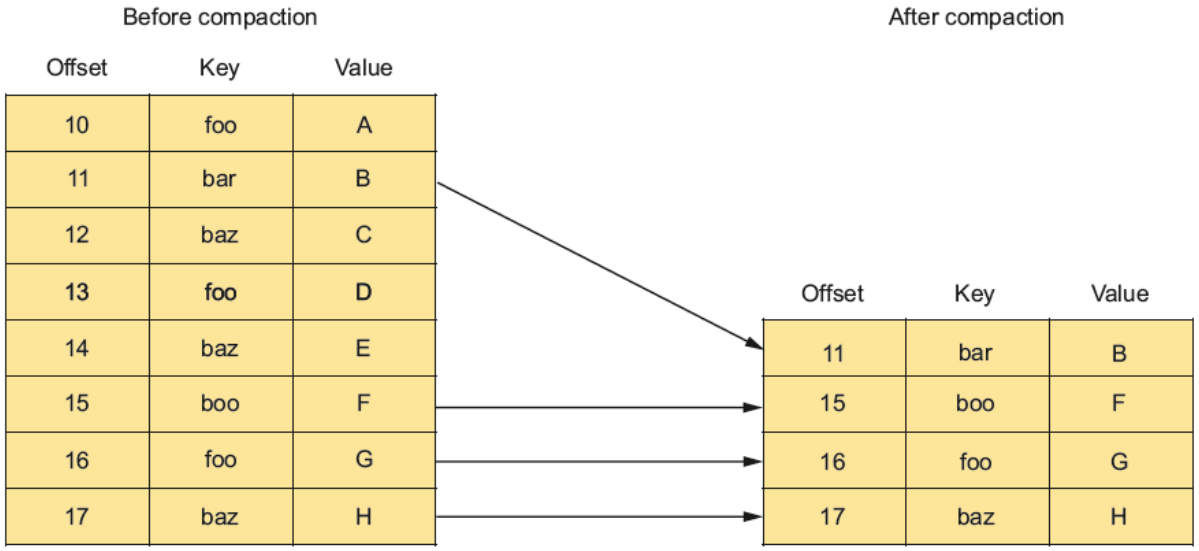


Figure 2.13 On the left is a log before compaction—you'll notice duplicate keys with

different values that are updates for the given key. On the right is the log after compaction—the latest value for each key is retained, but the log is smaller in size.

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

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 important 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: http://kafka.apache.org/documentation/\#compaction.

NOTE With a `cleanup.policy` of `compact`, you might wonder how you can remove a record from the log. With a compacted topic, deletion provides a `null` value for the given key, setting a tombstone marker. Any key with a `null` value ensures that any prior record with the same key is removed, and the tombstone marker itself is removed after a period of time.

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.

We've spent a good deal of time covering how Kafka handles data internally. Now it's time to move outside of Kafka and discuss how we can send messages to Kafka with producers and read messages from Kafka with consumers.

2.4 Sending messages with producers

Going back to ZMart's need for a centralized sales transaction data hub, let's look at how you'll send purchase transactions into Kafka. In Kafka, the *producer* is the client used for sending messages. Figure 2.14 revisits ZMart's data architecture with the producers highlighted, to emphasize where they fit into the data flow.

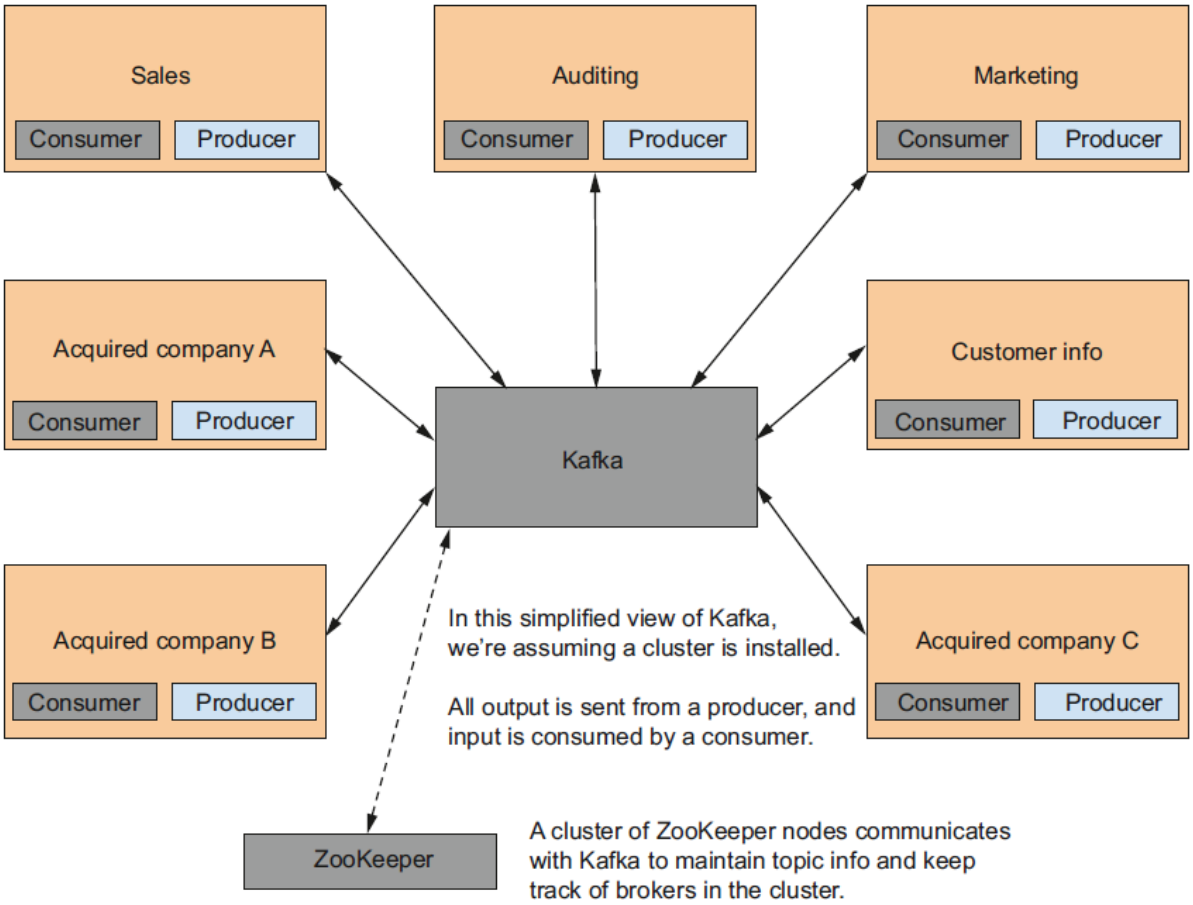


Figure 2.14 Producers are used to send messages to Kafka. Producers don't know which consumer will read the messages or when.

Although ZMart has a lot of sales transactions, we're going to consider the purchase of a single item for now: a book costing $10.99. When the customer completes the sales transaction, the information is converted into a key/value pair and sent to Kafka via a producer.

The key is the customer ID, `123447777`, and the value is in JSON format: `"{\"item\":\"book\",\"price\":10.99}"`. (I've escaped the double quotes so the JSON can be represented as a string literal in Java.) With the data in this format, you can use a producer to send the data to the Kafka cluster. The following example can be found in src/main/java/bbejeck.chapter_2/producer/SimpleProducer.java.

Listing 2.3 `SimpleProducer` example

```
Properties properties = new Properties();
properties.put("bootstrap.servers", "localhost:9092");
properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
properties.put("value.serializer",
[CA]"org.apache.kafka.common.serialization.StringSerializer");
properties.put("acks", "1");
properties.put("retries", "3");
properties.put("compression.type", "snappy");
```

```
properties.put("partitioner.class",
[CA]PurchaseKeyPartitioner.class.getName());   (1)

PurchaseKey key = new PurchaseKey("12334568", new Date());

 try(Producer<PurchaseKey, String> producer = new KafkaProducer<>   (2)
[CA](properties)) {   (2)
    ProducerRecord<PurchaseKey, String>  record = new ProducerRecord<>   (3)
[CA]("transactions", key, "{\"item\":\"book\",\"price\":10.99}");   (3)

  Callback callback = (metadata, exception) -> {
            if (exception != null) {
                System.out.println("Encountered exception "   (4)
[CA]+ exception);   (4)
            }
        };

  Future<RecordMetadata> sendFuture =   (5)
[CA]producer.send(record, callback);   (5)
}
```

(1) Properties for configuring a producer

(2) Creates the KafkaProducer

(3) Instantiates the ProducerRecord

(4) Builds a callback

(5) Sends the record and sets the returned Future to a variable

Kafka producers are thread-safe. All sends to Kafka are asynchronous—`Producer.send` returns immediately once the producer places the record in an internal buffer. The buffer sends records in batches. Depending on your configuration, there could be some blocking if you attempt to send a message while a producer's buffer is full.

The `Producer.send` method depicted here takes a `Callback` instance. Once the leader broker acknowledges the record, the producer fires the `Callback.onComplete` method. Only one of the arguments will be non-null in the `Callback.onComplete` method. In this case, you're only concerned with printing out the stacktrace in the event of error, so you check if the exception object is non-null. The returned `Future` yields a `RecordMetadata` object once the server acknowledges the record.

NOTE **Definition**

In listing 2.3, the `Producer.send` method returns a `Future` object. A `Future` object represents the result of an asynchronous operation. More important, a `Future` gives you the option to lazily retrieve asynchronous results instead of waiting for their completion. For more information on futures, see the Java documentation for "Interface Future<V>": http://mng.bz/0JK2.

2.4.1 Producer properties

When you created the `KafkaProducer` instance, you passed a `java.util.Properties` parameter containing the configuration for the producer. The configuration of a `KafkaProducer` isn't complicated, but there are key properties to consider when setting it up. These settings are where you'd specify a custom partitioner, for example. There are too many properties to cover here, so we'll just look at the ones used in listing 2.3:

- *Bootstrap servers*—`bootstrap.servers` is a comma-separated list of `host:port` values. Eventually the producer will use all the brokers in the cluster; this list is used for initially connecting to the cluster.
- *Serialization*—`key.serializer` and `value.serializer` instruct Kafka how to convert the keys and values into byte arrays. Internally, Kafka uses byte arrays for keys and values, so you need to provide Kafka with the correct serializers to convert objects to byte arrays before them sending across the wire.
- `acks`—`acks` specifies the minimum number of acknowledgments from a broker that the producer will wait for before considering a record send completed. Valid values for `acks` are `all`, `0`, and `1`. With a value of `all`, the producer will wait for a broker to receive confirmation that all followers have committed the record. When set to `1`, the broker writes the record to its log but doesn't wait for any followers to acknowledge committing the record. A value of `0` means the producer won't wait for any acknowledgments—this is mostly fire-and-forget.
- *Retries*—If sending a batch results in a failure, `retries` specifies the number of times to attempt to resend. If record order is important, you should consider setting `max.in.flight.requests.per.connection` to `1` to prevent the scenario of a second batch being sent successfully before a failed record being sent as the result a retry.
- *Compression type*—`compression.type` specifies what compression algorithm to apply, if any. If set, `compression.type` instructs the producer to compress a batch before sending. Note that it's the entire batch that's compressed, not individual records.
- *Partitioner class*—`partitioner.class` specifies the name of the class implementing the `Partitioner` interface. The `partitioner.class` is related to our earlier discussion of custom partitioners discussion in section 2.3.72.11.

For more information about producer configuration, see the Kafka documentation: http://kafka.apache.org/documentation/#producerconfigs.

2.4.2 Specifying partitions and timestamps

When you create a `ProducerRecord`, you have the option of specifying a partition, a timestamp, or both. When you instantiated the `ProducerRecord` in listing 2.3, you used one of four overloaded constructors. Other constructors allow for setting a partition and timestamp, or just a partition:

```
ProducerRecord(String topic, Integer partition, String key, String value)
ProducerRecord(String topic, Integer partition,
               Long timestamp, String key,
               String value)
```

2.4.3 Specifying a partition

In section 2.3.52.9, we discussed the importance of partitions in Kafka. We also discussed how the `DefaultPartitioner` works and how you can supply a custom partitioner. Why would you explicitly set the partition? There are a variety of business reasons why you might do so. Here's one example.

Suppose you have keyed data coming in, but it doesn't matter which partition the records go to, because the consumers have logic to handle any value that the key might contain. Additionally, the distribution of the keys might not be even, and you want to ensure that all partitions receive roughly the same amount of data. Here's a rough implementation that would do this.

Listing 2.4 Manually setting the partition

```
AtomicInteger partitionIndex = new AtomicInteger(0);  (1)

int currentPartition = Math.abs(partitionIndex.getAndIncrement()) %  (2)
[CA]numberPartitions;  (2)
ProducerRecord<String, String> record =
[CA]new ProducerRecord<>("topic", currentPartition, "key", "value");
```

(1) Creates an AtomicInteger instance variable

(2) Gets the current partition and uses it as a parameter

Here, you use the `Math.abs` call, so you don't have to keep track of the value of the integer if it goes beyond `Integer.MAX_VALUE`.

NOTE	**Definition**
	`AtomicInteger` belongs to the java.util.concurrent.atomic package, which contains classes that support lock-free, thread-safe operations on single variables. For more information, see the Java documentation for the java.util.concurrent.atomic package: http://mng.bz/PQ2q.

2.4.4 Timestamps in Kafka

Kafka version 0.10 added timestamps to records. You set the timestamp when you create a `ProducerRecord` via this overloaded constructor call:

```
ProducerRecord(String topic, Integer partition,
  [CA]Long timestamp, K key, V value)
```

If you don't set a timestamp, the producer will (using the current clock time) before sending the record to the Kafka broker.

Timestamps are also affected by the `log.message.timestamp.type` broker configuration setting, which can be set to either `CreateTime` (the default) or `LogAppendTime`. Like many other broker settings, the value configured on the broker applies to all topics as a default value, but when you create a topic, you can specify a different value for that topic. If you specify `LogAppendTime` and the topic doesn't override the broker's configuration, the broker will overwrite the timestamp with the current time when it appends the record to the log. Otherwise, the timestamp from `ProducerRecord` is used.

Why would you choose one setting over another? `LogAppendTime` is considered to be "processing time," and `CreateTime` is considered to be "event time." Which you should use depends on your business requirements. You'll need to decide whether you need to know when Kafka processed the record, or when the actual event occurred. In later chapters, you'll see the important role timestamps have in controlling data flow in Kafka Streams.

2.5 Reading messages with consumers

You've seen how producers work; now it's time to look at consumers in Kafka. Suppose you're building a prototype application to show the latest ZMart sales statistics. For this example, you'll consume the message you sent in the previous producer example. Because this prototype is in its earliest stages, all you'll do at this point is consume the message and print the information to the console.

> **NOTE** Because the version of Kafka Streams covered in the book requires Kafka version 0.10.2 or higher, we'll only discuss the new consumer that was part of the Kafka 0.9 release.

`KafkaConsumer` is the client you'll use to consume messages from Kafka. The `KafkaConsumer` class is straightforward to use, but there are a few operational considerations to take into account. Figure 2.15 shows the ZMart architecture, highlighting where consumers play a role in the data flow.

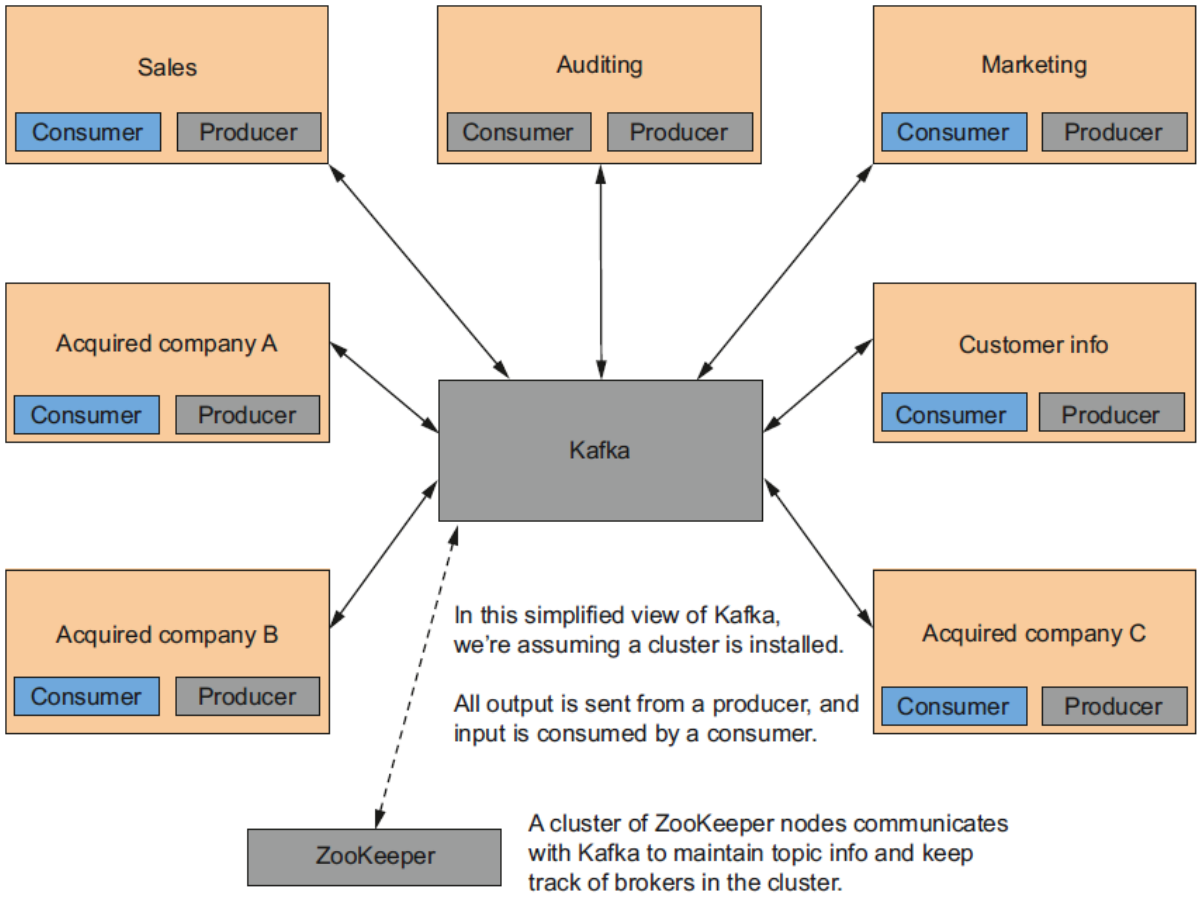


Figure 2.15 These are the consumers that read messages from Kafka. Just as producers have no knowledge of the consumers, consumers read messages from Kafka with no knowledge of who produced the messages.

2.5.1 Managing offsets

`KafkaProducer` is essentially stateless, but `KafkaConsumer` manages some state by periodically committing the offsets of messages consumed from Kafka. Offsets uniquely identify messages and represent the starting positions of messages in the log. Consumers periodically need to commit the offsets of messages they have received.

Committing an offset has two implications for a consumer:

- Committing implies the consumer has fully processed the message.
- Committing also represents the starting point for that consumer in the case of failure or a restart.

If you have a new consumer instance or some failure has occurred, and the last committed offset isn't available, where the consumer starts from will depend on your configuration:

- `auto.offset.reset="earliest"`—Messages will be retrieved starting at the earliest available offset. Any messages that haven't yet been removed by the log-management process will be retrieved.
- `auto.offset.reset="latest"`—Messages will be retrieved from the latest offset, essentially only consuming messages from the point of joining the cluster.
- `auto.offset.reset="none"`—No reset strategy is specified. The broker throws an exception to the consumer.

In figure 2.16, you can see the impact of choosing an `auto.offset.reset` setting. By selecting `earliest`, you receive messages starting at offset 1. If you choose `latest`, you'll get a message starting at offset 11.

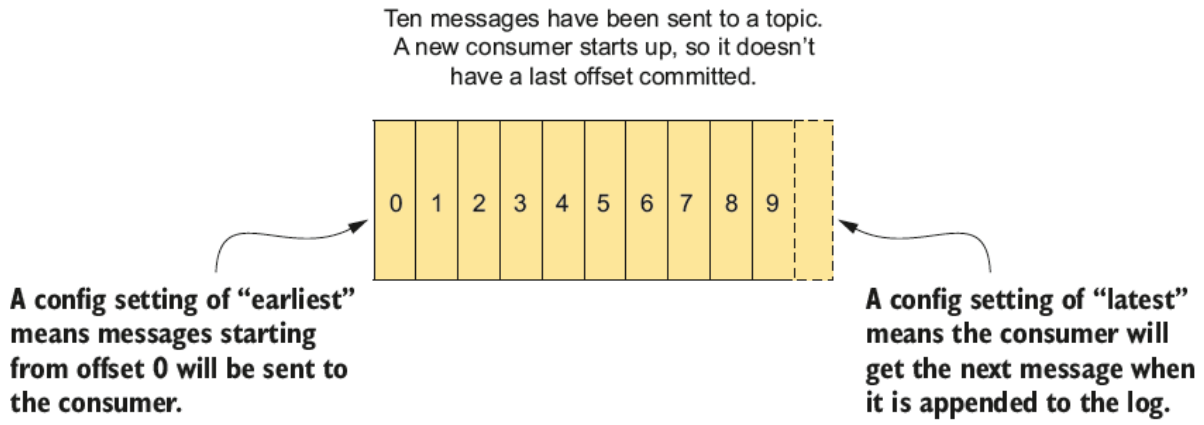


Figure 2.16 A graphical representation of setting auto.offset.reset to earliest versus latest. A setting of earliest will give you all messages not yet deleted; latest means you'll wait for the next available message to arrive.

Next, we need to discuss the options for committing offsets. You can do this either

automatically or manually.

2.5.2 Automatic offset commits

Automatic offset commits are enabled by default, and they're represented by the `enable.auto.commit` property. There's a companion configuration option, `auto.commit.interval.ms`, which specifies how often the consumer will commit offsets (the default value is 5 seconds). You should take care when adjusting this value. If it's too small, it will increase network traffic; if it's too large, it could result in the consumer receiving large amounts of repeated data in the event of a failure or restart.

2.5.3 Manual offset commits

There are two types of manually committed offsets—synchronous and asynchronous. These are the synchronous commits:

```
consumer.commitSync()
consumer.commitSync(Map<TopicPartition, OffsetAndMetadata>)
```

The no-arg `commitSync()` method blocks until all offsets returned from the last retrieval (poll) succeed. This call applies to all subscribed topics and partitions. The other version takes a `Map<TopicPartition, OffsetAndMetadata>` parameter, and it commits only the offsets, partitions, and topics specified in the map.

There are analogous `consumer.commitAsync()` methods that are completely asynchronous and return immediately. One of the overloaded methods accepts no arguments, and two of the `consumer.commitAsync` methods have an option to provide an `OffsetCommitCallback` object, which is called when the commit has concluded either successfully or with an error. Providing a callback instance allows for asynchronous processing and error handling. The advantage of using manual commits is that it gives you direct control over when a record is considered processed.

2.5.4 Creating the consumer

Creating a consumer is similar to creating a producer. You supply a configuration in the form of a Java `java.util.Properties` object, and you get back a `KafkaConsumer` instance. This instance then subscribes to topics from a supplied list of topic names or by specifying a regular expression. Typically, you'll run the consumer in a loop, where you poll for a period specified in milliseconds.

A `ConsumerRecords<K, V>` object is the result of the polling. `ConsumerRecords`

implements the `Iterable` interface, and each call to `next()` returns a `ConsumerRecord` object containing metadata about the message, in addition to the actual key and value.

After you've exhausted all of the `ConsumerRecord` objects returned from the last call to `poll`, you return to the top of the loop, polling again for the specified period. In practice, consumers are expected to run indefinitely in this manner, unless an error occurs or the application needs to be shut down and restarted (this is where committed offsets come into play—on reboot, the consumer will pick up where it left off).

2.5.5 Consumers and partitions

You'll generally want multiple consumer instances—one for each partition of a topic. It's possible to have one consumer read from multiple partitions, but it's not uncommon to have a thread pool with as many threads as there are partitions, and with each thread running a consumer that's assigned to one partition.

This consumer-per-partition pattern maximizes throughput, but if you spread your consumers across multiple applications or machines, the *total* thread count across all instances shouldn't exceed the total number of partitions in the topic. Any threads in excess of the total partition count will be idle. If a consumer fails, the leader broker assigns its partitions to another active consumer.

> **NOTE** This example shows a consumer subscribing to one topic, but this is for demonstration purposes only. You can subscribe a consumer to an arbitrary number of topics.

The leader broker assigns topic partitions to all available consumers with the same `group.id`. The `group.id` is a configuration setting that identifies the consumer as belonging to a *consumer group*—that way, consumers don't need to reside on the same machine. In fact, it's probably preferable to have your consumers spread out across a few machines. That way, in the case of one machine failing, the leader broker can assign topic partitions to consumers on good machines.

2.5.6 Rebalancing

The process of adding and removing topic-partition assignments to consumers described in the previous section is called *rebalancing*. Topic-partition assignments to a consumer aren't static—they're dynamic. As you add consumers with the same group ID, some of the current topic-partition assignments are taken from active consumers and given to the new consumers. This reassignment process continues until every partition has been assigned to a consumer that's reading data.

After that equilibrium point, any additional consumers will remain idle. When consumers leave the group for whatever reason, their topic-partition assignments are reassigned to other consumers.

2.5.7 Finer-grained consumer assignment

In section 2.5.52.32, I described the use of a thread pool and subscribing multiple consumers (in the same consumer group) to the same topics. Although Kafka will balance the load of topic-partitions across all consumers, the assignment of the topic and partition isn't deterministic. You won't know what topic-partition pairings each consumer will receive.

`KafkaConsumer` has methods that allow you to subscribe to a particular topic and partition:

```
TopicPartition fooTopicPartition_0 = new TopicPartition("foo", 0);
TopicPartition barTopicPartition_0 = new TopicPartition("bar", 0);

consumer.assign(Arrays.asList(fooTopicPartition_0, barTopicPartition_0));
```

There are trade-offs to consider when using manual topic-partition assignment:

- Failures won't result in topic partitions being reassigned, even for consumers with the same group ID. Any changes in assignments will require another call to `consumer.assign`.
- The group specified by the consumer is used for committing, but because each consumer will be acting independently, it's a good idea to give each consumer a unique group ID.

2.5.8 Consumer example

Here's the consumer code for the ZMart prototype that consumes transactions and prints them to the console. You can find it in src/main/java/bbejeck.chapter_2/consumer/ThreadedConsumerExample.java.

Listing 2.5 `ThreadedConsumerExample` example

```
public void startConsuming() {
        executorService = Executors.newFixedThreadPool(numberPartitions);
        Properties properties = getConsumerProps();

        for (int i = 0; i < numberPartitions; i++) {
            Runnable consumerThread = getConsumerThread(properties);   (1)
            executorService.submit(consumerThread);
        }
    }

    private Runnable getConsumerThread(Properties properties) {
        return () -> {
            Consumer<String, String> consumer = null;
            try {
                consumer = new KafkaConsumer<>(properties);
                consumer.subscribe(Collections.singletonList(   (2)
[CA]"test-topic"));   (2)
                while (!doneConsuming) {
                    ConsumerRecords<String, String> records =   (3)
[CA]consumer.poll(5000);   (3)
                    for (ConsumerRecord<String, String> record : records) {
                        String message = String.format("Consumed: key =
[CA]%s value = %s with offset = %d partition = %d",
                                record.key(), record.value(),
                                record.offset(), record.partition());
                        System.out.println(message);   (4)
                    }

                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (consumer != null) {
                    consumer.close();   (5)
                }
            }
        };
    }
```

(1) Builds a consumer thread

(2) Subscribes to the topic

(3) Polls for 5 seconds

(4) Prints a formatted message

(5) Closes the consumer—will leak resources otherwise

This example leaves out other sections of the class for clarity—it won't stand on its own. You can find the full example in this chapter's source code.

2.6 Installing and running Kafka

As I write this, Kafka 1.0.0 is the most recent version. Because Kafka is a Scala project, each release comes in two versions: one for Scala 2.11 and another for Scala 2.12. I use the 2.12 Scala version of Kafka in this book. Although you can download the release, the book's source code includes a binary distribution of Kafka that will work with Kafka Streams as demonstrated and described in this book. To install Kafka, extract the .tgz file found in the book's source code repo (source code can be found on the book's website here: https://manning.com/books/kafka-streams-in-action), to somewhere in the libs folder on your machine.

> **NOTE** The binary distribution of Kafka includes Apache ZooKeeper, so no extra installation work is required.

2.6.1 Kafka local configuration

Running Kafka locally on your machine requires minimal configuration if you accept the default values. By default, Kafka uses port 9092, and ZooKeeper uses port 2181. Assuming you have no applications already using those ports, you're all set.

Kafka writes its logs to /tmp/Kafka-logs, and ZooKeeper uses /tmp/zookeeper for its log storage. Depending on your machine, you may need to change permission or ownership of those directories or to modify the location where you want to write the logs.

To change the Kafka logs directory, `cd` into <kafka-install-dir>/config and open the server.properties file. Find the `log.dirs` entry, and change the value to what you'd rather use. In the same directory, open the zookeeper.properties file and change the `dataDir` entry.

We'll look at configuring Kafka in detail later in this book, but that's all the configuration you need to do for now. Keep in mind that these "logs" are the actual data used by Kafka and ZooKeeper, and not application-level logs that track the application's behavior. The application logs are found in the <kafka-install-dir>/logs directory.

2.6.2 Running Kafka

Kafka is simple to get started. Because ZooKeeper is essential for the Kafka cluster to function properly (ZooKeeper determines the leader broker, holds topic information, performs health checks on cluster members, and so on), you'll need to start ZooKeeper before starting Kafka.

NOTE From now on, all directory references assume you're working in your Kafka installation directory. If you're using a Windows machine, the directory is <kafka-install-dir>/bin/windows.

STARTING ZOOKEEPER

To start ZooKeeper, open a command prompt and enter the following command:

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

You'll see a lot of information run by on the screen, and it should end up looking something like figure 2.17.

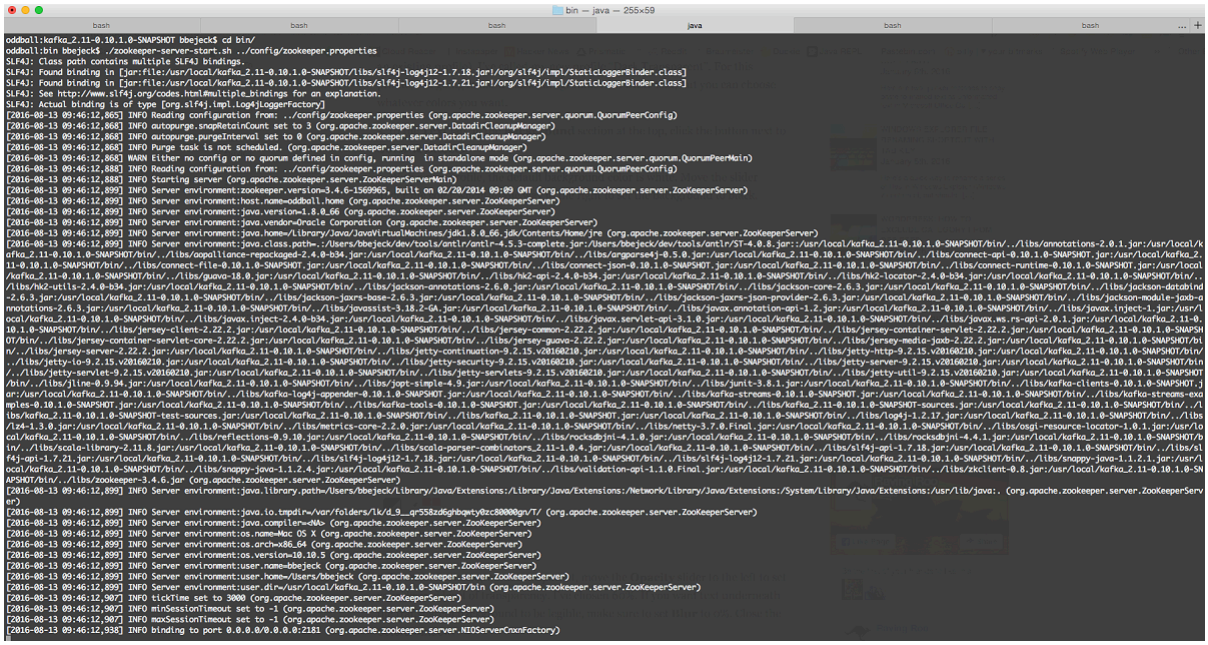


Figure 2.17 Output visible on the console when ZooKeeper starts up

STARTING KAFKA

To start Kafka, open another command prompt and type this command:

```
bin/Kafka-server-start.sh  config/server.properties
```

Again, you'll see text scroll by on the screen. When Kafka has fully started, you should see something similar to figure 2.18.

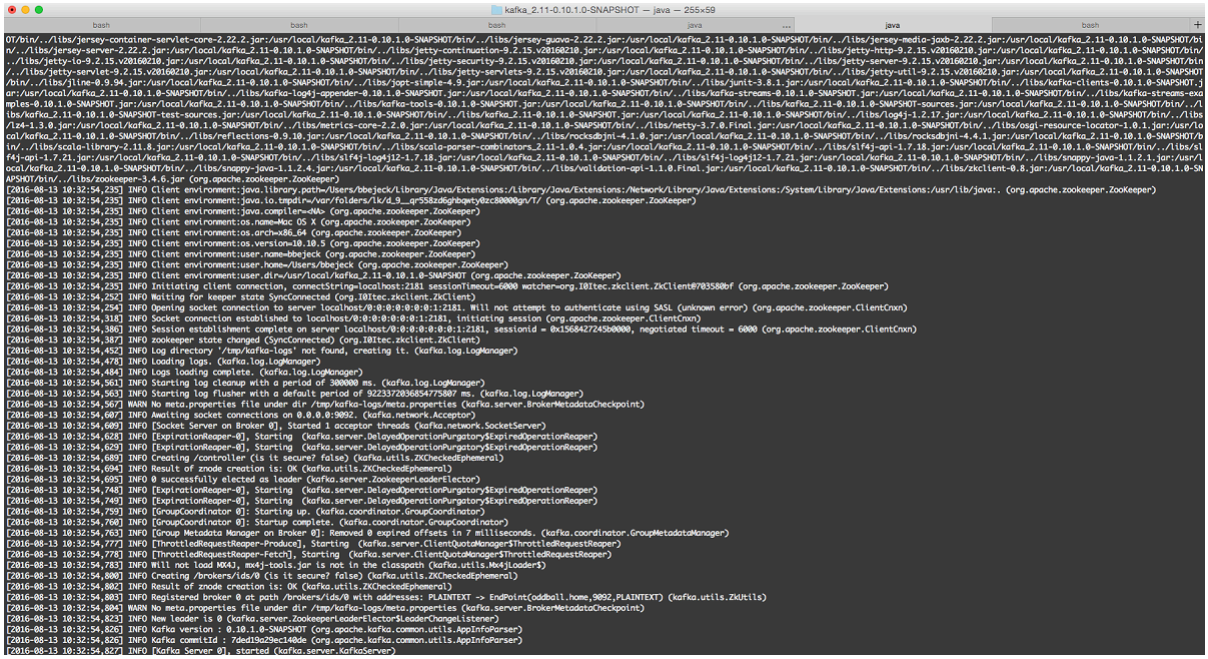


Figure 2.18 Output from Kafka when starting up

> **TIP** ZooKeeper is essential for Kafka to run, so it's important to reverse the order when shutting down: stop Kafka first, and then stop ZooKeeper. To stop Kafka, you can press Ctrl-C from the terminal Kafka is running in, or run `kafka-server-stop.sh` from another terminal. The same goes for ZooKeeper, except the shutdown script is `zookeeper-server-stop.sh`.

2.6.3 Sending your first message

Now that you have Kafka up and running, it's time to use Kafka for what it's meant to do: sending and receiving messages. But before you send a message, you'll need to define a topic for a producer to send a message to.

YOUR FIRST TOPIC

Creating a topic in Kafka is simple. It's just a matter of running a script with some configuration parameters. The configuration is easy, but the settings you provide have broad performance implications.

By default, Kafka is configured to autocreate topics, meaning that if you attempt to send to or read from a nonexistent topic, the Kafka broker will create one for you (using default configurations in the server.properties file). It's rarely a good practice to rely on the broker to create topics, even in development, because the first produce/consume attempt will fail, as it takes time for the metadata about the topic's existence to propagate. Be sure to always proactively to create your topics.

CREATING A TOPIC

To create a topic, you need to run the kafka-topics.sh script. Open a terminal window and run this command:

```
bin/kafka-topics.sh --create --topic first-topic --replication-factor 1
[CA]--partitions 1 --zookeeper localhost:2181
```

When the script executes, you should see something similar to figure 2.19 in your terminal.

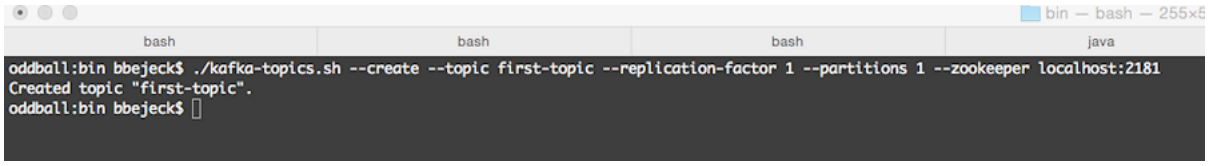


Figure 2.19 These are the results from creating a topic. It's important to create your topics ahead of time so you can supply topic-specific configurations. Otherwise, autocreated topics will use default configuration or the configuration in the server.properties file.

Most of the configuration flags in the previous command are obvious, but let's quickly review two of them:

- `replication-factor`—This flag determines how many copies of the message the leader broker distributes in the cluster. In this case, with a replication factor of `1`, no copies will be made. Just the original message will reside in Kafka. A replication factor of `1` is fine for a quick demo or prototype, but in practice you'll almost always want a replication factor of `2` or `3` to provide data availability in the case of machine failures.
- `partitions`—This flag specifies the number of partitions that the topic will use. Again, just one partition is fine here, but if you have greater load, you'll certainly want more partitions. Determining the correct number of partitions is not an exact science.

SENDING A MESSAGE

Sending a message in Kafka generally involves writing a producer client, but Kafka also comes with a handy script called `kafka-console-producer` that allows you to send a message from a terminal window. We'll use the console producer in this example, but we'll cover the producer client in some detail in section 5 of this book.

To send your first message, run the following command (also shown in figure 2.20):

```
# command assumes running from bin directory
./kafka-console-producer.sh --topic first-topic --broker-list localhost:9092
```

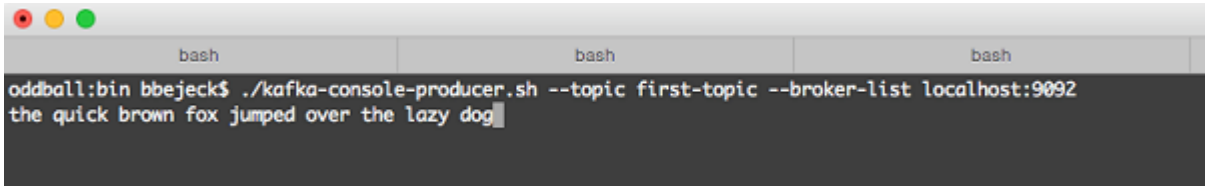


Figure 2.20 The console producer is a great tool for quickly testing your configuration and ensuring end-to-end functionality.

There are several options for configuring the console producer, but for now we'll only use the required ones: the topic to send the message to, and a list of Kafka brokers to connect to (in this case, just the one on your local machine).

Starting a console producer is a "blocking script," so after executing the preceding command, you enter some text and press Enter. You can send as many messages as you like, but for our demo purposes you can type a single message, "the quick brown fox jumped over the lazy dog," press Enter, and then press Ctrl-C to exit the producer.

READING A MESSAGE

Kafka also provides a console consumer for reading messages from the command line. The console consumer is similar to the console producer: once started, it will keep reading messages from the topic until the script is stopped by you (with Ctrl-C).

To launch the console consumer, run this command:

```
bin/kafka-console-consumer.sh --topic first-topic
[CA]--bootstrap-server localhost:9092 --from-beginning
```

After starting the console consumer, you should see something like figure 2.21 in your terminal.

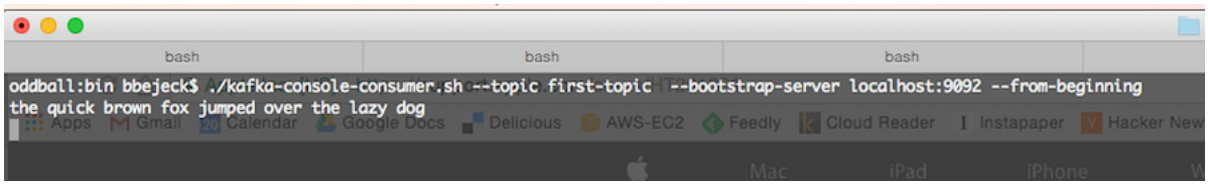


Figure 2.21 The console consumer is a handy tool for quickly getting a feel for whether data is flowing and if messages contain the expected information.

The `--from-beginning` parameter specifies that you'll receive any message not deleted from that topic. The console consumer won't have any committed offsets, so if you didn't have the `--from-beginning` setting, you'd only get messages sent after the console consumer had started.

You've just completed a whirlwind tour of Kafka and produced and consumed your first message. If you haven't read the first part of this chapter, it's time to go back to the beginning this chapter to learn the details of how Kafka works!

2.7 Summary

- Kafka is a message broker that receives messages and stores them in a way that makes it easy and fast to respond to consumer requests. Messages are never pushed out to consumers, and message retention in Kafka is entirely independent of when and how often messages are consumed.
- Kafka uses partitions for achieving high throughput and to provide a means for grouping messages with the same keys in order.
- Producers are used for sending messages to Kafka.
- Null keys mean round-robin partition assignment; otherwise, the producer uses the hash of the key, modulus the number of partitions, for partition assignment.
- Consumers are what you use to read messages from Kafka.
- Consumers that are part of a consumer group are given topic-partition allocations in an attempt to distribute messages evenly.

In the next chapter, we'll start looking at Kafka Streams with a concrete example from the world of retail sales. Although Kafka Streams will handle the creation of all consumer and producer instances, you should be able to see the concepts we introduced here come into play.

Part 2: Kafka Streams Development

This part of the book builds on the previous part and puts the mental model of Kafka Streams into action as you develop your first Kafka Streams application. Once you've gotten your feet wet, we'll walk through the significant Kafka Streams APIs.

You'll learn about providing state to a streaming application and how to use state for performing joins, much like the joins you perform when running SQL queries. Then we'll move on to a new abstraction from Kafka Streams: the `KTable` API. This part of the book begins with the high-level DSL, but we'll wrap up by discussing the lower-level Processor API and how you can use it to make Kafka Streams do pretty much anything you need it to do.

3 Developing Kafka Streams

This chapter covers

- Introducing the Kafka Streams API
- Building Hello World for Kafka Streams
- Exploring the ZMart Kafka Streams application in depth
- Splitting an incoming stream into multiple streams

In chapter 1, you learned about the Kafka Streams library. You learned about building a topology of processing nodes, or a graph that transforms data as it's streaming into Kafka. In this chapter, you'll learn how to create this processing topology with the Kafka Streams API.

The Kafka Streams API is what you'll use to build Kafka Streams applications. You'll learn how to assemble Kafka Streams applications; but, more important, you'll gain a deeper understanding of how the components work together and how they can be used to achieve your stream-processing goals.

3.1 The Streams Processor API

The Kafka Streams DSL is the high-level API that enables you to build Kafka Streams applications quickly. The high-level API is very well thought out, and there are 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 Java 8 lambda expressions. Taking these factors into account, you can imagine the simplicity and ease with which you can build a Kafka Streams program.

Back in 2005, Martin Fowler and Eric Evans developed the concept of the *fluent interface*—an interface where the return value of a method call is the same instance that originally called the method (https://martinfowler.com/bliki/FluentInterface.html). This approach is useful when constructing objects with several parameters, such as `Person.builder().firstName("Beth").withLastName("Smith").withOccupation("CE`. In Kafka Streams, there is one small but important difference: the returned `KStream` object is a new instance, not the same instance that made the original method call.

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 chapter 6. With that introduction out of the way, let's dive into the requisite Hello World program for Kafka Streams.

3.2 Hello World for Kafka Streams

For the first Kafka Streams example, we'll deviate from the problem outlined in chapter 1 to a simpler use case. This will get off the ground quickly so you can see how Kafka Streams works. We'll get back to the problem from chapter 1 later in section 3.5.1 for a more realistic, concrete example.

Your first program will be a toy application that takes incoming messages and converts them to uppercase characters, effectively yelling at anyone who reads the message. You'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. You'll follow the same pattern as in chapter 1, where you built up a processing graph topology with each node in the graph having a particular function. The main difference is that this graph will be simpler, as you can see in figure 3.1.

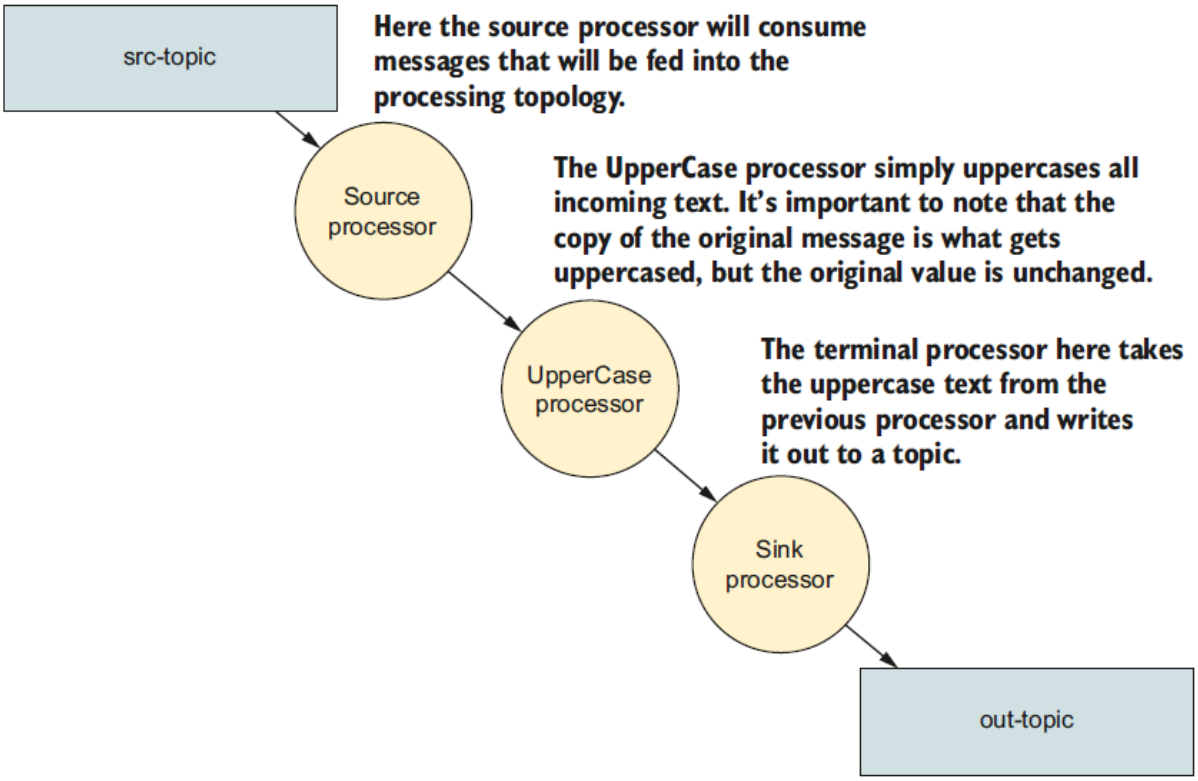


Figure 3.1 Graph (topology) of the Yelling App

As you can see, you're building 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 structure:

1. Define the configuration items.
2. Create `Serde` instances, either custom or predefined.
3. Build the processor topology.
4. Create and start the `KStream`.

When we get into the more advanced examples, the principal difference will be in the complexity of the processor topology. With that in mind, it's time to build your first application.

3.2.1 Creating the topology for the Yelling App

The first step to creating any Kafka Streams application is to create a source node. The source node is responsible for consuming the records, from a topic, that will flow through the application. Figure 3.2 highlights the source node in the graph.

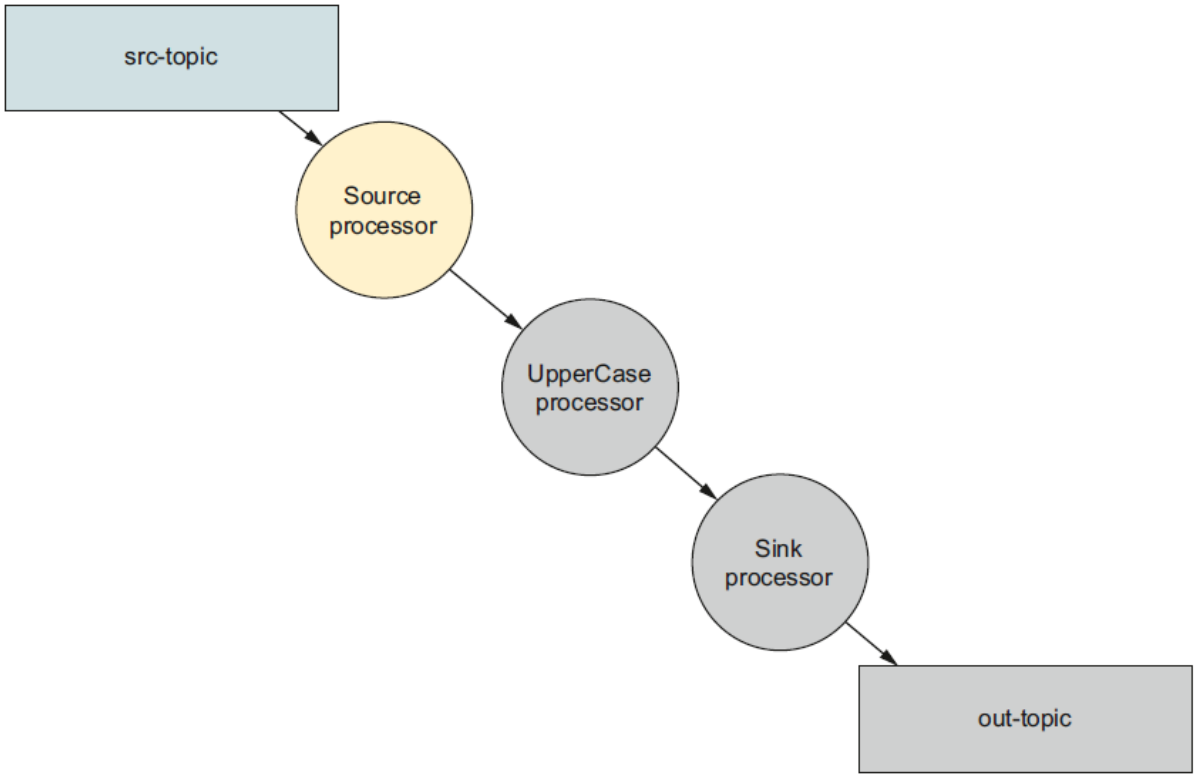


Figure 3.2 Creating the source node of the Yelling App

The following line of code creates the source, or parent, node of the graph.

Listing 3.1 Defining the source for the stream

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

The `simpleFirstStreamKStream` instance is set to consume messages written to the `src-topic` topic. In addition to specifying the topic name, you also provide `Serde` objects (via a `Consumed` instance) for deserializing the records from Kafka. You'll use the `Consumed` class for any optional parameters whenever you create a source node in Kafka Streams.

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 3.3. The code used to attach the processor (a child node of the source node) is shown in the following listing. With this

line, you create another `KStream` instance that's a child node of the parent node.

Listing 3.2 Mapping incoming text to uppercase

```
KStream<String, String> upperCasedStream =
[CA]simpleFirstStream.mapValues(String::toUpperCase);
```

By calling the `KStream.mapValues` function, you're creating a new processing node whose inputs are the results of going through the `mapValues` call.

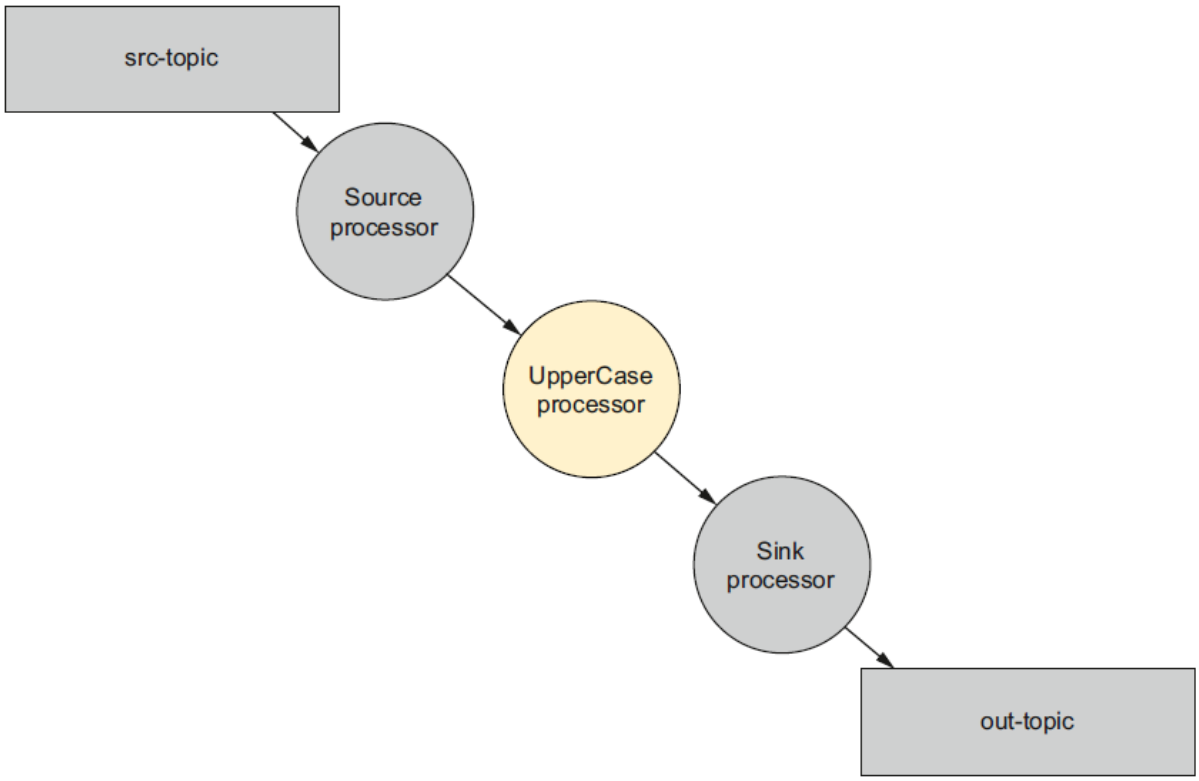


Figure 3.3 Adding the uppercase processor to the Yelling App

It's important to remember that you shouldn't modify the *original* value in the `ValueMapper` provided to `mapValues`. The `upperCasedStream` instance receives transformed copies of the initial value from the `simpleFirstStream.mapValues` call. In this case, it's uppercase text.

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 Java 8 lambda expression. This is what you've done here with `String::toUpperCase`, which is a method reference, an even shorter form of a Java 8 lambda expression.

NOTE Many Java 8 tutorials are available for lambda expressions and method references. Good starting points can be found in Oracle's Java documentation: "Lambda Expressions" (http://mng.bz/J0Xm) and "Method References" (http://mng.bz/BaDW).

You could have used the form `s  s.toUpperCase()`, but because `toUpperCase()` is an instance method on the `String` class, you can use a method reference.

Using lambda expressions instead of concrete implementations is a pattern you'll see over and over with the Streams Processor API in this book. Because most of the methods expect types that are single method interfaces, you can easily use Java 8 lambdas.

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 3.4 shows where you are in the construction of the topology.

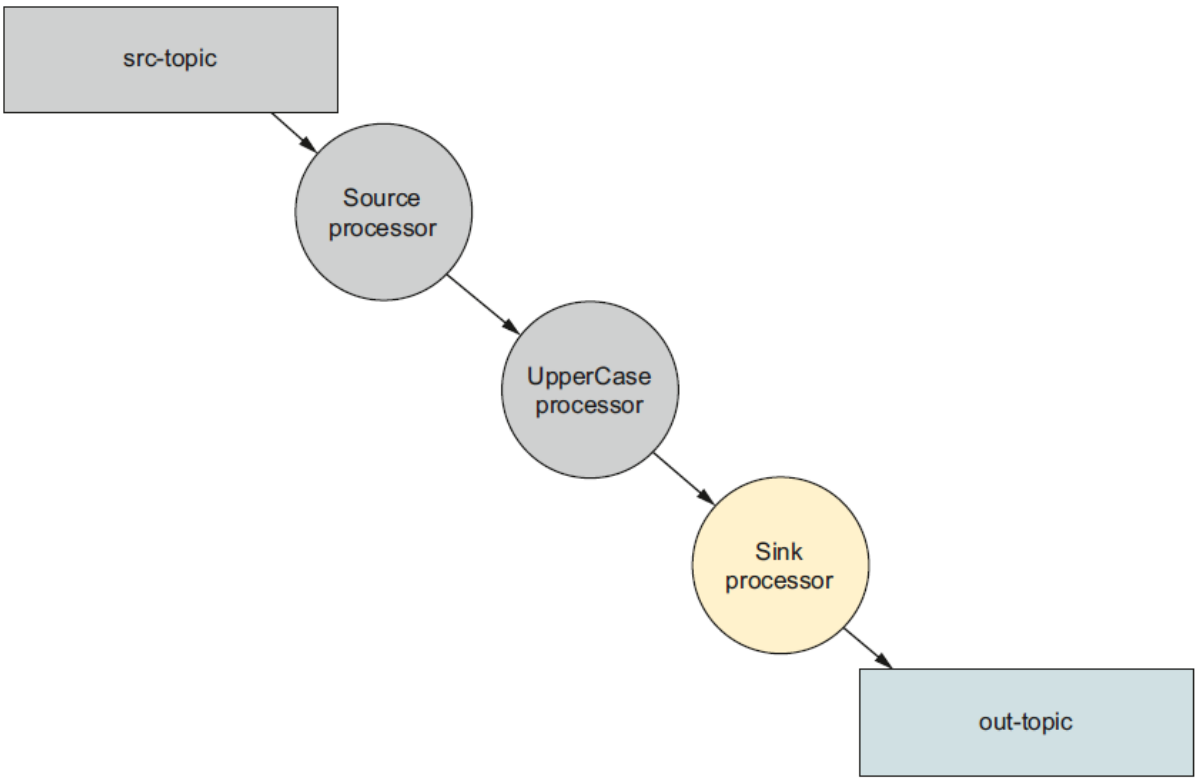


Figure 3.4 Adding a processor for writing the Yelling App results

The following code line adds the last processor in the graph.

Listing 3.3 Creating a sink node

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

The `KStream.to` method creates a sink-processing node in the topology. Sink processors write records back out to Kafka. This sink node takes records from the `upperCasedStream` processor and writes them to a topic named `out-topic`. 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 =
[CA]builder.stream("src-topic", Consumed.with(stringSerde, stringSerde));
KStream<String, String> upperCasedStream =
[CA]simpleFirstStream.mapValues(String::toUpperCase);
    upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));
```

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 mentioned earlier. To demonstrate this idea, here's another way you could construct the Yelling App topology:

```
builder.stream("src-topic", Consumed.with(stringSerde, stringSerde))
[CA].mapValues(String::toUpperCase)
[CA].to("out-topic", Produced.with(stringSerde, stringSerde));
```

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.

3.2.2 Kafka Streams configuration

Although Kafka Streams is highly configurable, with several properties you can adjust for your specific needs, the first example uses only two 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 settings are required because no default values are provided. 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 identifies your Kafka Streams application, and it must be a unique value for the entire cluster. It also serves as a default value for the client ID prefix and group ID parameters if you don't set either value. The client ID prefix is the user-defined value that uniquely identifies clients connecting to Kafka. The group ID is used to manage the membership of a group of consumers reading from the same topic, ensuring that all consumers in the group can effectively read subscribed topics.

The `StreamsConfig.BOOTSTRAP_SERVERS_CONFIG` property can be a single `hostname:port` pair or multiple `hostname:port` comma-separated pairs. The value of this setting points the Kafka Streams application to the locaction of the Kafka cluster. We'll cover several more configuration items as we explore more examples in the book.

3.2.3 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
- Long

- Integer
- Double

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 an upcoming example, you'll 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_3/KafkaStreamsYellingApp.java (source code can be found on the book's website here: https://manning.com/books/kafka-streams-in-action).

Listing 3.4 Hello World: the Yelling App

```
public class KafkaStreamsYellingApp {

 public static void main(String[] args) {


  Properties props = new Properties();

  props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");       (1)
  props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");


  StreamsConfig streamingConfig = new StreamsConfig(props);       (2)

  Serde<String> stringSerde = Serdes.String();       (3)

  StreamsBuilder builder = new StreamsBuilder();       (4)

  KStream<String, String> simpleFirstStream = builder.stream("src-topic",       (5)
  [CA]Consumed.with(stringSerde, stringSerde));       (5)

  KStream<String, String> upperCasedStream =       (6)
  [CA]simpleFirstStream.mapValues(String::toUpperCase);       (6)

  upperCasedStream.to( "out-topic",       (7)
  [CA]Produced.with(stringSerde, stringSerde));       (7)

  KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),streamsConfig);


  kafkaStreams.start();       (8)
  Thread.sleep(35000);
  LOG.info("Shutting down the Yelling APP now");
  kafkaStreams.close();

  }
}
```

1. Properties for configuring the Kafka Streams program
2. Creates the StreamsConfig with the given properties
3. Creates the Serdes used to serialize/deserialize keys and values
4. Creates the StreamsBuilder instance used to construct the processor topology
5. Creates the actual stream with a source topic to read from (the parent node in the graph)
6. A processor using a Java 8 method handle (the first child node in the graph)
7. Writes the transformed output to another topic (the sink node in the graph)
8. Kicks off the Kafka Streams threads

You've now constructed your first Kafka Streams application. Let's quickly review the steps involved, as it's a general pattern you'll see in most of your Kafka Streams applications:

1. Create a `StreamsConfig` instance.
2. Create a `Serde` object.
3. Construct a processing topology.
4. Start the Kafka Streams program.

Apart from the general construction of a Kafka Streams application, a key takeaway here is to use lambda expressions whenever possible, to make your programs more concise.

We'll now move on to a more complex example that will allow us to explore more of the Streams Processor API. The example will be new, but the scenario is one you're already familiar with: ZMart data-processing goals.

3.3 Working with customer data

In chapter 1, we discussed ZMart's new requirements for processing customer data, intended to help ZMart do business more efficiently. We demonstrated how you could build a topology of processors that would work on purchase records as they come streaming in from transactions in ZMart stores. Figure 3.5 shows the completed graph again.

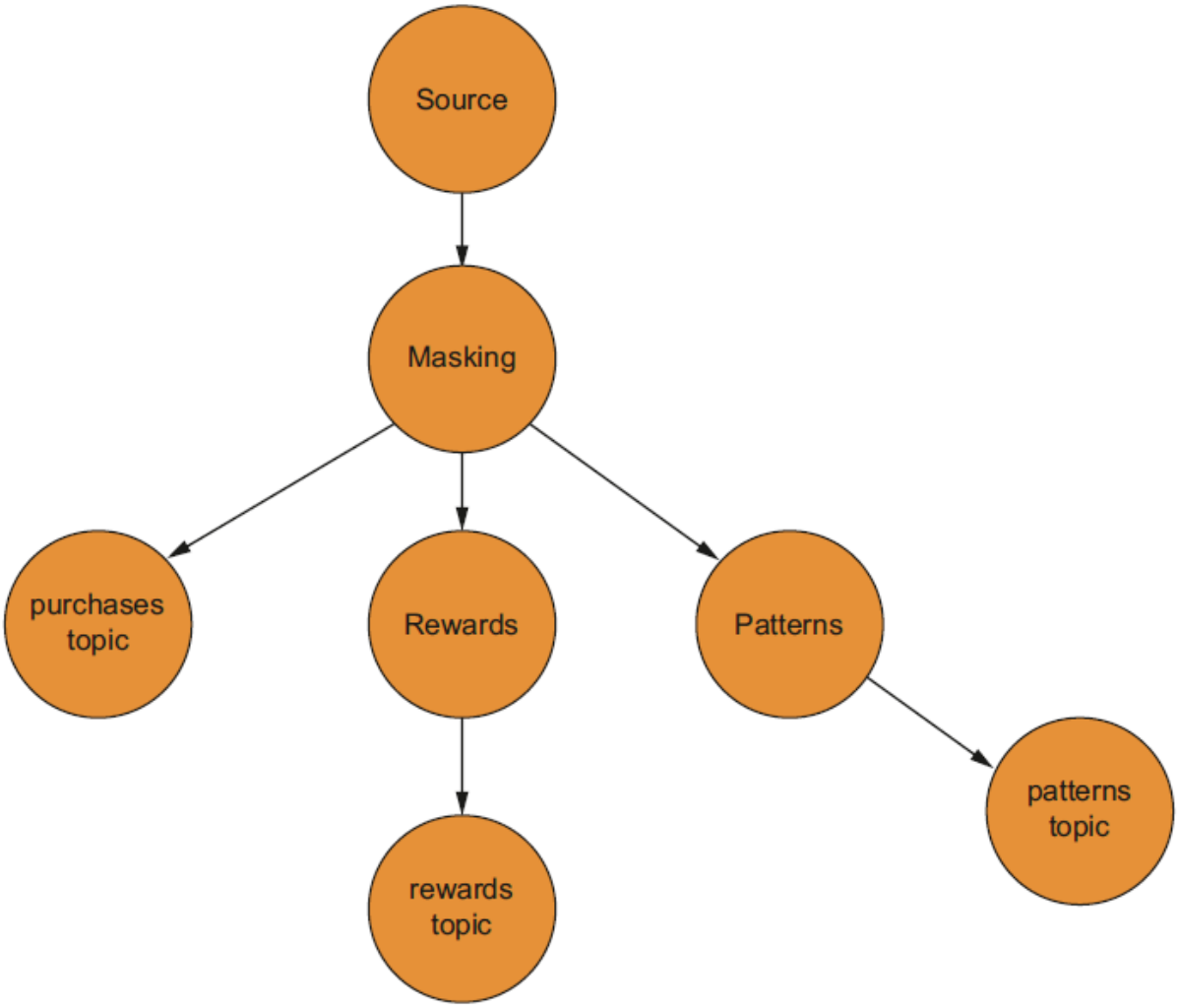


Figure 3.5 Topology for ZMart Kafka Streams program

Let's briefly review the requirements for the streaming program, which will also serve as a good description of what the program will do:

- All records need to have credit card numbers protected, in this case by masking the first 12 digits.
- You need to extract the items purchased and the ZIP code to determine purchase patterns. This data will be written out to a topic.
- 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.
- You need to write the entire transaction out to topic, which will be consumed by a storage engine for ad hoc analysis.

As in the Yelling App, you'll combine the fluent interface approach with Java 8 lambdas when building the application. Although it's sometimes clear that the return type of a

method call is a `KStream` object, other times it may not be. Keep in mind that the majority of the methods in the `KStream` API return *new*`KStream` instances. Now, let's build a streaming application that will satisfy ZMart's business requirements.

3.3.1 Constructing a topology

Let's dive into building the processing topology. To help make the connection between the code you'll create here and the processing topology graph from chapter 1, I'll highlight the part of the graph that you're currently working on.

BUILDING THE SOURCE NODE

You'll start by building the source node and first processor of the topology by chaining two calls to the `KStream` API together (highlighted in figure 3.6). It should be fairly obvious by now what the role of the origin node is. The first processor in the topology will be responsible for masking credit card numbers to protect customer privacy.

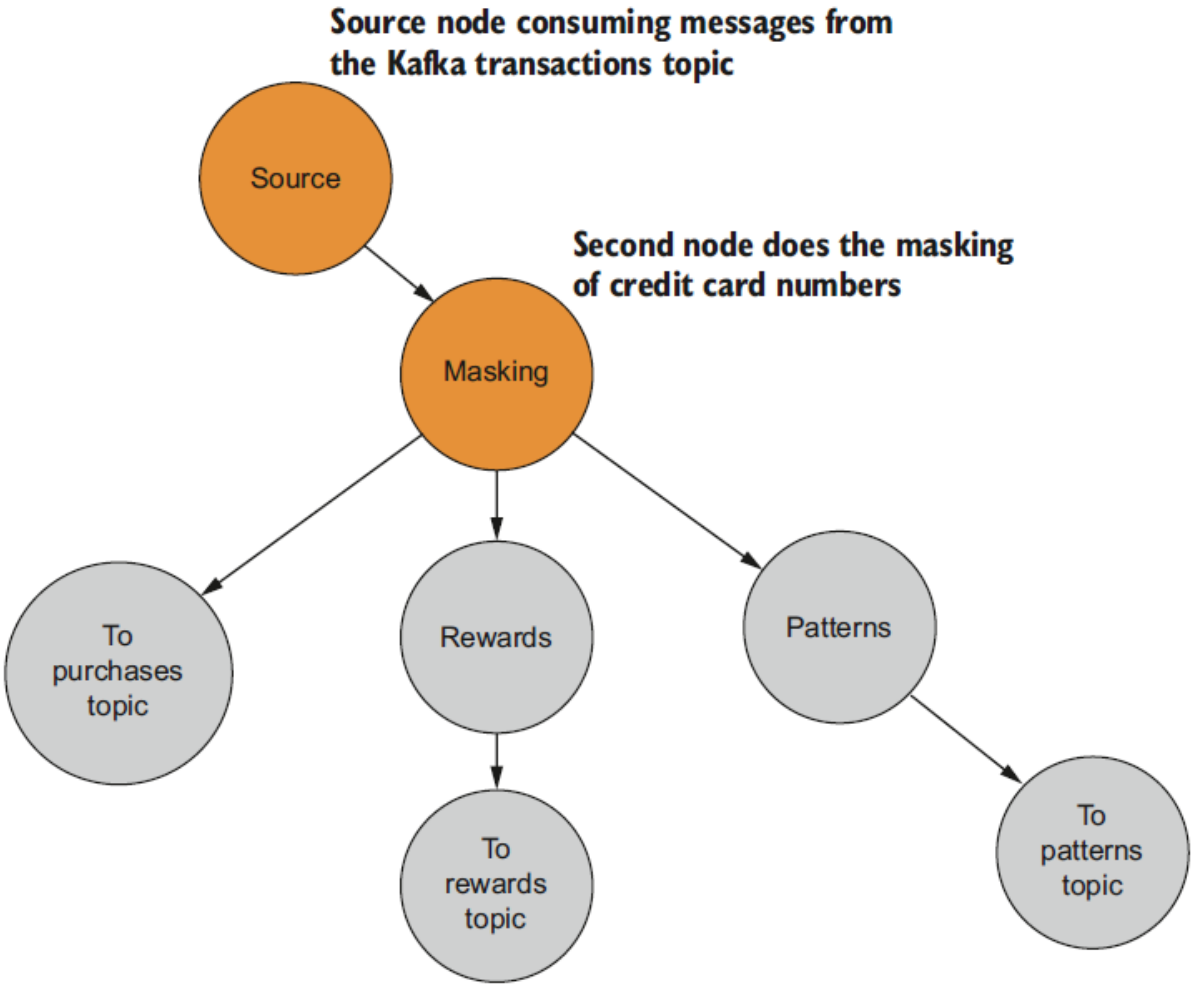


Figure 3.6 The source processor consumes from a Kafka topic, and it feeds the masking processor exclusively, making it the source for the rest of the topology.

Listing 3.5 Building the source node and first processor

```
KStream<String,Purchase> purchaseKStream =
[CA]streamsBuilder.stream("transactions",
[CA]Consumed.with(stringSerde, purchaseSerde))
[CA].mapValues(p -> Purchase.builder(p).maskCreditCard().build());
```

You create the source node with a call to the `StreamsBuilder.stream` method using a default `String` serde, a custom serde for `Purchase` 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 listing 3.5, 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 `Purchase` 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 (`Purchase`), but with a masked credit card number.

Note that when using the `KStream.mapValues` method, the original key is unchanged and isn't factored into mapping a new value. If you wanted to generate a new key/value pair or include the key in producing a new value, you'd use the `KStream.map` method that takes a `KeyValueMapper<K, V, KeyValue<K1, V1>>` instance.

HINTS ABOUT FUNCTIONAL PROGRAMMING

An important concept to keep in mind with the `map` and `mapValues` functions is that they're expected to operate without side effects, meaning the functions don't modify the object or value presented as a parameter. This is because of the functional programming aspects in the `KStream` API. Functional programming is a deep topic, and a full discussion is beyond the scope of this book, but we'll briefly look at two central principles of functional programming here.

The first principle is avoiding state modification. If an object requires a change or update, you pass the object to a function, and a copy or entirely new instance is made, containing the desired changes or updates. In listing 3.5, the lambda passed to `KStream.mapValues` is used to update the `Purchase` object with a masked credit card number. The credit card field on the original `Purchase` object is left unchanged.

The second principle is building complex operations by composing several smaller single-purpose functions together. The composition of functions is a pattern you'll frequently see when working with the `KStream` API.

NOTE **Definition**

For the purposes of this book, I define *functional programming* as a programming approach in which functions are first-class objects. Furthermore, functions are expected to avoid creating side effects, such as modifying state or mutable objects.

BUILDING THE SECOND PROCESSOR

Now you'll build the second processor, responsible for extracting pattern data from a topic, which ZMart can use to determine purchase patterns in regions of the country. You'll also add a sink node responsible for writing the pattern data to a Kafka topic. The construction of these is demonstrated in figure 3.7.

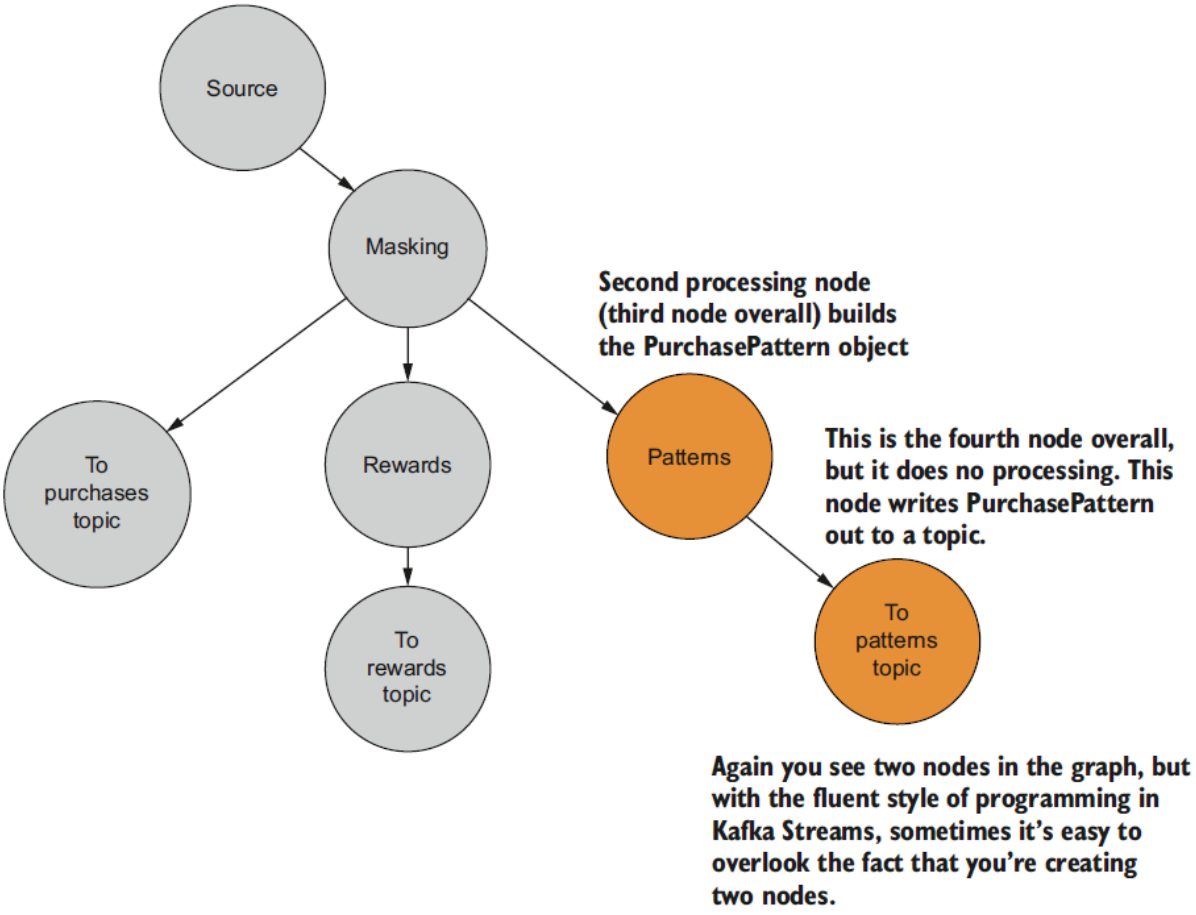


Figure 3.7 The second processor builds purchase-pattern information. The sink node writes the PurchasePattern object out to a Kafka topic.

In listing 3.6, you can see the `purchaseKStream` processor using the familiar `mapValues` call to create a new `KStream` instance. This new `KStream` will start to receive `PurchasePattern` objects created as a result of the `mapValues` call.

Listing 3.6 Second processor and a sink node that writes to Kafka

```
KStream<String, PurchasePattern> patternKStream =
[CA]purchaseKStream.mapValues(purchase ->
[CA]PurchasePattern.builder(purchase).build());

patternKStream.to("patterns",
[CA]Produced.with(stringSerde,purchasePatternSerde));
```

Here, you declare a variable to hold the reference of the new `KStream` instance, because you'll use it to print the results of the stream to the console with a `print` call. This is very useful during development and for debugging. 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`.

The `KStream.to` method is a mirror image of the `KStream.source` method. Instead of setting a source for the topology to read from, the `KStream.to` method 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 overloaded versions in which you can leave out the `Produced` parameter and use the default `Serdes` defined in the configuration. One of the optional parameters you can set with the `Produced` class is `StreamPartitioner`, which we'll discuss next.

BUILDING THE THIRD PROCESSOR

The third processor in the topology is the customer rewards accumulator node shown in figure 3.8, 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.

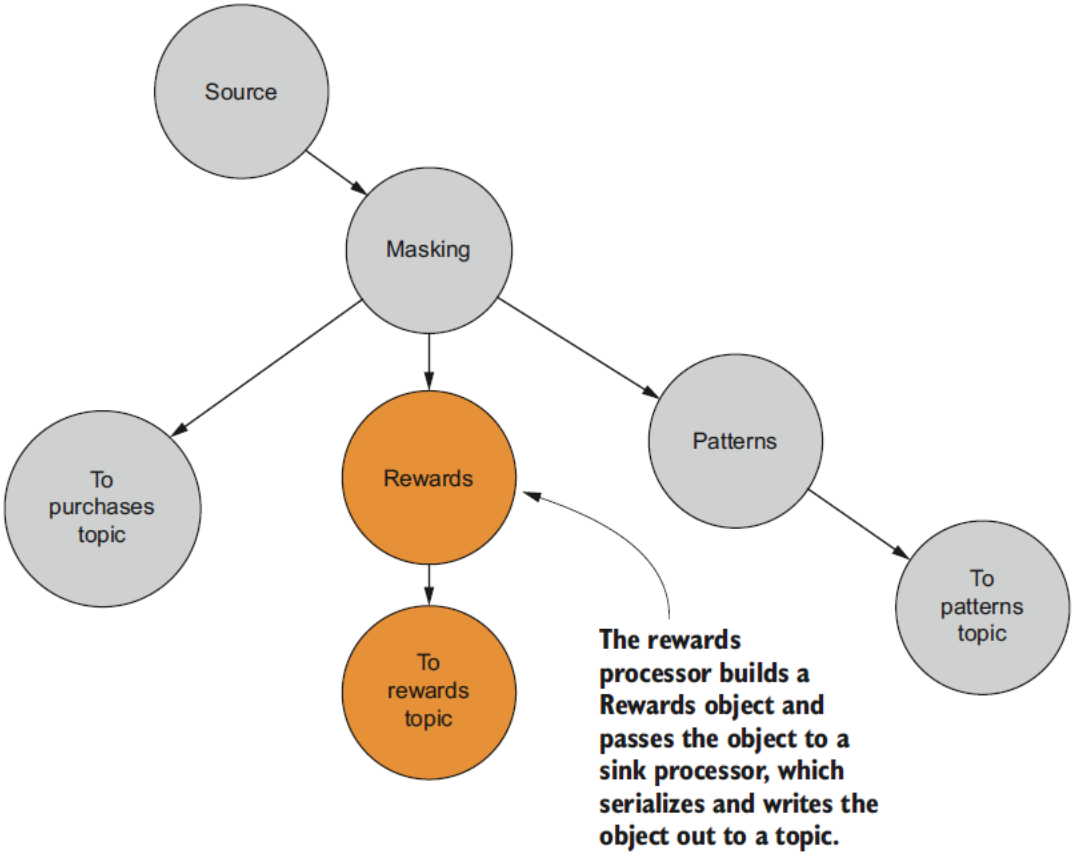


Figure 3.8 The third processor creates the RewardAccumulator object from the purchase data. The terminal node writes the results out to a Kafka topic.

Listing 3.7 Third processor and a terminal node that writes to Kafka

```
KStream<String, RewardAccumulator> rewardsKStream =
[CA]purchaseKStream.mapValues(purchase ->
[CA]RewardAccumulator.builder(purchase).build());
rewardsKStream.to("rewards",
[CA]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 record 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.

BUILDING THE LAST PROCESSOR

Finally, you'll take the first `KStream` you created, `purchaseKStream`, and attach a sink node to write out the raw purchase records (with credit cards masked, of course) to a topic called `purchases`. The `purchases` topic will be used to feed into a NoSQL store such as Cassandra (http://cassandra.apache.org/), Presto (https://prestodb.io/), or Elastic Search (www.elastic.co/webinars/getting-started-elasticsearch) to perform ad hoc analysis. Figure 3.9 shows the final processor.

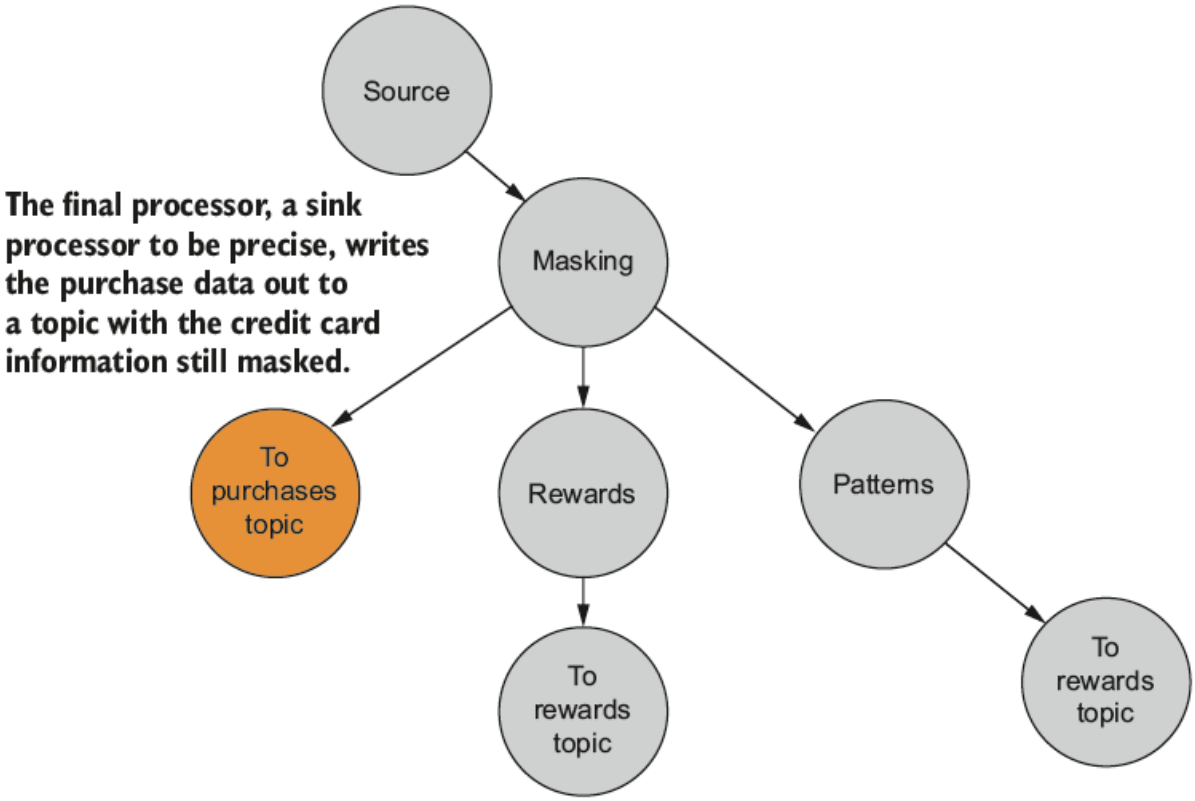


Figure 3.9 The last node writes out the entire purchase transaction to a topic whose consumer is a NoSQL data store.

Listing 3.8 Final processor

```
purchaseKStream.to("purchases", Produced.with(stringSerde, purchaseSerde));
```

Now that you've built the application piece by piece, let's look at the entire application (src/main/java/bbejeck/chapter_3/ZMartKafkaStreamsApp.java). You'll quickly notice it's more complicated than the previous Hello World (the Yelling App) example.

Listing 3.9 ZMart customer purchase `KStream` program

```
public class ZMartKafkaStreamsApp {
```

```
    public static void main(String[] args) {
        // some details left out for clarity

        StreamsConfig streamsConfig = new StreamsConfig(getProperties());

        JsonSerializer<Purchase> purchaseJsonSerializer = new
[CA]JsonSerializer<>();
        JsonDeserializer<Purchase> purchaseJsonDeserializer =      (1)
[CA]new JsonDeserializer<>(Purchase.class);                        (2)
        Serde<Purchase> purchaseSerde =
[CA]Serdes.serdeFrom(purchaseJsonSerializer, purchaseJsonDeserializer);
        //Other Serdes left out for clarity

        Serde<String> stringSerde = Serdes.String();

        StreamsBuilder streamsBuilder = new StreamsBuilder();


        KStream<String,Purchase> purchaseKStream =                 (2)
[CA]streamsBuilder.stream("transactions",                          (3)
[CA]Consumed.with(stringSerde, purchaseSerde))                     (3)
[CA].mapValues(p -> Purchase.builder(p).maskCreditCard().build());      (4)


        KStream<String, PurchasePattern> patternKStream =          (4)
[CA]purchaseKStream.mapValues(purchase ->                          (5)
[CA]PurchasePattern.builder(purchase).build());                    (5)

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


        KStream<String, RewardAccumulator> rewardsKStream =        (4)
[CA]purchaseKStream.mapValues(purchase ->                          (4)
[CA]RewardAccumulator.builder(purchase).build());                  (4)

         rewardsKStream.to("rewards",
[CA]Produced.with(stringSerde,rewardAccumulatorSerde));


        purchaseKStream.to("purchases",                            (5)
[CA]Produced.with(stringSerde,purchaseSerde));                     (5)

        KafkaStreams kafkaStreams =
[CA]new KafkaStreams(streamsBuilder.build(),streamsConfig);
        kafkaStreams.start();

    }
```

(1) Creates the Serde; the data format is JSON.

(2) Builds the source and first processor

(3) Builds the PurchasePattern processor

(4) Builds the RewardAccumulator processor

(5) Builds the storage sink, the topic used by the storage consumer

> **NOTE** I've left out some details in listing 3.9 for 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 `StreamsConfig` instance.
- Build one or more `Serde` instances.
- Construct the processing topology.
- Assemble all the components and start the Kafka Streams program.

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.

3.3.2 Creating a custom Serde

Kafka transfers data in byte array format. Because the data format is JSON, you need to tell Kafka how to convert an object first into JSON and then into a byte array when it sends data to a topic. Conversely, you need to specify how to convert consumed byte arrays into JSON, and then into the object type your processors will use. This conversion of data to and from different formats is why you need a `Serde`. 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 the types of the objects are arbitrary. We'll look at what's involved in building a `Serde` for the `Purchase` class. We won't cover the other `Serde` instances, because they follow the same pattern, just with different types.

Building a `Serde` requires implementations of the `Deserializer<T>` and `Serializer<T>` interfaces. We'll use the implementations in listings 3.10 and 3.11 throughout the examples. Also, you'll use the Gson library from Google to convert objects to and from JSON. Here's the serializer, which you can find in src/main/java/bbejeck/util/serializer/JsonSerializer.java.

Listing 3.10 Generic serializer

```
public class JsonSerializer<T> implements Serializer<T> {

    private Gson gson = new Gson();                                    (1)

    @Override
    public void configure(Map<String, ?> map, boolean b) {

    }

    @Override
    public byte[] serialize(String topic, T t) {
        return gson.toJson(t).getBytes(Charset.forName("UTF-8"));      (2)
    }

    @Override
    public void close() {

    }
}
```

(1) Creates the Gson object

(2) Serializes an object to bytes

For serialization, you first convert an object to JSON, and then get the bytes from the string. To handle the conversions from and to JSON, the example uses Gson (https://github.com/google/gson).

For the deserializing process, you take different steps: create a new string from a byte array, and then use Gson to convert the JSON string into a Java object. This generic deserializer can be found in src/main/java/bbejeck/util/serializer/JsonDeserializer.java.

Listing 3.11 Generic deserializer

```
public class JsonDeserializer<T> implements Deserializer<T> {

    private Gson gson = new Gson();                                    (1)
    private Class<T> deserializedClass;                                (2)

    public JsonDeserializer(Class<T> deserializedClass) {
        this.deserializedClass = deserializedClass;
    }

    public JsonDeserializer() {
    }

    @Override
    @SuppressWarnings("unchecked")
    public void configure(Map<String, ?> map, boolean b) {
        if(deserializedClass == null) {
            deserializedClass = (Class<T>) map.get("serializedClass");
        }
    }

    @Override
    public T deserialize(String s, byte[] bytes) {
```

```
        if(bytes == null){
            return null;
        }

        return gson.fromJson(new String(bytes),deserializedClass);

    }

    @Override
    public void close() {

    }
}
```

1. Creates the Gson object
2. Instance variable of Class to deserialize
3. Deserializes bytes to an instance of expected Class

Now, let's go back to the following lines from listing 3.9:

```
JsonDeserializer<Purchase> purchaseJsonDeserializer =
[CA]new JsonDeserializer<>(Purchase.class);
JsonSerializer<Purchase> purchaseJsonSerializer =
[CA]new JsonSerializer<>();
Serde<Purchase> purchaseSerde =
[CA]Serdes.serdeFrom(purchaseJsonSerializer,purchaseJsonDeserializer);
```

1. Creates the Deserializer for the Purchase class
2. Creates the Serializer for the Purchase class
3. Creates the Serde for Purchase objects

As you can see, a `Serde` object is useful because it serves as a container for the serializer and deserializer for a given object.

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.

3.4 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 it would be good to have a more convenient solution, like the ability to watch data flowing through the topology in the console, as shown in figure 3.10.

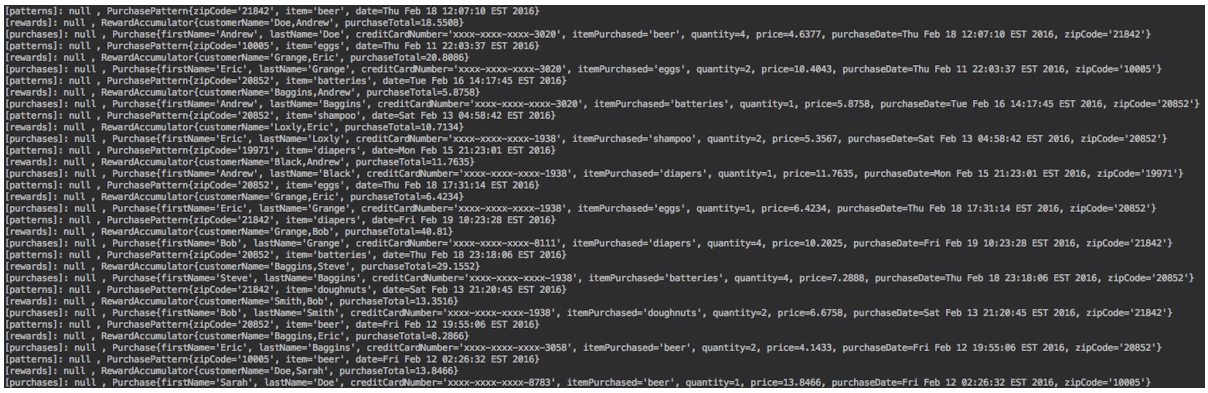


Figure 3.10 A great tool while you're developing is the capacity to print the data that's output from each node to the console. To enable printing to the console, just replace any of the to methods with a call to print.

There's a method on the `KStream` interface that can be useful during development: the `KStream.print` method, which takes an instance of the `Printed<K, V>` class. `Printed` provides two static methods allowing you print to stdout, `Printed.toSysOut()`, or to write results to a file, `Printed.toFile(filePath)`.

Additionally, you can label your printed results by chaining the `withLabel()` method, allowing you to print an initial header with the records. This is useful when you're dealing with results from different processors. It's important that your objects provide a meaningful `toString` implementation to create useful results when printing your stream either to the console or a file.

Finally, if you don't want to use `toString`, or you want to customize how Kafka Streams prints records, there's the `Printed.withKeyValueMapper` method, which takes a `KeyValueMapper` instance so you can format your records in any way you want. The same caveat I mentioned earlier—that you shouldn't modify the original records—applies here as well.

In this book, I focus on printing records to the console for all examples. Here are some examples of using `KStream.print` in listing 3.11:

```
patternKStream.print(Printed.<String, PurchasePattern>toSysOut()
[CA].withLabel("patterns"));

rewardsKStream.print(Printed.<String, RewardAccumulator>toSysOut()
[CA].withLabel("rewards"));

purchaseKStream.print(Printed.<String, Purchase>toSysOut()
[CA].withLabel("purchases"));
```

1. Sets up to print the PurchasePattern transformation to the console
2. Sets up to print the RewardAccumulator transformation to the console
3. Prints the purchase data to the console

Let's take a quick look at the output you'll see on the screen (figure 3.11) and how it can help you during development. With printing enabled, you can run the Kafka Streams application directly from your IDE as you make changes, stop and start the application, and confirm that the output is what you expect. This is no substitute for unit and integration tests, but viewing streaming results directly as you develop is a great tool.

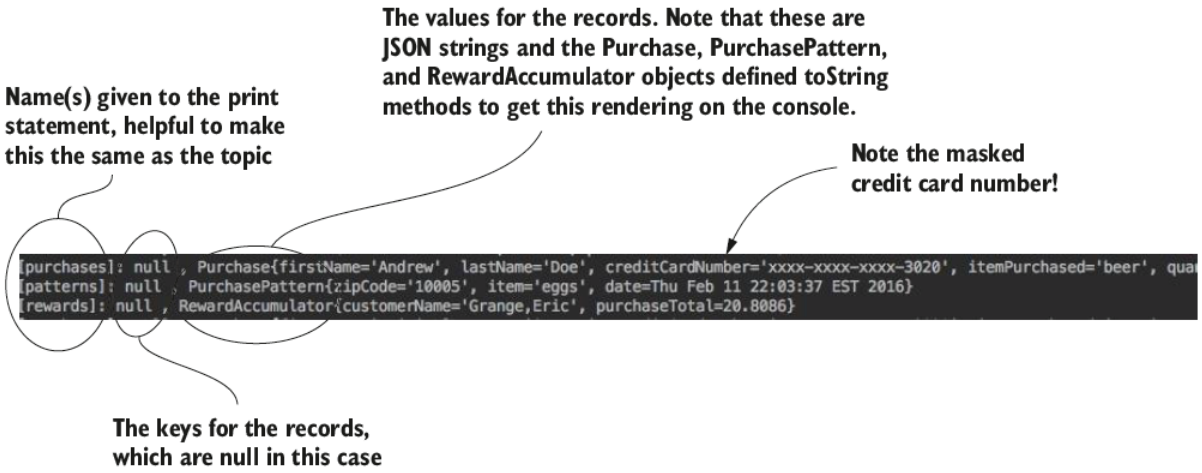


Figure 3.11 This a detailed view of the data on the screen. With printing to the console enabled, you'll quickly see if your processors are working correctly.

One downside of using the `print()` method is that it creates a terminal node, meaning you can't embed it in a chain of processors. You need to have a separate statement. However, there's also the `KStream.peek` method, which takes a `ForeachAction` instance as a parameter and returns a new `KStream` instance. The `ForeachAction` interface has one method, `apply()`, which has a return type of `void`, so nothing from `KStream.peek` is forwarded downstream, making it ideal for operations like printing. You can embed it in a chain of processors without the need for a separate print statement. You'll see the `KStream.peek` method used in this manner in other examples in the book.

3.5 Next steps

At this point, you have your Kafka Streams purchase-analysis program running well. Other applications have also been developed to consume the messages written to the `patterns`, `rewards`, and `purchases` topics, and the results for ZMart have been good. But alas, no good deed goes unpunished. Now that the ZMart executives can see what your streaming program can provide, a slew of new requirements come your way.

3.5.1 New requirements

You now have new requirements for each of the three categories of results you're producing. The good news is that you'll still use the same source data. You're being asked to refine, and in some cases further break down, the data you're providing. The new requirements may be able to be applied to current topics, or they may require you to create entirely new topics:

- Purchases under a certain dollar amount need to be filtered out. Upper management isn't much interested in the small purchases for general daily articles.
- ZMart has expanded and has bought an electronics chain and a popular coffee house chain. All purchases from these new stores will flow through the streaming application you've set up. You need to send the purchases from these new subsidiaries to their topics.
- The NoSQL solution you've chosen stores items in key/value format. Although Kafka also uses key/value pairs, the records coming into your Kafka cluster don't have keys defined. You need to generate a key for each record before the topology forwards it to the `purchases` topic.

More requirements will inevitably come your way, but you can start to work on the current set of new requirements now. If you look through the `KStream` API, you'll be relieved to see that there are several methods already defined that will make fulfilling these new demands easy.

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_3/ZMartKafkaStreamsAdvancedReqsApp.java.

FILTERING PURCHASES

Let's start with filtering out purchases that don't reach the minimum threshold. To remove low-dollar purchases, you'll need to insert a filter-processing node between the `KStream` instance and the sink node. You'll update the processor topology graph as shown in figure 3.12.

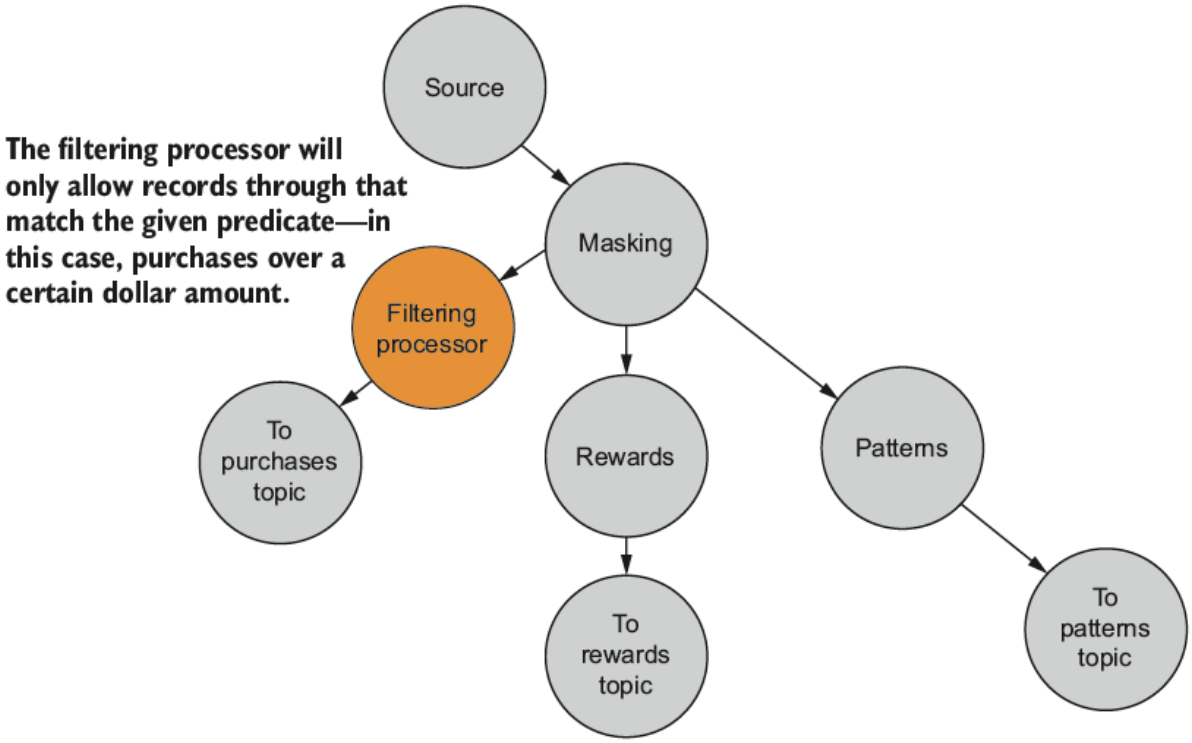


Figure 3.12 You're placing a processor between the masking processor and the terminal node that writes to Kafka. This filtering processor will drop purchases under a given dollar amount.

You can use the `KStream` method, which takes a `Predicate<K,V>` instance as a parameter. Although you're chaining method calls together here, you're creating a new processing node in the topology.

Listing 3.12 Filtering on `KStream`

```
KStream<Long, Purchase> filteredKStream =
  [CA]purchaseKStream((key, purchase) ->
  [CA]purchase.getPrice() > 5.00).selectKey(purchaseDateAsKey);
```

This code filters purchases that are less than $5.00 and selects the purchase date as a long value for a key.

The `Predicate` interface 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. Again, you can use a Java 8 lambda in place of a concrete type defined in the `KStream` API.

NOTE

Definition

If you're familiar with functional programming, you should feel right at home with the `Predicate` interface. If the term *predicate* is new to you, it's nothing more than a given statement, such as `x < 100`. An object either matches the predicate statement or doesn't.

Additionally, you want to use the purchase timestamp as a key, so you use the `selectKey` processor, which uses the `KeyValueMapper` mentioned in section 3.4 to extract the purchase date as a long value. I cover details about selecting the key in the section "Generating a key."

A mirror-image function, `KStreamNot`, performs the same filtering functionality but in reverse. Only records that *don't* match the given predicate are processed further in the topology.

SPLITTING/BRANCHING THE STREAM

Now you need to split the stream of purchases into separate streams that can write to different topics. Fortunately, the `KStream.branch` method is perfect. The `KStream.branch` method takes an arbitrary number of `Predicate` instances and returns an array of `KStream` instances. The size of the returned array matches the number of predicates supplied in the call.

In the previous change, you modified an existing leaf on the processing topology. With this requirement to branch the stream, you'll create brand-new leaf nodes on the graph of processing nodes, as shown in figure 3.13.

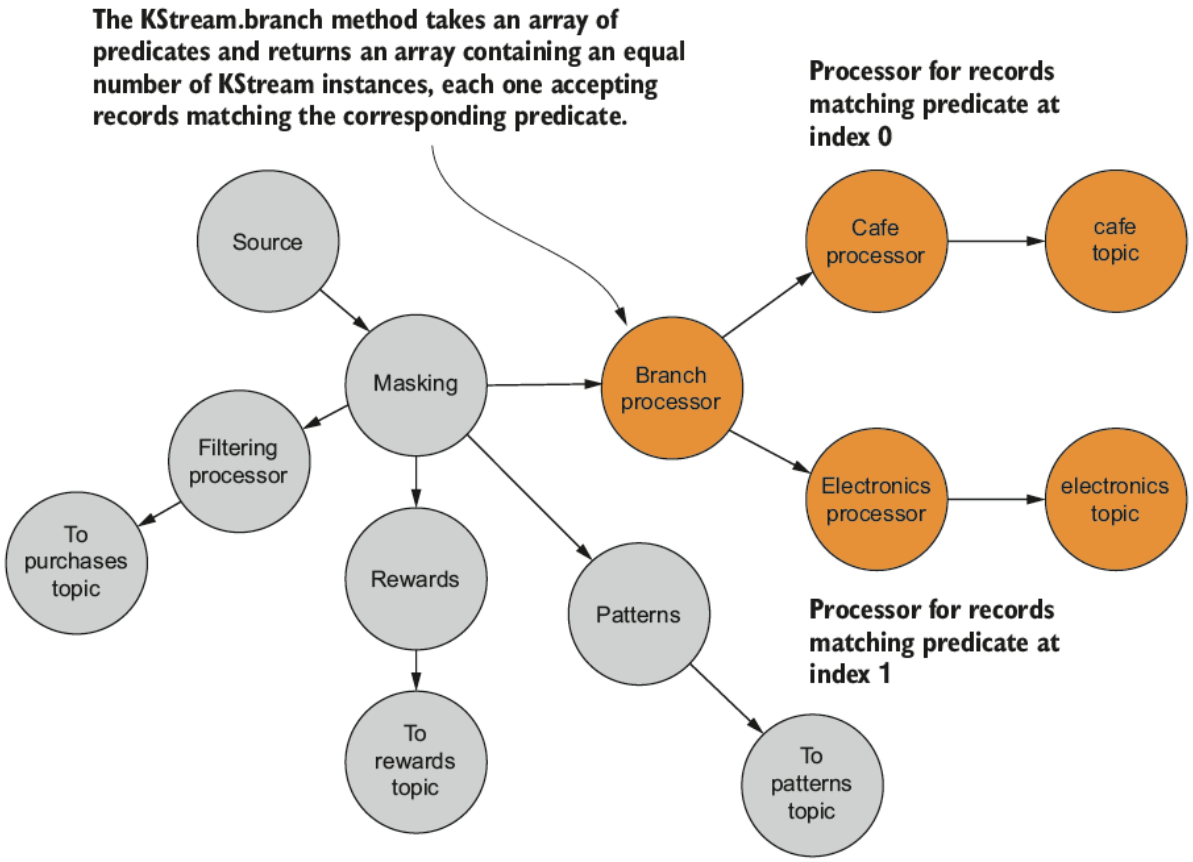


Figure 3.13 The branch processor splits the stream into two: one stream consists of purchases from the cafe, and the other stream contains purchases from the electronics store.

As records from the original stream flow through the branch processor, each record is matched against the supplied predicates in the order that they're provided. The processor assigns records to a stream on the first match; no attempts are made to match additional predicates.

The branch processor drops records if they don't match any of the given predicates. The order of the streams in the returned array matches the order of the predicates provided to the `branch()` method. A separate topic for each department may not be the only approach, but we'll stick with this for now. It satisfies the requirement, and it can be revisited later.

Listing 3.13 Splitting the stream

```
Predicate<String, Purchase> isCoffee =                    (1)
[CA](key, purchase) ->                                    (1)
[CA]purchase.getDepartment().equalsIgnoreCase("coffee");  (1)

Predicate<String, Purchase> isElectronics =
[CA](key, purchase) ->
```

```
[CA]purchase.getDepartment().equalsIgnoreCase("electronics");

int coffee = 0;                                                    (2)
int electronics = 1;

KStream<String, Purchase>[] kstreamByDept =                        (3)
[CA]purchaseKStream.branch(isCoffee, isElectronics);               (3)

kstreamByDept[coffee].to( "coffee", Produced.with(stringSerde, purchaseSerde));
kstreamByDept[electronics].to("electronics",                       (4)
[CA]Produced.with(stringSerde, purchaseSerde));                    (4)
```

(1) 2 CO7-3)) Creates the predicates as Java 8 lambdas

(2) Labels the expected indices of the returned array

(3) Calls branch to split the original stream into two streams

(4) Writes the results of each stream out to a topic

WARNING The example in listing 3.13 sends records to several different topics. Although Kafka can be configured to automatically create topics when it attempts to produce or consume for the first time from nonexistent topics, it's not a good idea to rely on this mechanism. If you rely on autocreating topics, 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 level of partitions, and the replication factor ahead of time, and create them before running your Kafka Streams application.

In listing 3.13, you define the predicates ahead of time, because passing four lambda expression parameters would be a little unwieldy. The indices of the returned array are also labeled, to maximize readability.

SIDEBAR

Splitting vs. partitioning streams

Although *splitting* and *partitioning* may seem like similar ideas, they're unrelated in Kafka and Kafka Streams. Splitting a stream with the `KStream.branch` method results in creating one or more streams that could ultimately send records to another topic. Partitioning is how Kafka distributes messages for one topic across servers, and aside from configuration tuning, it's the principal means of achieving high throughput in Kafka.

This example demonstrates the power and flexibility of Kafka Streams. You've been able to take the original stream of purchase transactions and split them into four streams with very few lines of code. Also, you're starting to build up a more complex processing

topology, all while reusing the same source processor.

So far, so good. You've met two of the three new requirements with ease. Now it's time to implement the last additional requirement, generating a key for the purchase record to be stored.

GENERATING A KEY

Kafka messages are in key/value pairs, so all records flowing through a Kafka Streams application are key/value pairs as well. But there's no requirement stating that keys can't be null. In practice, if there's no need for a particular key, having a null key will reduce the overall amount of data that travels the network. All the records flowing into the ZMart Kafka Streams application have null keys.

That's been fine, until you realize that your NoSQL storage solution stores data in key/value format. You need a way to create a key from the `Purchase` data before it gets written out to the `purchases` topic. You certainly could use `KStream.map` to generate a key and return a new key/value pair (where only the key would be new), but there's a more succinct `KStream.selectKey` method that returns a new `KStream` instance that produces records with a new key (possibly a different type) and the same value. This change to the processor topology is similar to filtering, in that you add a processing node between the filter and the sink processor, shown in figure 3.14.

Listing 3.14 Generating a new key

```
KeyValueMapper<String, Purchase, Long> purchaseDateAsKey =
[CA](key, purchase) -> purchase.getPurchaseDate().getTime();

KStream<Long, Purchase> filteredKStream =
[CA]purchaseKStream((key, purchase) ->
[CA]purchase.getPrice() > 5.00).selectKey(purchaseDateAsKey);

filteredKStream.print(Printed.<Long,
[CA]Purchase>toSysOut().withLabel("purchases"));
filteredKStream.to("purchases",
[CA]Produced.with(Serdes.Long(),purchaseSerde));
```

1. The KeyValueMapper extracts the purchase date and converts to a long.
2. 4 CO8-5)) Filters out purchases and selects the key in one statement
3. Prints the results to the console
4. Materializes the results to a Kafka topic

To create the new key, you take the purchase date and convert it to a long. Although you could pass a lambda expression, it's assigned to a variable here to help with readability. Also, note that you need to change the serde type used in the `KStream.to` method, because you've changed the type of the key.

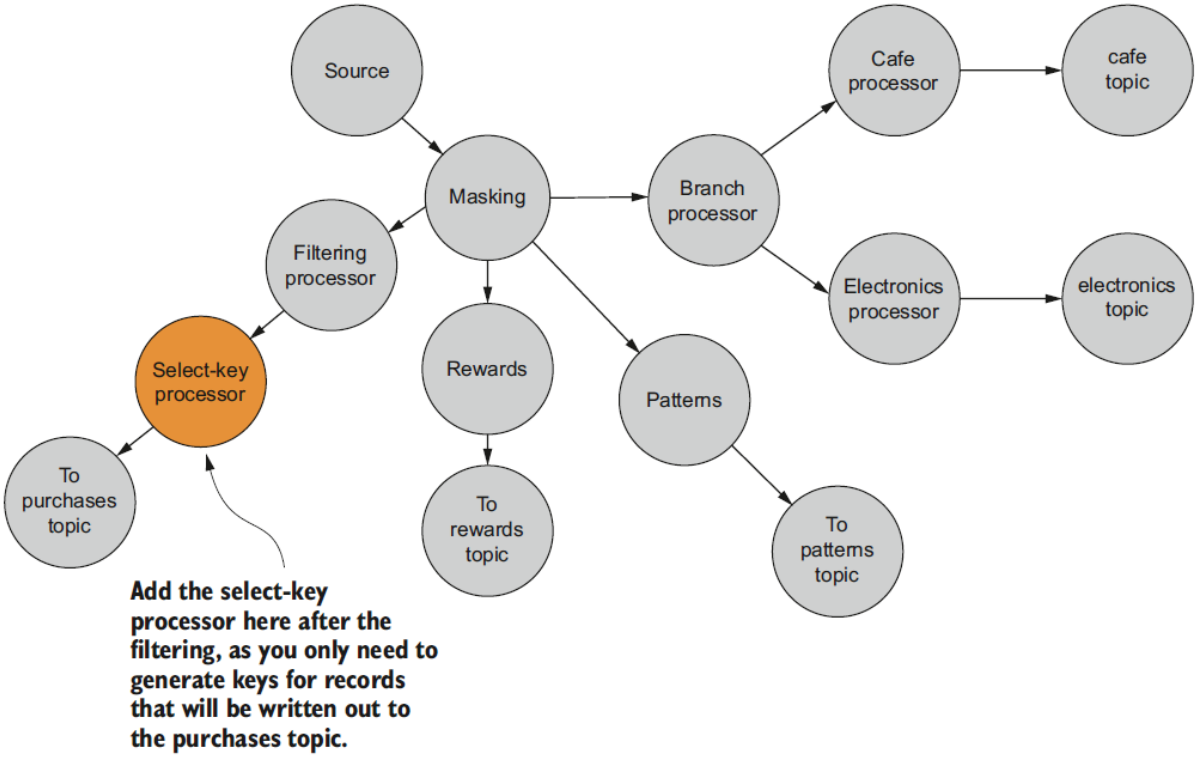


Figure 3.14 The NoSQL data store will use the purchase date as a key for the data it stores. The new selectKey processor will extract the purchase date to be used as a key, right before you write the data to Kafka.

This is a simple example of mapping to a new key. Later, in another example, you'll select keys to enable joining separate streams. Also, all the examples up until this point have been stateless, but there are several options for stateful transformations as well, which you'll see a little later on.

3.5.2 Writing records outside of Kafka

The security department at ZMart has approached you. Apparently, in one of the stores, there's a suspicion of fraud. There have been reports that a store manager is entering invalid discount codes for purchases. Security isn't sure what's going on, but they're asking for your help.

The security folks don't want this information to go into a topic. You talk to them about securing Kafka, about access controls, and about how you can lock down access to a

topic, but the security folks are standing firm. These records need to go into a relational database where they have full control. You sense this is a fight you can't win, so you relent and resolve to get this task done as requested.

FOREACH ACTIONS

The first thing you need to do is create a new `KStream` that filters results down to a single employee ID. Even though you have a large amount of data flowing through your topology, this filter will reduce the volume to a tiny amount.

Here, you'll use `KStream` with a predicate that looks to match a specific employee ID. This filter will be completely separate from the previous filter, and it'll be attached to the source `KStream` instance. Although it's entirely possible to chain filters, you won't do that here; you want full access to the data in the stream for this filter.

Next, you'll use a `KStream.foreach` method, as shown in figure 3.15. `KStream.foreach` takes a `ForeachAction<K, V>` instance, and it's another example of a terminal node. It's a simple processor that uses the provided `ForeachAction` instance to perform an action on each record it receives.

Listing 3.15 `Foreach` operations

```
ForeachAction<String, Purchase> purchaseForeachAction = (key, purchase) ->
[CA]SecurityDBService.saveRecord(purchase.getPurchaseDate(),
[CA]purchase.getEmployeeId(), purchase.getItemPurchased());

purchaseKStream((key, purchase) ->
[CA]purchase.getEmployeeId()
[CA].equals("1234567"))
[CA].foreach(purchaseForeachAction);
```

`ForeachAction` uses a Java 8 lambda (again), and it's stored in a variable, `purchaseForeachAction`. This requires an extra line of code, but the clarity gained by doing so more than makes up for it. On the next line, another `KStream` instance sends the filtered results to the `ForeachAction` defined directly above it.

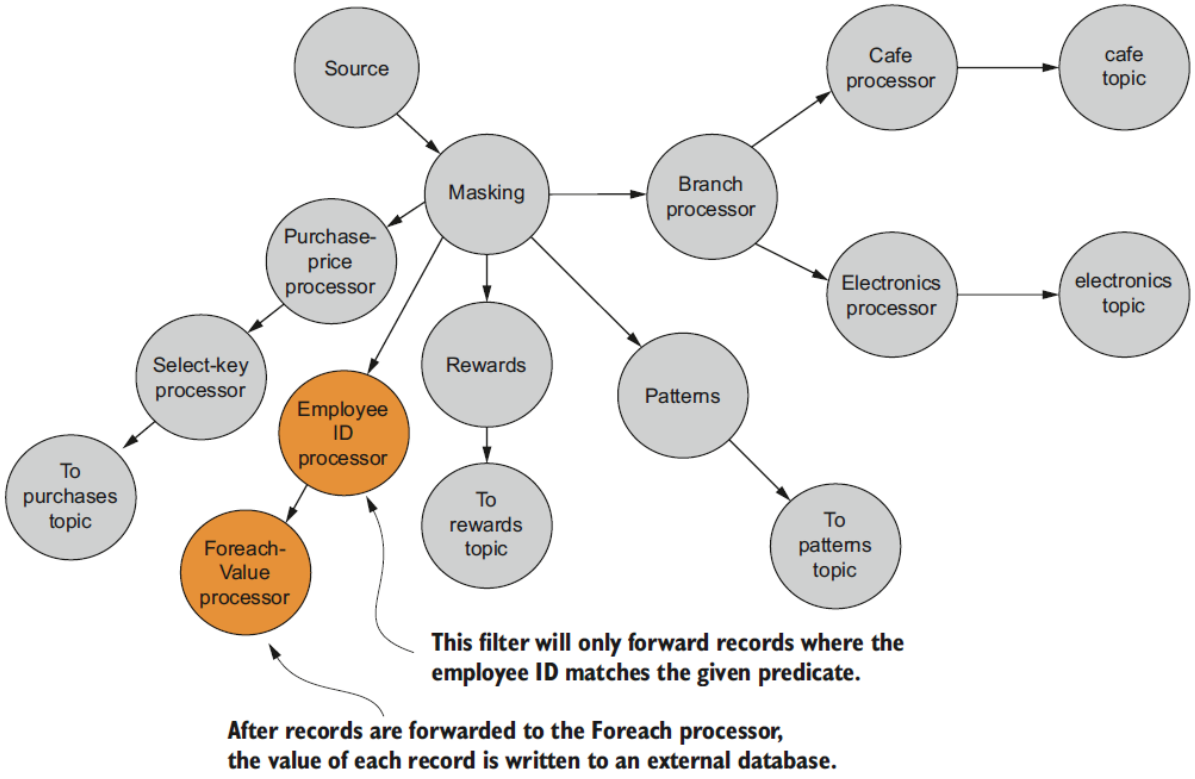


Figure 3.15 To write purchases involving a given employee outside of the Kafka Streams application, you'll first add a filter processor to extract purchases by employee ID, and then you'll use a foreach operator to write each record to an external relational database.

Note that `KStream.foreach` is stateless. If you need state to perform some action for each record, you can use the `KStream.process` method. The `KStream.process` method will be discussed in the next chapter when you add state to a Kafka Streams application.

If you step back and look at what you've accomplished so far, it's pretty impressive, considering the amount of code written. Don't get too comfortable, though, because upper management at ZMart has taken notice of your productivity. More changes and refinements to the purchase-streaming analysis program are coming.

3.6 Summary

- 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.
- 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. You used predicates in the filter function to prevent records that didn't match a given predicate from being forwarded in the topology.
- 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.
- You can modify an existing key or create a new one using the `KStream.selectKey` method.

In the next chapter, we'll start to look at state, the required properties for using state with a steaming application, and why you might need to add state at all. Then you'll add state to a `KStream` application, first by using stateful versions of `KStream` methods you've seen in this chapter (`KStream.mapValues`). For a more advanced example, you'll perform joins between two different streams of purchases to help ZMart improve customer service.

4 Streams and state

This chapter covers

- Applying stateful operations to Kafka Streams
- Using state stores for lookups and remembering previously seen data
- Joining streams for added insight
- How time and 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 purchases at ZMart locations. 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.

Another example of state we'll discuss is 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.

4.1 Thinking of events

When it comes to event processing, events sometimes require no further information or context. At other times, an event on its own may be understood in a literal sense, but without some added context, you might miss the significance of what is occurring; you might think of the event in a whole new light, given some additional information.

An example of an event that doesn't require additional information is the attempted use of a stolen credit card. The transaction is canceled immediately once the stolen card's use is detected. You don't need any additional information to make that decision.

But sometimes a singular event won't give you enough information to make a decision. Consider a series of stock purchases by three individual investors within a short period. On the face of it, there's nothing about the purchases of XYZ Pharmaceutical stock, shown in figure 4.1, that would give you pause. Investors buying shares of the same stock is something that happens every day on Wall Street.

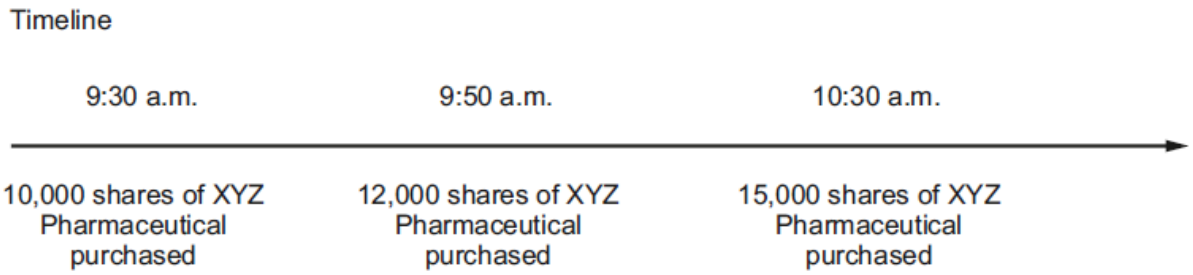


Figure 4.1 Stock transactions without any extra information don't look like anything out of the ordinary.

Now let's add some context. Within a short period of the individual stock purchases, XYZ Pharmaceutical announced government approval for a new drug, which sent the stock price to historic highs. Additionally, those three investors had close ties to XYZ Pharmaceutical. Now the transactions, shown in figure 4.2, can be viewed in a whole new light.

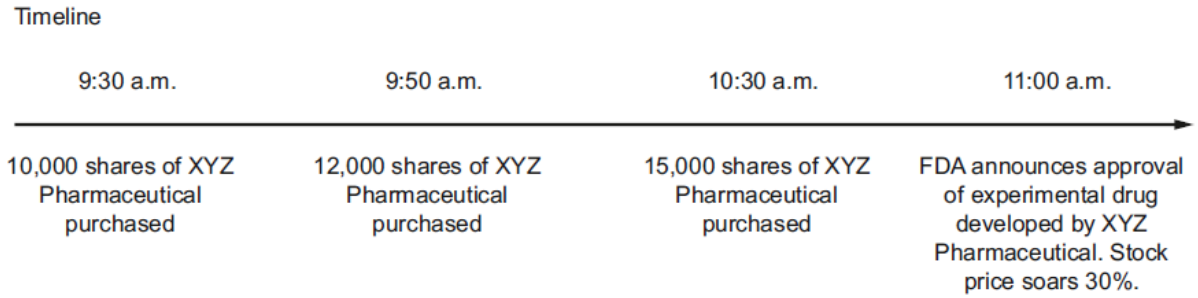


Figure 4.2 When you add some additional context about the timing of the stock purchases, you'll see them in an entirely new light.

The timing of these purchases and the information release raises some questions. Were these investors leaked information ahead of time? Or do the transactions represent one investor with inside information trying to cover their tracks?

4.1.1 Streams need state

The preceding fictional scenario illustrates something that most of us already know instinctively. Sometimes it's easy to reason about what's going on, but usually you need some context to make good decisions. When it comes to stream processing, we call that added context *state*.

At first glance, the notions of state and stream processing may seem to be at odds with each other. Stream processing implies a constant flow of discrete events that don't have much to do with each other and need to be dealt with as they occur. The notion of state might evoke images of a static resource, such as a database table.

In actuality, you can view these as one and the same. But the rate of change in a stream is potentially much faster and more frequent than in a database table.[8]

Footnote 8 Jay Kreps, "Why Local State Is a Fundamental Primitive in Stream Processing," http://mng.bz/sfoI.

You don't always need state to work with streaming data. In some cases, you may have discrete events or records that carry enough information to be valuable on their own. But more often than not, the incoming stream of data will need enrichment from some sort of store, either using information from events that arrived before, or joining related events with events from different streams.

4.2 Applying stateful operations to Kafka Streams

In this section, we'll look at how you can add a stateful operation to an existing stateless one to improve the information collected by our application. You're going to modify the original topology from chapter 3, shown in figure 4.3 to refresh your memory.

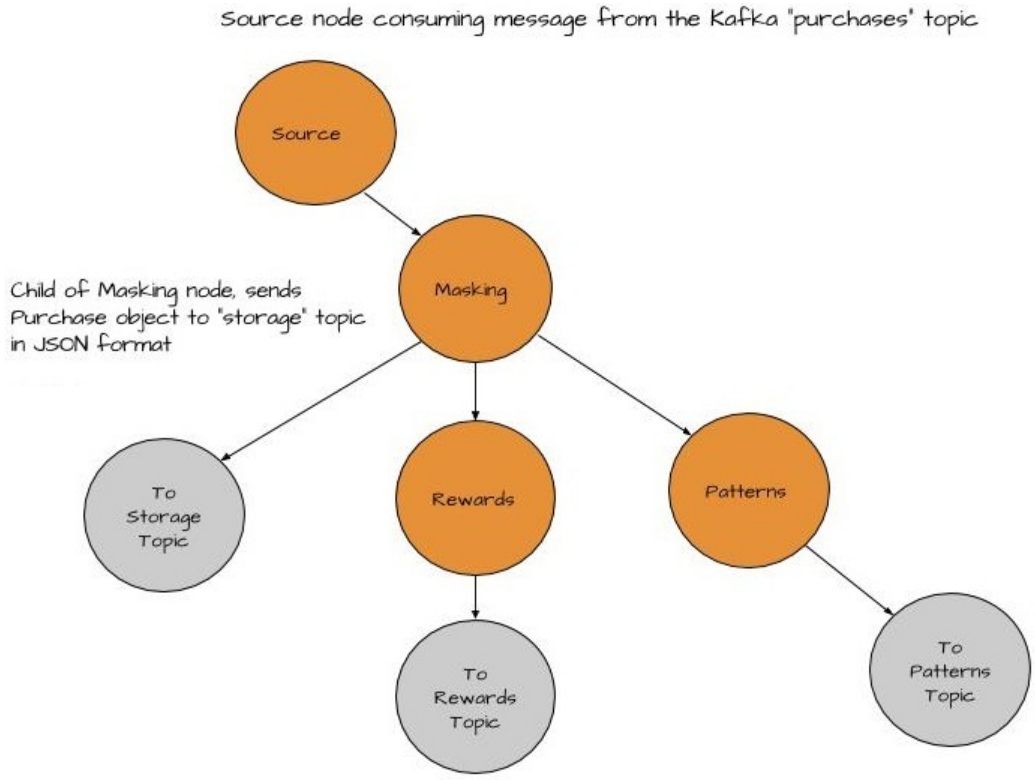


Figure 4.3 Here's another look at the topology from chapter 3.

In this topology, you produced a stream of purchase-transaction events. One of the processing nodes in the topology calculated reward points for customers based on the amount of the sale. But in that processor, you just calculated the total number of points for the single transaction and forwarded the results.

If you added some state to the processor, you could keep track of the cumulative number of reward points. Then, the consuming application at ZMart would need to check the total and send out a reward if needed.

Now that you have a basic idea of how state can be useful in Kafka Streams (or any other streaming application), let's look at some concrete examples. You'll start with transforming the stateless rewards processor into a stateful processor using `transformValues`. You'll keep track of the total bonus points achieved so far and the amount of time between purchases, to provide more information to downstream consumers.

4.2.1 The transformValues processor

The most basic of the stateful functions is `KStream.transformValues`. Figure 4.4 illustrates how the `KStream.transformValues()` method operates.

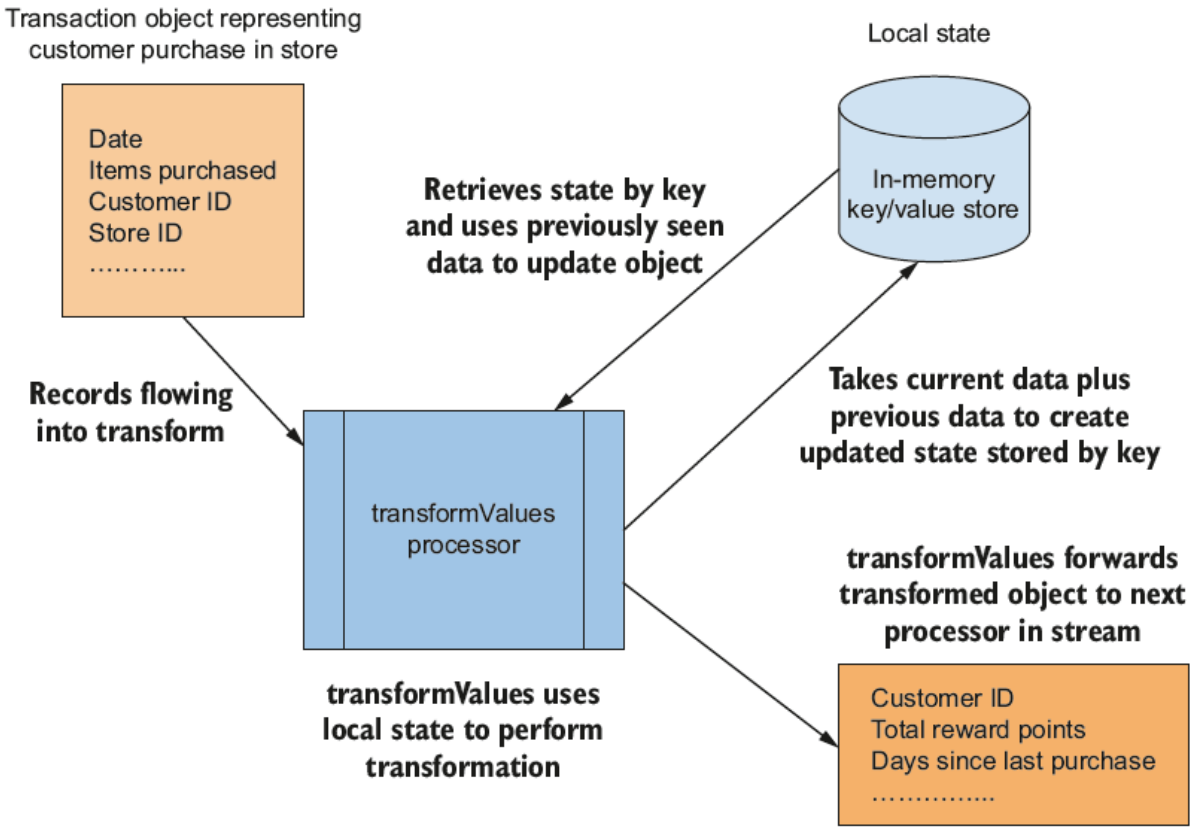


Figure 4.4 The transformValues processor uses information stored in local state to update incoming records. In this case, the customer ID is the key used to retrieve and store the state for a given record.

This method is semantically the same as `KStream.mapValues()`, with a few exceptions. One difference is that `transformValues` has access to a `StateStore` instance to accomplish its task. The other difference is its ability to schedule operations to occur at regular intervals via a `punctuate()` method. The `punctuate()` method will be discussed in detail when we cover the Processor API in chapter 6.

4.2.2 Stateful customer rewards

The rewards processor from the chapter 3 topology (see figure 4.3) for ZMart extracts information for customers belonging to ZMart's rewards program. Initially, the rewards processor used the `KStream.mapValues()` method to map the incoming `Purchase` object into a `RewardAccumulator` object.

The `RewardAccumulator` object originally consisted of just two fields, the customer ID

and the purchase total for the transaction. Now, the requirements have changed some, and points are being associated with the ZMart rewards program:

```
public class RewardAccumulator {

    private String customerId;          (1)
    private double purchaseTotal;       (2)
    private int currentRewardPoints;    (3)

  //details left out for clarity
}
```

(1) Customer ID

(2) Total dollar amount of purchase

(3) Current number of reward points

Whereas before, an application read from the rewards topic and calculated customer achievements, now management wants the point system to be maintained and calculated by the streaming application. Additionally, you need to capture the amount of time between the customer's current and last purchase.

When the application reads records from the rewards topic, the consuming application will only need to check whether the total points are above the threshold to distribute an award. To meet this new goal, you can add the `totalRewardPoints` and `daysFromLastPurchase` fields to the `RewardAccumulator` object, and use the local state to keep track of accumulated points and the last date of purchase. Here's the refactored `RewardAccumulator` code (found in src/main/java/bbejeck/model/RewardAccumulator.java; source code can be found on the book's website here: https://manning.com/books/kafka-streams-in-action) needed to support these changes.

Listing 4.1 Refactored `RewardAccumulator` object

```
public class RewardAccumulator {

    private String customerId;
    private double purchaseTotal;
    private int currentRewardPoints;
    private int daysFromLastPurchase;
    private long totalRewardPoints;     (1)


  //details left out for clarity
}
```

❶ Field added for tracking total points

The updated rules for the purchase program are simple. The customer earns a point per dollar, and transaction totals are rounded down to the nearest dollar. The overall structure of the topology won't change, but the rewards-processing node will change from using the `KStream.mapValues()` method to using `KStream.transformValues`. Semantically, these two methods operate the same way, in that you still map the `Purchase` object into a `RewardAccumulator` object. The difference lies in the ability to use local state to perform the transformation.

Specifically, you'll take two main two steps:

- Initialize the value transformer.
- Map the `Purchase` object to a `RewardAccumulator` using state.

The `KStream.transformValues()` method takes a `ValueTransformerSupplier<V, R>` object, which supplies an instance of the `ValueTransformer<V, R>` interface. Your implementation of the `ValueTransformer` will be `PurchaseRewardTransformer<Purchase, RewardAccumulator>`. For the sake of clarity, I won't reproduce the entire class here in the text. Instead, we'll walk through the important methods for the example application. Also note that these code snippets aren't meant to stand alone, and some details will be left out for clarity. The full code can be found in the chapter source code (found on the book's website here: https://manning.com/books/kafka-streams-in-action). Let's move on and initialize the processor.

4.2.3 Initializing the value transformer

The first step is to set up or create any instance variables in the transformer `init()` method. In the `init()` method, you retrieve the state store created when building the processing topology (we'll cover how you add the state store in section 4.7).

Listing 4.2 `init()` method

```
private KeyValueStore<String, Integer> stateStore;    ❶

private final String storeName;
private ProcessorContext context;

public void init(ProcessorContext context) {
    this.context = context;        ❷
```

```
    stateStore = (KeyValueStore)
[CA]this.context.getStateStore(storeName);
}
```

(2) (3)

(1) Instance variables

(2) Sets a local reference to ProcessorContext

(3) Retrieves the StateStore instance by storeName variable. storeName is set in the constructor.

Inside the transformer class, you cast to a `KeyValueStore` type. You're not concerned with the implementation inside the transformer at this point, just that you can retrieve values by key (more on state store implementation types in the next section).

There are other methods (such as `punctuate()` and `close`()) not listed here that belong to the `ValueTransformer` interface. We'll discuss `punctuate` and `close` when we discuss the Processor API in chapter 6.

4.2.4 Mapping the Purchase object to a RewardAccumulator using state

Now that you've initialized the processor, you can move on to transforming a `Purchase` object using state. A few simple steps for performing the transformation are as follows:

1. Check for points accumulated so far by customer ID.
2. Sum the points for the current transaction and present the total.
3. Set the reward points on the `RewardAccumulator` to the new total amount.
4. Save the new total points by customer ID in the local state store.

Listing 4.3 Transforming `Purchase` using state

```
public RewardAccumulator transform(Purchase value) {
    RewardAccumulator rewardAccumulator =
[CA]RewardAccumulator.builder(value).build();
    Integer accumulatedSoFar =
[CA]stateStore.get(rewardAccumulator.getCustomerId());

    if (accumulatedSoFar != null) {
         rewardAccumulator.addRewardPoints(accumulatedSoFar);
    }
    stateStore.put(rewardAccumulator.getCustomerId(),
                   rewardAccumulator.getTotalRewardPoints());

    return rewardAccumulator;
}
```

(1) (2) (2) (3) (3)

(1) Builds the RewardAccumulator object from Purchase

2. Retrieves the latest count by customer ID
3. If an accumulated number exists, adds it to the current total
4. Stores the new total points in stateStore
5. Returns the new accumulated rewards points

In the `transform()` method, you first map a `Purchase` object into the `RewardAccumulator`—this is the same operation used in the `mapValues()` method. In the next few lines, the state gets involved in the transformation process. You do a lookup by key (customer ID) and add any points accumulated so far to the points from the current purchase. Then, you place the new total in the state store until it's needed again.

All that's left is to update the rewards processor. But before you do, you need to consider the fact that you're accessing all sales by customer ID. Gathering information per sale for a given customer implies that all transactions for that customer are on the same partition. But because the transactions come into the application without a key, the producer assigns the transactions to partitions in a round-robin fashion. We covered round-robin partition assignment in chapter 2, but it's worth reviewing it here again—see figure 4.5.

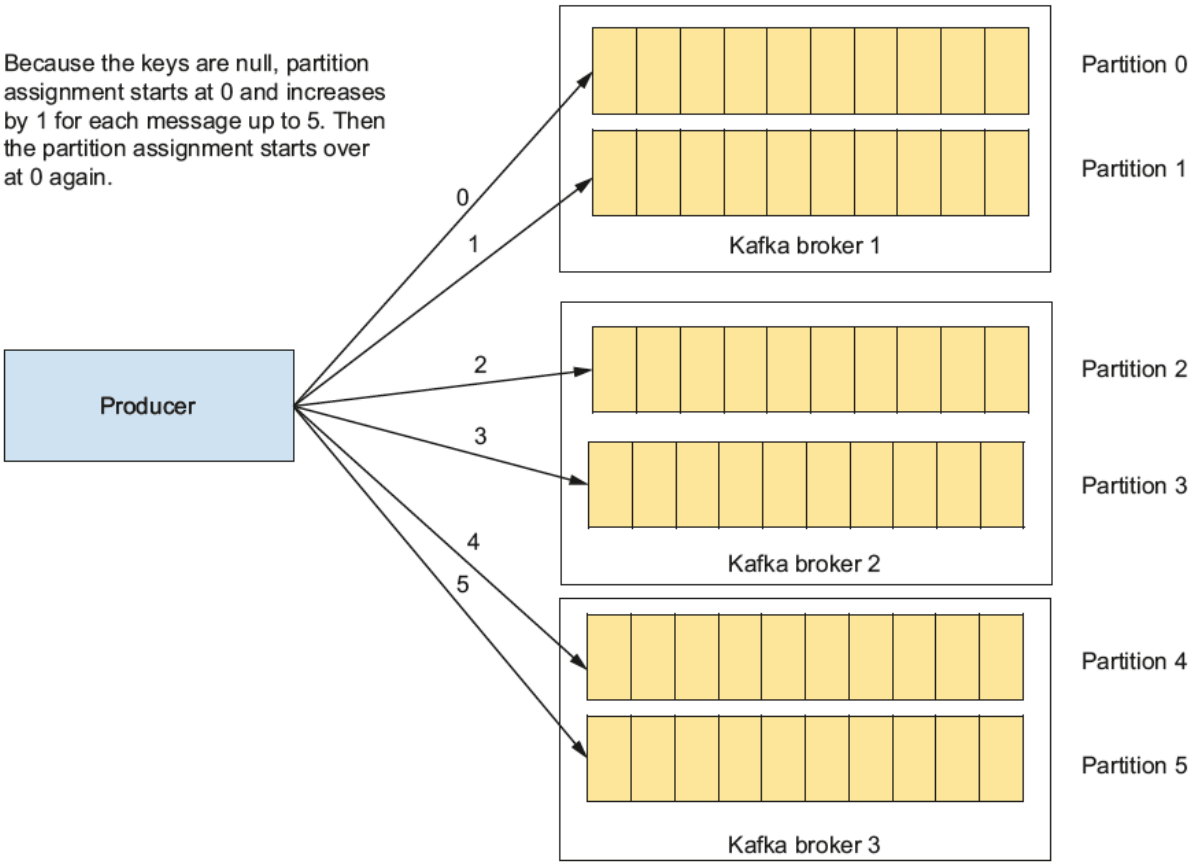


Figure 4.5 A Kafka producer distributes records evenly (round-robin) when the keys are null.

You'll have an issue here (unless you're using topics with only one partition). Because the key isn't populated, round-robin assignment means the transactions for a given customer won't land on the same partition.

Placing customer transactions with the same ID on the same partition is important, because you need to look up records by ID in the state store. Otherwise, you'll have customers with the same ID spread across different partitions, requiring you to look up the same customer in multiple state stores. (This statement could be interpreted to mean that each partition has its own state store, but that's not the case. Partitions are assigned to a `StreamTask`, and each `StreamTask` has its own state store.)

The way to solve this problem is to repartition the data by customer ID. We'll look at how to do this next.

REPARTITIONING THE DATA

First, let's have a general discussion on how repartitioning works (see figure 4.6). To repartition records, first you may modify or change the key on the original record, and then you write out the record to a new topic. Next, you consume those records again; but as a result of repartitioning, those records may come from different partitions than they were in originally.

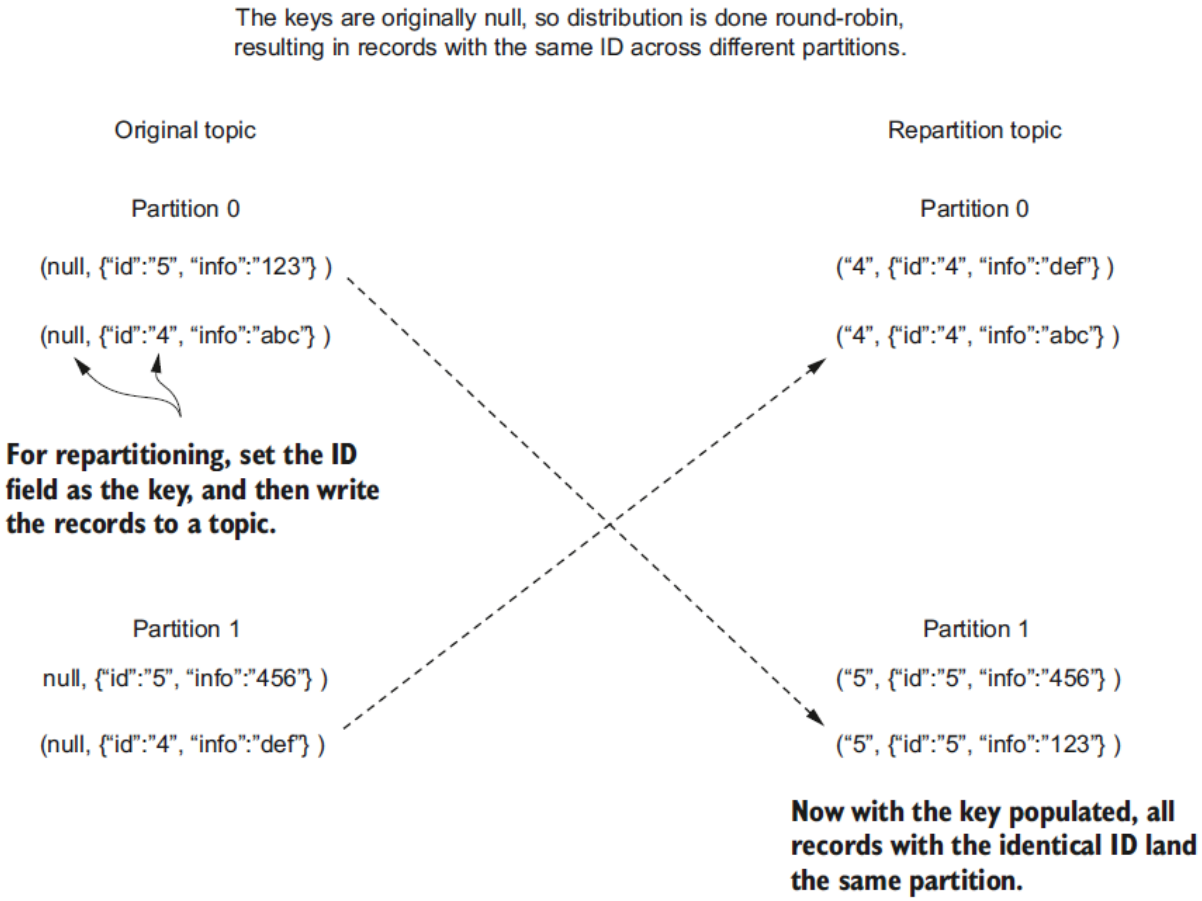


Figure 4.6 Repartitioning: changing the original key to move records to a different partition

Although, in this simple example, you replaced the null key with a concrete value, repartitioning need not always change the key. By using `StreamPartitioner` (http://mng.bz/9Z8A), you can apply just about any partition strategy you can think of, such as partitioning on the value or part of the value instead of the key. In the next section, we'll demonstrate using `StreamPartitioner` in Kafka Streams.

REPARTITIONING IN KAFKA STREAMS

Repartitioning in Kafka Streams is easily accomplished by using the `KStream.through()` method, as illustrated in figure 4.7. The `KStream.through()` method creates an intermediate topic, and the *current*`KStream` instance will start writing records to that topic. A *new*`KStream` instance is returned from the `through()` method call, using the same intermediate topic for its source. This way, the data is seamlessly repartitioned.

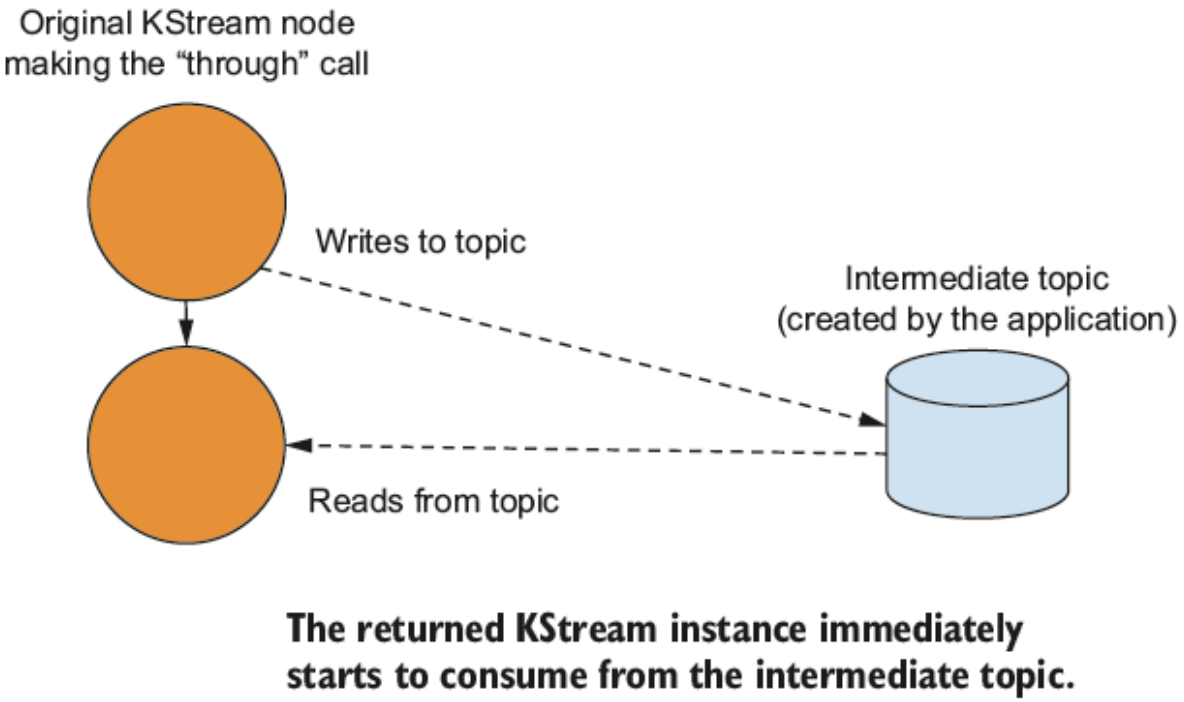


Figure 4.7 Writing out to an intermediate topic and then reading from it in a new KStream instance

Under the covers, Kafka Streams creates a sink and source node. The sink node is a child processor of the calling `KStream` instance, and the new `KStream` instance uses the new source node for its source of records. You could write the same type of sub-topology yourself using the DSL, but using the `KStream.through` method is more convenient.

If you've modified or changed keys and you don't need a custom partition strategy, you can rely on the `DefaultPartitioner` of the internal Kafka Streams `KafkaProducer` to handle the partitioning. But if you'd like to apply your own partitioning approach, you can use `StreamPartitioner`. You'll do just that in the next example.

The code for using the `KStream.through` method is shown in the following listing. In this example, `KStream.through` takes two parameters: the topic name and a `Produced` instance that provides the key `Serde`, the value `Serde`, and a `StreamPartitioner`. Note that if you want to use the default key and value `Serde` instances and have no need for a custom partitioning strategy, there's a version of `KStream.through` where you only provide the topic name.

Listing 4.4 Using the `KStream.through` method

```
RewardsStreamPartitioner streamPartitioner =
[CA]new RewardsStreamPartitioner();   (1)

KStream<String, Purchase> transByCustomerStream =
```

```
[CA]purchaseKStream.through("customer_transactions",
                            Produced.with(stringSerde,
                                          purchaseSerde,
                                          streamPartitioner));  (1)
```

(1) Instantiates the concrete StreamPartitioner instance

(2) Creates a new KStream with the KStream.through method

Here, you've instantiated a `RewardsStreamPartitioner`. Let's take a quick look at how it works as well as demonstrate how to create a `StreamPartitioner`.

USING A STREAMPARTITIONER

Typically, the partition assignment is calculated by taking the hash of an object, modulo the number of partitions. In this case, you want to use the customer ID found in the `Purchase` object so that all data for a given customer ends up in the same state store. The following listing shows the `StreamPartitioner` implementation (found in src/main/java/bbejeck/chapter_4/partitioner/RewardsStreamPartitioner.java).

Listing 4.5 `RewardsStreamPartitioner`

```
public class RewardsStreamPartitioner implements
[CA]StreamPartitioner<String, Purchase> {

    @Override
    public Integer partition(String key,
                             Purchase value,
                             int numPartitions) {

      return value.getCustomerId().hashCode() % numPartitions;  (1)
    }
}
```

(1) Determines the partition by customer ID

Notice that you haven't generated a new key. You're using a property of the value to determine the correct partition. The key point to take away from this quick detour is that when you're using state to update and modify records, it's necessary for those records to be on the same partition.

WARNING Don't mistake this simple repartitioning demonstration for something you can be cavalier with. Although repartitioning is sometimes necessary, it comes at the cost of duplicating your data and incurs processing overhead. My advice is to use `mapValues`, `transformValues`, or `flatMapValues` operations whenever possible, because `map`, `transform`, and `flatMap` can trigger automatic repartitioning. It's best to use repartitioning logic sparingly.

Now, let's get back to making changes in the rewards processor node to support stateful transformation.

4.2.5 Updating the rewards processor

Up to this point, you've created a new processing node that writes purchase objects out to a topic, partitioned by customer ID. This new topic will also be the source for your soon-to-be-updated rewards processor. You did this to ensure that all purchases for a given customer are written to the same partition; hence, you'll use the same state store for all purchases by a given customer. Figure 4.8 shows the updated processing topology with the new through processor between the credit card–masking node (the source for all purchase transactions) and the rewards processor.

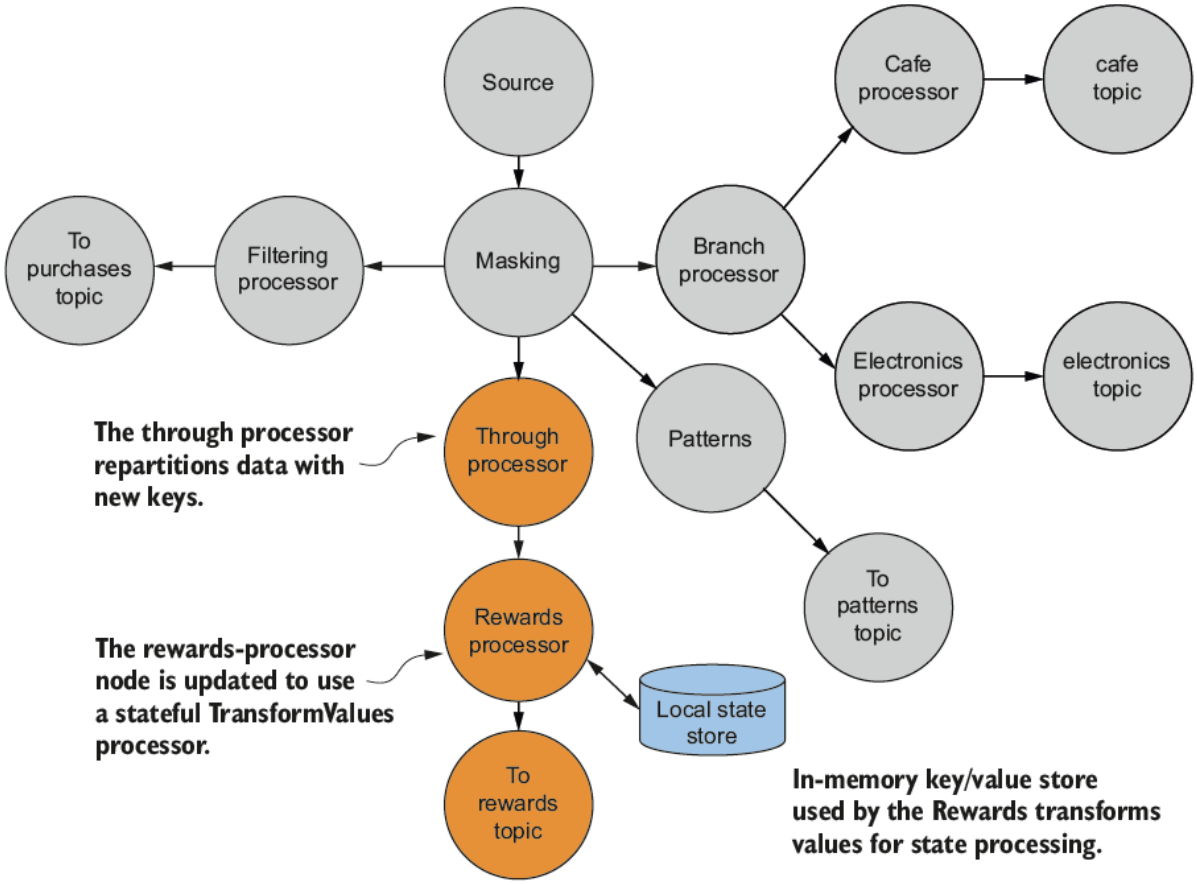


Figure 4.8 The new through processor ensures that you send purchases to partitions by customer ID, allowing the rewards processor to make the right updates using local state.

Now, you'll use the new `Stream` instance (created by the `KStream.through` method) to update the rewards processor and use the stateful transform approach with the following code.

Listing 4.6 Changing the rewards processor to use stateful transformation

```
KStream<String, RewardAccumulator> statefulRewardAccumulator =
[CA]transByCustomerStream.transformValues(() ->
[CA]new PurchaseRewardTransformer(rewardsStateStoreName),
                                  rewardsStateStoreName);   (1)
statefulRewardAccumulator.to("rewards",
                             Produced.with(stringSerde,
                                      rewardAccumulatorSerde));   (1)
```

(1) Uses a stateful transformation

(2) Writes the results out to a topic

The `KStream.transformValues` method takes a `ValueTransformerSupplier<V, R>` instance, which is provided via a Java 8 lambda expression.

In this section, you've added stateful processing to a stateless node. By adding state to the processor, ZMart can take action sooner after a customer makes a reward-qualifying purchase. You've seen how to use a state store and the benefits using one provides, but we've glossed over important details that you'll need to understand about how state can impact your applications. With this in mind, the next section will discuss which type of state store to use, what requirements you'll need to make state efficient, and how you can add the state stores to a Kafka Streams program.

4.3 Using state stores for lookups and previously seen data

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.

So far, we've discussed the need for using state with streams, and you've seen an example of one of the more basic stateful operations available in Kafka Streams. Before we get into more detail about using state stores in Kafka Streams, let's briefly look at two important attributes of state: data locality and failure recovery.

4.3.1 Data locality

Data locality is critical for performance. Although key lookups are typically very fast, the latency introduced by using remote storage becomes a bottleneck when you're working at scale.

Figure 4.9 illustrates the principle behind data locality. The dashed line represents a

network call to retrieve data from a remote database. The solid line depicts a call to an in-memory data store located on the same server. As you can see, making a call to get data locally is more efficient than making a call across a network to a remote database.

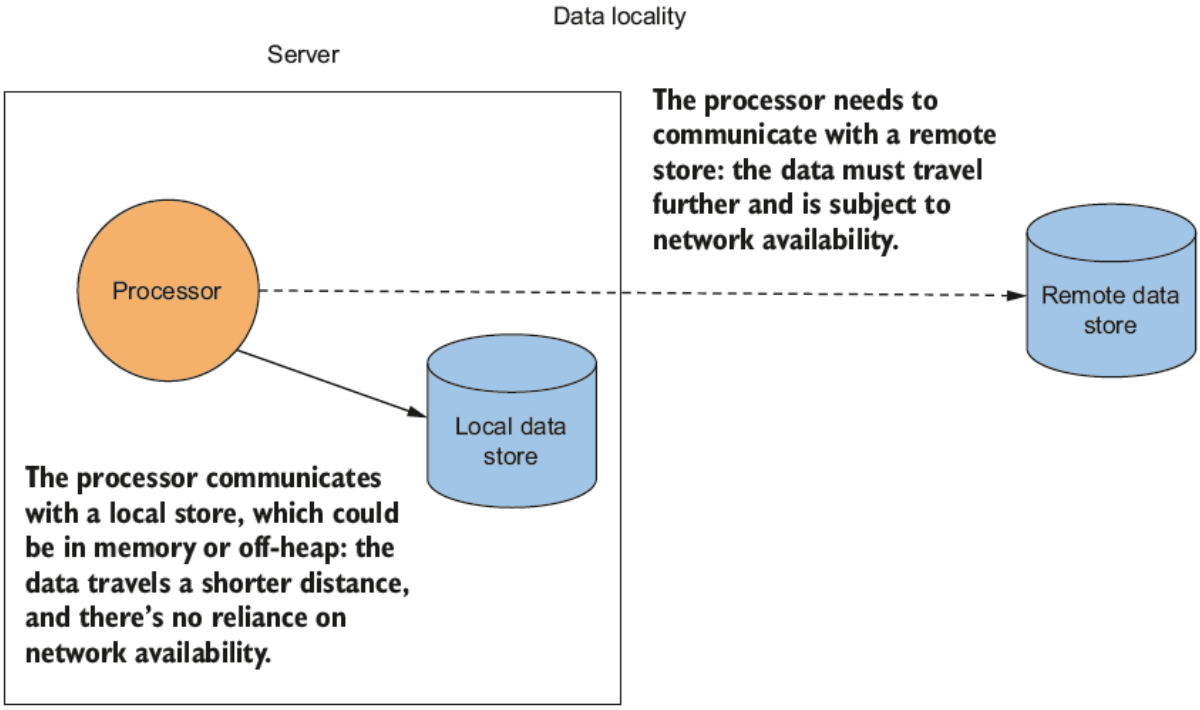


Figure 4.9 Data locality is necessary for stream processing.

The key point here isn't the degree of latency per record retrieval, which may be minimal. The important factor is that you'll potentially process millions or billions of records through a streaming application. When multiplied by a factor that large, even a small degree of network latency can have a huge impact.

Data locality also implies that the store is local to each processing node, and there's no sharing across processes or threads. This way, if a process fails, it shouldn't have an impact on the other stream-processing processes or threads.

The key point here is that although streaming applications will sometimes require state, it should be local to where the processing occurs. Each server or node in the application should have an individual data store.

4.3.2 Failure recovery and fault tolerance

Application failure is inevitable, especially when it comes to distributed applications. We need to shift our focus from preventing failure to recovering quickly from failure, or even from restarts.

Figure 4.10 depicts the principles of data locality and fault tolerance. Each processor has its local data store, and a changelog topic is used to back up the state store.

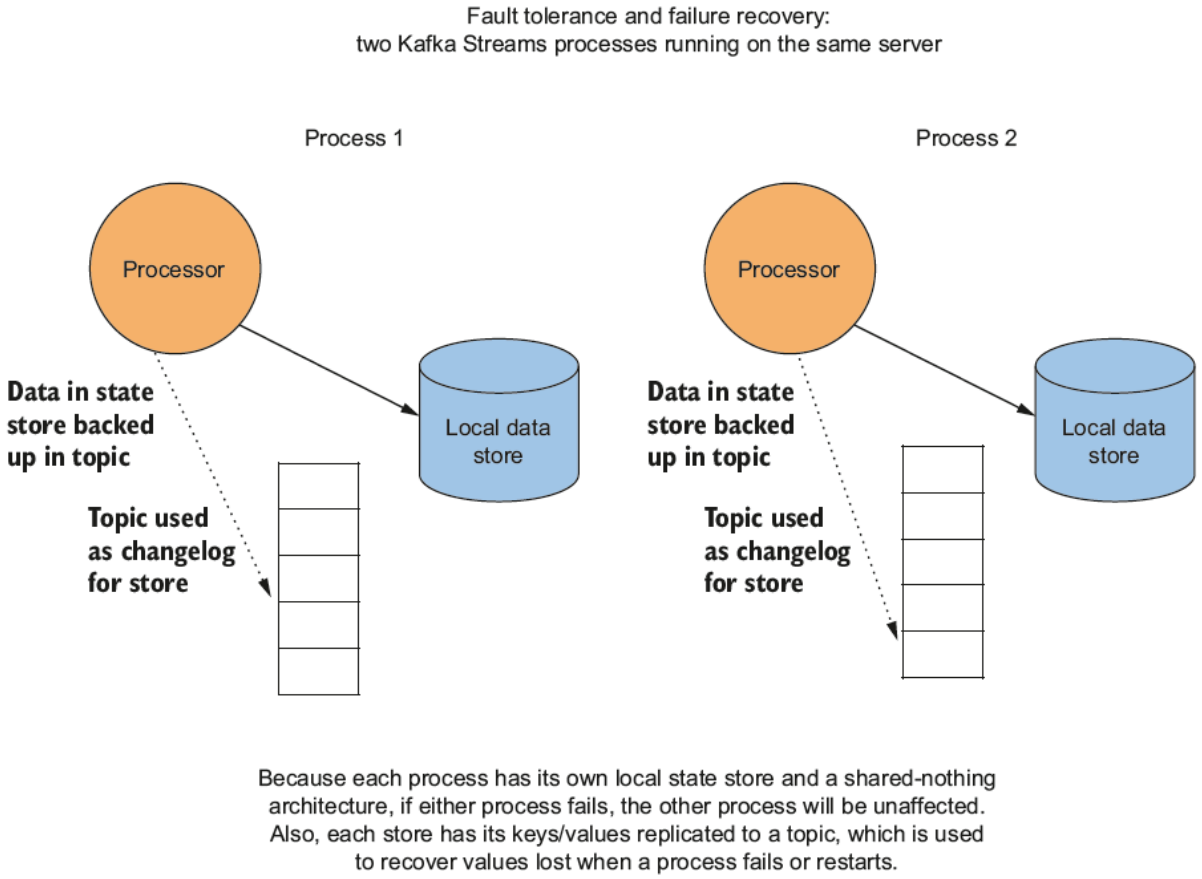


Figure 4.10 The ability to recover from failure is important for stream-processing applications. Kafka Streams persists data from the local in-memory stores to an internal topic, so when you resume operations after a failure or a restart, the data is repopulated.

Backing up a state store with a topic may seem expensive, but there are a couple of mitigating factors at play: a `KafkaProducer` sends records in batches, and by default, records are cached. It's only on cache flush that Kafka Streams writes records to the store, so only the latest record for a given key is persisted. We'll discuss this caching mechanism with state stores in more detail in chapter 5.

The state stores provided by Kafka Streams meet both the locality and fault-tolerance requirements. They're local to the defined processors and don't share access across processes or threads. State stores also use topics for backup and quick recovery.

We've now covered the requirements for using state with a streaming application. The next step is to look at how you can enable the use of state in a Kafka Streams application.

4.3.3 Using state stores in Kafka Streams

Adding a state store is a simple matter of creating a `StoreSupplier` instance with one of the static factory methods on the `Stores` class. There are two additional classes for customizing the state store: the `Materialized` and `StoreBuilder` classes. Which one you'll use depends on how you add the store to the topology. If you use the high-level DSL, you'll typically use the `Materialized` class; when you work with the lower-level Processor API, you'll use the `StoreBuilder`.

Even though the current example uses the high-level DSL, you add a state store to a `Transformer`, which provides Processor API semantics. So, you'll use the `StoreBuilder` for state store customization.

Listing 4.7 Adding a state store

```
String rewardsStateStoreName = "rewardsPointsStore";
KeyValueBytesStoreSupplier storeSupplier =

[CA]Stores.inMemoryKeyValueStore(rewardsStateStoreName);   (1)

StoreBuilder<KeyValueStore<String, Integer>> storeBuilder =
[CA]Stores.keyValueStoreBuilder(storeSupplier,
                                Serdes.String(),

                                Serdes.Integer());   (1)


builder.addStateStore(storeBuilder);   (2)
```

(1) Creates the StateStore supplier

(2) Creates the StoreBuilder and specifies the key and value types

(3) Adds the state store to the topology

You first create a `StoreSupplier` that provides an in-memory key/value store. Then, you provide the `StoreSupplier` as a parameter to create a `StoreBuilder`, additionally specifying `String` keys and `Integer` values. Finally, you add the `StateStore` to the topology by providing the `StoreBuilder` to the `StreamsBuilder`.

Here, you've created an in-memory key/value store with `String` keys and `Integer` values, and you've added the store to the application with the `StreamsBuilder.addStateStore` method. As a result, you can now use the state in your processors by using the name `rewardsStateStoreName` created above, for the state store.

You've now seen an example of building an in-memory state store, but you have options

for creating different types of `StateStore` instances. Let's look at those options.

4.3.4 Additional key/value store suppliers

In addition to the `Stores.inMemoryKeyValueStore` method, you can use these other static factory methods for producing store suppliers:

- `Stores.persistentKeyValueStore`
- `Stores.lruMap`
- `Stores.persistentWindowStore`
- `Stores.persistentSessionStore`

It's worth noting that all persistent `StateStore` instances provide local storage using RocksDB (http://rocksdb.org).

Before we move on from state stores, I'd like to cover two other important aspects of Kafka Streams state stores: how fault tolerance is provided with changelog topics, and how you can configure changelog topics.

4.3.5 StateStore fault tolerance

All the `StateStoreSupplier` types have logging enabled as a default. *Logging*, in this context, means a Kafka topic used as a changelog to back up the values in the store and provide fault tolerance.

For example, suppose you lost a machine running Kafka Streams. Once you recovered your server and restarted your Kafka Streams application, the state stores for that instance would be restored to their original contents (the last committed offset in the changelog before crashing).

This logging can be disabled when using the `Stores` factory with the `disableLogging()` method. But you shouldn't disabling logging without serious consideration, because doing so removes fault tolerance from the state stores and eliminates their ability to recover after a crash.

4.3.6 Configuring changelog topics

The changelogs for state stores are configurable via the `withLoggingEnabled(Map<String, String> config)` method. You can use any configuration parameters available for topics in the map. The configuration of changelogs for state stores is important when building a Kafka Streams application. But keep in mind that you never need to create changelog topics—Kafka Streams handles changelog topic creation for you.

> **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.

With Kafka topics, the default setting for data retention for a log segment is one week, and the size is unlimited. Depending on your volume of data, this might be acceptable, but there's a good chance you'll want to adjust those settings. Additionally, the default cleanup policy is `delete`.

Let's first take a look at how you can configure your changelog topic to have a retention size of 10 GB and a retention time of 2 days.

Listing 4.8 Setting changelog properties

```
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("retention.ms","172800000" );
changeLogConfigs.put("retention.bytes", "10000000000");

// to use with a StoreBuilder
storeBuilder.withLoggingEnabled(changeLogConfigs);

// to use with Materialized
Materialized.as(Stores.inMemoryKeyValueStore("foo")
                      .withLoggingEnabled(changeLogConfigs));
```

In chapter 2, we discussed compacted topics offered by Kafka. To refresh your memory: compacted topics use a different approach to cleaning a topic. 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. By default, Kafka Streams creates changelog topics with a delete policy of `compact`.

But if you have a changelog topic with a lot of unique keys, compaction might not be enough, as the size of the log segment will keep growing. In that case, the solution is simple. You specify a cleanup policy of `delete` and `compact`.

Listing 4.9 Setting a cleanup policy

```
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("retention.ms","172800000" );
changeLogConfigs.put("retention.bytes", "10000000000");
changeLogConfigs.put("cleanup.policy", "compact,delete");
```

Now your changelog topic will be kept at a reasonable size even with unique keys. This has been a brief section on topic configuration; appendix A provides more information about changelog topics and internal topic configuration.

You've been introduced to the basics of stateful operations and state stores. You've learned about the in-memory and persistent state stores Kafka Streams provides and how you can include them in your streaming applications. You've also learned about the importance of data locality and fault tolerance when using state in a streaming application. Let's move on to joining streams.

4.4 Joining streams for added insight

As we discussed earlier in the chapter, streams need state when events in the stream don't stand alone. Sometimes the state or context you need is another stream. 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 ZMart scenario. As you'll recall, ZMart opened a new line of stores that carried electronics and related merchandise (CDs, DVDs, smart phones, and so on). Trying a new approach, ZMart has partnered with a national coffee house and has embedded a cafe in each store.

In chapter 3, you were asked to separate the purchase transactions in those stores into two distinct streams. Figure 4.11 shows the topology for this requirement.

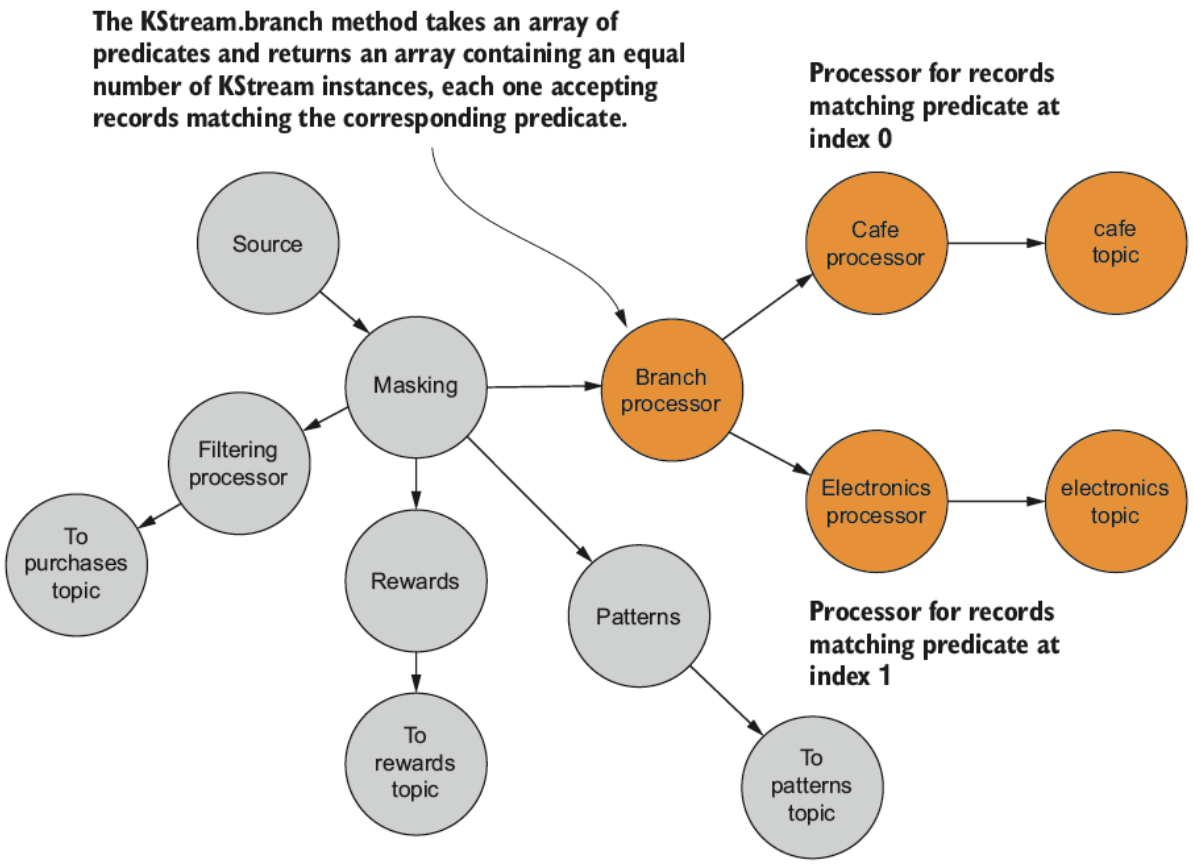


Figure 4.11 The branch processor, and where it stands in the overall topology

This approach of embedding the cafe has been a big success for ZMart, and the company would like to see this trend continue, so they've decided to start a new program. ZMart wants to keep traffic in the electronics store high by offering coupons for the cafe (hoping that increased traffic leads to additional purchases).

ZMart wants to identify customers who have bought coffee and made a purchase in the electronics store and give them coupons almost immediately after their second transaction (see figure 4.12). ZMart intends to see if it can generate some sort of Pavlovian response in their customers.

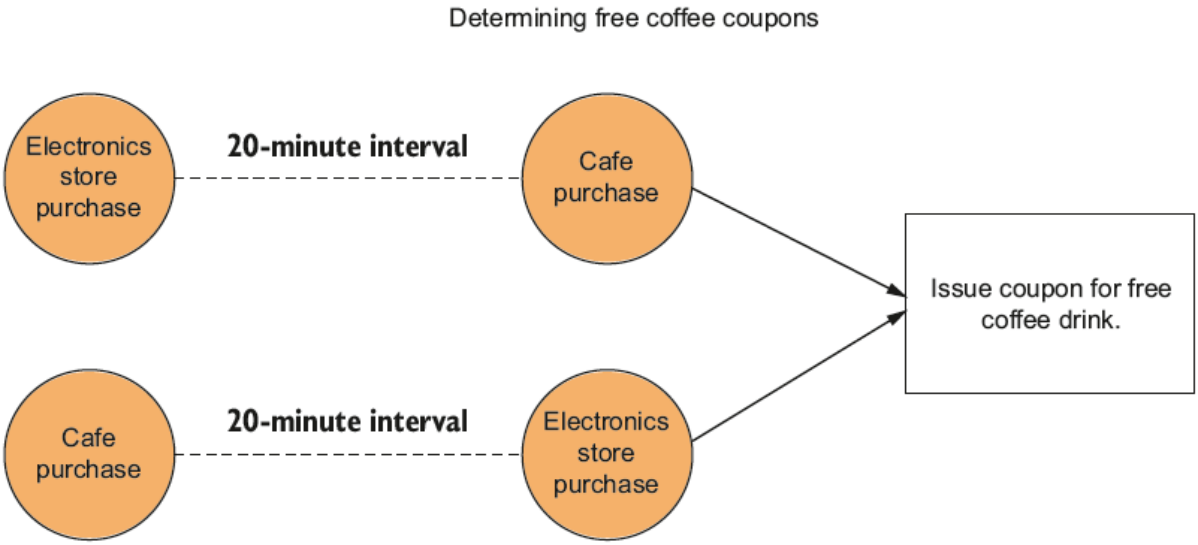


Figure 4.12 Purchase records with timestamps within 20 minutes of each other are joined by customer ID and used to issue a reward to the customer—a free coffee drink, in this case.

To determine when to issue a coupon, you'll need to join the sales from the cafe with the sales in the electronics store. Joining streams is relatively straightforward in terms of the code you need to write. Let's start by setting up the data you need to process for doing joins.

4.4.1 Data setup

First, let's take another look at the portion of the topology responsible for branching the streams (figure 4.13). In addition, let's review the code used to implement the branching requirement (found in src/main/java/bbejeck/chapter_3/ZMartKafkaStreamsAdvancedReqsApp.java).

Listing 4.10 Branching into two streams

```
Predicate<String, Purchase> coffeePurchase = (key, purchase) ->
[CA]purchase.getDepartment().equalsIgnoreCase("coffee");

Predicate<String, Purchase> electronicPurchase = (key, purchase) ->
[CA]purchase.getDepartment().equalsIgnoreCase("electronics");   (1)

final int COFFEE_PURCHASE = 0;
final int ELECTRONICS_PURCHASE = 1;   (2)

KStream<String, Purchase>[] branchedTransactions =
[CA]transactionStream.branch(coffeePurchase, electronicPurchase);   (3)
```

(1) Defines the predicates for matching records

(2) Uses labeled integers for clarity when accessing the corresponding array

❸ Creates the branched stream

This code shows how to perform branching: you use predicates to match incoming records into an array of `KStream` instances. The order of matching is the same as the position of `KStream` objects in the array. The branching process drops any record not matching any predicate.

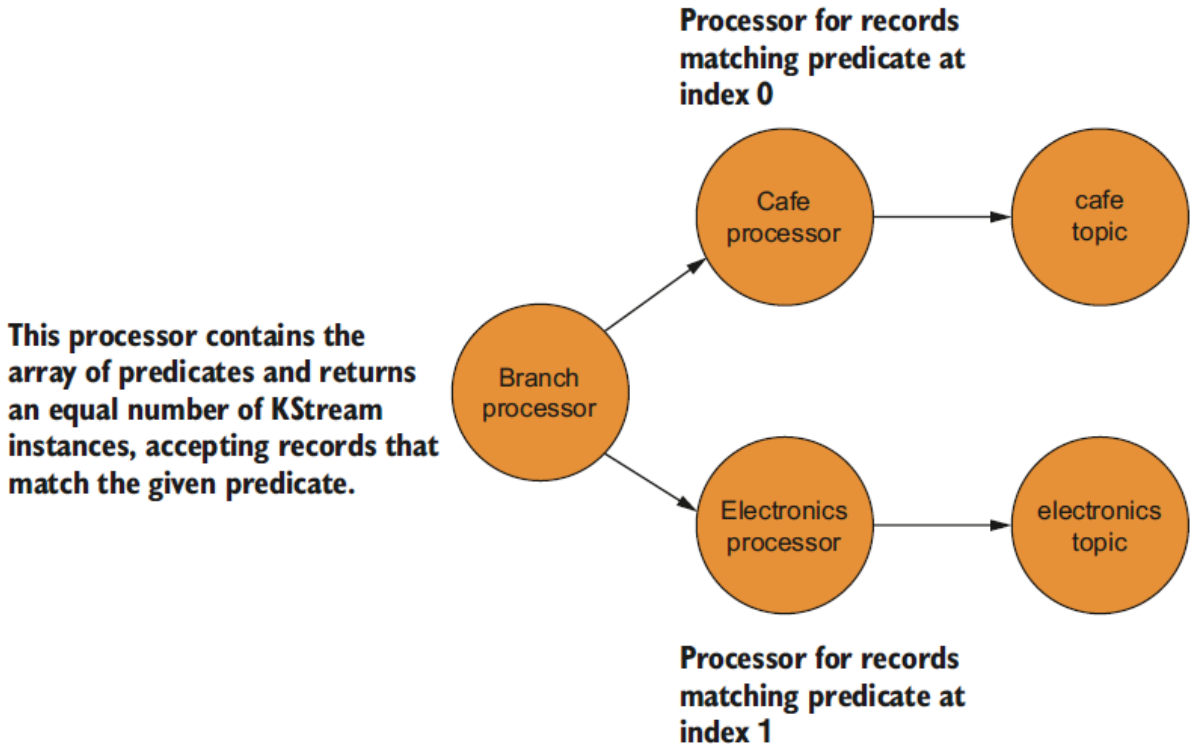


Figure 4.13 To perform a join, you need more than one stream. The branch processor takes care of this by creating two streams: one containing cafe purchases and the other containing electronics purchases.

Although you have two streams to join, there's an additional step to perform. Remember that purchase records come into the Kafka Streams application with no keys. As a result, you need to add another processor to generate a key containing the customer ID. You need populated keys because that's what you'll use to join the records together.

4.4.2 Generating keys containing customer IDs to perform joins

To generate a key, you select the customer ID from the purchase data in the stream. To do so, you need to update the original `KStream` instance (`transactionStream`) and create another processing node between it and the branch node. This is shown in the following code (found in src/main/java/bbejeck/chapter_4/KafkaStreamsJoinsApp.java).

Listing 4.11 Generating new keys

```
KStream<String, Purchase>[] branchesStream =
 [CA]transactionStream.selectKey((k,v)->
 [CA]v.getCustomerId()).branch(coffeePurchase, electronicPurchase);   ❶
```

❶ 0-1)) Inserts the selectKey processing node

Figure 4.14 shows an updated view of the processing topology based on listing 4.11. You've seen before that changing the key may require you to repartition the data. That's true in this example as well, so why isn't there a repartitioning step?

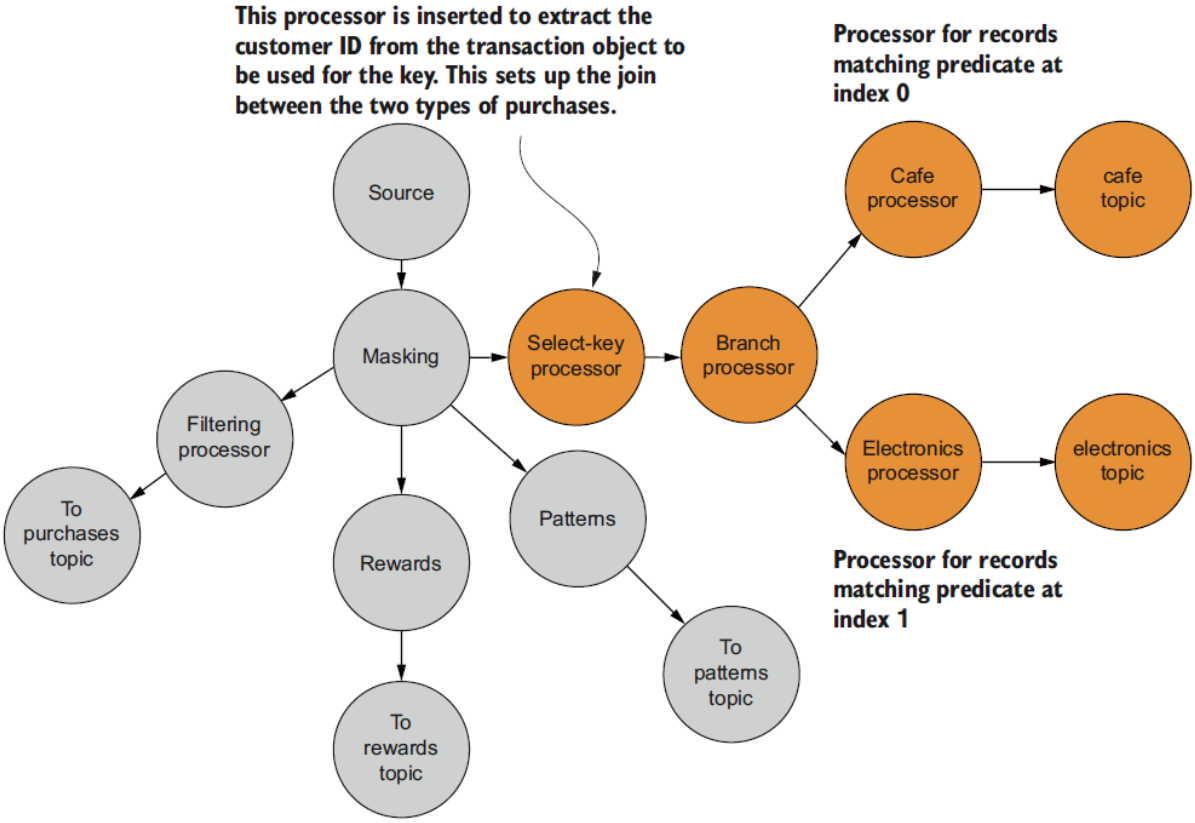


Figure 4.14 You need to remap the key/value purchase records into records where the key contains the customer ID. Fortunately, you can extract the customer ID from the Purchase object.

In Kafka Streams, whenever you invoke a method that could result in generating a new key (`selectKey`, `map`, or `transform`), an internal Boolean flag is set to `true`, indicating that the new `KStream` instance requires repartitioning. With this Boolean flag set, if you perform a join, reduce, or aggregation operation, the repartitioning is handled for you automatically.

In this example, you perform a `selectKey` operation on the `transactionStream`, so the resulting `KStream` is flagged for repartitioning. Additionally, you immediately perform a branching operation, so each `KStream` resulting from the `branch` call is flagged for

repartitioning as well.

> **NOTE** In the example, you repartition by the key only. But there may be times when you either don't want to use the key or want to use some combination of the key and value. In these cases, you can use the `StreamPartitioner<K, V>` interface, as you saw in the example in listing 4.5 in the section, "Using a StreamPartitioner."

Now that you have two separate streams with populated keys, you're ready for the next step: joining the streams by key.

4.4.3 Constructing the join

The next step is to perform the actual join. You'll take the two branched streams and join them with the `KStream.join` method. The topology is shown in figure 4.15.

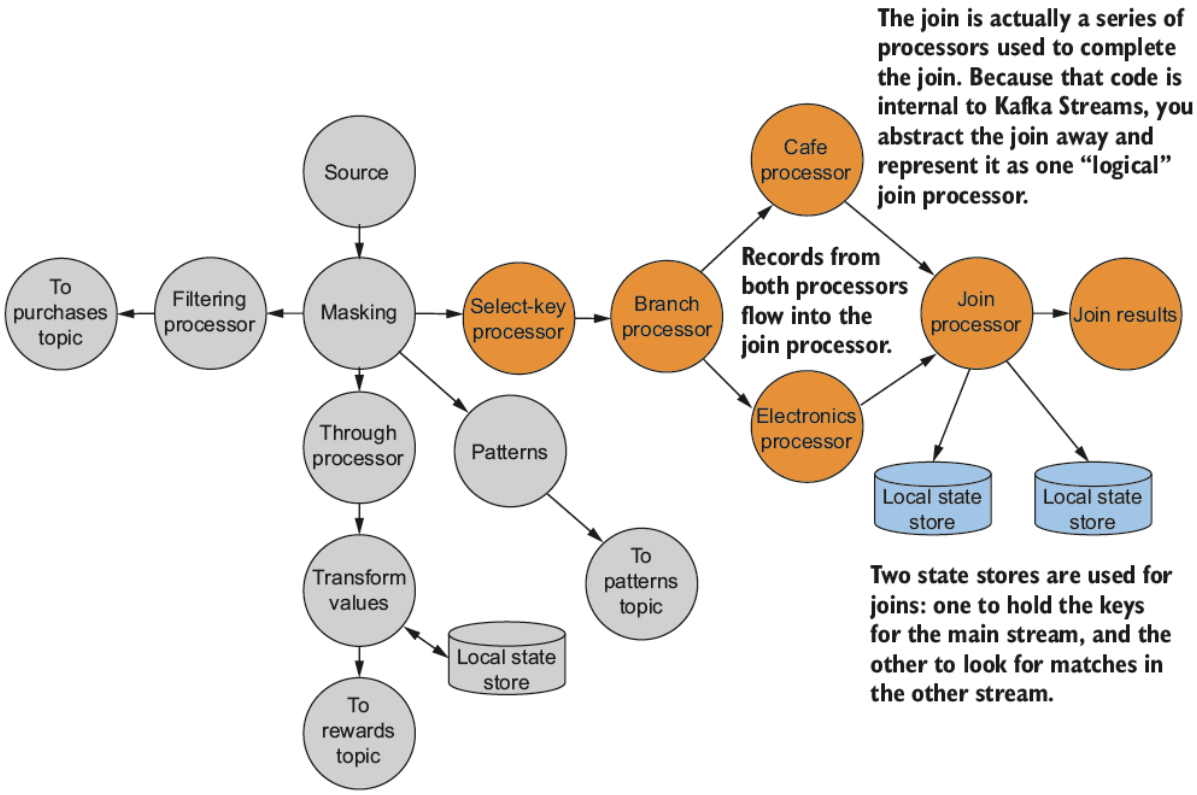


Figure 4.15 In the updated topology, both join() methodthe cafe and electronics processors forward their records to the join processor. The join processor uses two state stores to search for matches for a record in the other stream.

JOINING PURCHASE RECORDS

To create the joined record, you need to create an instance of a `ValueJoiner<V1, V2, R>`. `ValueJoiner` takes two objects, which may or may not be of the same type, and it returns a single object, possibly of a different type. In this case, `ValueJoiner` takes two `Purchase` objects and returns a `CorrelatedPurchase` object. Let's take a look at the code (found in src/main/java/bbejeck/chapter_4/joiner/PurchaseJoiner.java).

Listing 4.12 `ValueJoiner` implementation

```
public class PurchaseJoiner
[CA]implements ValueJoiner<Purchase, Purchase, CorrelatedPurchase> {

    @Override
    public CorrelatedPurchase apply(Purchase purchase, Purchase purchase2) {
        CorrelatedPurchase.Builder builder =
[CA]CorrelatedPurchase.newBuilder();  (1)

        Date purchaseDate =
[CA]purchase != null ? purchase.getPurchaseDate() : null;

        Double price = purchase != null ? purchase.getPrice() : 0.0;
        String itemPurchased =
[CA]purchase != null ? purchase.getItemPurchased() : null;  (2)

        Date otherPurchaseDate =
[CA]otherPurchase != null ? otherPurchase.getPurchaseDate() : null;

        Double otherPrice =
[CA]otherPurchase != null ? otherPurchase.getPrice() : 0.0;

        String otherItemPurchased =
[CA]otherPurchase != null ? otherPurchase.getItemPurchased() : null;  (3)

        List<String> purchasedItems = new ArrayList<>();

        if (itemPurchased != null) {
            purchasedItems.add(itemPurchased);
        }

        if (otherItemPurchased != null) {
            purchasedItems.add(otherItemPurchased);
        }

        String customerId =
[CA]purchase != null ? purchase.getCustomerId() : null;

        String otherCustomerId =
[CA]otherPurchase != null ? otherPurchase.getCustomerId() : null;

        builder.withCustomerId(customerId != null ? customerId : otherCustomerId)
                .withFirstPurchaseDate(purchaseDate)
                .withSecondPurchaseDate(otherPurchaseDate)
                .withItemsPurchased(purchasedItems)
                .withTotalAmount(price + otherPrice);


        return builder.build();  (4)
    }
}
```

1. Instantiates the builder
2. Handles a null Purchase in the case of an outer join
3. Handles a null Purchase in the case of a left outer join
4. Returns the new CorrelatedPurchase object

To create the `CorrelatedPurchase` object, you extract some information from each `Purchase` object. Because of the number of items you need to construct the new object, you use the builder pattern, which makes the code clearer and eliminates any errors due to misplaced parameters. Additionally, the `PurchaseJoiner` checks for null values with both of the provided `Purchase` objects, so it can be used for inner, outer, and left-outer joins. We'll discuss the different join options in section 4.4.4. For now, we'll move on to implementing the join between streams.

IMPLEMENTING THE JOIN

You've seen how to merge records resulting from the join between streams, so let's move on to calling the actual `KStream.join` method (found in src/main/java/bbejeck/chapter_4/KafkaStreamsJoinsApp.java).

Listing 4.13 Using the `join()` method

```
KStream<String, Purchase> coffeeStream =
[CA]branchesStream[COFFEE_PURCHASE];        (1)
KStream<String, Purchase> electronicsStream =
[CA]branchesStream[ELECTRONICS_PURCHASE];

ValueJoiner<Purchase, Purchase, CorrelatedPurchase> purchaseJoiner =
[CA] new PurchaseJoiner();                  (2)

JoinWindows twentyMinuteWindow =  JoinWindows.of(60 * 1000 * 20);

KStream<String, CorrelatedPurchase> joinedKStream =
[CA]coffeeStream.join(electronicsStream,    (3)
                      purchaseJoiner,
                      twentyMinuteWindow,
                      Joined.with(stringSerde,
                                  purchaseSerde,
                                  purchaseSerde));   (4)

joinedKStream.print("joinedStream");        (5)
```

1. Extracts the branched streams
2. ValueJoiner instance used to perform the join
3. Calls the join method, triggering automatic repartitioning of coffeeStream and electronicsStream

4. Constructs the join
5. Prints the join results to the console

You supply four parameters to the `KStream.join` method:

- `electronicsStream`—The stream of electronic purchases to join with.
- `purchaseJoiner`—An i mplementation 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` (maybe a whole new type). In this example, `purchaseJoiner` will add some relevant information from both `Purchase` objects, and it will return a `CorrelatedPurchase` object.
- `twentyMinuteWindow`—A `JoinWindows` instance. The `JoinWindows.of` method specifies a maximum time difference between the two values to be included in the join. In this case, the timestamps must be within 20 minutes of each other.
- A `Joined` 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.

NOTE `Serde`s are required for joins because join participants are materialized in windowed state stores. This example provides only one `Serde` for the key, because both sides of the join must have a key of the same type.

You've specified that the purchases need to be within 20 minutes of each other, but no order is implied. As long as the timestamps are within 20 minutes of each other, the join will occur.

Two additional `JoinWindows()` methods are available, which you can use to specify the order of events:

- `JoinWindows.after`—`streamA.join(streamB,...,twentyMinuteWindow.after(5000)....)` This specifies that the timestamp of the `streamB` record is at most 5 seconds *after* the timestamp of the `streamA` record. The starting time boundary of the window is unchanged.
- `JoinWindows.before`—`streamA.join(streamB,...,twentyMinuteWindow.before(5000),...)` This specifies that the timestamp of the `streamB` record is at most 5 seconds *before* the timestamp of the `streamA` record. The ending time boundary of the window is unchanged.

With both the `before()` and `after()` methods, the time difference is expressed in milliseconds. The timespans used for the join are an example of *sliding windows*. We'll look at windowing operations in detail in the next chapter.

NOTE In listing 4.13, you're relying on the actual timestamps of the transaction, not timestamps set by Kafka. In order to use the timestamps embedded in the transaction, you specify a custom timestamp extractor by setting `StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG` to use `TransactionTimestampExtractor.class`.

You've now constructed a joined stream: electronics purchases made within 20 minutes of a coffee purchase will result in a coupon for a free drink on the customer's next visit to ZMart.

Before we go any further, I'd like to take a minute to explain an important requirement for joining data—co-partitioning.

CO-PARTITIONING

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. As a result, when you call the `join()` method in listing 4.13, both `KStream` instances will be checked to see if a repartition is required.

NOTE `GlobalKTable` instances don't require repartitioning when involved in a join.

In section 4.2, you use the `selectKey()` method on the `transactionStream` and immediately branched on the returned `KStreams`. Because the `selectKey()` method modifies the key, both `coffeeStream` and `electronicsStream` require repartitioning. It's worth repeating that repartitioning is necessary because you need to ensure that identical keys are written to the same partition. This repartitioning is handled automatically. Additionally, when you start your Kafka Streams application, topics involved in a join are checked to make sure they have the same number of partitions; if any mismatches are found, a `TopologyBuilderException` is thrown. It's the developer's responsibility to ensure the keys involved in a join are of 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.

Let's continue on with joins and look at the other options available to you.

4.4.4 Other join options

The join in listing 4.13 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 `CorrelatedPurchase` 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.

OUTER JOINS

Outer joins always output a record, but the forwarded join record may not include both of the events specified by the join. If either side of the join isn't present when the time window expires, an outer join sends the record that's available downstream. Of course, if both events are present within the window, the issued record contains both events.

For example, if you wanted to use an outer join in listing 4.13, you'd do so like this: `coffeeStream.outerJoin(electronicsStream,..)`. Figure 4.16 demonstrates the three possible outcomes of the outer join.

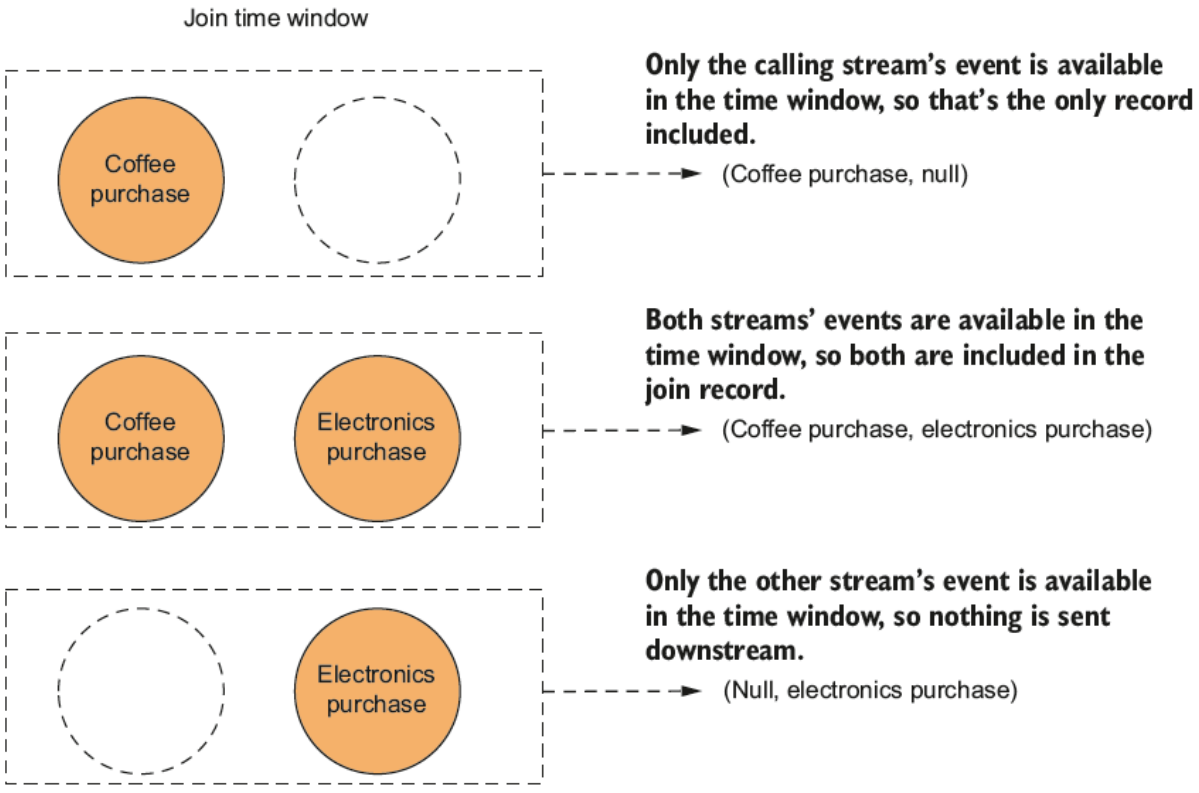


Figure 4.16 Three outcomes are possible with outer joins: only the calling stream's event, both events, and only the other stream's event.

LEFT-OUTER JOIN

The records sent downstream from a left-outer join are similar to an outer join, with one exception. When the only event available in the join window is from the other stream, there's no output at all. If you wanted to use a left-outer join in listing 4.13, you'd do so like this: `coffeeStream.leftJoin(electronicsStream..)`. Figure 4.17 shows the outcomes of the left-outer join.

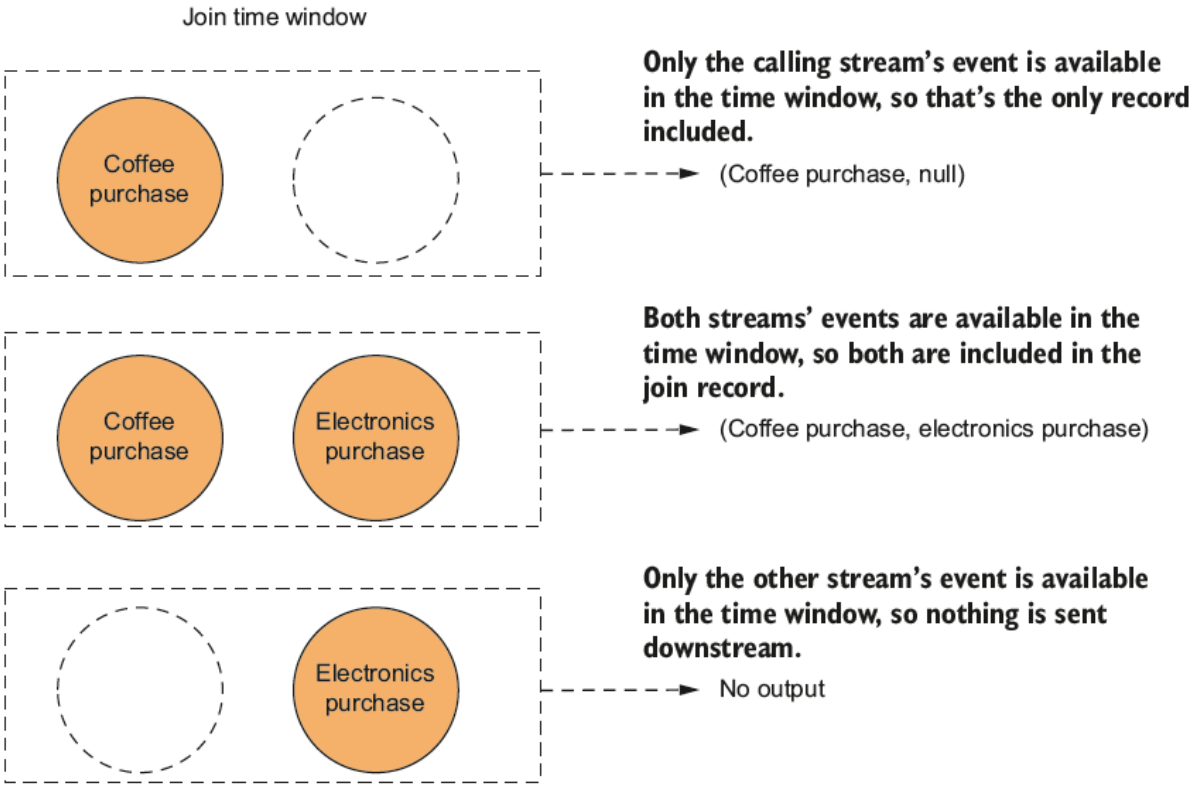


Figure 4.17 Three outcomes are possible with the left-outer join, but there's no output if only the other stream's record is available.

We've now covered joining streams, but there's one concept that deserves a more detailed discussion: timestamps and the impact they have on your Kafka Streams application. In the join example, you specified a maximum time difference between events of 20 minutes. In this case, it's the time between purchases, but how you set or extract these timestamps wasn't specified. Let's take a closer look at that.

4.5 Timestamps in Kafka Streams

Section 2.4.4 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)

We haven't covered the `KTable` or Processor APIs yet, but that's OK. You don't need them to understand this section.

In stream processing, you can group timestamps into three categories, as shown in figure 4.18:

- *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.

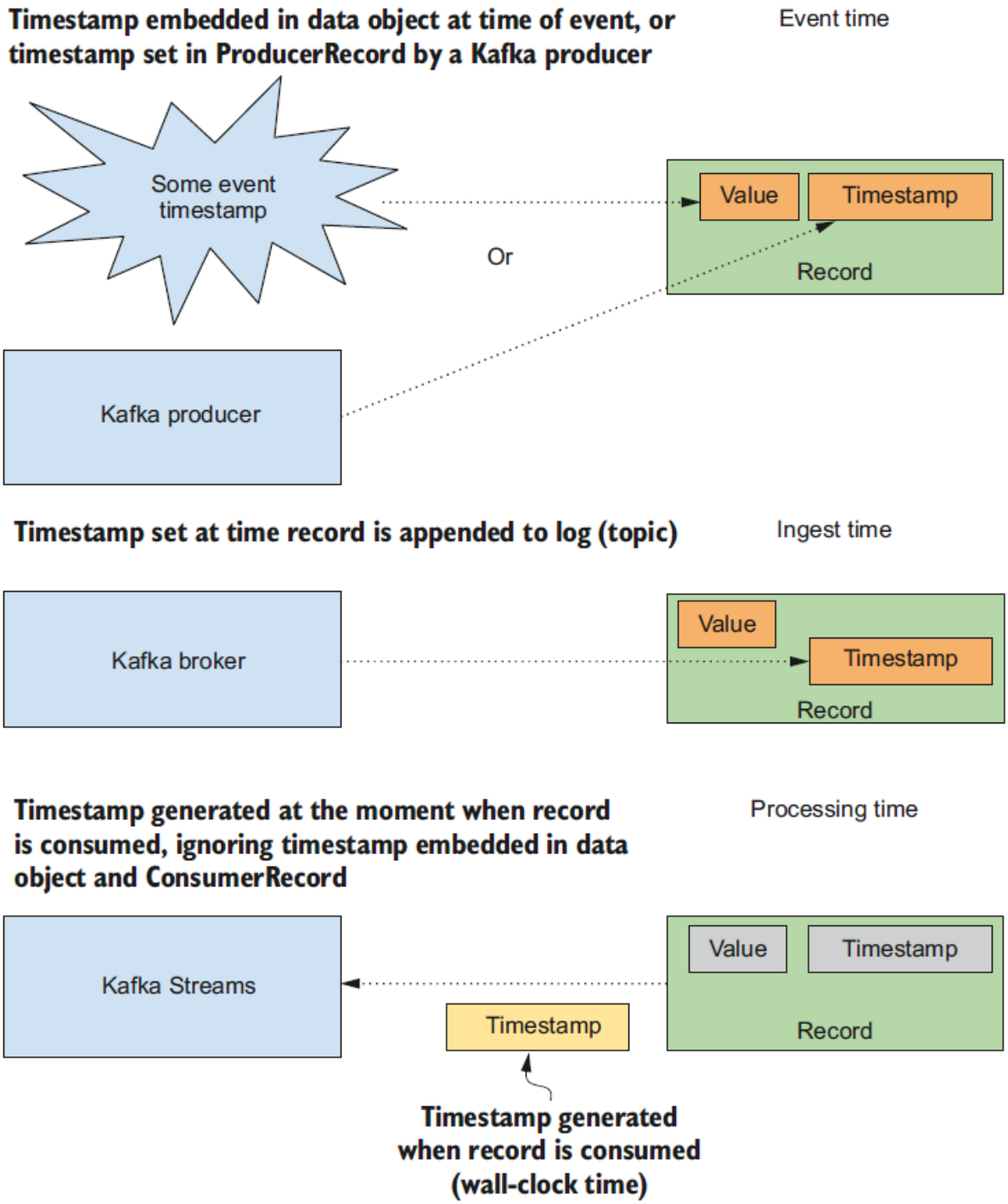


Figure 4.18 There are three categories of timestamps in Kafka Streams: event time, ingestion time, and processing time.

You'll see in this section how the Kafka Streams API supports all three types of processing timestamps.

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.

We'll consider three cases of timestamp-processing semantics:

- An timestamp embedded in the actual event or message object (event-time semantics)
- Using the timestamp set in the record metadata when creating the `ProducerRecord` (event-time semantics)
- Using the current timestamp (current local time) when the Kafka Streams application ingests the record (processing-time semantics)

For event-time semantics, using 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 consumes records as opposed to using the timestamps of the records.

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`.

4.5.1 Provided TimestampExtractor implementations

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.

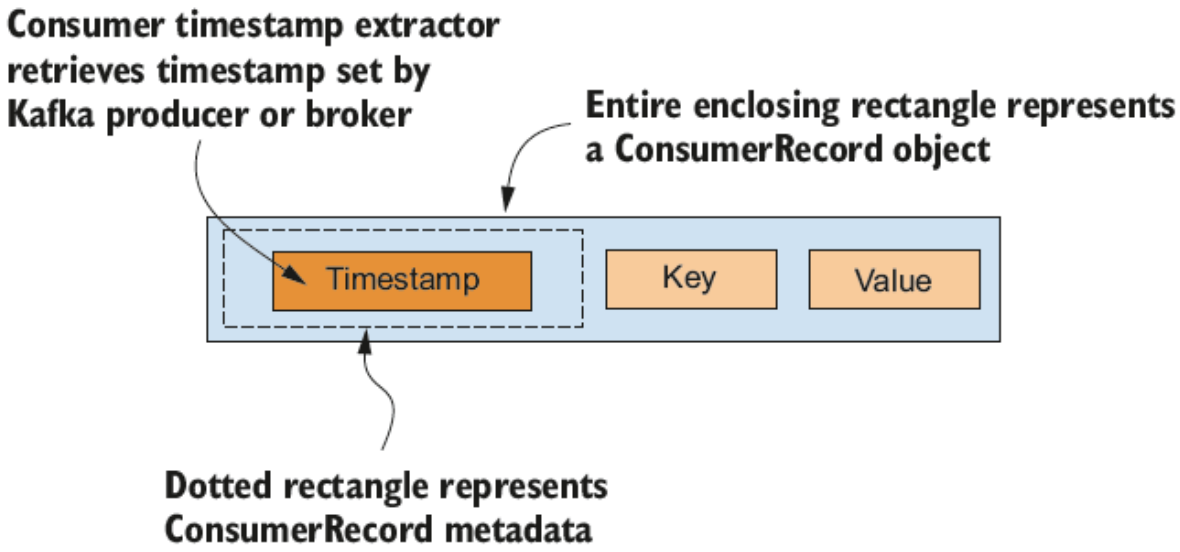


Figure 4.19 Timestamps in the ConsumerRecord object: either the producer or broker set this timestamp, depending on your configuration.

Although you're assuming the default configuration setting of `CreateTime` for the timestamp, bear in mind that if you were to use `LogAppendTime`, this would return the timestamp value for when the Kafka broker appended the record to the log. `ExtractRecordMetadataTimestamp` is an abstract class that provides the core functionality for extracting the metadata timestamp from the `ConsumerRecord`. Most of the concrete implementations extend this class. Implementors override the abstract method, `ExtractRecordMetadataTimestamp#onInvalidTimestamp`, to handle invalid timestamps (when the timestamp is less than 0).

Here's a list of classes that extend the `ExtractRecordMetadataTimestamp` class:

- `FailOnInvalidTimestamp`—Throws an exception in the case of an invalid timestamp.
- `LogAndSkipOnInvalidTimestamp`—Returns the invalid timestamp and logs a warning message that the record will be discarded due to the invalid timestamp.
- `UsePreviousTimeOnInvalidTimestamp`—In the case of an invalid timestamp, the last valid extracted timestamp is returned.

We've covered the event-time timestamp extractors, but there's one more provided timestamp extractor to cover.

4.5.2 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.

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

4.5.3 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. Figure 4.20 depicts using the timestamp embedded in the value object versus one set by Kafka (either producer or broker).

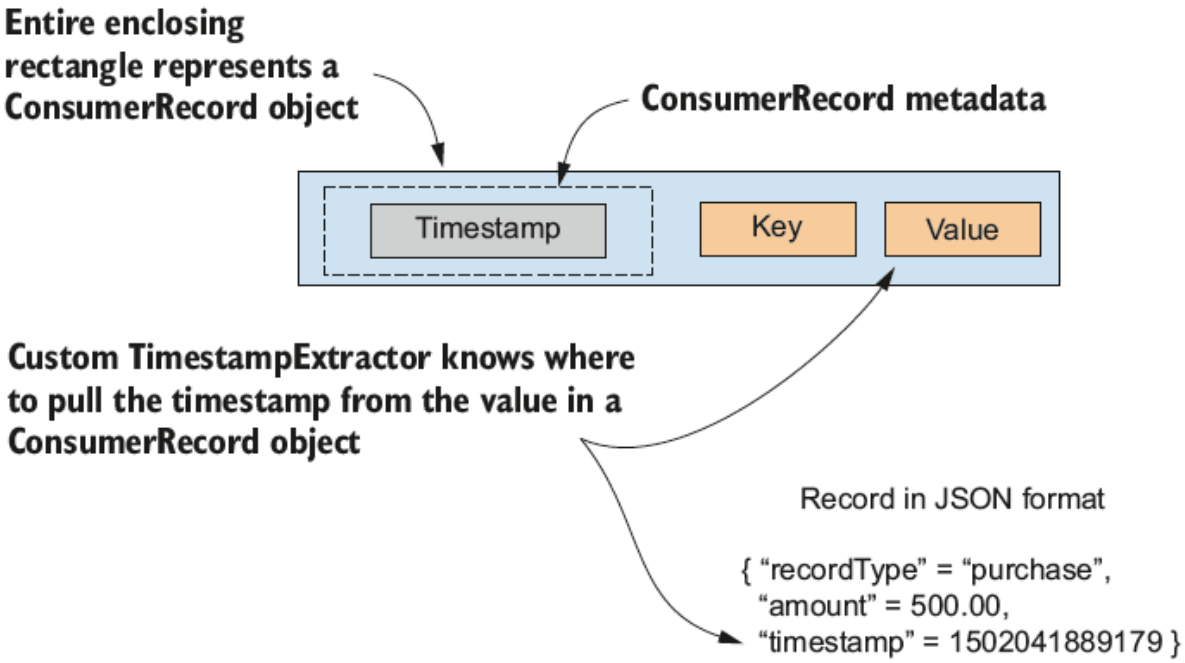


Figure 4.20 A custom TimestampExtractor provides a timestamp based on the value contained in the ConsumerRecord. This timestamp could be an existing value or one calculated from properties contained in the value object.

Here's an example of a `TimestampExtractor` implementation (found in src/main/java/bbejeck/chapter_4/timestamp_extractor/TransactionTimestampExtractor.java), also used in the join example from listing 4.12 in the section "Implementing the Join" (although not shown in the text, because it's a configuration parameter).

Listing 4.14 Custom `TimestampExtractor`

```
public class TransactionTimestampExtractor implements TimestampExtractor {

    @Override
    public long extract(ConsumerRecord<Object, Object> record,
[CA]long previousTimestamp) {
        Purchase purchaseTransaction = (Purchase) record.value();    ❶
        return purchaseTransaction.getPurchaseDate().getTime();    ❷
```

```
    }
}
```

1. Retrieves the Purchase object from the key/value pair sent to Kafka
2. Returns the timestamp recorded at the point of sale

In the join example, you used a custom `TimestampExtractor` because you wanted to use the timestamps of the actual purchase time. This approach allows you to join the records even if there are delays in delivery or out-of-order arrivals.

WARNING When you create a custom `TimestampExtractor`, take care not to get too clever. Log retention and log rolling are timestamp based, and the timestamp provided by the extractor may become the message timestamp used by changelogs and output topics downstream.

4.5.4 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,
[CA]TransactionTimestampExtractor.class);
```

The second option is to provide a `TimestampExtractor` instance via a `Consumed` object:

```
Consumed.with(Serdes.String(), purchaseSerde)
         .withTimestampExtractor(new TransactionTimestampExtractor()))
```

The advantage of doing this is that you have one `TimestampExtractor` per input source, whereas the other option requires you to handle records from different topics in one `TimestampExtractor` instance.

We've come to the end of our discussion of timestamp usage. In the coming chapters, you'll run into situations where the difference between timestamps drives some action,

such as flushing the cache of a `KTable`. I don't expect you remember all three types of timestamp extractors, but it's vital to understand that timestamps are an important part of how Kafka and Kafka Streams function.

4.6 Summary

- Stream processing needs state. Sometimes events can stand on their own, but usually you'll need additional information to make good decisions.
- Kafka Streams provides useful abstractions for stateful transformations, including joins.
- State stores in Kafka Streams provide the type of state required for stream processing: data locality and fault tolerance.
- Timestamps control the flow of data in Kafka Streams. The choice of timestamp sources needs careful consideration.

In the next chapter, we'll continue exploring state in streams with more-significant operations, like aggregations and grouping. We'll also explore the `KTable` API. Whereas the `KStream` API concerns itself with individual discrete records, a `KTable` is an implementation of a changelog, where records with the same key are considered updates. We'll also discuss joins between `KStream` and `KTable` instances. Finally, we'll explore one of the most exciting developments in Kafka Streams: queryable state. Queryable state allows you to directly observe the state of your stream, without having to materialize the information by reading data from a topic in an external application.

5 The KTable API

This chapter covers

- Defining the relationship between streams and tables
- Updating records, and the `KTable` abstraction
- Aggregations, and windowing and joining `KStream`s and `KTable`s
- Global `KTable`s
- Queryable state stores

So far, we've covered the `KStream` API and adding state to a Kafka Streams application. In this chapter, we're going to look deeper into adding state. Along the way, you'll be introduced to a new abstraction, the `KTable`.

In discussing the `KStream` API, we've talked about individual events or an event stream. In the original ZMart example, when Jane Doe made a purchase, you considered the purchase to be an individual event. You didn't keep track of how many purchases Jane made, or how often. In the context of a database, the purchase event stream could be considered a series of inserts. Because each record is new and unrelated to any other record, you could continually insert them into a table.

Now let's add a primary key (customer ID) to each purchase event. You have a series of *related* purchase events or updates for Jane Doe. Because you're using a primary key, each purchase is updated with respect to Jane's purchase activity. Treating an event stream as inserts, and events with keys as updates, is how you'll define the relationship between streams and tables.

In this chapter, we'll cover the relationship between streams and tables in more depth. This relationship is important, as it will help you understand how the `KTable` operates.

Second, we'll discuss the `KTable`. The `KTable` API is necessary because it's designed to work with updates to records. We need the ability to work with updates for operations like aggregations and counts. We touched on updates in chapter 4 when introducing stateful transformations; in section 4.2.5, you updated the rewards processor to keep track of customer purchases.

Third, we'll get into windowing operations. Windowing is the process of bucketing data for a given period. For example, how many purchases have there been over the past hour, updated every ten minutes? Windowing allows you to gather data in chunks, as opposed to having an unbounded collection.

> **NOTE** *Windowing* and *bucketing* are somewhat synonymous terms. Both operate by placing information into smaller chunks or categories. Windowing implies categorizing by time, but the result of either operation is the same.

Our final topic in this chapter will be queryable state stores. Queryable state stores are an exciting feature of Kafka Streams: they allow you to run direct queries against state stores. In other words, you can view stateful data without having to consume it from a Kafka topic or read it from a database. Let's move on to our first topic.

5.1 The relationship between streams and tables

In chapter 1, I defined a stream as an infinite sequence of events. That wording is pretty generic, so let's narrow it down with a specific example.

5.1.1 The record stream

Suppose you want to view a series of stock price updates. You can recast the generic marble diagram from chapter 1 to look like figure 5.1. 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.

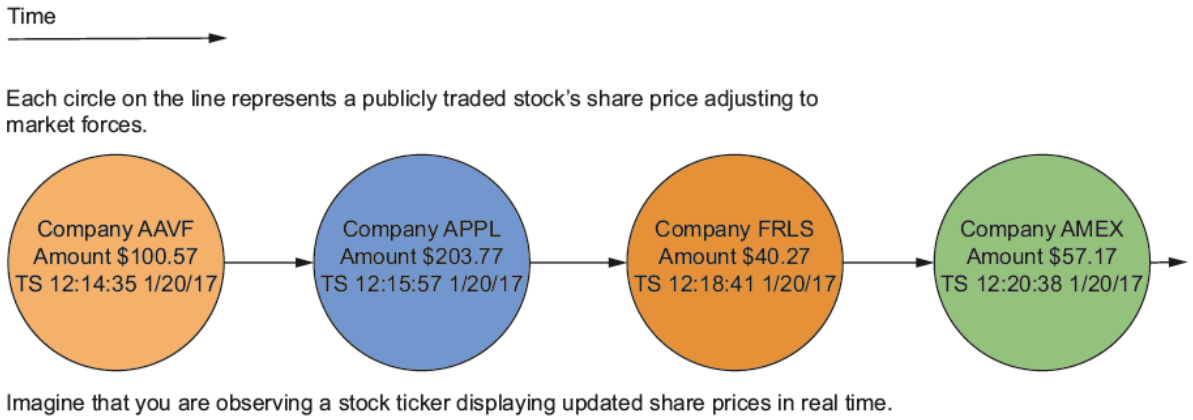
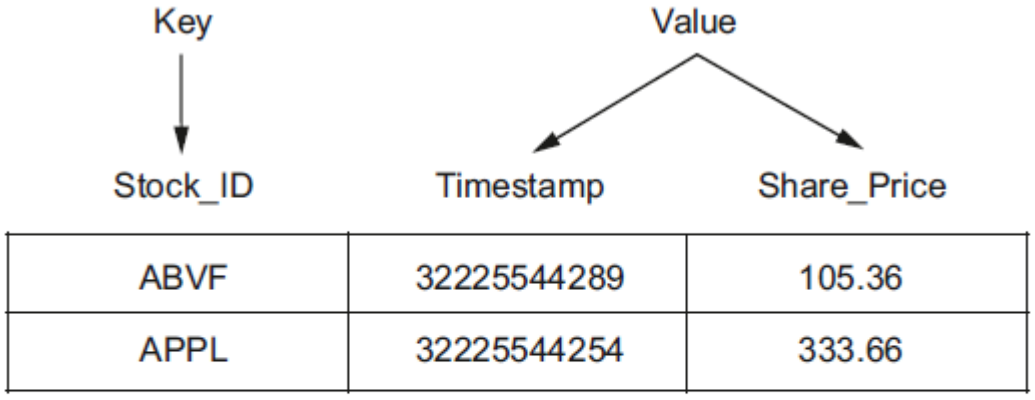


Figure 5.1 A marble diagram for an unbounded stream of stock quotes

Now, let's see how this concept ties into database tables. Look at the simple stock quote table in figure 5.2.



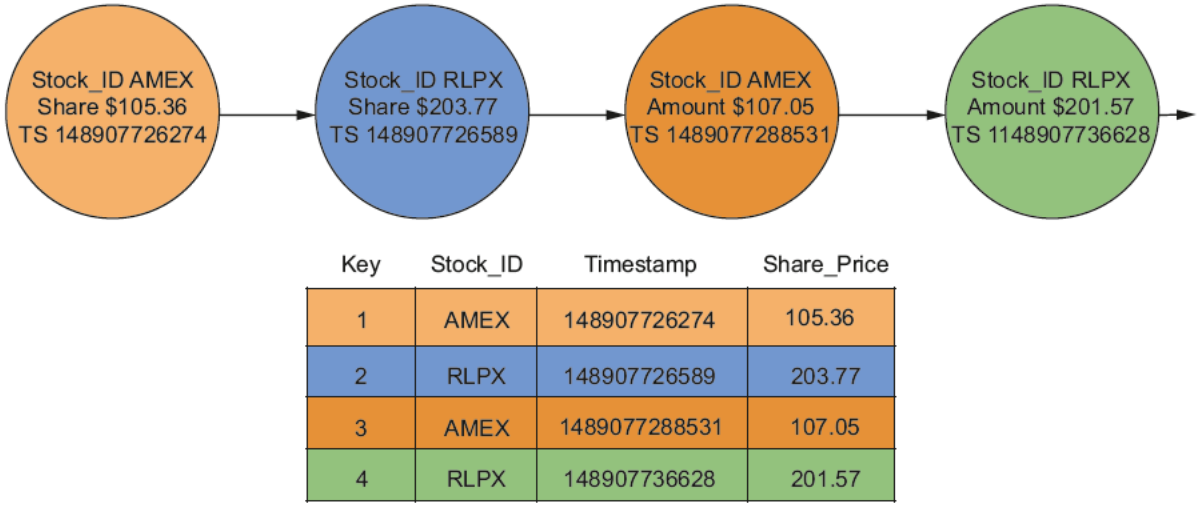
The rows from table above can be recast as key/value pairs. For example, the first row in the table can be converted to this key/value pair:

{key:{stockid:1235588}, value:{ts:3225544289, price:105.36}}

Figure 5.2 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.

> **NOTE** To keep our discussion straightforward, we'll assume a key to be a singular value.

Next, let's take another look at the record stream. Because each record stands on its own, the stream represents inserts into a table. Figure 5.3 combines the two concepts to illustrate this point.



Key	Stock_ID	Timestamp	Share_Price
1	AMEX	148907726274	105.36
2	RLPX	148907726589	203.77
3	AMEX	1489077288531	107.05
4	RLPX	148907736628	201.57

This shows the relationship between events and inserts into a database. Even though it's stock prices for two companies, it counts as four events because you consider each item on the stream as a singular event.

As a result, each event is an insert, and you increment the key by one for each insert into the table.

With that in mind, each event is a new, independent record or insert into a database table.

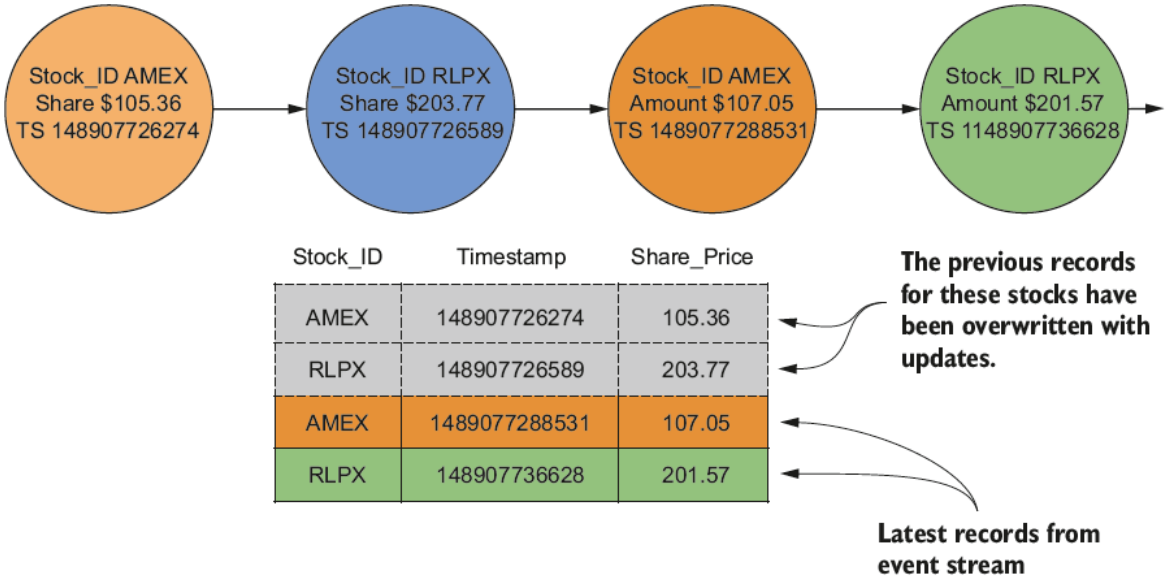
Figure 5.3 A stream of individual events compares to inserts into a database table. You could similarly imagine streaming each row from the table.

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

5.1.2 Updates to records or the changelog

Let's 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 5.4 demonstrates this concept.



If you use the stock ID as a primary key, subsequent events with the same key are updates in a changelog. In this case, you only have two records, one per company. Although more records can arrive for the same companies, the records won't accumulate.

Figure 5.4 In a changelog, each incoming record overwrites the previous one with the same key. With a record stream, you'd have a total of four events, but in the case of updates or a changelog, you have only two.

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

NOTE With both a log and a changelog, records are appended to the end of the file as they come in. The distinction between the two is that in a log, you want to see *all* records, but in a changelog, you only want the *latest* record for each key.

To trim a log while maintaining the latest records per key, you can use log compaction, which we discussed in chapter 2. You can see the impact of compacting a log in figure 5.5. Because you only care about the latest values, you can remove older key/value pairs. [9]

Footnote 9 This section derived information from Jay Kreps's "Introducing Kafka Streams: Stream Processing Made Simple" (http://mng.bz/49HO) and "The Log: What Every Software Engineer Should Know About Real-time Data's Unifying Abstraction" (http://mng.bz/eE3w).

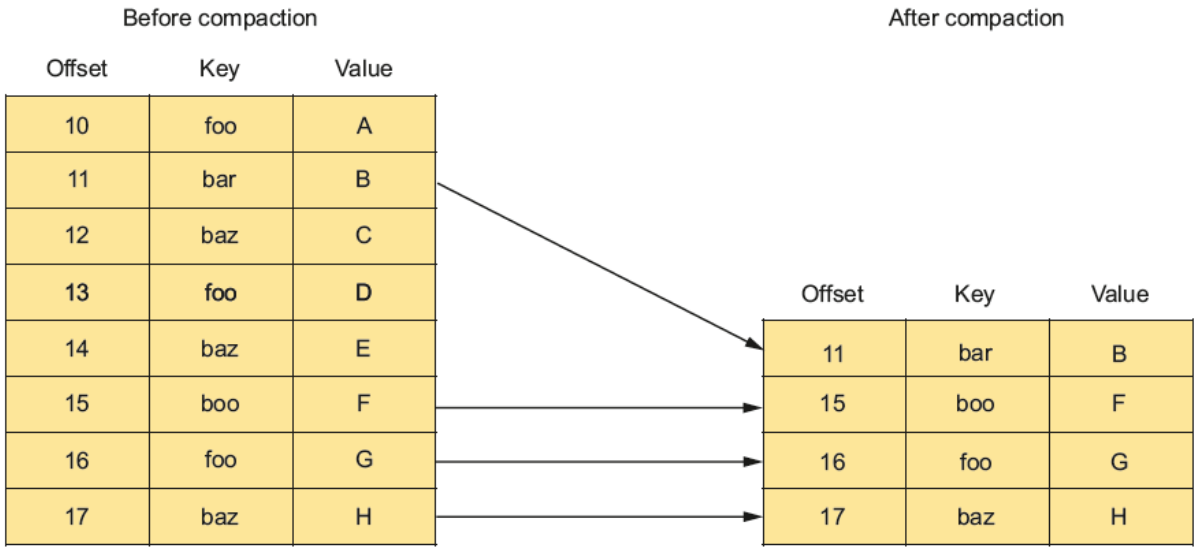


Figure 5.5 On the left is a log before compaction—you'll notice duplicate keys with different values, which are updates. On the right is the log after compaction—you keep the latest value for each key, but the log is smaller in size.

You're already familiar with event streams from working with `KStreams`. For a changelog or stream of updates, we'll use an abstraction known as the `KTable`. Now that we've established the relationship between streams and tables, the next step is to compare an event stream to an update stream.

5.1.3 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.

Figure 5.6 shows the results of running the application. As you can see, the `KStream` printed all nine records. We'd expect the `KStream` to behave this way because it views each record individually. In contrast, the `KTable` printed only three records, because the `KTable` views records as updates to previous ones.

A simple stock ticker for three fictitious companies with a data generator producing three updates for the stocks. The KStream printed all records as they were received. The KTable only printed the last batch of records because they were the latest updates for the given stock symbol.

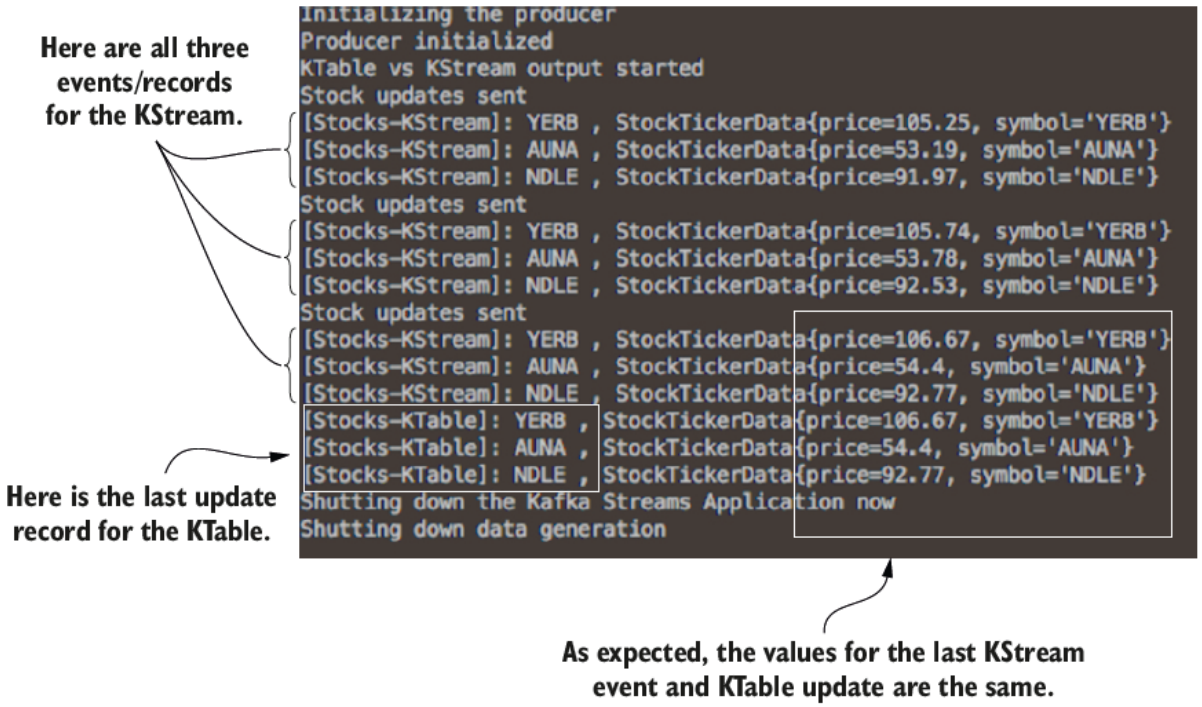


Figure 5.6 KTable versus KStream printing messages with the same keys

> **NOTE** Figure 5.6 demonstrates how a `KTable` works with updates. I made an implicit assumption that I'll make explicit here. When working with a `KTable`, your records must have populated keys in the key/value pairs. Having a key is essential for the `KTable` to work, just as you can't update a record in a database table without having the key.

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_5/KStreamVsKTableExample.java; source code can be found on the book's website here: https://manning.com/books/kafka-streams-in-action).

Listing 5.1 `KTable` and `KStream` printing to the console

```
KTable<String, StockTickerData> stockTickerTable =
```

```
{CA}builder.table(STOCK_TICKER_TABLE_TOPIC);                    ❶
KStream<String, StockTickerData> stockTickerStream =
{CA}builder.stream(STOCK_TICKER_STREAM_TOPIC);                  ❷

stockTickerTable.toStream()
{CA}.print(Printed.<String, StockTickerData>toSysOut()
{CA}.withLabel("Stocks-KTable"));                               ❸

stockTickerStream
{CA}.print(Printed.<String, StockTickerData>toSysOut()
{CA}.withLabel("Stocks-KStream"));                              ❹
```

❶ Creates the KTable instance

❷ Creates the KStream instance

❸ KTable prints results to the console

❹ KStream prints results to the console

SIDEBAR

Using default serdes

In creating the `KTable` and `KStream`, you didn't specify any serdes to use. The same is true with both calls to the `print()` method. You were able to do this because you registered a default serdes in the configuration. like so:

```
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
{CA}Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
{CA}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`.

You've now seen the `KTable` in action, so let's discuss the mechanisms behind its functionality.

5.2 Record updates and KTable configuration

To figure out how the `KTable` functions, we should ask the following two questions:

- Where are records stored?
- How does a `KTable` make the determination to emit records?

As we get into aggregation and reducing operations, the answers to these questions are

necessary. For example, when performing an aggregation, you'll want to see updated counts, but you probably won't want an update each time the count increments by one.

To answer the first question, let's look at the line that creates the `KTable`:

```
builder.table(STOCK_TICKER_TABLE_TOPIC);
```

With this simple statement, the `StreamsBuilder` creates a `KTable` instance and simultaneously, under the covers, creates a `StateStore` for tracking the state of stream, thus creating an update stream. The `StateStore` created by this approach has an internal name and won't be available for interactive queries.

There's an overloaded version of `StreamsBuilder#table` that accepts a `Materialized` instance, allowing 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.

That gives us the answer to our first question: the `KTable` uses the local state integrated with Kafka Streams for storage. (We covered state stores in sections 4.3.1 through 4.3.5.)

Now let's move on to the next question: what determines when the `KTable` emits updates to downstream processors? To answer this question, we need to consider a few factors:

- The number of records flowing into the application. Higher data rates will tend to increase the rate of emitting updated records.
- How many distinct keys are in the data. Again, a greater number of distinct keys may lead to more updates being sent downstream.
- The configuration parameters `cache.max.bytes.buffering` and `commit.interval.ms`.

From this list, we'll only cover what we can control: configuration parameters. First, let's look at the `cache.max.bytes.buffering` configuration.

5.2.1 Setting cache buffering size

The `KTable` cache serves to deduplicate updates to records with the same key. This deduplication allows child nodes to receive only the most recent update instead of all updates, reducing the amount of data processed. Additionally, only the most recent update is placed in the state store, which can amount to significant performance improvements when using persistent state stores.

Figure 5.7 illustrates the cache operation. As you can see, with caching enabled, not all the record updates get forwarded downstream. The cache keeps only the latest record for

any given key.

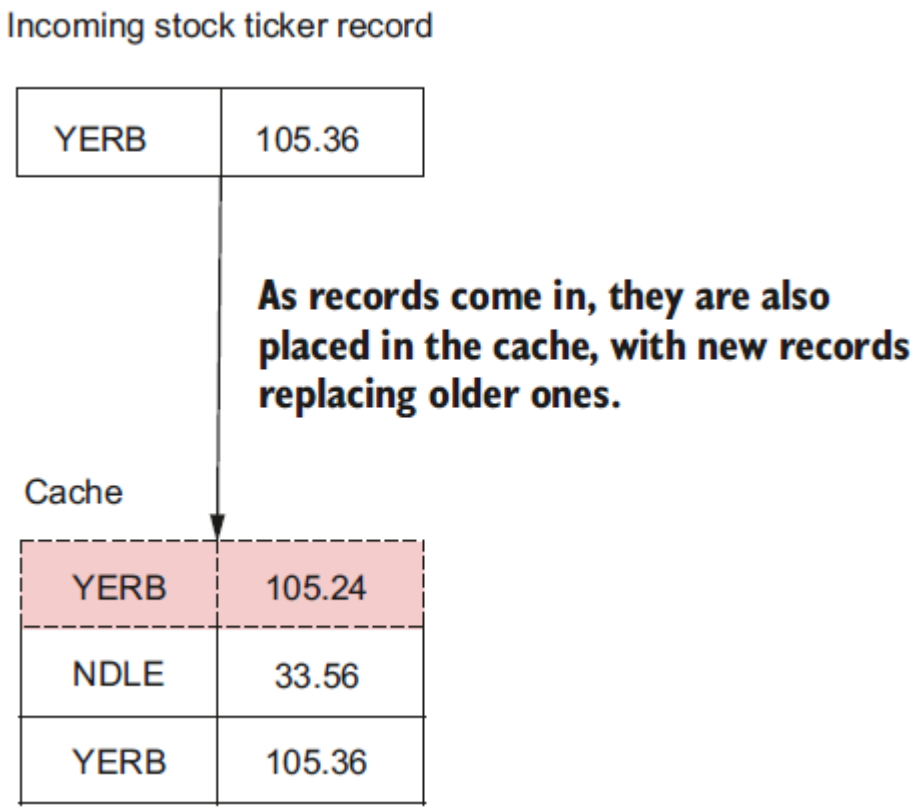


Figure 5.7 KTable caching deduplicates updates to records with the same key, preventing a flood of consecutive updates to child nodes of the KTable in the topology.

NOTE A Kafka Streams application is a topology or graph of connected nodes (processors). Any given node may have one or more child nodes, unless it's a terminal processor. Once a processor has finished working with a record, it forwards the record "downstream" to its child nodes.

Because the `KTable` represents a changelog of events in a stream, you'll expect to work with only the latest update at any given point. Using the cache enforces this behavior. If you wanted to process all records in the stream, you'd use an event stream, the `KStream`, covered earlier.

A larger cache will reduce the number of updates emitted. Additionally, caching reduces the amount of data written to disk by persistent stores (RocksDB), and if logging is enabled, the number of records sent to the changelog topic for any store.

Cache size is controlled by the `cache.max.bytes.buffering` setting, which specifies the amount of memory allocated for the record cache. The amount of memory specified is divided evenly across the number of stream threads. (The number of stream threads is

specified by the `StreamsConfig.NUM_STREAM_THREADS_CONFIG` setting, with 1 being the default.)

> **WARNING** To turn off caching, you can set `cache.max.bytes.buffering` to 0. But this setting will result in every `KTable` update being sent downstream, effectively turning your changelog stream into an event stream. Also, no caching means more I/O, as persistent stores will now write each update to disk instead of only writing the latest update.

5.2.2 Setting the commit interval

The other setting is the `commit.interval.ms` parameter. The commit interval specifies how often (in milliseconds) the state of a processor should be saved. When the state of the processor is saved (committing), it forces a cache flush, sending the latest updated, deduplicated records downstream.

In the full caching workflow (figure 5.8), you can see two forces at play when it comes to sending records downstream. Either a commit or the cache reaching its maximum size will send records downstream. Conversely, disabling the cache will send all records downstream, including duplicate keys. Generally speaking, it’s best to have caching enabled when using a `KTable`.

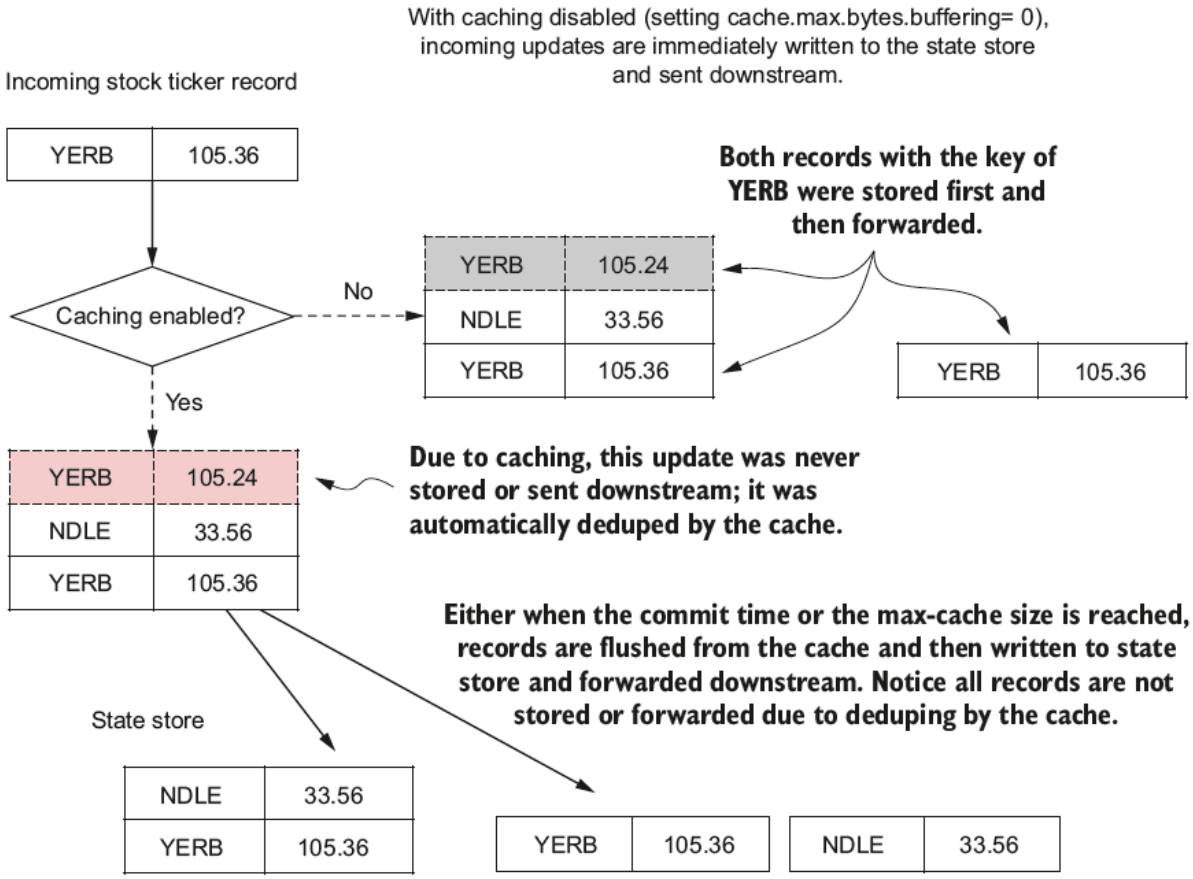


Figure 5.8 Full caching workflow: if caching is enabled, records are deduped and sent downstream on cache flush or commit.

As you can see, we need to strike a balance between cache size and commit time. A large cache with a small commit time will still result in frequent updates. A longer commit interval could lead to fewer updates (depending on the memory settings) because cache evictions occur to free-up space. There are no hard rules here—only trial and error will determine what works best for you. It's best to start with the default values of 30 seconds (commit time) and 10 MB (cache size). The key thing to remember is that the rate of updated records sent from a `KTable` is configurable.

Next, let's take a look at how you can use the `KTable` in your applications.

5.3 Aggregations and windowing operations

In this section, we'll move on to cover some of the most potent parts of Kafka Streams. So far, we've looked at several aspects of Kafka Streams:

- How to set up a processing topology
- How to use state in your streaming application
- How to perform joins between streams
- The difference between an event stream (`KStream`) and an update stream (`KTable`)

In the examples that follow, we'll tie all these elements together. Additionally, I'll introduce windowing, another powerful tool in streaming applications. The first example is a straightforward aggregation.

5.3.1 Aggregating share volume by industry

Aggregation and grouping are necessary tools when you're working with streaming data. Reviewing single records as they arrive is often not enough. To gain any insight, you'll need grouping and combining of some sort.

In this example, you'll take on the role of being a day trader, and you'll track the share volume of companies across a list of selected industries. In particular, you're interested in the top five companies (by share volume) in each industry.

To do this aggregation, a few steps will be required to set up the data in the correct format. At a high level, these are the steps:

1. *Create a source from a topic publishing raw stock-trade information.* You'll need to map a `StockTransaction` object into a `ShareVolume` object. The reason for performing this mapping step is simple: the `StockTransaction` object contains metadata about the trade, but you only want the volume of shares involved in the trade.
2. *Group* `ShareVolume` *by its ticker symbol.* Once it's grouped by symbol, you can reduce it to a rolling total of share volume. I should note here that calling `KStream.groupBy` returns a `KGroupedStream` instance. Then, calling `KGroupedStream.reduce` is what will get you to a `KTable` instance.

SIDEBAR

What is the KGroupedStream?

When you use `KStream.groupBy` or `KStream.groupByKey`, the returned instance is a `KGroupedStream`. The `KGroupedStream` is an intermediate representation of the event stream after grouping by keys and is never meant for you to work with directly. Instead, the `KGroupedStream` is required to perform aggregation operations, which always result in a `KTable`. Because the aggregate operations produce a `KTable` and use a state store, not all updates may end up being forwarded downstream.

There's an analogous `KGroupedTable` resulting from the `KTable.groupBy` method, which is the intermediate representation of the update stream regrouped by key.

Let's pause for a minute and look at figure 5.9, which shows what you've built so far. This topology is something you're familiar with by now.

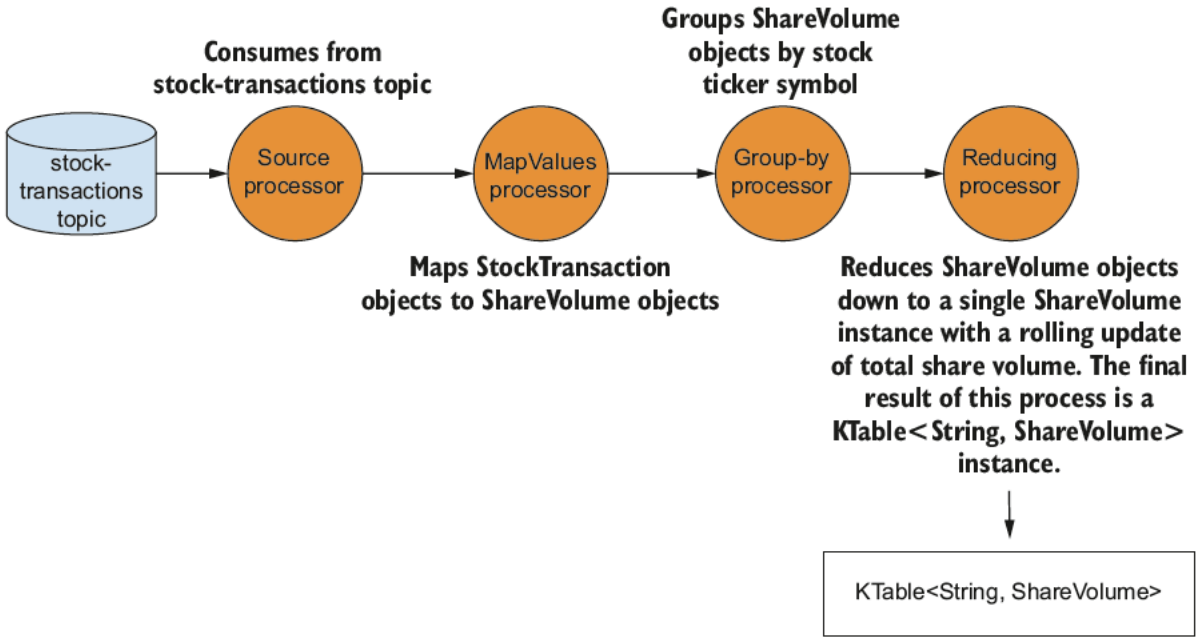


Figure 5.9 Mapping and reducing StockTransaction objects into ShareVolume objects and then reducing to a rolling total

Now, let's look at the code behind the topology (found in src/main/java/bbejeck/chapter_5/AggregationsAndReducingExample.java).

Listing 5.2 Source for the map-reduce of stock transactions

```
KTable<String, ShareVolume> shareVolume =
 {CA}builder.stream(STOCK_TRANSACTIONS_TOPIC,
                    Consumed.with(stringSerde, stockTransactionSerde)
 {CA}.withOffsetResetPolicy(EARLIEST))                              (1)

 {CA}.mapValues(st -> ShareVolume.newBuilder(st).build())           (2)
 {CA}.groupBy((k, v) -> v.getSymbol(),
                       Serialized.with(stringSerde, shareVolumeSerde))   (2)
               {CA}.reduce(ShareVolume::reduce);                    (3)
```

1. The source processor consumes from a topic.
2. Maps StockTransaction objects to ShareVolume objects
3. Groups the ShareVolume objects by their stock ticker symbol
4. Reduces ShareVolume objects to contain a rolling aggregate of share volume

This code is concise and squeezes a lot of power into a few lines. If you look at the first parameter of the `builder.stream` method, you'll see something new: the `AutoOffsetReset.EARLIEST` enum (there's also a `LATEST`) that you set with the

`Consumed.withOffsetResetPolicy` method. This enum allows you to specify the offset-reset strategy for each `KStream` or `KTable`. Using the enum takes precedence over the offset-reset setting in the streams configuration.

SIDEBAR

GroupByKey vs. GroupBy

`KStream` has two methods for grouping records: `GroupByKey` and `GroupBy`. Both return a `KGroupedTable`, so you might wonder what the difference is and when you should use which one.

The `GroupByKey` method is for when your `KStream` already has non-null keys. More importantly, the "needs repartitioning" flag is never set.

The `GroupBy` method assumes you've modified the key for the grouping, so the repartition flag is set to `true`. After calling `GroupBy`, joins, aggregations, and the like will result in automatic repartitioning.

The bottom line is that you should prefer `GroupByKey` over `GroupBy` whenever possible.

It's clear what `mapValues` and `groupBy` are doing, but let's look into the `sum()` method (found in src/main/java/bbejeck/model/ShareVolume.java).

Listing 5.3 The `ShareVolume.sum` method

```
public static ShareVolume sum(ShareVolume csv1, ShareVolume csv2) {
      Builder builder = newBuilder(csv1);  (1)
      builder.shares = csv1.shares + csv2.shares;  (2)
      return builder.build();  (2)
}
```

1. Uses a Builder for a copy constructor
2. Sets the number of shares to the total of both ShareVolume objects
3. Calls build and returns a new ShareVolume object

NOTE

You've seen the builder pattern in use earlier in the book, but it's used here in a somewhat different context. In this example, you're using the builder to make a copy of an object and update a field without modifying the original object.

The `ShareVolume.sum` method gives you the rolling total of share volume, and the outcome of the entire processing chain is a `KTable<String, ShareVolume>` object. Now, you can see the role of the `KTable`. As `ShareVolume` objects come in, the

associated `KTable` keeps the most recent update. It's important to remember that every update is reflected in the preceding `shareVolumeKTable`, but each update is not necessarily emitted.

> **NOTE** Why do a reduce instead of an aggregation? Although reducing is a form of aggregation, a reduce operation will yield the same type of object. An aggregation also sums results, but it could return a different type of object.

Next, you'll take the `KTable` and use it to perform a top-five aggregation (by share volume) summary. The steps you'll take here are similar to the steps for the first aggregation:

1. Perform another `groupBy` operation to group the individual `ShareVolume` objects by industry.
2. Start to add the `ShareVolume` objects. This time, the aggregation object is a fixed-size priority queue. The fixed-size queue keeps only the top-five companies by share volume.
3. Map the queue into a string, reporting the top-five stocks per industry by share volume.
4. Write out the string result to a topic.

Figure 5.10 shows a topology graph of the data flow. As you can see, this second round of processing is straightforward.

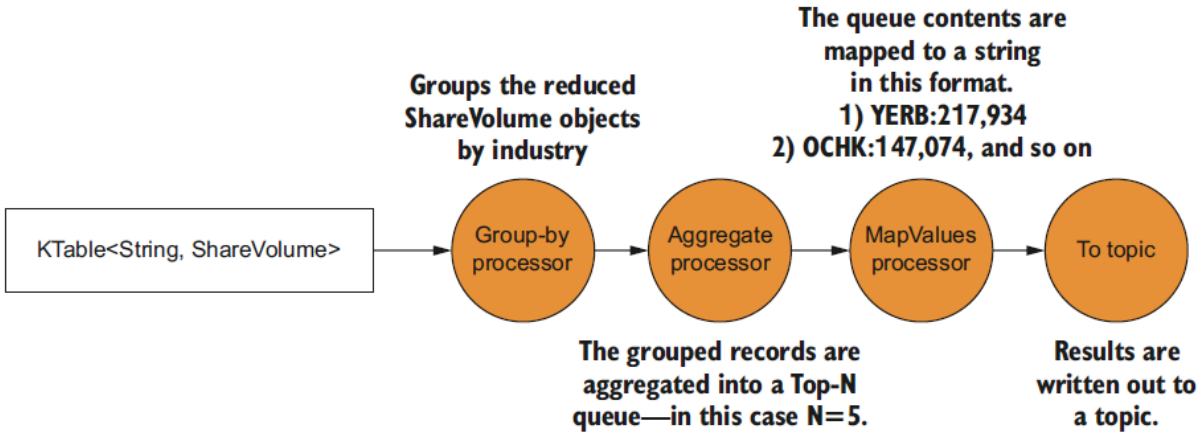


Figure 5.10 Topology for grouping by industry, aggregating by top five, mapping the top-five queue to a string, and writing out the string to a topic

Now that you have a clear picture of the structure of this second round of processing, it's time to look at the source code (found in src/main/java/bbejeck/chapter_5/AggregationsAndReducingExample.java).

Listing 5.4 `KTable`groupBy and aggregation

```
Comparator<ShareVolume> comparator =
{CA}(sv1, sv2) -> sv2.getShares() - sv1.getShares()

FixedSizePriorityQueue<ShareVolume> fixedQueue =
{CA}new FixedSizePriorityQueue<>(comparator, 5);

shareVolume.groupBy((k, v) -> KeyValue.pair(v.getIndustry(), v),
{CA}Serialized.with(stringSerde, shareVolumeSerde))                    (1)
  .aggregate(() -> fixedQueue,                                         (2)
             (k, v, agg) -> agg.add(v),                                (2)
             (k, v, agg) -> agg.remove(v),                             (3)
             Materialized.with(stringSerde, fixedSizePriorityQueueSerde))   (3)
  .mapValues(valueMapper)                                              (4)
  .toStream().peek((k, v) ->
{CA}LOG.info("Stock volume by industry {} {}", k, v))                  (5)
  .to("stock-volume-by-company", Produced.with(stringSerde,
{CA}stringSerde));                                                     (8)
```

1. Groups by industry and provides required serdes
2. The Aggregate initializer is an instance of the FixedSizePriorityQueue class (for demonstration purposes only!).
3. Aggregate adder adds new updates
4. Aggregate remover removes old updates
5. Serde for the aggregator
6. ValueMapper instance converts aggregator to a string that's used for reporting
7. Calls toStream() to log the results (to the console) via the peak method
8. Writes the results to the stock-volume-by-company topic

In this initializer, there's a `fixedQueue` variable. This is a custom object that wraps a `java.util.TreeSet`, which is used to keep track of the top *N* results in decreasing order of share volume.

You've seen the `groupBy` and `mapValues` calls before, so we won't go over them again (you call the `KTable#toStream` method, as `KTable#print` is deprecated). But you haven't seen the `KTable` version of `aggregate` before, so let's take a minute to discuss it.

As you'll recall, what makes a `KTable` unique is that records with the same key are updates. The `KTable` replaces the old record with the new one. Aggregation works in the same manner. It aggregates the most recent records with the same key. As a record arrives, you add it to the `FixedSizePriorityQueue` using the `adder()` method (the second parameter in the `aggregate` call), but if another record exists with the same key,

you remove the old record with the `subtractor` (the third parameter in the `aggregate` call).

What this means is that your aggregator, `FixedSizePriorityQueue`, doesn't aggregate *all* values with the same key. Instead, it keeps a running tally of the top *N* stocks that have the largest volume. Each record coming in has the total volume of shares traded so far. Your `KTable` will show you which companies have the top number of shares traded at the moment; you don't want a running aggregation of each update.

You've now learned how to do two important things:

- Group values in a `KTable` by a common key
- Perform useful operations like reducing and aggregation with those grouped values

The ability to perform these operations is important when you're trying to make sense of your data, or to determine what your data is telling you, as it flows through your Kafka Streams application.

We've also brought together some of the key concepts discussed earlier in the book. In chapter 4, you learned about the importance of having fault-tolerant, local state for streaming applications. The first example in this chapter showed why local state is so important—it allows you to keep track of what you've seen. Having *local* access avoids network latency, making your application more robust and performant.

Any time you execute a reduction or aggregation operation, you provide the name of a state store. Reduction and aggregation operations return a `KTable` instance, and the `KTable` uses the state store to replace older results with the newer ones. As you've seen, not every update gets forwarded downstream, and that's important because you perform aggregation operations to gather *summary* information. If you didn't use local state, the `KTable` would forward *every* aggregation or reduction result.

Next, we'll look at how you can perform aggregation-like operations over distinct periods of time, a process called *windowing*.

5.3.2 Windowing operations

In the previous section, we looked at a "rolling" reduction and aggregation. The application performed a continuous reduction of share volume and then a top-five aggregation of shares traded in the stock market.

Sometimes, you'll want an ongoing aggregation and reduction of results like this. At

other times, you'll only want to perform operations for a given time range. For example, how many stock transactions have involved a particular company in the last 10 minutes? How many users have clicked to view a new advertisement in the last 15 minutes? An application may perform these operations many times, but the results may only be for a defined period or window of time.

COUNTING STOCK TRANSACTIONS BY CUSTOMER

In the next example, you'll track stock transactions for a handful of traders. These could be large institutional traders or financially savvy individuals.

There are two likely reasons for doing this tracking. One reason is that you may want to see where the market leaders are buying and selling. If these big hitters or savvy investors see an opportunity in the market, you'll follow the same strategy. The other reason is that you may want to identify any indications of insider trading. You'll want to look into the timing of large spikes in trading and correlate them with significant news releases.

Here are the steps to do this tracking:

1. Create a stream to read from the stock-transactions topic.
2. Group incoming records by the customer ID and stock ticker symbol. The `groupBy` call returns a `KGroupedStream` instance.
3. Use the `KGroupedStream.windowedBy` method to return a windowed stream, so you can perform some sort of windowed aggregation. Depending on the window type provided, you'll get either a `TimeWindowedKStream` or a `SessionWindowedKStream` in return.
4. Perform a count for the aggregation operation. The windowing stream determines whether the record is included in the count.
5. Write the results to a topic, or print the results to the console during development.

The topology for this application is straightforward, but it's helpful to have a mental picture of the structure. Take a look at figure 5.11.

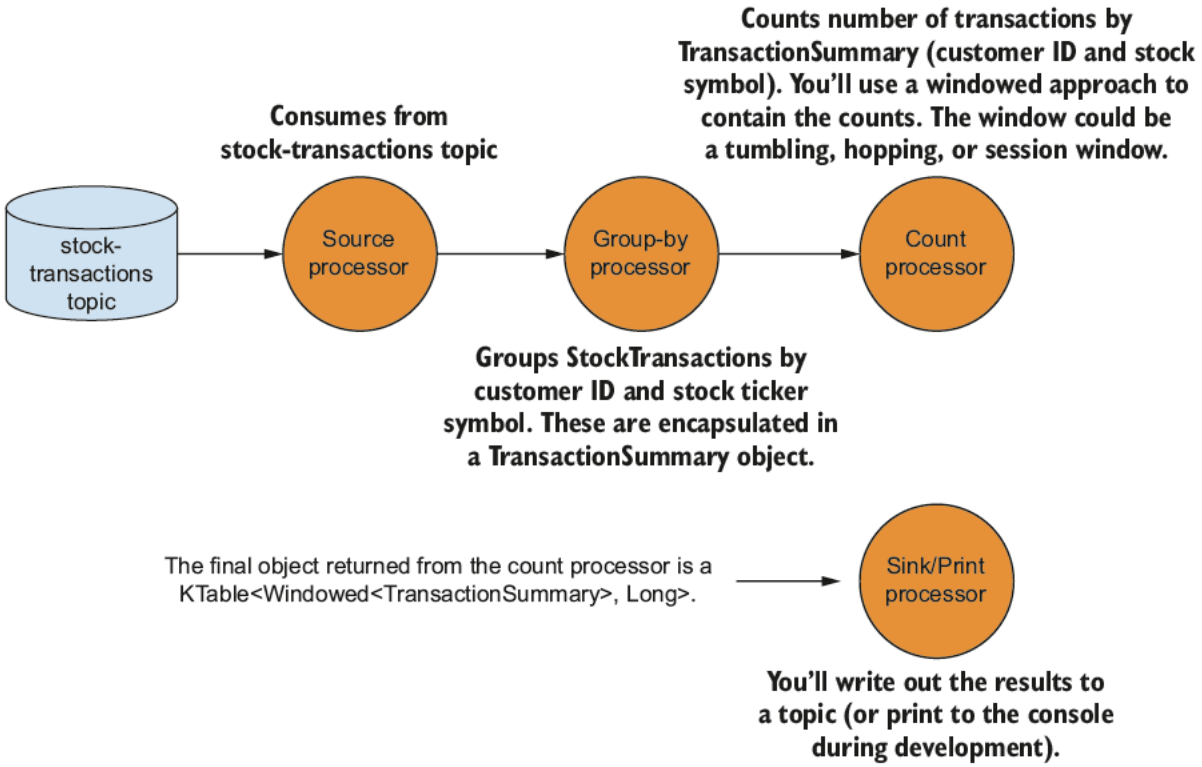


Figure 5.11 Counting windows topology

Next, let's look at the windowing functionality and corresponding code.

WINDOW TYPES

In Kafka Streams, three types of windows are available:

- Session windows
- Tumbling windows
- Sliding/hopping windows

Which type you choose depends on your business requirements. The tumbling and hopping windows are time bound, whereas session windows are more about user activity; the length of the session(s) is determined solely by how active the user is. A key point to keep in mind for all windows is that they're based on the timestamps in the records and not wall-clock time.

Next, you'll implement the topology with each of the window types. We'll only look at the full code in the first example. Other than changing the type of window operation, everything will remain the same for the other windows.

SESSION WINDOWS

Session windows are very different from other windows. Session windows aren't bound strictly by time as much as by user activity (or the activity of anything you want to track). You delineate session windows by a period of inactivity.

Figure 5.12 shows how you can view session windows. The smaller session will be merged with the session on the left. But the session on the right will be a new session because it follows a large inactivity gap. Session windows are based on user activity, but they use timestamps in the records to determine which session a record belongs to.

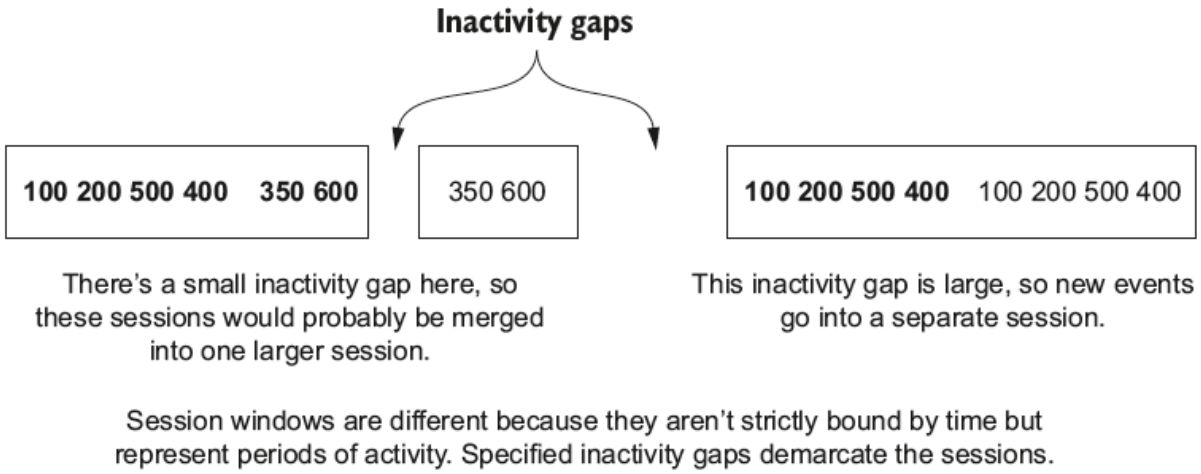


Figure 5.12 Session windows separated by a small inactivity gap are combined to form a new, larger session.

USING SESSION WINDOWS TO TRACK STOCK TRANSACTIONS

Let's use session windows to capture the stock transactions. The following code (found in src/main/java/bbejeck/chapter_5/CountingWindowingAndKTableJoinExample.java) shows how to implement the session windows.

Listing 5.5 Tracking stock transactions with session windows

```
Serde<String> stringSerde = Serdes.String();
Serde<StockTransaction> transactionSerde =
{CA}StreamsSerdes.StockTransactionSerde();

Serde<TransactionSummary> transactionKeySerde =
{CA}StreamsSerdes.TransactionSummarySerde();


long twentySeconds = 1000 * 20;
long fifteenMinutes = 1000 * 60 * 15;
KTable<Windowed<TransactionSummary>, Long>
{CA}customerTransactionCounts =                    (1)
{CA}builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,
{CA}transactionSerde)
```

```
.withOffsetResetPolicy(LATEST))                    (1)
.groupBy((noKey, transaction) ->

{CA}TransactionSummary.from(transaction),          (2)
{CA}Serialized.with(transactionKeySerde, transactionSerde))
.windowedBy(SessionWindows.with(twentySeconds).

{CA}until(fifteenMinutes)).count();                (2)


customerTransactionCounts.toStream()
{CA}.print(Printed.<Windowed<TransactionSummary>, Long>toSysOut()

{CA}.withLabel("Customer Transactions Counts"));   (3)
```

1. KTable resulting from the groupBy and count calls
2. Creates the stream from the STOCK_TRANSACTIONS_TOPIC (a String constant). Uses the offset-reset-strategy enum of LATEST for this stream.
3. Groups records by customer ID and stock symbol, which are stored in the TransactionSummary object.
4. Windows the groups with SessionWindow with an inactivity time of 20 seconds and a retention time of 15 minutes. Then performs an aggregation as a count.
5. Converts the KTable output to a KStream and prints the result to the console

You've seen most of the operations specified in this topology before, so we don't need to cover them again. But there are a couple of new items, and we'll discuss those now.

Any time you do a `groupBy` operation, you'll typically perform some sort of aggregation operation (aggregate, reduce, or count). You can perform a cumulative aggregation where previous results continue to build up, or you can perform windowed aggregations where records are combined for the specified time of the window.

The code in listing 5.5 does a count over session windows. Figure 5.13 breaks it down.

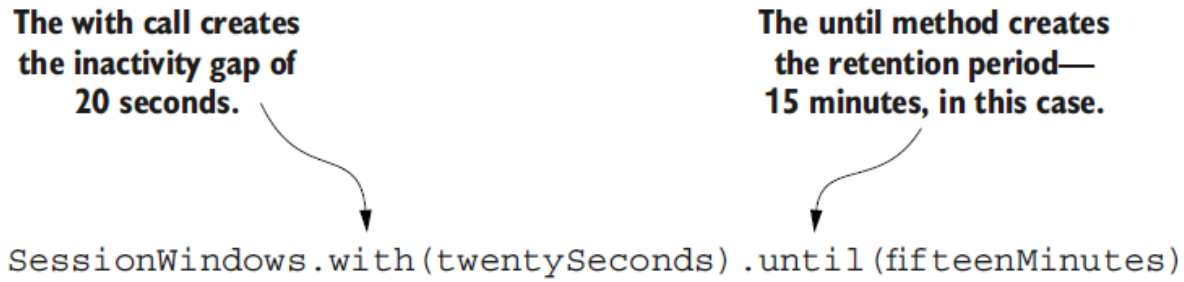


Figure 5.13 Creating session windows with inactivity periods and retention

With the call to `windowedBy(SessionWindows.with(twentySeconds).until(fifteenMinutes))`, you create a session window with an inactivity gap of 20 seconds and a retention period of 15 minutes. An inactivity time of 20 seconds means the application includes any

record arriving within 20 seconds of the current session's ending or start time within the current (active) session.

You then specify the aggregation operation to perform—a `count`, in this case—on the session window. If an incoming record falls outside the inactivity gap (on either side of the timestamp), the application creates a new session. The retention period maintains the session for the specified amount of time and allows for late-arriving data that's outside the inactivity period of a session but can still be merged. Additionally, as sessions are combined, the newly created session uses the earliest timestamp and latest timestamp for the start and end of the new session, respectively.

Let's walk through a few records from the `count()` method to see sessions in action: see table 5.1.

Table 5.1 Sessioning table with a 20-second inactivity gap

Arrival order	Key	Timestamp
1	{123-345-654,FFBE}	00:00:00
2	{123-345-654,FFBE}	00:00:15
3	{123-345-654,FFBE}	00:00:50
4	{123-345-654,FFBE}	00:00:05

As records come in, you look for existing sessions with the same key, with ending times less than `current timestamp – inactivity gap`, and with starting times greater than `current timestamp + inactivity gap`. With that in mind, here's how the four records in table 5.15.1 end up being merged into a single session:

1. Record 1 is first, so start and end are 00:00:00.
2. Record 2 arrives, and you look for sessions with an earliest ending of 23:59:55 and a latest start of 00:00:35. You find record 1, so you merge sessions 1 and 2. You keep the session 1 start time (earliest) and the session 2 end time (latest), so you have one session starting at 00:00:00 and ending at 00:00:15.
3. Record 3 arrives, and you look for sessions between 00:00:30 and 00:01:10 and find none. You add a second session for key 123-345-654, FFBE starting and ending at 00:00:50.
4. Record 4 arrives, and you search for sessions between 23:59:45 and 00:00:25. This time you find both sessions 1 and 2. All three are merged into one session with a start time of 00:00:00 and an end time of 00:00:15.

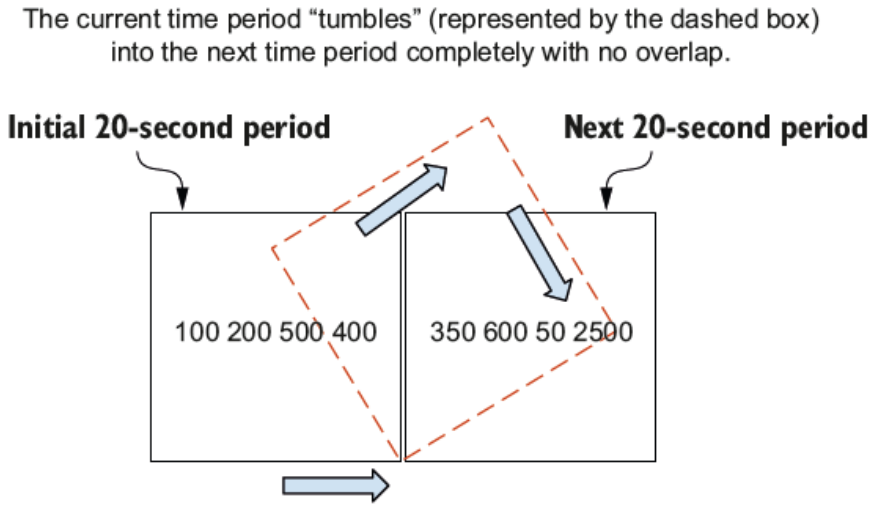
There are a couple of key points to remember from this section:

- Sessions are not a fixed-size window. Rather, the size of a session is driven by the amount of activity within a given time frame.
- Timestamps in the data determine whether an event fits into an existing session or falls into an inactivity gap.

Now, we'll move on to the next windowing option, tumbling windows.

TUMBLING WINDOWS

Fixed or *tumbling* windows capture events within a given period. Imagine you want to capture all stock transactions for a company every 20 seconds, so you collect every event for that time. After the 20-second period is over, your window will "tumble" to a new 20-second observation period. Figure 5.14 shows this situation.



The box on the left is the first 20-second window. After 20 seconds, it "tumbles" over or updates to capture events in a new 20-second period.

There is no overlapping of events. The first event window contains [100, 200, 500, 400] and the second event window contains [350, 600, 50, 2500].

Figure 5.14 Tumbling windows reset after a fixed period.

As you can see, each event that has come in for the last 20 seconds is included in the window. A new window is created after the specified time.

Here's how you could use tumbling windows to capture stock transactions every 20 seconds (found in src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java).

Listing 5.6 Using tumbling windows to count user transactions

```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =
{CA}builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,
                                                           transactionSerde)
{CA}.withOffsetResetPolicy(LATEST))
    .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),
             {CA}Serialized.with(transactionKeySerde, transactionSerde))

    .windowedBy(TimeWindows.of(twentySeconds)).count();    (1)
```

❶ Specifies a tumbling window of 20 seconds

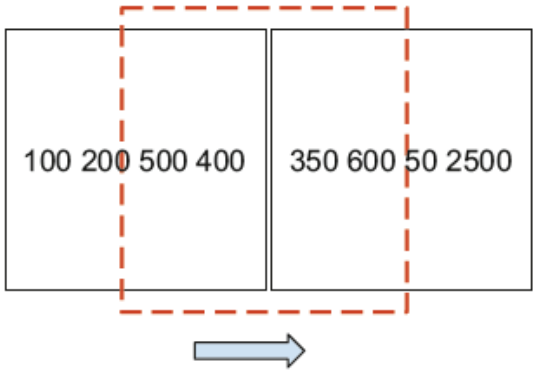
With this minor change of the `TimeWindows.of` call, you can use a tumbling window. This example doesn't include the `until()` method. By not specifying the duration of the window, you'll get the default retention of 24 hours.

Finally, we'll move on to the last of the windowing options: hopping windows.

SLIDING OR HOPPING WINDOWS

Sliding or *hopping* windows are like tumbling windows but with a small difference. A sliding window doesn't wait the entire time before launching another window to process recent events. Sliding windows perform a new calculation after waiting for an interval smaller than the duration of the entire window.

To illustrate how hopping windows differ from tumbling windows, let's recast the stock transaction count example. You still want to count the number of transactions, but you don't want to wait the entire period before you update the count. Instead, you'll update the count at *smaller* intervals. For example, you'll still count the number of transactions every 20 seconds, but you'll update the count every 5 seconds, as shown in figure 5.15. You now have three result windows with overlapping data.



The box on the left is the first 20-second window, but the window "slides" over or updates after 5 seconds to start a new window. Now you see an overlapping of events. Window 1 contains [100, 200, 500, 400], window 2 contains [500, 400, 350, 600], and window 3 is [350, 600, 50, 2500].

Figure 5.15 Sliding windows update frequently and may contain overlapping data.

Here's how to specify hopping windows with code (found in src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java).

Listing 5.7 Specifying hopping windows to count transactions

```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =
{CA}builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,
{CA}transactionSerde)
{CA}.withOffsetResetPolicy(LATEST))
.groupBy((noKey, transaction) -> TransactionSummary.from(transaction),
{CA}Serialized.with(transactionKeySerde, transactionSerde))
.windowedBy(TimeWindows.of(twentySeconds)
{CA}.advanceBy(fiveSeconds).until(fifteenMinutes)).count();    (1)
```

(1) Uses a hopping window of 20 seconds, advancing every 5 seconds

With the addition of the `advanceBy()` method, you can convert a tumbling window to a hopping window. This example specifies a retention time of 15 minutes.

NOTE You'll notice in all the windowing examples presented here that the only code that changes is the `windowedBy` call. Instead of having four nearly identical example classes in the sample code, I've included four different windowing lines in the src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java file. To see a different windowing operation in action, comment out the current windowing operation and uncomment the one you want to execute.

You've now seen how to put your aggregation results into time windows. In particular, I want you to remember three things from this section:

- Session windows aren't fixed by time but are driven by user activity.
- Tumbling windows give you a set picture of events within the specified time frame.
- Hopping windows are of fixed length, but they're frequently updated and can contain overlapping records in each window.

Next, we'll look at how to convert a `KTable` back into a `KStream` to perform a join.

5.3.3 Joining KStreams and KTables

In chapter 4, we discussed joining two `KStreams`. Now, you're going to join a `KTable` and a `KStream`. The reason to do this is simple. `KStreams` are record streams, and `KTables` are streams of record updates, but sometimes you may need to add some additional context to your record stream with the updates from a `KTable`.

Let's take the stock transaction counts and join them with financial news from relevant industry sectors. Here are the steps to make this happen with the existing code:

1. Convert the `KTable` of stock transaction counts into a `KStream` where you change the key to the industry of the count by ticker symbol.

2. Create a `KTable` that reads from a topic of financial news. The new `KTable` will be categorized by industry.
3. Join the news updates with the stock transaction counts by industry.

With these steps laid out, let's walk through how you can accomplish these tasks.

CONVERTING THE KTABLE TO A KSTREAM

To do the `KTable`-to-`KStream` conversion, you'll take the following steps:

1. Call the `KTable.toStream()` method.
2. Use the `KStream.map` call to change the key to the industry name, and extract the `TransactionSummary` object from the `Windowed` instance.

These steps are chained together in the following manner (found in src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java).

Listing 5.8 Converting a `KTable` to a `KStream`

```
KStream<String, TransactionSummary> countStream =
{CA}customerTransactionCounts.toStream().map((window, count) -> {    (1)
    TransactionSummary transactionSummary = window.key();            (1)
    String newKey = transactionSummary.getIndustry();                (2)
    transactionSummary.setSummaryCount(count);                       (2)
    return KeyValue.pair(newKey, transactionSummary);                (2)
        });
```

(1) Calls toStream, immediately followed by the map call

(2) Extracts the TransactionSummary object from the Windowed instance

(3) Sets the key to the industry segment of the stock purchase

(4) Takes the count value from the aggregation and places it in the TransactionSummary object

(5) Returns the new KeyValue pair for the KStream

Because you're performing a `KStream.map` operation, repartitioning for the returned `KStream` instance is done automatically when it's used in a join.

Now that you have the conversion process completed, the next step is to create the `KTable` to read the financial news.

CREATING THE FINANCIAL NEWS KTABLE

Fortunately, creating the `KTable` involves just one line of code (found in src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java).

Listing 5.9 `KTable` for financial news

```
KTable<String, String> financialNews =
{CA}builder.table( "financial-news", Consumed.with(EARLIEST));
```
(1)

(1) Creates the KTable with the EARLIEST enum, topic financial-news

It's worth noting here that you don't need to provide any serdes because the configuration is using string serdes. Also, by using the enum `EARLIEST`, you populate the table with records on startup.

Now, we'll move on to the last step, setting up the join.

JOINING NEWS UPDATES WITH TRANSACTION COUNTS

Setting up the join is very simple. You'll use a left join, in case there's no financial news for the industry involved in the transaction (found in src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java).

Listing 5.10 Setting up the join between the `KStream` and `KTable`

```
ValueJoiner<TransactionSummary, String, String> valueJoiner =
{CA}(txnct, news) ->
{CA}String.format("%d shares purchased %s related news [%s]",
{CA}txnct.getSummaryCount(), txnct.getStockTicker(), news);

KStream<String,String> joined =
{CA}countStream.leftJoin(financialNews, valueJoiner,
{CA}Joined.with(stringSerde, transactionKeySerde, stringSerde));

joined.print(Printed.<String, String>toSysOut()
{CA}.withLabel("Transactions and News"));
```
(1) (2) (3)

(1) 0-1)) ValueJoiner combines the values from the join result.

(2) 0-2)) The leftJoin statement for the countStream KStream and the financial news KTable, providing serdes with a Joined instance

(3) 0-3)) Prints results to the console (in production this would be written to a topic with a to("topic-name") call)

The `leftJoin` statement is straightforward. Unlike the joins in chapter 4, you don't

provide a `JoinWindow` because, when performing a `KStream`-to-`KTable` join, there's only one record per key in the `KTable`. The join is unrelated to time; the record is either present in the `KTable` or not. The key point here is that you can use `KTables` to provide less-frequently updated lookup data to enrich your `KStream` counterparts.

Next, we'll look at a more efficient way to enhance `KStream` events.

5.3.4 GlobalKTables

We've established the need to enrich or add context to our event streams. You've also seen joins between two `KStreams` in chapter 4, and the previous section demonstrated a join between a `KStream` and a `KTable`. In all of these cases, when you map the keys to a new type or value, the stream needs to be repartitioned. Sometimes you'll do the repartitioning explicitly yourself, and other times Kafka Streams will do it automatically. Repartitioning makes sense, because the keys have been changed and will end up on new partitions or the join won't happen (this was discussed in the chapter 4 section, "Repartitioning the data").

REPARTITIONING HAS A COST

Repartitioning isn't free. There's additional overhead in this process: creating intermediate topics, storing duplicate data in another topic, and increased latency due to writing to and reading from another topic. Additionally, if you want to join on more than one facet or dimension, you'll need to chain joins, map the records with new keys, and repeat the repartitioning process.

JOINING WITH SMALLER DATASETS

In some cases, the lookup data you want to join against will be relatively small, and entire copies of the lookup data could fit locally on each node. For situations where the lookup data is reasonably small, Kafka Streams provides the `GlobalKTable`.

`GlobalKTables` are unique because the application replicates all the data to each node. Because the entirety of the data is on each node, the event stream doesn't need to be partitioned by the key of the lookup data in order to make it available to all partitions. `GlobalKTables` also allow you to do non-key joins. Let's revisit one of the previous examples to demonstrate this capability.

JOINING KSTREAMS WITH GLOBALKTABLES

In section 5.3.2, you performed a windowed aggregation of stock transactions per customer. The output of the aggregation looked like this:

```
{customerId='074-09-3705', stockTicker='GUTM'}, 17
{customerId='037-34-5184', stockTicker='CORK'}, 16
```

Although this output accomplished the goal, it would have more impact if you could see the client's name and the full company name. You could perform regular joins to fill out the customer and company names, but you'd need to do two key mappings and repartitioning. With `GlobalKTables`, you can avoid the expense of those operations. To accomplish this, you'll use the `countStream` from the following listing (found in src/main/java/bbejeck/chapter_5/GlobalKTableExample.java) and join it against two `GlobalKTables`.

Listing 5.11 Aggregating stock transactions using session windows

```
KStream<String, TransactionSummary> countStream =
  builder.stream( STOCK_TRANSACTIONS_TOPIC,
{CA}Consumed.with(stringSerde, transactionSerde)
{CA}.withOffsetResetPolicy(LATEST))
         .groupBy((noKey, transaction) ->
{CA}TransactionSummary.from(transaction),
{CA}Serialized.with(transactionSummarySerde, transactionSerde))
        .windowedBy(SessionWindows.with(twentySeconds)).count()
        .toStream().map(transactionMapper);
```

We covered this previously, so we won't review it again here. But note that the code in the `toStream().map` function is abstracted into a function object instead of having an in-line lambda, for readability purposes.

The next step is to define two `GlobalKTable` instances (found in src/main/java/bbejeck/chapter_5/GlobalKTableExample.java).

Listing 5.12 Defining the `GlobalKTables` for lookup data

```
GlobalKTable<String, String> publicCcompanies =
{CA}builder.globalTable(COMPANIES.topicName());  (1)

GlobalKTable<String, String> clients =
{CA}builder.globalTable(CLIENTS.topicName());    (2)
```

(1) The publicCompanies lookup is for finding companies by their stock ticker symbol.

(2) The clients lookup is for getting customer names by customer ID.

Note that the topic names are defined using enums.

Now that the components are in place, you need to construct the join (found in

src/main/java/bbejeck/chapter_5/GlobalKTableExample.java).

Listing 5.13 Joining a `KStream` with two `GlobalKTables`

```
countStream.leftJoin(publicCompanies, (key, txn) ->
{CA}txn.getStockTicker(),TransactionSummary::withCompanyName)    (1)
          .leftJoin(clients, (key, txn) ->
{CA}txn.getCustomerId(), TransactionSummary::withCustomerName)    (2)
          .print(Printed.<String, TransactionSummary>toSysOut()
{CA}.withLabel("Resolved Transaction Summaries"));    (3)
```

(1) Sets up the leftJoin with the publicCompanies table, keys by stock ticker symbol, and returns the transactionSummary with the company name added

(2) Sets up the leftJoin with the clients table, keys by customer ID, and returns the transactionSummary with the customer named added

(3) Prints the results out to the console

Although there are two joins here, they're chained together because you don't use any of the results alone. You print the results at the end of the entire operation.

If you run the join operation, you'll now get results like this:

```
{customer='Barney, Smith' company="Exxon", transactions= 17}
```

The facts haven't changed, but these results are clearer for reading.

Including chapter 4, you've seen several types of joins in action. They're listed in table 5.2. This table represents the state of join options as of Kafka Streams 1.0.0, but this may change in future releases.

Table 5.2 Kafka Streams joins

Left join	Inner join	Outer join
`KStream-KStream`	`KStream-KStream`	`KStream-KStream`
`KStream-KTable`	`KStream-KTable`	N/A
`KTable-KTable`	`KTable-KTable`	`KTable-KTable`
`KStream-GlobalKTable`	`KStream-GlobaKTable`	N/A

In conclusion, the key thing to remember is that you can combine event streams (`KStream`) and update streams (`KTable`), using local state. Additionally, when the lookup data is of a manageable size, you can use a `GlobalKTable`. `GlobalKTables` replicate all partitions to each node in the Kafka Streams application, making all data available, regardless of which partition the key maps to.

Next, we'll look at a capability of Kafka Streams that allows you to observe changes to state without having to consume data from a Kafka topic.

5.3.5 Queryable state

You've performed several operations involving state, and you've always printed the results to the console (for development) or written them to a topic (for production). When you write the results to a topic, you need to use a Kafka consumer to view the results.

Reading the data from these topics could be considered a form of *materialized views*. For our purposes, we can use Wikipedia's definition of a materialized view: "a database object that contains the results of a query. For example, it may be a local copy of data located remotely, or may be a subset of the rows and/or columns of a table or join result, or may be a summary using an aggregate function" (https://en.wikipedia.org/wiki/Materialized_view).

Kafka Streams also offers *interactive queries* from state stores, giving you the ability to read these materialized views directly. It's important to note that querying state stores is a read-only operation. By making the queries read-only, you don't have to worry about creating an inconsistent state while the application continues to process data.

The impact of making the state stores directly queryable is significant. It means you can create dashboard applications without having to consume the data from a Kafka consumer first. There are also some gains in efficiency resulting from not writing the data out again:

- Because the data is local, you can access it quickly.
- You avoid duplicating data by not copying it to an external store.[10]

Footnote 10 For more details, see Eno Thereska, "Unifying Stream Processing and Interactive Queries in Apache Kafka," http://mng.bz/dh1H.

The main thing I want you to remember here is that you can query state from the application directly. I can't stress enough the power this feature gives you. Instead of consuming from Kafka and storing records in a database to feed your application, you can directly query the state stores for the same results. The impact of direct queries on state stores means less code (no consumer) and less software (no need for a database table to store results).

We've covered a lot in this chapter, so we'll stop here in our discussion of interactive queries on state stores. But fear not: in chapter 9, you'll build a simple dashboard

application with interactive queries. It will use some of the examples from this and previous chapters to demonstrate interactive queries and how you can add them to your Kafka Streams applications.

5.4 Summary

- `KStreams` represent event streams that are comparable to inserts into a database. `KTables` are update streams and are more akin to updates to a database. The size of a `KTable` doesn't continue to grow; older records are replaced with newer records.
- `KTables` are essential for performing aggregation operations.
- You can place your aggregated data into time buckets with windowing operations.
- Global`KTables` give you lookup data across the entire application, regardless of the partitions.
- You can perform joins with `KStreams`, `KTables`, and `GlobalKTables`.

So far, we've focused on the high-level `KStreams` DSL to build Kafka Streams applications. Although the high-level approach gives nice, concise programs, everything is a trade-off. By working with the `KStreams` DSL, you relinquish a level of control and gain more-concise code. In the next chapter, we'll cover the low-level Processor API and make different trade-offs. You won't have the conciseness of the applications you've built so far, but you'll gain the ability to create virtually any kind of processor you need.

6 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.

6.1 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.

6.2 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 6.1. 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.

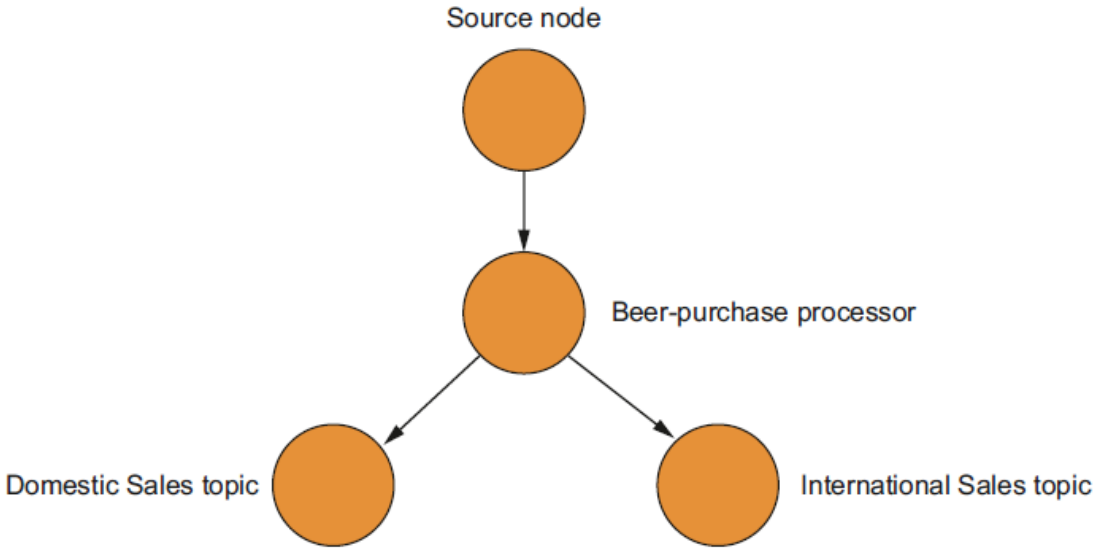


Figure 6.1 Beer sales distribution pipeline

6.2.1 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_6/PopsHopsApplication.java) sets the data source for the new topology.

Listing 6.1 Creating the beer application source node

```
topology.addSource(LATEST,                                          (1)
                   purchaseSourceNodeName,                          (2)
                   new UsePreviousTimeOnInvalidTimestamp(),         (3)
                   stringDeserializer,                              (4)
```

```
                beerPurchaseDeserializer,                    5
                Topics.POPS_HOPS_PURCHASES.topicName())      5
```

1. Specifies the offset reset to use
2. Specifies the name of this node
3. Specifies the TimestampExtractor to use for this source
4. Sets the key deserializer
5. Sets the value deserializer
6. Specifies the name of the topic to consume data from

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. In section 4.5.1, we discussed the different timestamp extractors available to use for each stream source. Here, you're using the `UsePreviousTimeOnInvalidTimestamp` class; all other sources in the application will use the default `FailOnInvalidTimestamp` class.

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 work with purchase records coming into the application.

6.2.2 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_6/PopsHopsApplication.java).

Listing 6.2 Adding a processor node

```
BeerPurchaseProcessor beerProcessor =
```

```
[CA]new BeerPurchaseProcessor(domesticSalesSink, internationalSalesSink);

topology.addSource(LATEST,
                   purchaseSourceNodeName,
                   new UsePreviousTimeOnInvalidTimestamp(),
                   stringDeserializer,
                   beerPurchaseDeserializer,
                   Topics.POPS_HOPS_PURCHASES.topicName())
       .addProcessor(purchaseProcessor,          (1)
                     () -> beerProcessor,        (2)
                     purchaseSourceNodeName);    (2)
```

(1) Names the processor node

(2) Adds the processor defined above

(3) Specifies the name of the parent node or nodes

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 the processor instantiated on the first line of the code example. 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 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 6.2. 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 6.3 reviews what you've built so far.

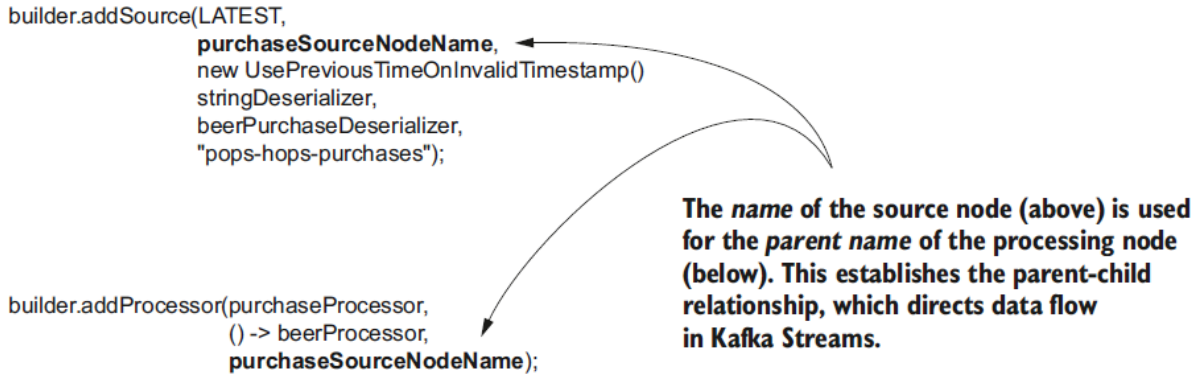


Figure 6.2 Wiring up parent and child nodes in the Processor API

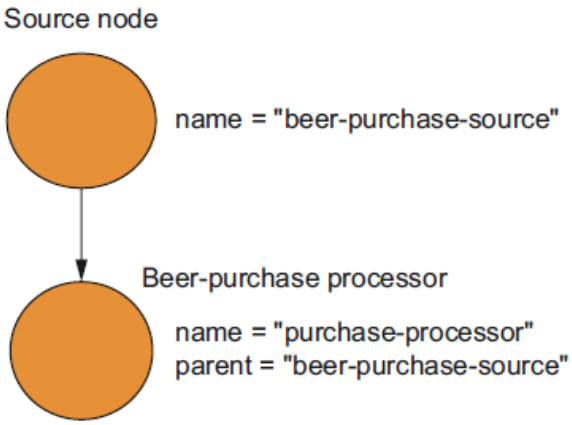


Figure 6.3 The Processor API topology so far, including node names and parent names

Let's take a second to discuss the `BeerPurchaseProcessor`, created in listing 6.1, 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_6/processor/BearPurchaseProcessor.java).

Listing 6.3 `BeerPurchaseProcessor`

```
public class BeerPurchaseProcessor extends
[CA]AbstractProcessor<String, BeerPurchase> {

    private String domesticSalesNode;
    private String internationalSalesNode;

    public BeerPurchaseProcessor(String domesticSalesNode,
                                 String internationalSalesNode) {
        this.domesticSalesNode = domesticSalesNode;

        this.internationalSalesNode = internationalSalesNode;          (1)
    }

    @Override

    public void process(String key, BeerPurchase beerPurchase) {      (2)
```

```
        Currency transactionCurrency = beerPurchase.getCurrency();

        if (transactionCurrency != DOLLARS) {
            BeerPurchase dollarBeerPurchase;
            BeerPurchase.Builder builder =
            [CA]BeerPurchase.newBuilder(beerPurchase);
            double internationalSaleAmount = beerPurchase.getTotalSale();
            String pattern = "###.##";
            DecimalFormat decimalFormat = new DecimalFormat(pattern);
            builder.currency(DOLLARS);
            builder.totalSale(Double.parseDouble(decimalFormat.
[CA]format(transactionCurrency
[CA].convertToDollars(internationalSaleAmount))));
            dollarBeerPurchase = builder.build();
            context().forward(key,
            [CA}dollarBeerPurchase, internationalSalesNode);
        } else {
            context().forward(key, beerPurchase, domesticSalesNode);
        }

    }
}
```

1. Sets the names for different nodes to forward records to
2. The process() method, where the action takes place
3. Converts international sales to US dollars
4. Uses the ProcessorContext (returned from the context() method) and forwards records to the international child node
5. Sends records for domestic sales to the domestic child node

This example extends `AbstractProcessor`, 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()`, `punctuate()`, 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 mostly use the `AbstractProcessor` class, so you'll only override the methods you want. The `AbstractProcessor` 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 6.3 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.

6.2.3 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 6.4 shows the updated topology.

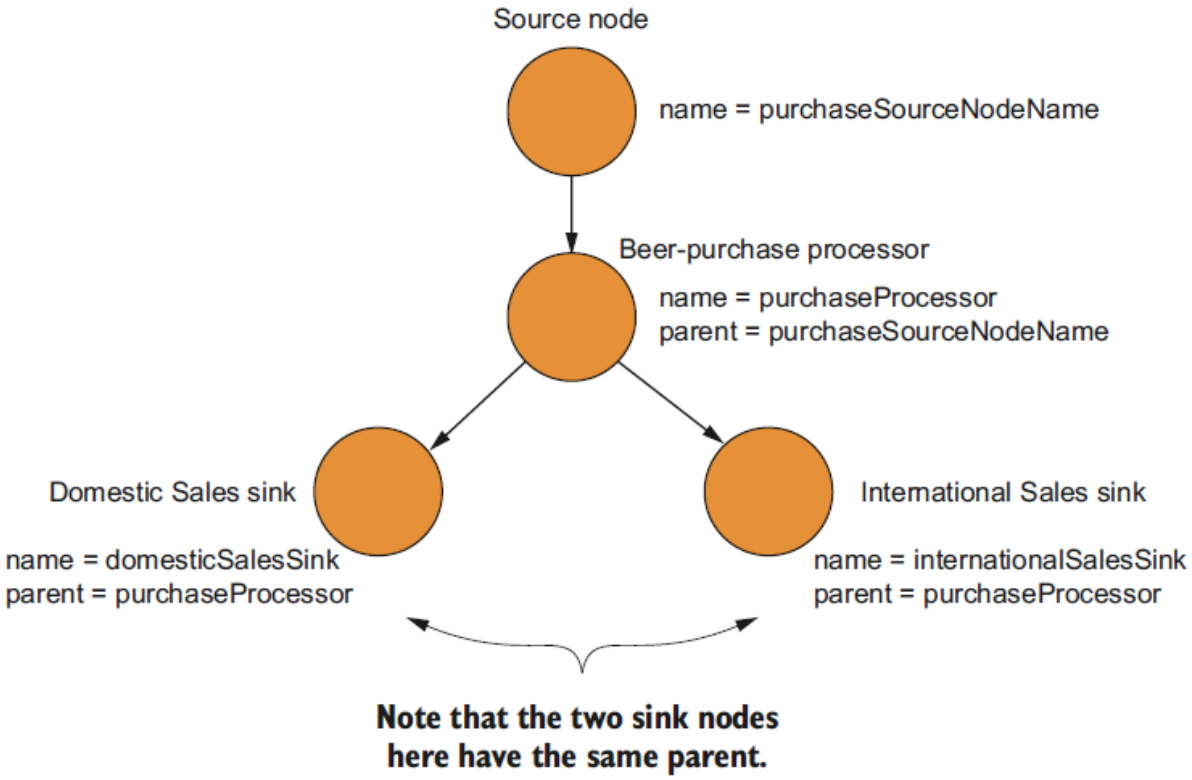


Figure 6.4 Completing the topology by adding sink nodes

You can update the topology you're building by adding sink nodes in the code now (found in src/main/java/bbejeck/chapter_6/PopsHopsApplication.java).

Listing 6.4 Adding a sink node

```
topology.addSource(LATEST,
                   purchaseSourceNodeName,
                   new UsePreviousTimeOnInvalidTimestamp(),
                   stringDeserializer,
                   beerPurchaseDeserializer,
                   Topics.POPS_HOPS_PURCHASES.topicName())
       .addProcessor(purchaseProcessor,
                     () -> beerProcessor,
                     purchaseSourceNodeName)
```

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

.addSink(domesticSalesSink,               (4)
         "domestic-sales",                (5)
         stringSerializer,                (8)
         beerPurchaseSerializer,          (4)
         purchaseProcessor);              (5)
```

(1) Name of the sink

(2) The topic this sink represents

(3) Serializer for the key

(4) Serializer for the value

(5) Parent node for this sink

In this listing, you add two sink nodes, one for dollars and another for euros. 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.

6.3 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.

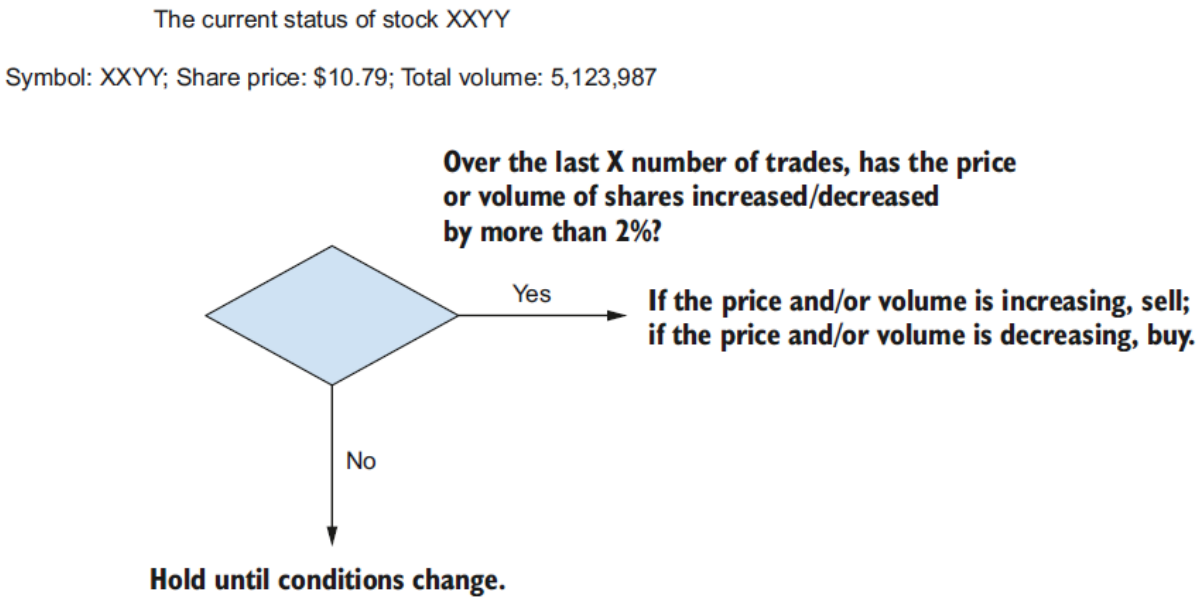


Figure 6.5 Stock trend updates

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.

SIDEBAR **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!

6.3.1 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 6.5 Stock-performance application with custom processor

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

KeyValueBytesStoreSupplier storeSupplier =
[CA]Stores.inMemoryKeyValueStore(stocksStateStore);                      (1)
StoreBuilder<KeyValueStore<String, StockPerformance>> storeBuilder
[CA]= Stores.keyValueStoreBuilder(
[CA]storeSupplier, Serdes.String(), stockPerformanceSerde);              (2)


  topology.addSource("stocks-source",
                     stringDeserializer,
                     stockTransactionDeserializer,
                     "stock-transactions")
          .addProcessor("stocks-processor",
[CA]() -> new StockPerformanceProcessor(
[CA]stocksStateStore, differentialThreshold), "stocks-source")           (2)
          .addStateStore(storeBuilder,"stocks-processor")                (3)
          .addSink("stocks-sink",
                   "stock-performance",
                   stringSerializer,
                   stockPerformanceSerializer,
                   "stocks-processor");                                  (6)
```

(1) Sets the percentage differential for forwarding stock information

(2) Creates an in-memory key/value state store

(3) Creates the StoreBuilder to place in the topology

(4) Adds the processor to the topology

(5) Adds the state store to the stocks processor

(6) Adds a sink for writing results out, although you could use a printing sink as well

This topology has the same flow as the previous example, so 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 `AbstractProcessor#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).

Listing 6.6 `init()` method tasks

```
@Override
public void init(ProcessorContext processorContext) {
  super.init(processorContext);                                    (1)
  keyValueStore =
[CA](KeyValueStore) context().getStateStore(stateStoreName);       (1)
  StockPerformancePunctuator punctuator =
[CA]new StockPerformancePunctuator(differentialThreshold,
                                   context(),
                                   keyValueStore);                 (2)
  context().schedule(10000, PunctuationType.WALL_CLOCK_TIME,
[CA]punctuator);                                                   (2)
 }
}
```

(1) Initializes ProcessorContext via the AbstractProcessor superclass

(2) Retrieving state store created when building topology

(3) Initializing the Punctuator to handle the scheduled processing

(4) Schedules the Punctuator#punctuate to be called every 10 seconds

First, you need to initialize the `AbstractProcessor` 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 6.5 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(long, 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 (http://mng.bz/YSKF). I don't have examples or a discussion here, but I present some examples in src/main/java/bbejeck/chapter_6/cancellation.

In the last line of listing 6.5, 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 `WALL_CLOCK_TIME`. Your other option is to specify `PunctuationType.STREAM_TIME`, which means the execution of `Punctuator.punctuate` is still scheduled every 10 seconds but driven by the time

elapsed according to timestamps in the data. Let's take a moment to discuss the difference between these two `PunctuationType` 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 `PartitionGroup`, then Kafka Streams calls that processor's `punctuate()` method.

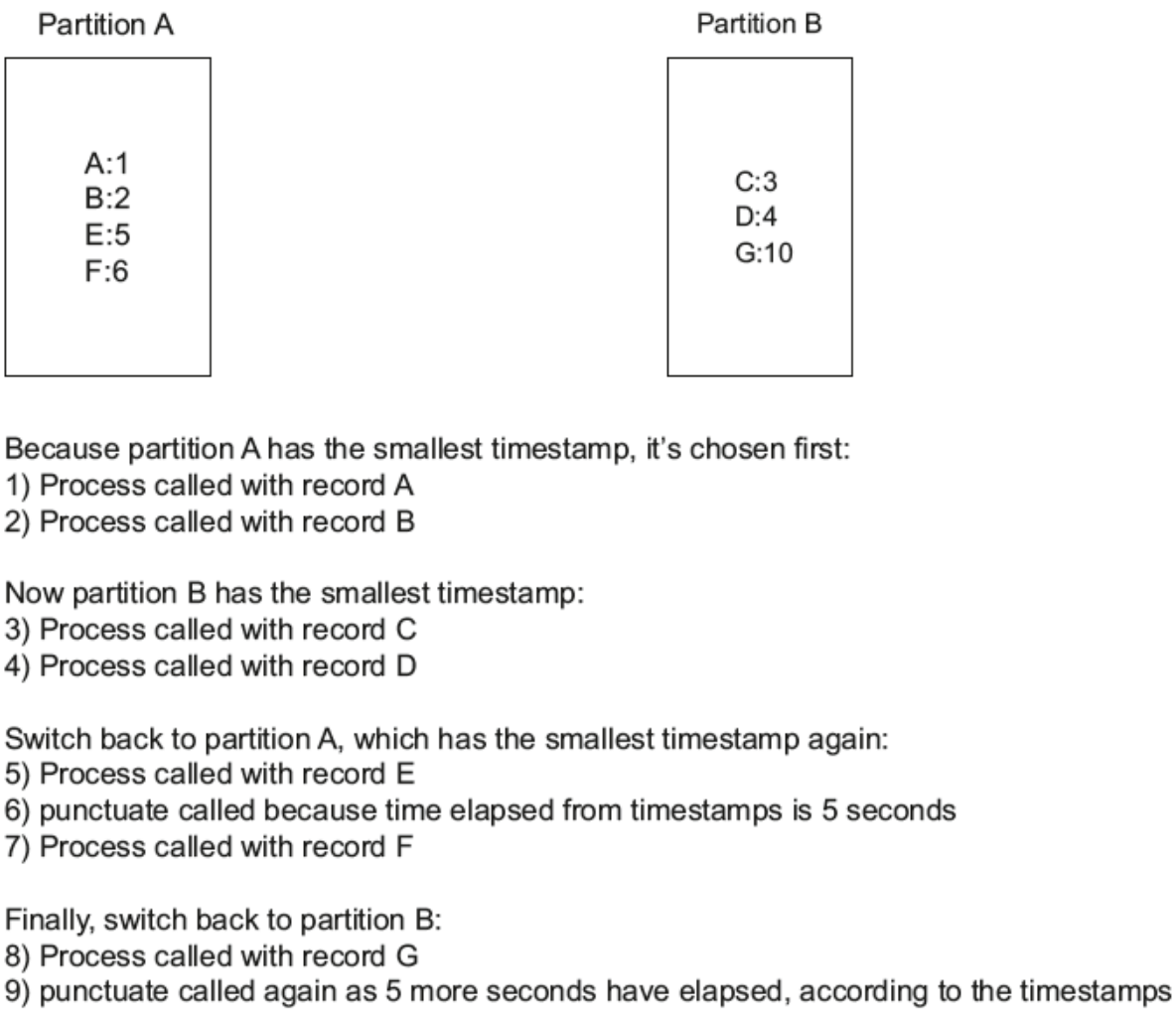


Figure 6.6 Punctuation scheduling using STREAM_TIME

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 6.6, 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.

6.3.2 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 6.7 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.

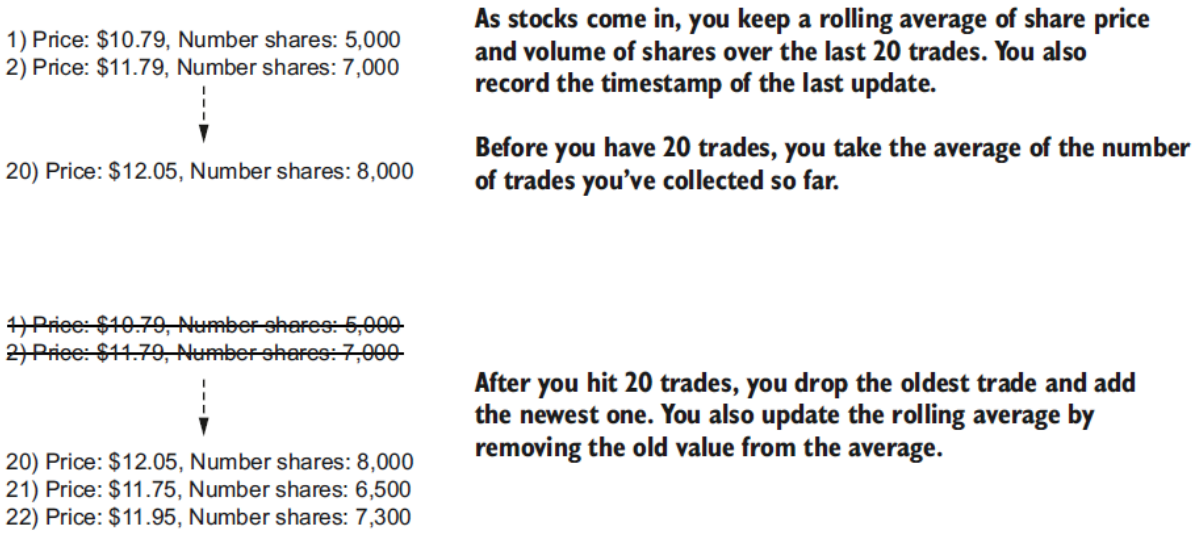


Figure 6.7 Stock analysis process() method walk-through

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

Listing 6.7 `process()` implementation

```
@Override
public void process(String symbol, StockTransaction transaction) {
 StockPerformance stockPerformance = keyValueStore.get(symbol);          (1)

 if (stockPerformance == null) {
  stockPerformance = new StockPerformance();                             (1)
 }

 stockPerformance.updatePriceStats(transaction.getSharePrice());        (1)
 stockPerformance.updateVolumeStats(transaction.getShares());           (2)
 stockPerformance.setLastUpdateSent(Instant.now());                      (3)

 keyValueStore.put(symbol, stockPerformance);                            (3)
}
```

(1) Retrieves previous performance stats, possibly null

(2) Creates a new StockPerformance object if one isn't in the state store

(3) Updates the price statistics for this stock

(4) Updates the volume statistics for this stock

(5) Sets the timestamp of the last update

(6)

- Places the updated StockPerformance object into the state store

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.

6.3.3 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_6/processor/punctuator/StockPerformancePunctuator.java).

Listing 6.8 Punctuation code

```
@Override
public void punctuate(long timestamp) {
  KeyValueIterator<String, StockPerformance> performanceIterator =
[CA]keyValueStore.all();   (1)

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

        if (stockPerformance != null) {
            if (stockPerformance.priceDifferential()
```

```
[CA}>= differentialThreshold ||
                stockPerformance.volumeDifferential()
[CA]>= differentialThreshold) {                                    ❶

                context.forward(key, stockPerformance);            ❷
            }
        }
  }
}
```

❶ Retrieves the iterator to go over all key values in the state store

❷ Checks the threshold for the current stock

❸ If you've met or exceeded the threshold, forwards the record

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, but you can take writing custom processors a bit further by adding a new data structure and 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 a co-group processor.

6.4 The co-group processor

Back in chapter 4, we discussed joins between two streams; specifically, we joined purchases from different departments within a given time frame to promote business. You can use joins to bring together records that have the same key and that arrive in the same time window. With joins, there's an implied one-to-one mapping of records from stream A to stream B. Figure 6.8 depicts this relationship.

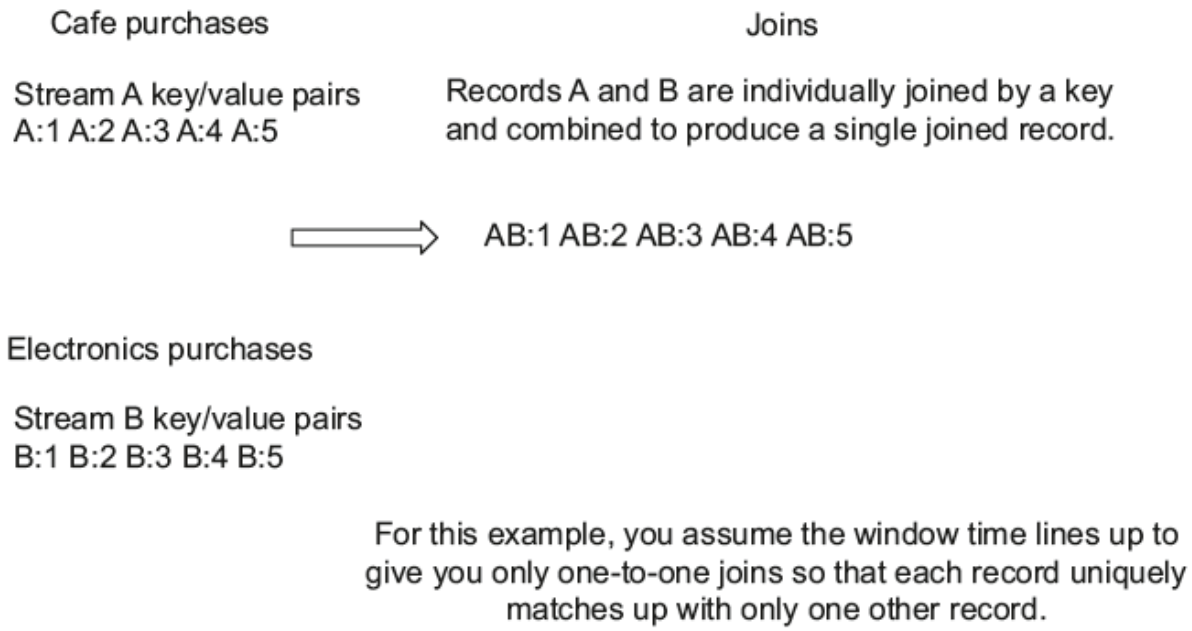


Figure 6.8 Records A and B joined by a common key

Now, let's imagine that you want to do a similar type of analysis, but instead of using a one-to-one join by key, you want two collections of data joined by a common key, a *co-grouping* of data. Suppose you're the manager of a popular online day-trading application. Day traders use your application for several hours a day—sometimes the entire time the stock market is open. One of the metrics your application tracks is the notion of events. You've defined an *event* as being when a user clicks on a ticker symbol to read more about a company and its financial outlook. You want to do some deeper analysis between those user clicks in the application and the purchase of stocks by users. You want course-grained results, comparing multiple clicks and purchases to determine some overall pattern. What you need is a tuple with two collections of each event type by the company trading symbol, as shown in figure 6.9.

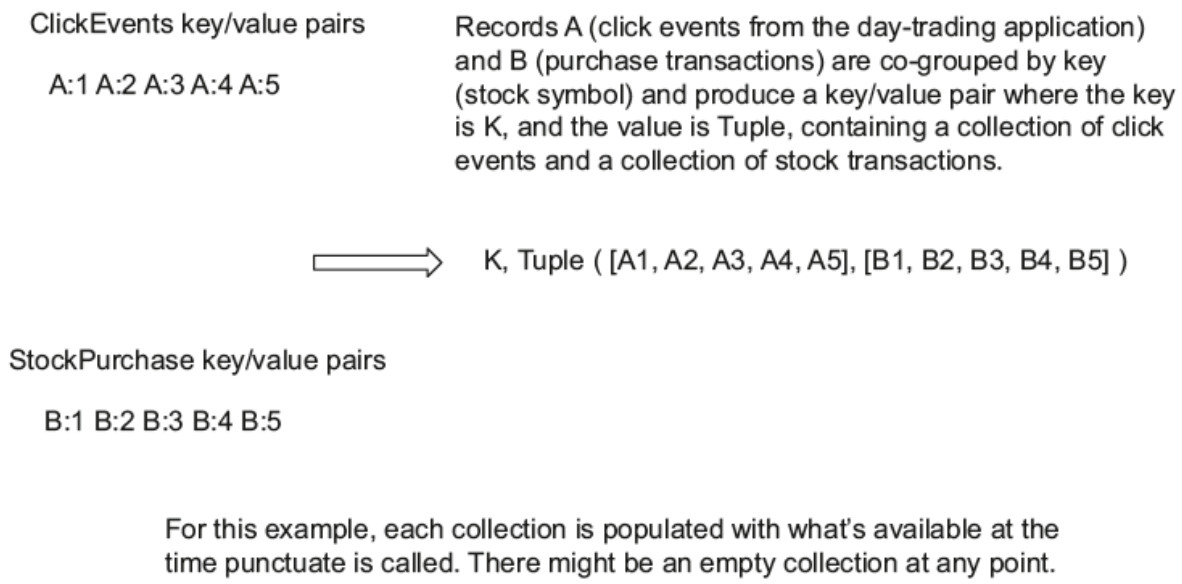


Figure 6.9 Output of a key with a tuple containing two collections of data—a co-grouped result

Your goal is to combine snapshots of click events and stock transactions for a given company, every *N* seconds, but you aren't waiting for records from either stream to arrive. When the specified amount of time goes by, you want a co-grouping of click events and stock transactions by company ticker symbol. If either type of event isn't present, one of the collections in the tuple will be empty. If you're familiar with Apache Spark or Apache Flink, this functionality is similar to the `PairRDDFunctions.cogroup` method (http://mng.bz/LaD4) and the `CoGroupDataSet` class (http:// mng.bz/FH9m), respectively. Let's walk through the steps you'll take in constructing this processor.

6.4.1 Building the co-grouping processor

To create the co-grouping processor, you need to tie a few pieces together:

1. Define two topics (`stock-transactions`, `events`).
2. Add two processors to consume records from the topics.
3. Add a third processor to act as an aggregator/co-grouping for the two preceding processors.
4. Add a state store for the aggregating processor to keep the state for both events.
5. Add a sink node to write the results to (and/or a printing processor to print results to console).

Now, lets walk through the steps to put this processor together.

DEFINING THE SOURCE NODES

You're already familiar with the first step, creating the source nodes. This time, you'll create two source nodes to support reading both the click event stream and the stock-transactions stream. To keep track of where we are in the topology, we'll build on figure 6.10. The code for creating the source nodes is shown in the following listing (found in src/main/java/bbejeck/chapter_6/CoGroupingApplication.java).

Listing 6.9 Source nodes for co-grouping processor

```
//I've left out configuration and (de)serializer creation for clarity.

topology.addSource("Txn-Source",
                   stringDeserializer,
                   stockTransactionDeserializer,
                   "stock-transactions")   (1)
       .addSource("Events-Source",
                   stringDeserializer,
                   clickEventDeserializer,
                   "events")   (2)
```

(1) Source node for the stock-transactions topic

(2) Source node for the events topic

With the sources for the topology in place, let's move on to the next step.

Figure 6.10 Co-grouping source nodes

ADDING THE PROCESSOR NODES

Now, you'll add the workhorses of the topology, the processors. Figure 6.11 shows the updated topology graph. Here's the code for adding these new processors (found in src/main/java/bbejeck/chapter_6/CoGroupingApplication.java).

Listing 6.10 The processor nodes

```
.addProcessor("Txn-Processor",
              StockTransactionProcessor::new,
              "Txn-Source")   (1)

.addProcessor("Events-Processor",
```

```
                ClickEventProcessor::new,
                "Events-Source")                    ❷

.addProcessor("CoGrouping-Processor",
                CogroupingProcessor::new,
                "Txn-Processor",
                "Events-Processor")                 ❸
```

❶ 0-1)) Adds the StockTransactionProcessor

❷ 0-2)) Adds the ClickEventProcessor

❸ 0-3)) Adds the CogroupingProcessor, which is a child node of both processors

In the first two lines, the parent names are the names of source nodes reading from the `stock-transactions` and `events` topics, respectively. The third processor has the names of both processors given as parent nodes. This means both processors will feed the aggregation processor.

For the `ProcessorSupplier` instances, you're again taking a Java 8 shortcut. This time, you've shortened the form, even more, to use a method handle: in this case, a constructor call to create the associated processor.

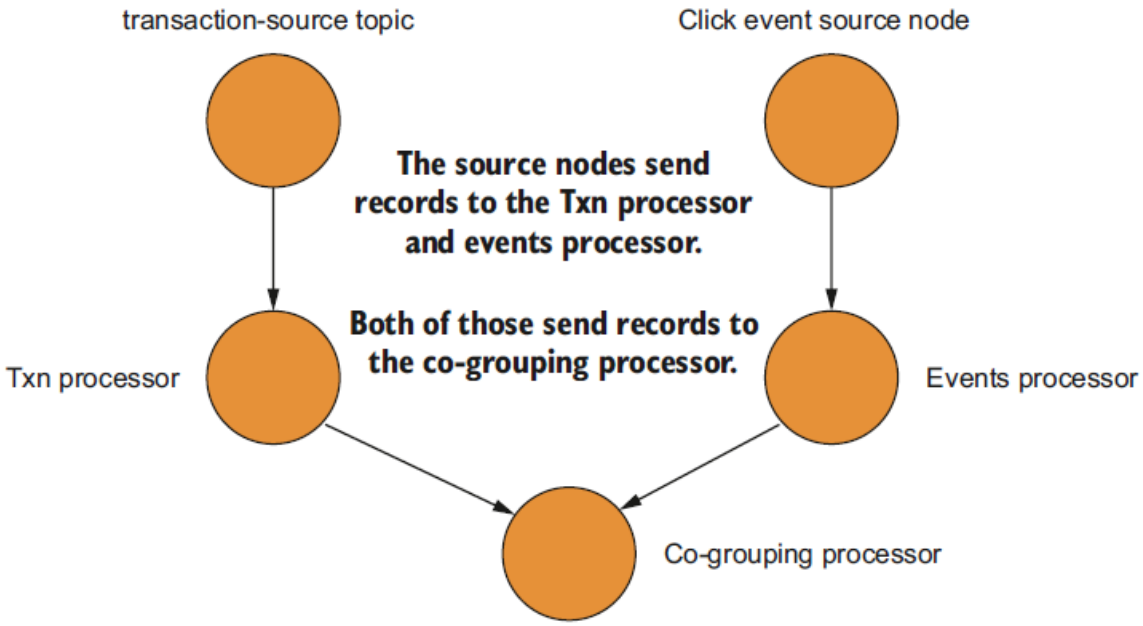


Figure 6.11 Adding processor nodes

TIP With single-method no-arg interfaces in Java 8, you can use a lambda in the form of `() doSomething`. But because the `ProcessorSupplier`'s only role is to return a (possibly new each time) `Processor` object, you can shorten the form even more to use a method handle for the constructor of the `Processor` type. Note that this only applies for no-arg constructors.

Let's look at why you've set up the processors in this manner. This example is an aggregation operation, and the roles of the `StockTransactionProcessor` and `ClickEventProcessor` are to wrap their respective objects into smaller aggregate objects and then forward them to another processor for a total aggregation. Both the `StockTransactionProcessor` and the `ClickEventProcessor` perform the smaller aggregation, and they forward their records to the `CogroupingProcessor`. The `CogroupingProcessor` then performs the co-grouping and forwards the results at regular intervals (intervals driven by timestamps) to an output topic.

The following listing shows the code for the processors (found in src/main/java/bbejeck/chapter_6/processor/cogrouping/StockTransactionProcessor.java).

Listing 6.11 `StockTransactionProcessor`

```
public class StockTransactionProcessor extends
[CA]AbstractProcessor<String, StockTransaction> {

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        super.init(context);
    }

    @Override
    public void process(String key, StockTransaction value) {
        if (key != null) {
            Tuple<ClickEvent, StockTransaction> tuple =
[CA]Tuple.of(null, value);  (1)

            context().forward(key, tuple);  (2)
        }
    }
}
```

(1) Creates an aggregate object with the StockTransaction

(2) Forwards the tuple to the CogroupingProcessor

As you can see, `StockTransactionProcessor` adds the `StockTransaction` to the aggregator (the `Tuple`) and forwards the record.

NOTE The `Tuple<L, R>` shown in listing 6.11 is a custom object for examples in this book. You can find it in src/main/java/bbejeck/util/collection/Tuple.java.

Now, let's look at the `ClickEventProcessor` code (found in src/main/java/bbejeck/chapter_6/processor/cogrouping/ClickEventProcessor.java).

Listing 6.12 `ClickEventProcessor`

```
public class ClickEventProcessor extends
[CA]AbstractProcessor<String, ClickEvent> {

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        super.init(context);

    }

    @Override
    public void process(String key, ClickEvent clickEvent) {
        if (key != null) {
            Tuple<ClickEvent, StockTransaction> tuple =
[CA}Tuple.of(clickEvent, null);  (1)

            context().forward(key, tuple);  (2)
        }
    }
}
```

(1) Adds the ClickEvent to the initial aggregator object

(2) Forwards the tuple to the CogroupingProcessor

As you can see, the `ClickEventProcessor` adds the `ClickEvent` to the `Tuple` aggregator, much like the previous listing.

To complete the picture of how to perform the aggregation, we need to look at the `CogroupingProcessor` code. It's more involved, so we'll examine each method in turn, starting with `CogroupingProcessor#init()` (found in src/main/java/bbejeck/chapter_6/processor/cogrouping/AggregatingProcessor.java).

Listing 6.13 The `CogroupingProcessorinit()` method

```
public class CogroupingProcessor extends
[CA]AbstractProcessor<String, Tuple<ClickEvent,StockTransaction>> {

    private KeyValueStore<String,
[CA}Tuple<List<ClickEvent>,List<StockTransaction>>> tupleStore;
    public static final  String TUPLE_STORE_NAME = "tupleCoGroupStore";

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        super.init(context);
        tupleStore = (KeyValueStore)
[CA]context().getStateStore(TUPLE_STORE_NAME);  (1)
        CogroupingPunctuator punctuator =
[CA]new CogroupingPunctuator(tupleStore, context());  (2)
        context().schedule(15000L, STREAM_TIME, punctuator);  (3)

    }
```

1. Retrieves the configured state store
2. Creates a Punctuator instance, CogroupingPunctuator, which handles all scheduled calls
3. Schedules a call to the Punctuator#punctuate method every 15 seconds

As you might expect, the `init()` method handles the details of setting up the class. You grab the state store configured in the main application and save it in a variable for use later on. You create the `CogroupingPunctuator` to handle the scheduled punctuation calls.

SIDEBAR

Method handles for Punctuator

You can specify a method handle for the `Punctuator` instance. To do so, declare a met processor that accepts a single parameter of type `long` and a `void` return type. Then, punctuation like this:

```
context().schedule(15000L, STREAM_TIME, this::myPunctuationMethod);
```

For an example of this, look src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingMethodHandleProce

Listing 6.13 schedules `punctuate` for every 15 seconds. Because you're using the `PunctuationType.STREAM_TIME` semantics, the timestamps in the *arriving* data drive the calls to `punctuate`. Remember that if the flow of data isn't relatively constant, you may have more than 15 seconds between calls to `Punctuator#punctuate`.

NOTE

You'll recall from our previous discussion of `punctuate` semantics that you have two choices: `PunctuationType.STREAM_TIME` and `PunctuationType.WALL_CLOCK_TIME`. Listing 6.13 uses `STREAM_TIME` semantics. There's an additional processor example showing `WALL_CLOCK_TIME` semantics in src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingSystemTimeProces so you can observe the differences in performance and behavior.

Next, let's look at how the `CogroupingProcessor` performs one of its main tasks in the `process()` method (found in src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingProcessor.java).

Listing 6.14 The `CogroupingProcessorprocess()` method

```
@Override
    public void process(String key,
[CA]Tuple<ClickEvent, StockTransaction> value) {

        Tuple<List<ClickEvent>, List<StockTransaction>> cogroupedTuple
[CA] = tupleStore.get(key);
        if (cogroupedTuple == null) {
             cogroupedTuple =
[CA]Tuple.of(new ArrayList<>(), new ArrayList<>());   (1)
        }

        if (value._1 != null) {
            cogroupedTuple._1.add(value._1);   (2)
        }

        if (value._2 != null) {
            cogroupedTuple._2.add(value._2);   (3)
        }

        tupleStore.put(key, cogroupedTuple);   (4)
    }
}
```

(1) Initializes the total aggregation if it doesn't exist yet

(2) If the ClickEvent is not null, adds it to the list of click events

(3) If the StockTransaction is not null, adds it to the list of stock transactions

(4) Places the updated aggregation into the state store

As you process incoming smaller aggregates of your overall co-grouping, the first is step checking if you have an instance in your state store already. If you don't, you create a `Tuple` with empty collections of `ClickEvent` and `StockTransaction`.

Next, you check the incoming smaller aggregation, and if either a `ClickEvent` or `StockTransaction` is present, you add it to the overall aggregation. The last step in the `process()` method is putting the `Tuple` back into the store, updating your aggregation total.

NOTE Although you have two processors forwarding records to one processor and accessing one state store, you don't have to be concerned about concurrency issues. Remember, parent processors forward records to child processors in a depth-first manner, so each parent processor serially calls the child processor. Additionally, Kafka Streams only uses one thread per task, so there are never any concurrency access issues.

The next step is to look at how punctuation is handled (found in src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingPunctuator.java). You use the updated API, so we won't look at the `Processor#punctuate` call, which is

deprecated.

Listing 6.15 The `CogroupingPunctuatorpunctuate()` method

```
// leaving out class declaration and constructor for clarity
  @Override
  public void punctuate(long timestamp) {
    KeyValueIterator<String, Tuple<List<ClickEvent>,
[CA}List<StockTransaction>>> iterator = tupleStore.all();   (1)

    while (iterator.hasNext()) {
      KeyValue<String, Tuple<List<ClickEvent>, List<StockTransaction>>>
[CA] cogrouping = iterator.next();   (2)

      // if either list contains values forward results
      if (cogrouping.value != null &&
[CA](!cogrouping.value._1.isEmpty() ||
[CA]!cogrouping.value._2.isEmpty())) {   (3)
          List<ClickEvent> clickEvents =
[CA]new ArrayList<>(cogrouping.value._1);   (4)
          List<StockTransaction> stockTransactions =
[CA]new ArrayList<>(cogrouping.value._2);   (4)

          context.forward(cogrouping.key,
[CA}Tuple.of(clickEvents, stockTransactions));   (5)
          cogrouped.value._1.clear();
          cogrouped.value._2.clear();

          tupleStore.put(cogrouped.key, cogrouped.value);   (6)
      }
    }
    iterator.close();
}
```

1. Gets iterator of all co-groupings in the store
2. Retrieves the next co-grouping
3. Ensures that the value is not null, and that either collection contains data
4. 4 CO15-5)) Makes defensive copies of co-grouped collections
5. Forwards the key and aggregated co-grouping
6. Puts the cleared-out tuple back into the store

During each `punctuate` call, you retrieve all of the stored records in a `KeyValueIterator`, and you start to pull out each co-grouped result contained in the iterator. Then, you make defensive copies of the collections, create a new co-grouped `Tuple`, and forward it downstream. In this case, you send the co-grouped results to a sink node. Finally, you remove the current co-grouped results and store the tuple back in the store, ready for the next round of records to arrive.

Now that we've covered the co-grouping functionality, let's complete building the

topology.

ADDING THE STATE STORE

As you've seen, the ability to perform aggregations in a Kafka streaming application requires having state. You'll need to add a state store to the `CogroupingProcessor` for it to function properly. Figure 6.12 shows the updated topology.

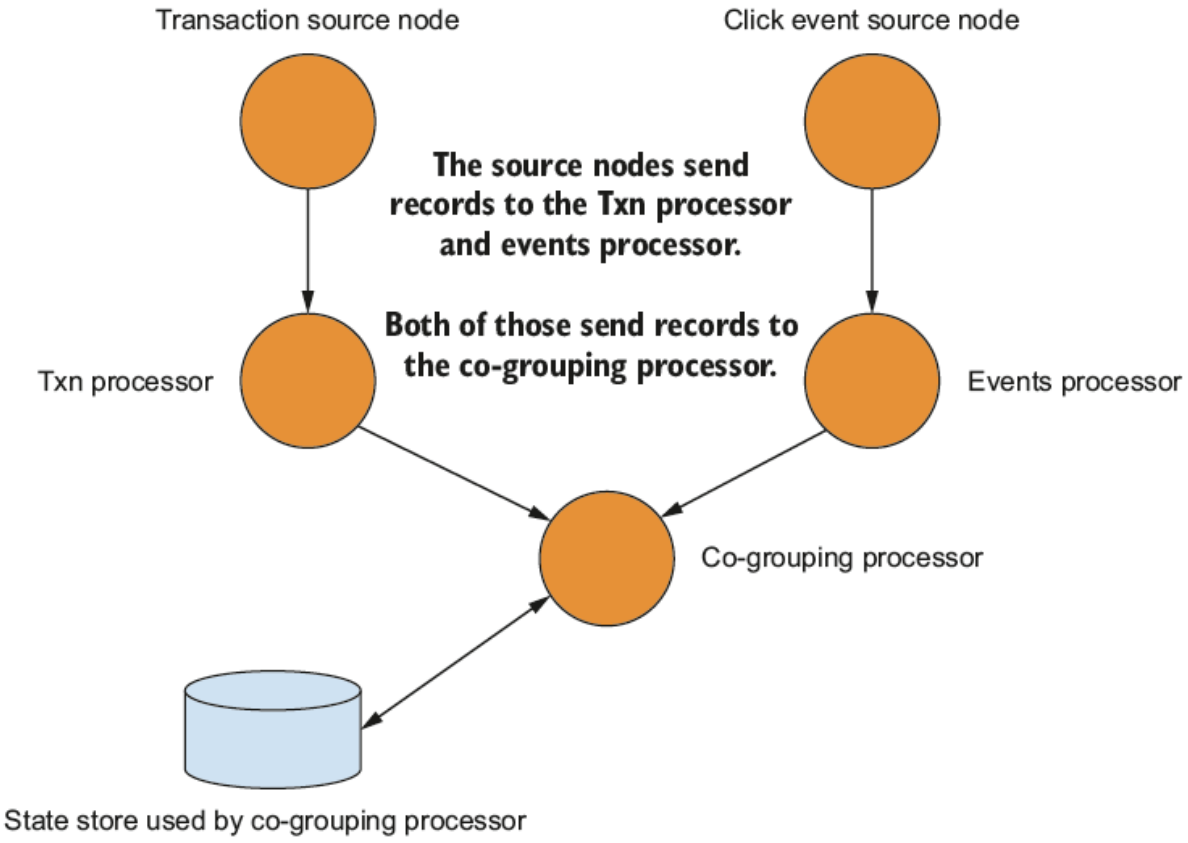


Figure 6.12 Adding a state store to the co-grouping processor in the topology

Now, let's look at the code for adding the state store (found in src/main/java/bbejeck/chapter_6/CoGroupingApplication.java).

Listing 6.16 Adding a state store node

```
// this comes earlier in source code, included here for context
 Map<String, String> changeLogConfigs = new HashMap<>();
 changeLogConfigs.put("retention.ms","120000" );                    (1)
 changeLogConfigs.put("cleanup.policy", "compact,delete");          (1)

 KeyValueBytesStoreSupplier storeSupplier =
[CA]Stores.persistentKeyValueStore(TUPLE_STORE_NAME);               (2)
        StoreBuilder<KeyValueStore<String,
[CA]Tuple<List<ClickEvent>, List<StockTransaction>>>> storeBuilder =   (3)
                Stores.keyValueStoreBuilder(storeSupplier,
```

```
                        Serdes.String(),
                        eventPerformanceTuple)
    [CA].withLoggingEnabled(changeLogConfigs);          4

.addStateStore(storeBuilder, "CoGrouping-Processor")    5
```

1. 1 CO16-2)) Specifies how long to keep records, and uses compaction and delete for cleanup
2. Creates the store supplier for a persistent store (RocksDB)
3. Creates the store builder
4. Adds the changelog configs to the store builder
5. Adds the store to the topology with the name of the processor that will access the store

Here, you add a persistent state store. This is a persistent store, because you might get infrequent updates for some keys. With the in-memory and LRU-based stores, infrequently used keys and values might eventually be removed, and here you'll want the ability to retrieve information for any key you've worked with before.

TIP The first three lines in listing 6.16 create specific configurations for the state store to keep the changelog at a manageable size. Remember: you can configure changelog topics with any valid topic configuration.

This code is straightforward. One point to notice, though, is that the `CoGrouping-Processor` is specified as the only processor that can access this store.

You now have one step left to complete the topology: the ability to read the results of the co-grouping.

ADDING THE SINK NODE

For the co-grouping topology to be of use, you need to write the data out to a topic (or the console). Let's update the topology one more time, as shown in figure 6.13.

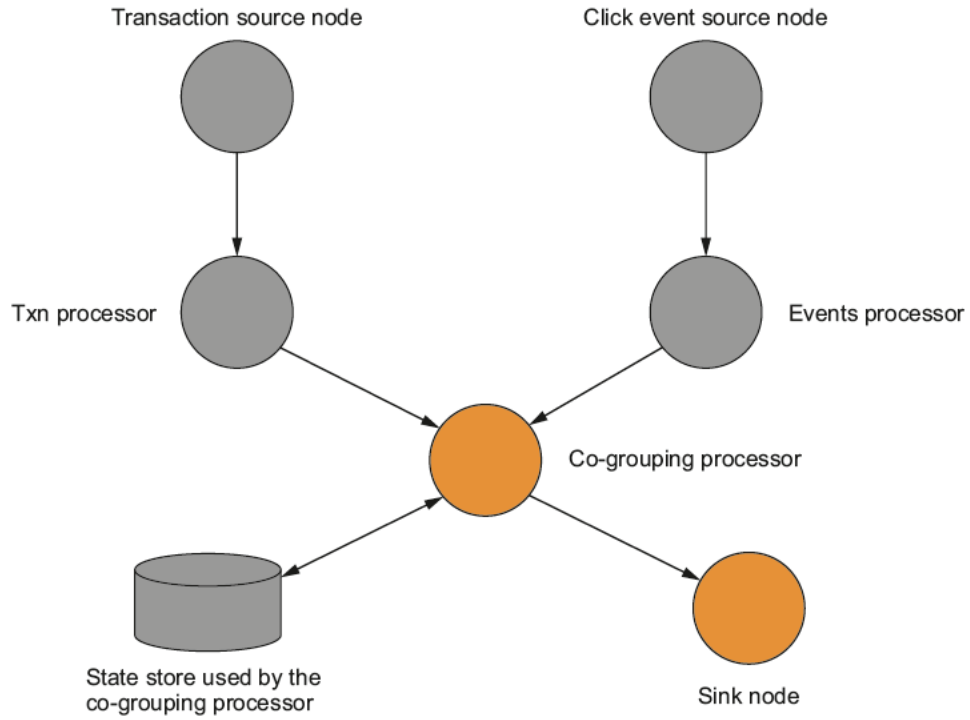


Figure 6.13 Adding a sink node completes the co-grouping topology

> **NOTE** In several examples, I talk about adding a sink node, but in the source code there's a sink that writes to the console; the sink that writes to a topic is commented out. For development purposes, I use the sink node writing to a topic and to stdout interchangeably.

Now, the co-grouped aggregation results are written out to a topic for use in further analysis. Here's the code (found in src/main/java/bbejeck/chapter_6/CoGroupingApplication.java).

Listing 6.17 The sink node and a printing processor

```
.addSink("Tuple-Sink",
         "cogrouped-results",
          stringSerializer,
          tupleSerializer,
          "CoGrouping-Processor");    (1)

 topology.addProcessor("Print",
                       new KStreamPrinter("Co-Grouping"),
                       "CoGrouping-Processor");    (2)
```

(1) The sink node writes the co-grouped tuples out to a topic.

(2) This processor prints results to stdout for use during development.

In this final piece of the topology, you add a sink node, as a child of the

`CoGrouping-Processor`, that writes the co-grouping results out to a topic. Listing 6.17 also adds an additional processor for printing results to the console during development—it's also a child node of the `CoGrouping-Processor`. Remember that with the Processor API, the order in which you define nodes doesn't establish a parent-child relationship. The parent-child relationship is determined by providing the names of previously defined processors.

You've now built the co-grouping processor. The key point I want you to remember from this section is that, although it involves more code, using the Processor API gives you the flexibility to create virtually any kind of streaming topology you need.

Let's wrap up this chapter with a look at how you can integrate some Processor API functionality into a KStreams application.

6.5 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 4.2.3.

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 yo `ProcessorContext#forward` method to send multiple values downstream. If you wanted to retu values instead, you'd return a `List<KeyValue<K,V>>` and then attach a `flatMap` or `flatMapValu` individual records downstream. An example of this can be f src/main/java/bbejeck/chapter_6/StockPerformanceStreamsAndProcessorMultipleValuesAppli To complete the replacement of a `Processor` instance, you'd plug in the `Transfo` `ValueTransformer`) instance using the `KStream#transform` or `KStream#transformValues` metho

A great example of combining the KStream and Processor APIs can be found in src/main/java/bbejeck/chapter_6/StockPerformanceStreamsAndProcessorApplication.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. 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.

6.6 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.

Part 3: Administering Kafka Streams

In these chapters, we'll shift focus to how you can measure the performance of your Kafka Streams application. Additionally, you'll learn how to monitor and test your Kafka Streams code so you know it's working as expected and will gracefully handle errors.

7 Monitoring and performance

This chapter covers

- Looking at basic Kafka monitoring
- Intercepting messages
- Measuring performance
- Observing the state of the application

So far, you've learned how to build a Kafka Streams application from the bottom up. You've worked with the high-level Kafka Streams DSL, and you've seen the power of using a declarative API. You've also learned about the Processor API and have seen how you can give up some convenience to gain more control in writing your streaming applications.

It's now time to change gears a bit. You're going to put on your forensic investigator hat and dig into your application from a different perspective. Your focus is going to shift from *how* you get things to work to *what* is going on. In some respects, the initial building of an application is the most comfortable part. Getting the application to run successfully, scale correctly, and work properly is always the more significant challenge. Despite your best efforts, there's almost always a situation you didn't account for.

In this chapter, you'll learn how to check the running state of your Kafka Streams application. You'll see how you can measure the performance of the application in order to spot performance bottlenecks. You'll also see techniques you can use to notify you about various states of the application and to view the structure of the topology. You'll

learn what metrics are available, how you can collect them, and how you can observe the collected metrics as the application is running. Let's start with monitoring a Kafka Streams application.

7.1 Basic Kafka monitoring

Because the Kafka Streams API is a part of Kafka, it goes without saying that monitoring your application will require some monitoring of Kafka as well. Full-blown surveillance of a Kafka cluster is a big topic, so we'll limit our discussion of Kafka performance to where the two meet—we'll talk about monitoring Kafka consumers and producers. More information on monitoring a Kafka cluster can be found in the documentation (https://kafka.apache.org/documentation/#monitoring).

> **NOTE** One thing I should note here is that to measure Kafka Streams performance, we also need to measure Kafka itself. At times, some of our coverage of performance will edge over to the Kafka side of things. But because this is a book on Kafka Streams, we'll focus on Kafka Streams.

7.1.1 Measuring consumer and producer performance

For our discussion of consumer and producer performance, let's start by looking at figure 7.1, which depicts one of the fundamental performance concerns for a producer and consumer. As you can see, producer and consumer performance are very similar in that both are concerned with throughput. But where we put the emphasis is just different enough that they can be considered two sides of the same coin.

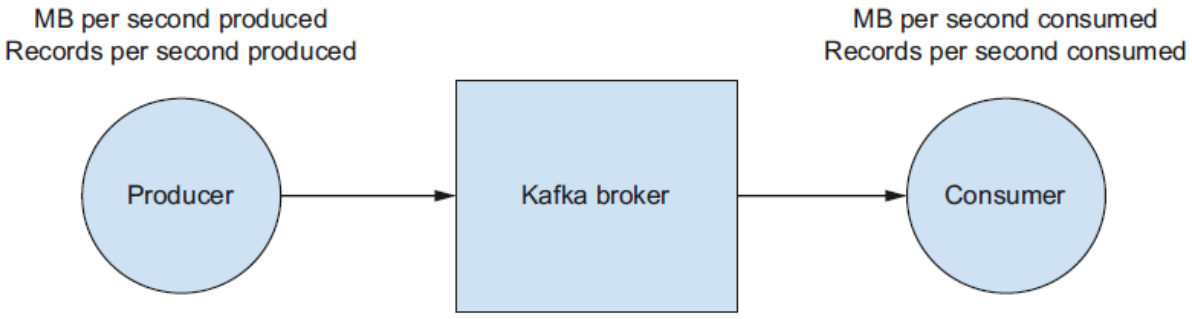


Figure 7.1 Kafka producer and consumer performance concerns, writing to and reading from a broker

For producers, we care mostly about how fast the producer is sending messages to the broker. Obviously, the higher the throughput, the better.

For consumers, we're also concerned with performance, or how fast we can read

messages from a broker. But there's another way to measure consumer performance: consumer lag. Take a look at figure 7.2. This measures producer and consumer throughput with a slightly different focus.

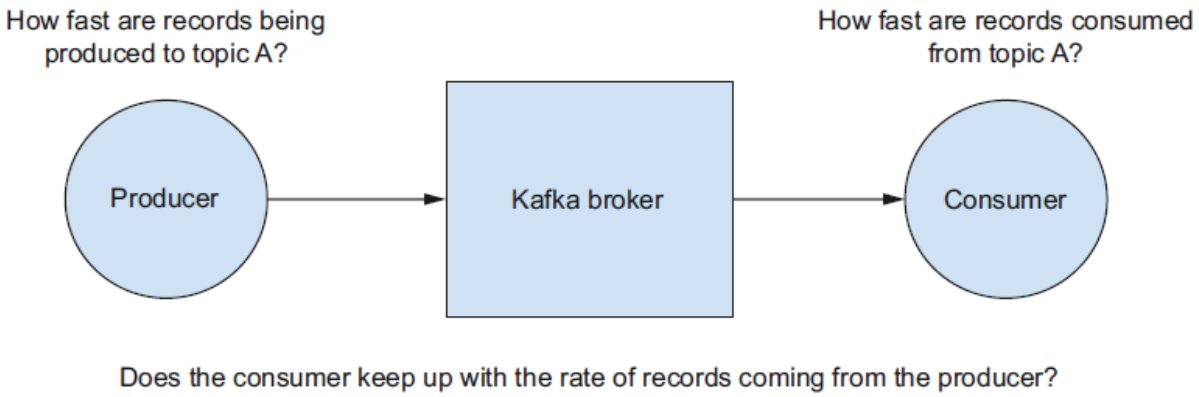


Figure 7.2 Kafka producer and consumer performance revisited

You can see that we care about how much and how fast our producers can publish to a broker, and we simultaneously care about how quickly our consumers can read those messages from the broker. The difference between how fast the producers place records on the broker and when consumers read those messages is called *consumer lag*.

Figure 7.3 illustrates that consumer lag is the difference between the last committed offset from the consumer and the last offset from a message written to the broker. There's bound to be some lag from the consumer, but ideally the consumer will catch up, or at least have a consistent lag rather than a gradually increasing lag.

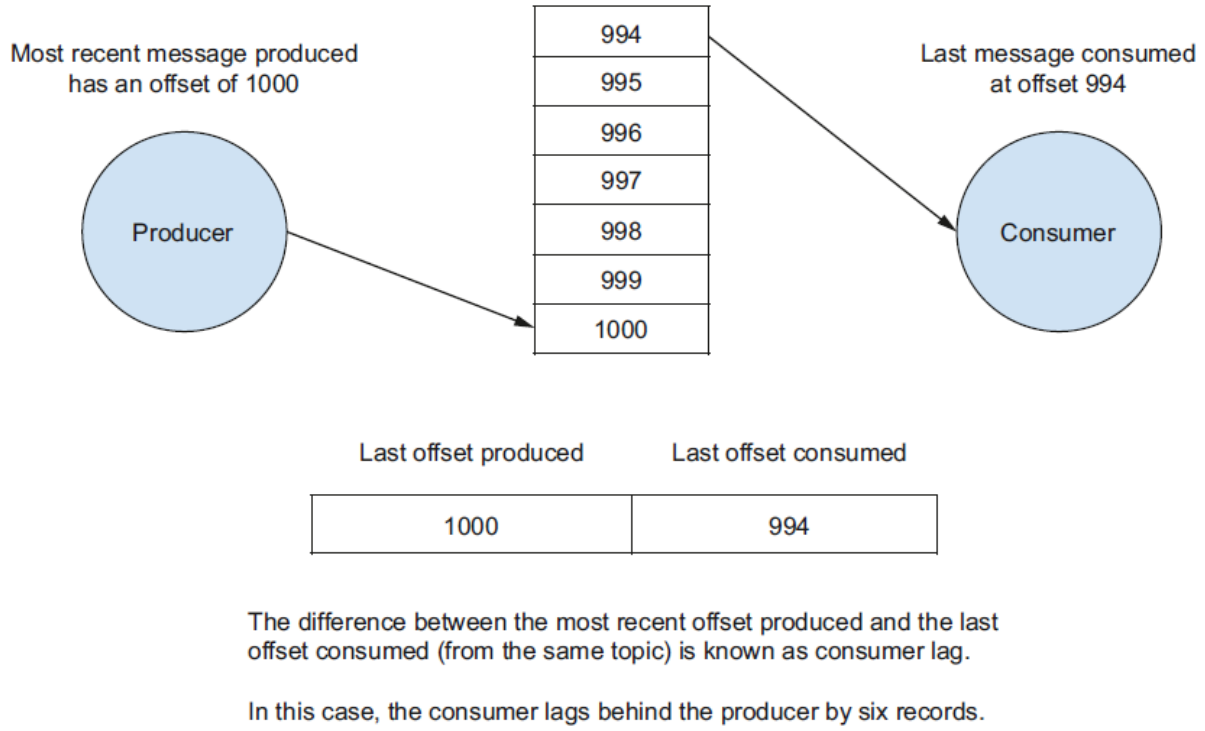


Figure 7.3 Consumer lag is the difference in offsets committed by the consumer and offsets written by the producer

Now that we've defined our performance parameters for producers and consumers, let's see how we can monitor them for performance and troubleshooting issues.

7.1.2 Checking for consumer lag

To check for consumer lag, Kafka provides a convenient command-line tool, `kafka-consumer-groups.sh`, found in the <kafka-install-dir>;/bin directory. The script has a few options, but here we'll focus on the `list` and `describe` options. These two options will give you the information you need about consumer group performance.

First, use the `list` command to find all active consumer groups. Figure 7.4 shows the results of running this command.

```
<kafka-install-dir>;/bin/kafka-consumer-groups.sh \
                --bootstrap-server localhost:9092 \
                --list
```

```
oddball:bin bbejeck$ ./kafka-consumer-groups.sh --list --bootstrap-server localhost:9092
Note: This will only show information about consumers that use the Java consumer API (non-ZooKeeper-based consumers).

console-consumer-59026
```

Figure 7.4 Listing available consumer groups from the command line

With this information, you can choose a consumer group name and run the following

command:

```
<kafka-install-dir>;/bin/kafka-consumer-groups.sh \
                        --bootstrap-server localhost:9092 \
                        --group <GROUP-NAME>; \
                        --describe
```

Figure 7.5 shows the results: the status of how this consumer is performing.

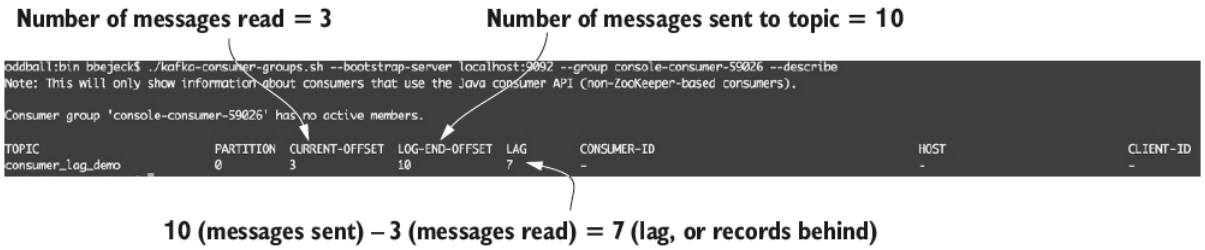


Figure 7.5 Status of a consumer group

These results show that you have a small consumer lag. Having a consumer lag isn't always indicative of a problem—consumers read messages in batches and won't retrieve another batch until they're finished processing the current batch. Processing the records takes time, so a little lag is not entirely surprising.

A small lag or one that stays constant is OK, but a lag that continues to grow over time is an indication you'll need to give your consumer more resources. For example, you might need to increase the partition count and hence increase the number of threads consuming from the topic. Or maybe your processing after reading the message is too heavyweight. After consuming a message, you could hand it off to an async queue, where another thread can pick up the message and do the processing.

In this section, you've learned how to determine how quickly a consumer is reading messages from a broker. Next, we'll dig a little deeper into observing behavior for debugging purposes—you'll see how to intercept what the producers are sending and consumers are receiving *before* your Kafka Streams application sends or consumes records.

7.1.3 Intercepting the producer and consumer

Early in 2016, Kafka Improvement Proposal 42 (KIP-42) introduced the ability to monitor or "intercept" information on client (consumer and producer) behavior. The goal of the KIP was to provide "the ability to quickly deploy tools to observe, measure, and monitor Kafka client behavior, down to the message level."[11]

Footnote 11 Apache Kafka, “KIP-42: Add Producer and Consumer Interceptors,” http://mng.bz/g8oX.

Although interceptors aren’t typically your first line for debugging, they can prove useful in observing the behavior of your Kafka streaming application, and they’re a valuable addition to your toolbox. An excellent example of using an interceptor (producer) is using one to keep track of the message offsets your Kafka Streams application is producing back to Kafka.

NOTE Because Kafka Streams can consume or produce any number of key and value types, the internal `Consumer` and `Producer` are configured to work with `byte[]` keys and `byte[]` values; hence, they always handle unserialized data. Serialized data means you can’t inspect messages without an extra deserialization/serialization step.

Let’s get started by discussing the consumer interceptor.

CONSUMER INTERCEPTOR

The consumer interceptor gives you two access points to intercept. The first is `ConsumerInterceptor#onConsume`, which reads `ConsumerRecords` between the point where they’re retrieved from the broker, and before the messages are returned from the `Consumer#poll` method. The following pseudocode will give you an idea of where the consumer interceptor is doing its work:

```
ConsumerRecords<String, String>; poll(long timeout) {
   ConsumerRecords<String, String>; consumerRecords =

   ...consuming records                                    (1)

   return interceptors.onConsume(consumerRecords);         (2)
```

(1) Fetches new records from the broker

(2) Runs records through the interceptor chain and returns the results

Although this pseudocode bears no resemblance to the actual `KafkaConsumer` code, it illustrates the point. Interceptors accept the `ConsumerRecords` returned from the broker inside the `Consumer#poll` method and have the opportunity to perform any operation, including filtering or modification, before the `KafkaConsumer` returns the records from the `poll` method.

`ConsumerInterceptors` are specified via `ConsumerConfig#INTERCEPTOR_CLASSES_CONFIG` with a `Collection` of one or more

`ConsumerInterceptor` implementor classes. Multiple interceptors are chained together and executed in the order specified in the configuration.

A `ConsumerInterceptor` accepts and returns a `ConsumerRecords` instance. If there are multiple interceptors, the returned `ConsumerRecords` from one interceptor serves as the input parameter for the next interceptor in the chain. Thus, any modifications made by one interceptor are propagated to the next interceptor in the chain.

Exception handling is an important consideration when chaining multiple interceptors together. If an `Exception` occurs in an interceptor, it logs the error, but it *doesn't* short-circuit the chain. Thus, `ConsumerRecords` continues to work its way through the remaining interceptors.

For example, suppose you have three interceptors: A, B, and C. All three modify the records and rely on changes made by the previous interceptor in the chain. But if interceptor A encounters an error, the `ConsumerRecords` object continues to interceptors B and C, but without the expected modifications, rendering the results from the interceptor chain invalid. For this reason, it's best not to have an interceptor rely on `ConsumerRecords` modified by a previous interceptor in the chain.

The second interception point is the `ConsumerInterceptor#onCommit` method. After the consumer commits its offsets to the broker, the broker returns a `Map<TopicPartition, OffsetAndMetadata>;` containing information with the topic, partition, and committed offsets, along with associated metadata (time of commit, and so on). The commit information can be useful for tracking purposes. Here's an example of a simple `ConsumerInterceptor` used for logging purposes (found in src/main/java/bbejeck/chapter_7/interceptors/StockTransactionConsumerInterceptor.java).

Listing 7.1 Logging consumer interceptor

```
public class StockTransactionConsumerInterceptor implements
[CA]ConsumerInterceptor<Object, Object>; {

    // some details left out for clarity
    private static final Logger LOG =
[CA]LoggerFactory.getLogger(StockTransactionConsumerInterceptor.class);

    public StockTransactionConsumerInterceptor() {
        LOG.info("Built StockTransactionConsumerInterceptor");
    }

    @Override
    public ConsumerRecords<Object, Object>; (ConsumerRecords<Object,
[CA]Object>; consumerRecords) {
        LOG.info("Intercepted ConsumerRecords {}",

                 buildMessage(consumerRecords.iterator()));
        return consumerRecords;
```

```
    }

    @Override
    public void onCommit(Map<TopicPartition, OffsetAndMetadata>; map) {
         LOG.info("Commit information {}",  map);     (2)
    }
```

(1) Logs the consumer records and metadata before the records are processed

(2) Logs the commit information once the Kafka Streams consumer commits offsets to the broker

Now let's cover the producing side of intercepting.

PRODUCER INTERCEPTOR

The `ProducerInterceptor` works similarly and has two access points: `ProducerInterceptor#onSend` and `ProducerInterceptor#onAcknowledgement`. With the `onSend` method, the interceptor can perform any action, including mutating the `ProducerRecord`. Each producer interceptor in the chain receives the returned object from the previous interceptor.

Exception handling is the same as on the consumer side, so the same caveats apply here as well. The `ProducerInterceptor#onAcknowledgement` method is called when the broker acknowledges the record. If sending the record fails, `onAcknowledgement` is called at that point as well.

Here's a simple logging `ProducerInterceptor` example (found in src/main/java/bbejeck/chapter_7/interceptors/ZMartProducerInterceptor.java).

Listing 7.2 Logging producer interceptor

```
public class ZMartProducerInterceptor implements
[CA]ProducerInterceptor<Object, Object>; {
 // some details left out for clarity
 private static final Logger LOG =
[CA]LoggerFactory.getLogger(ZMartProducerInterceptor.class);

 @Override
 public ProducerRecord<Object, Object>; onSend(ProducerRecord<Object,
[CA]Object>; record) {
  LOG.info("ProducerRecord being sent out {} ", record);     (1)
  return record;
 }

 @Override
 public void onAcknowledgement(RecordMetadata metadata,Exception exception) {
   if (exception != null) {
       LOG.warn("Exception encountered producing record {}",
[CA]exception);     (2)
   } else {
```

```
      LOG.info("record has been acknowledged {} ", metadata);
    }
}
```

1. Logs right before the message is sent to the broker
2. Logs broker acknowledgement or whether error occurred (broker-side) during the produce phase

The `ProducerInterceptor` is specified with `ProducerConfig#INTERCEPTOR_CLASSES_CONFIG` and takes a `Collection` of one or more `ProducerInterceptor` classes.

TIP When configuring interceptors in a Kafka Streams application, you need to prefix the consumer and producer interceptors' property names with `props.put(StreamsConfig.consumerPrefix(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG)` and `StreamsConfig.producerPrefix(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG)`, respectively.

If you want to see the interceptors in action, src/main/java/bbejeck/chapter_7/StockPerformanceStreamsAndProcessorMetricsApplication.ja uses a consumer interceptor, and src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java uses a producer interceptor. Both classes include the configuration required for using interceptors.

As a side note, because interceptors work on every record in the Kafka Streams application, the output of the logging interceptors is significant. The interceptor results are output to consumer_interceptor.log and producer_interceptor.log, found in the logs directory at the base of the source code installation.

We've spent some time looking at metrics on consumer performance and how you can intercept records coming into and out of a Kafka Streams application. But this information is coarse grained and *outside* of a Kafka Streams application. Let's now go *inside* a Kafka Streams application and see what's going on under the hood. The next step is to measure performance inside the topology by gathering metrics.

7.2 Application metrics

When it comes to measuring the performance of an application, you can get a sense of how long it takes to process one record, and measuring end-to-end latency is undoubtedly a good indicator of overall performance. But if you want to improve performance, you'll need to know exactly where things are slowing down.

Measuring performance is essential for streaming applications. The mere fact that you're using a streaming application implies you want to process data or information as it becomes available. It stands to reason that if your business needs dictate a streaming solution, you'll want the most *efficient* and *correct* streaming process you can get.

Before we discuss the actual metrics we'll focus on, let's revisit one of the applications you built in chapter 3, the advanced ZMart application. That app is a good candidate for metrics tracking because there are several processing nodes, so we'll use that topology for this example. Figure 7.6 shows the topology you created.

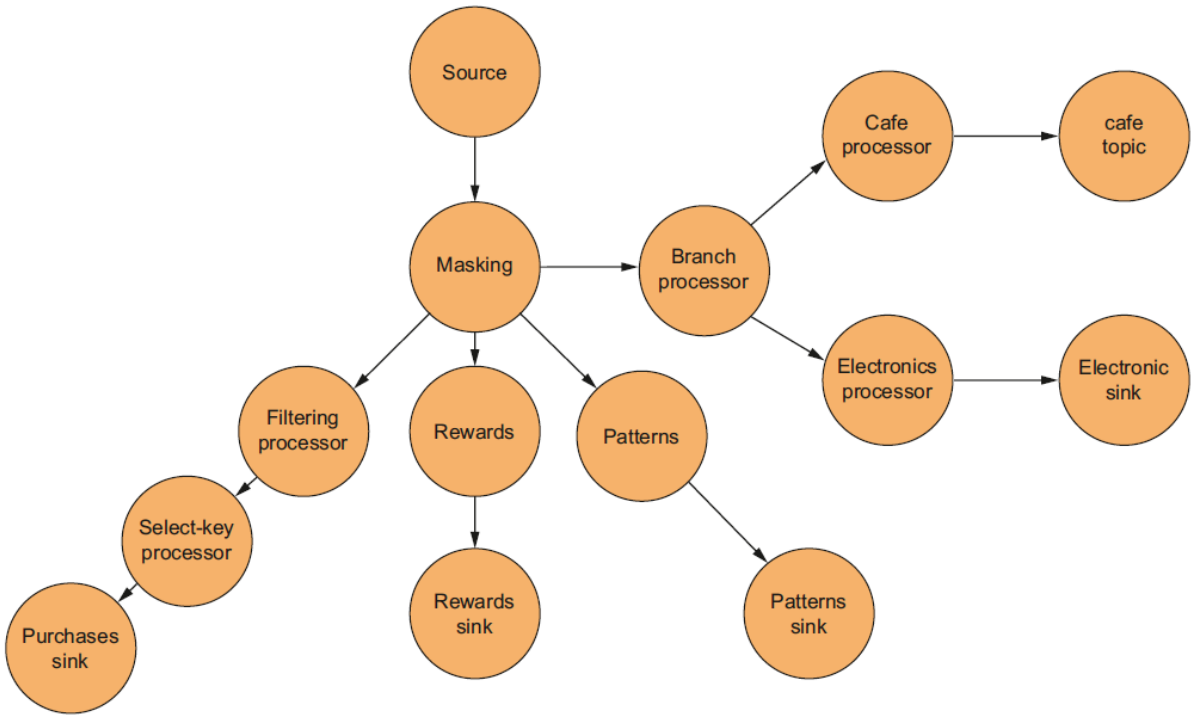


Figure 7.6 ZMart advanced application topology with a lot of nodes

Keeping the ZMart topology in mind, let's take a look at the categories of metrics:

- Thread metrics
  - Average time for commits, poll, process operations
  - Tasks created per second, tasked closed per second
- Task metrics

    - Average number of commits per second
    - Average commit time

- Processor node metrics
    - Average and max processing time
    - Average number of process operations per second
    - Forward rate

- State store metrics
    - Average execution time for put, get, and flush operations
    - Average number put, get, and flush operations per second

Note that this isn't an exhaustive list of the possible metrics. I've chosen these because they offer excellent coverage of the most common performance scenarios. You can find a full list on the Confluent website: http://mng.bz/4bcA.

Now that we have what we're going to measure, let's look at how to capture the information.

7.2.1 Metrics configuration

Kafka Streams already provides the mechanism for collecting performance metrics. For the most part, you just need to provide some configuration values. Because the collection of metrics does incur a performance cost, there are two levels, `INFO` and `DEBUG`. An individual metric may not be that expensive on its own, but when you consider that some metrics may involve every record flowing through the Kafka Streams application, you can see how the impact on performance can add up.

The metric levels are used like logging levels. When you're troubleshooting an issue or observing how the application behaves, you'll want more information, so you can use the `DEBUG` level. Other times, you don't need all the information, so you can use the `INFO` level.

Typically, you won't want to use `DEBUG` in production, as the cost of performance would be too high. Each of the previously listed metrics are available at different levels, as shown in table 7.1. As you can see, thread metrics are available at any level, whereas the rest of the metric categories are only collected when using the `DEBUG` level.

Table 7.1 Metrics availability by levels

Metrics category	DEBUG	INFO
Thread	x	x
Task	x	
Processor node	x	
State store	x	
Record cache	x	

You set the level when you set the configuration of your Kafka Streams application. That setting has been there all along with your other application configurations; you've accepted the default settings up to this point. The default level of metrics collection is `INFO`.

Let's update the configs in the advanced ZMart application and turn on the collection of all metrics (src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java).

Listing 7.3 Updating the configs for `DEBUG` metrics

```
private static Properties getProperties() {
   Properties props = new Properties();
   props.put(StreamsConfig.CLIENT_ID_CONFIG,
[CA]"metrics-client-id");     (1)
   props.put(ConsumerConfig.GROUP_ID_CONFIG,
[CA]"metrics-group-id");      (2)
   props.put(StreamsConfig.APPLICATION_ID_CONFIG,
[CA]"metrics-app-id");        (2)
   props.put(StreamsConfig.METRICS_RECORDING_LEVEL_CONFIG,
[CA]"DEBUG");                 (3)
   props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
[CA]"localhost:9092");        (3)
  return props;
}
```

(1) Client ID

(2) Group ID

(3) Application ID

(4) Sets the metrics recording level to DEBUG

(5) Sets the connection for the brokers

You've now enabled the collection and recording of `DEBUG`-level metrics. The key points I want you to remember from this section is there are built-in metrics to measure the full scope of a Kafka Steams application, and that you should carefully consider the

performance impact before turning on metrics collection at the `DEBUG` level.

Now that we've discussed what metrics are available and how they're collected, the next step is to observe the collected metrics.

7.2.2 How to hook into the collected metrics

The metrics in a Kafka Streams application are collected and distributed to metrics reporters. As you might have guessed, Kafka Streams provides a default reporter via Java Management Extensions (JMX).

Once you've enabled collecting metrics at the `DEBUG` level, you have nothing left to do but observe them. One thing to keep in mind is that JMX only works with live running applications, so the metrics we'll look at will be when the application is running.

TIP You can also access metrics programmatically. For an example of programmatic metrics acc
t a k e a l o o k
src/main/java/bbejeck/chapter_7/StockPerformanceStreamsAndProcessorMetricsApplication.j

You're likely familiar with using JMX or have at least heard of it. In the next section, I'll provide a brief overview on how to get started using JMX, but if you're an experienced JMX user, feel free to skip this next section.

7.2.3 Using JMX

JMX is a standard way of viewing the behavior of programs running on the Java VM. You can also use JMX to see how the Java Virtual Machine (Java VM) is performing. In a nutshell, JMX gives you the infrastructure to expose parts of your running program.

Fortunately, you won't need to write any code to do this monitoring. You'll just connect either Java VisualVM (http://mng.bz/euif), JConsole (http://mng.bz/Ea71), or Java Mission Control (JMC) (http://mng.bz/0r5B).

TIP Java Mission Control (JMC) is powerful and can be a great tool for monitoring, but it requires a commercial license for use in production. Because JMC ships with the JDK, you can start JMC directly from the command line with the command `jmc` (assuming the JDK bin directory is on your path). Additionally, you'll need to add these flags when starting your Kafka streaming application: `-XX:+UnlockCommercialFeatures -XX:+FlightRecorder`.

As JConsole is the most straightforward approach, we'll start with it for now.

SIDEBAR

What is JMX?

Oracle says the following in "Lesson: Overview of the JMX Technology" http://mng.bz/Ej29):

> The Java Management Extensions (JMX) technology is a standard part of the Java Platform, Standard Edition (Java SE platform), added to the platform in the 5.0 release.
>
> The JMX technology provides a simple, standard way of managing resources such as applications, devices, and services. Because the JMX technology is dynamic, you can use it to monitor and control resources as they are created, installed, and implemented. You can also use the JMX technology to monitor and manage the Java Virtual Machine (Java VM).
>
> The JMX specification defines the architecture, design patterns, APIs, and services in the Java programming language for management and monitoring of applications and networks.

STARTING JCONSOLE

JConsole ships with the JDK, so if you've got Java installed, you already have JConsole. Starting JConsole is as simple as running `jconsole` from a command prompt (assuming Java is on your path). Once it's started, a GUI will come up, as shown in figure 7.7.

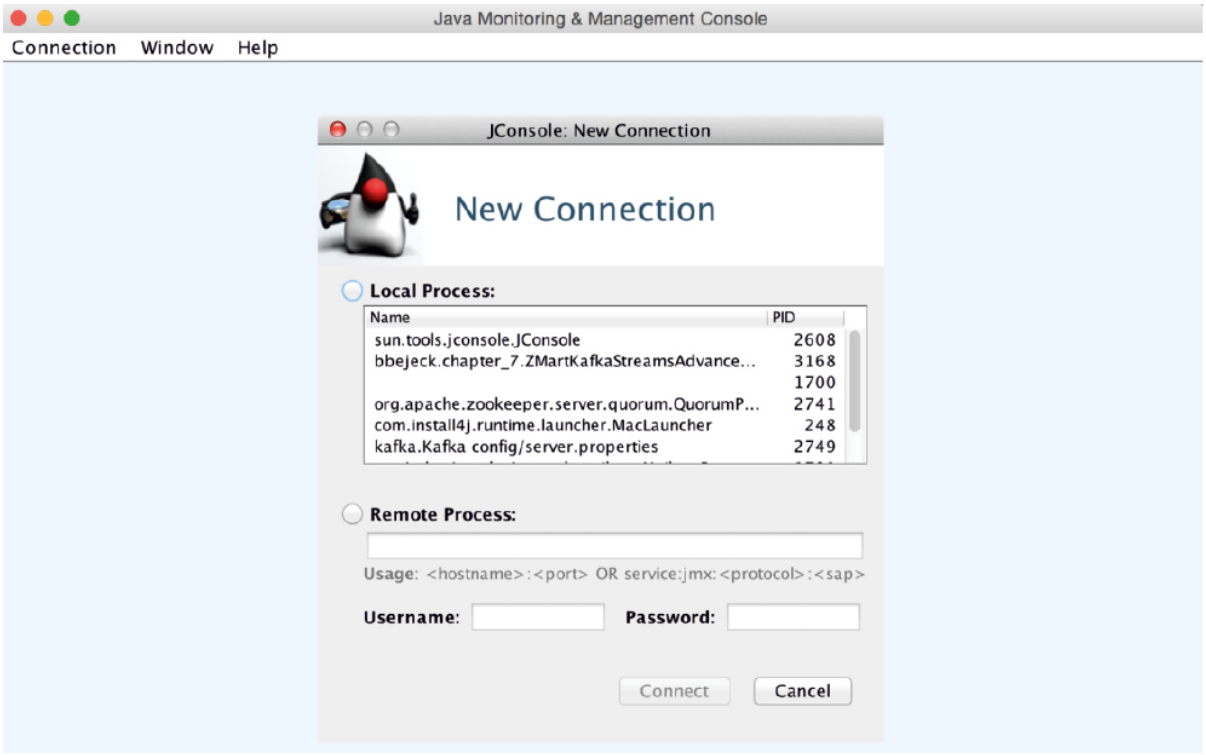


Figure 7.7 JConsole start menu

Once JConsole is up, the next step is to use it to look at some metric data!

STARTING TO MONITOR A RUNNING PROGRAM

If you look at the center of the JConsole GUI, you'll see a New Connection dialog box. Figure 7.8 shows the starting point for JConsole. For now, we're only concerned with the Java processes listed in the Local Process section.

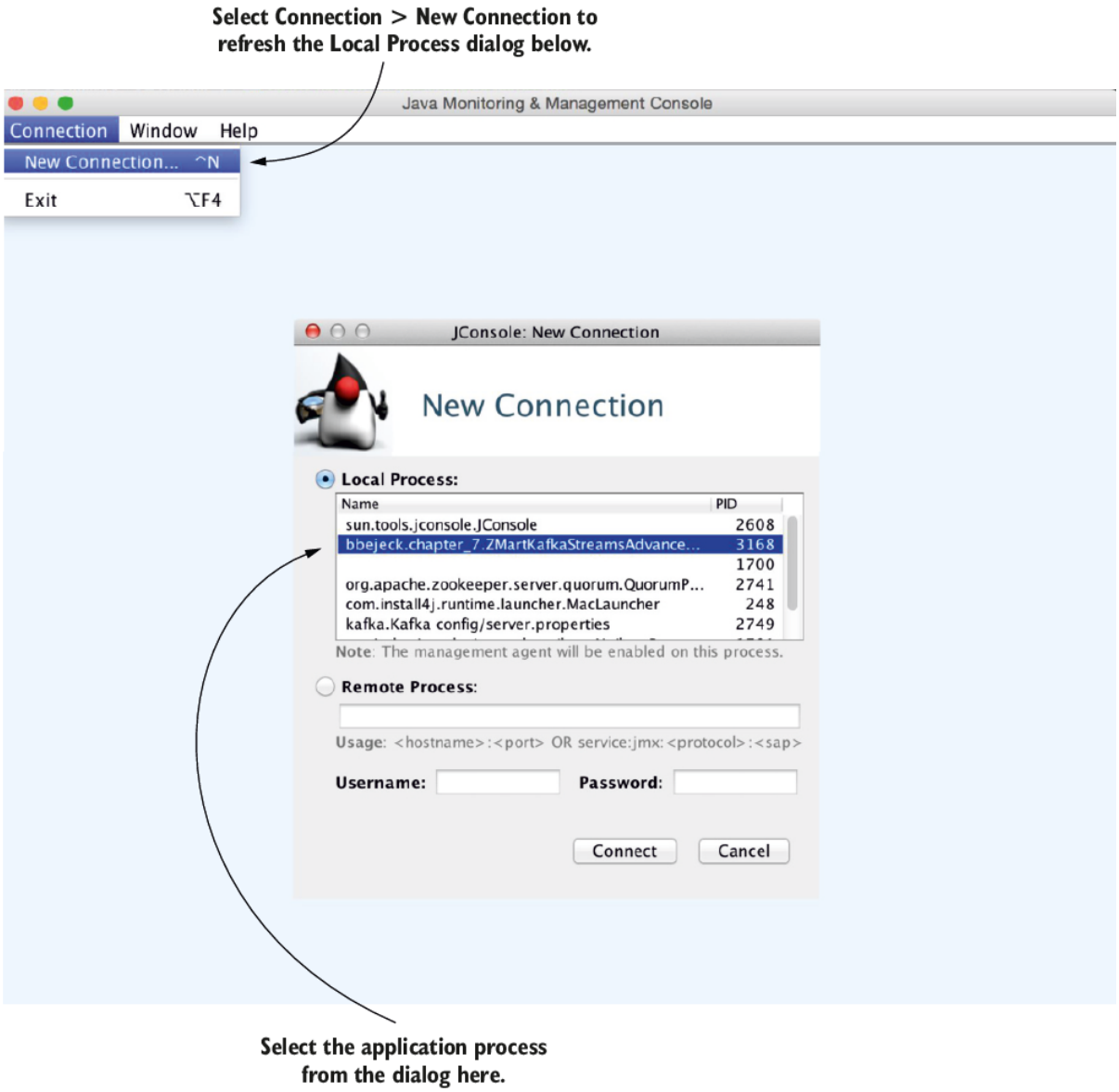


Figure 7.8 JConsole connect to program

> **NOTE** You can use JConsole to monitor remote applications, and you can secure access to JMX. You can see the Remote Process, Username, and Password text boxes in figure 7.8. In this book, however, we'll limit our discussion to local access during development. The internet is full of instructions on remote and secure JConsole access, and Oracle's documentation is a great starting point (http://mng.bz/Ea71).

If you haven't already started your Kafka Streams application, do so now. Then, to get your application to show up in the Local Process window, click Connection > New Connection (as in figure 7.8). The processes listed under Local Process should refresh, and you'll see your Kafka Streams application. Double-click the Kafka Streams application process.

Chances are that after you double-click the program you want to connect to, you'll be greeted with a warning similar to the one in figure 7.9. Because you're on your local machine, you can click the Insecure Connection button.

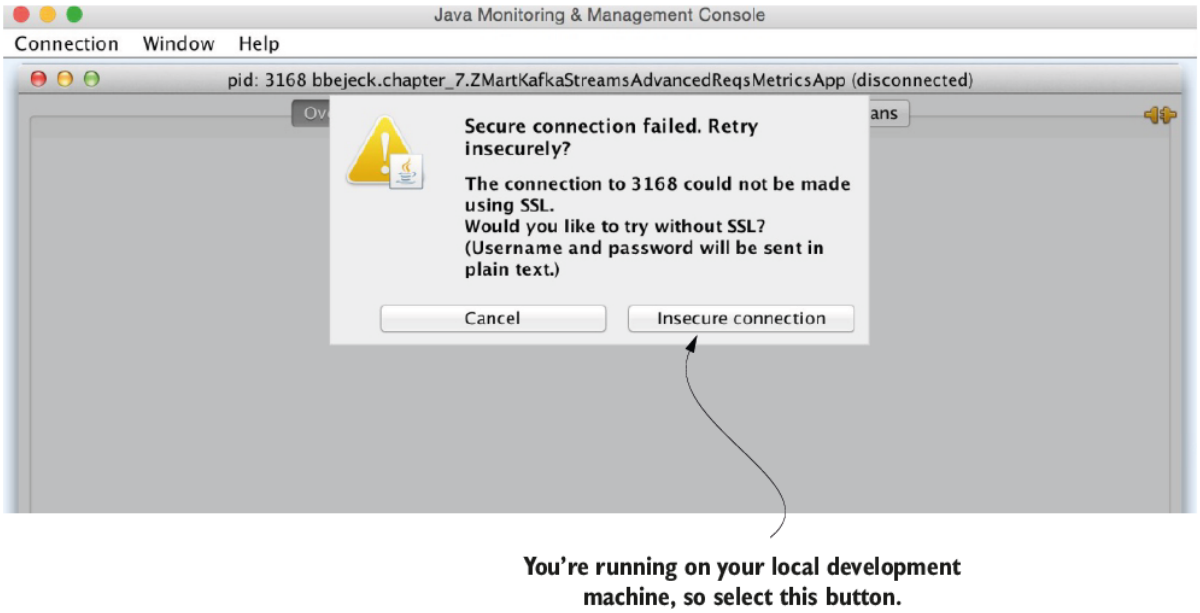


Figure 7.9 JConsole connect warning, no SSL

Now you're all set to look at the metrics being collected by your Kafka Streams application. The next step is to look at the information available.

WARNING You're using an insecure connection for development on a local machine. In practice, you should always secure access to any remote services accessing the internal state of your application.

VIEWING THE INFORMATION

Once you're connected, you'll see a GUI screen looking something like figure 7.10. JConsole offers several handy options for peeking inside the internals of running applications.

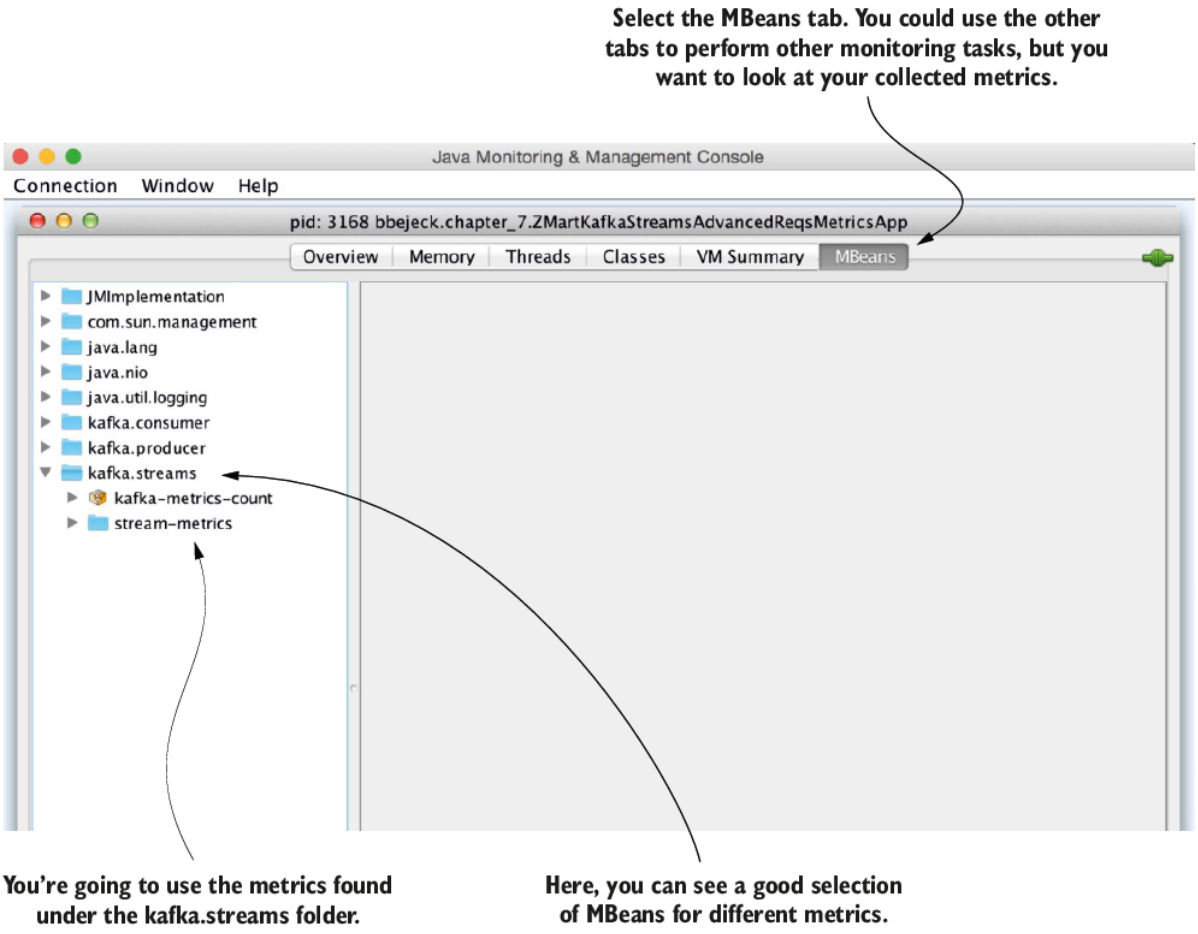


Figure 7.10 JConsole started

Of the Overview, Memory, Threads, Classes, VM Summary, and MBeans tabs, you're only going use the MBeans tab. MBeans contain the collected statistics about the performance of your Kafka Streams program. The other tabs provide relevant information, but it's info that relates more to overall application health and how the program is utilizing resources. The metrics collected in the MBeans contain information about the internal performance of the topology.

That's the end of your introduction to using JConsole. The next step is to start viewing the recorded metrics for the topology.

7.2.4 Viewing metrics

Figure 7.11 shows how you can view some metrics via JConsole while running the ZMart application (src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java). As you can see, you can drill down to all processors and nodes in the topology to view performance (either throughput or latency).

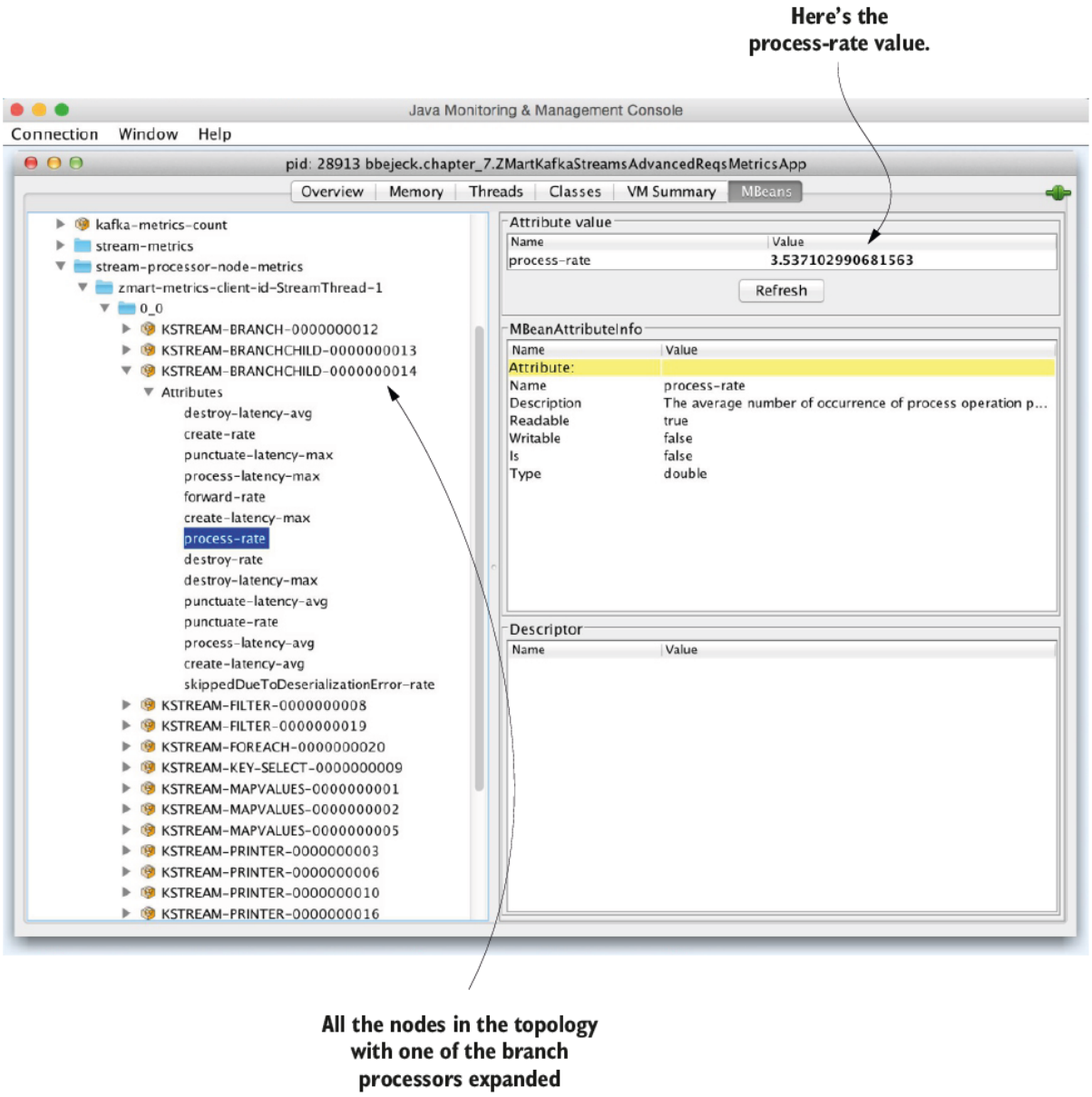


Figure 7.11 JConsole metrics for ZMart

> **TIP** Because JMX only works with running applications, some of the example applications in src/main/java/bbejeck/chapter_7 will run continuously so that you can play with the metrics. As a result, you'll need to explicitly stop them either in the IDE or by pressing Ctrl-C from the command line.

Figure 7.11 shows the `process-rate` metric, which tells you the average number of records processed per millisecond. If you look at the upper right, under Attribute Value, you can see that the process rate averages 3.537 records per millisecond (3,537 records per second). Additionally, as discussed earlier, you can see the producer and consumer metrics from JConsole.

TIP Although the provided metrics are comprehensive, there may be cases where you want custom metrics. This is a low-level detail and probably not a very common use case, so we won't walk through an example in detail. But you can look at the `StockPerformanceMetricsTransformer#init` method for an example of how you can add a custom metric and the `StockPerformanceMetricsTransformer#transform` method for an example of how you can utilize it. The `StockPerformanceMetricsTransformer` is found in src/main/java/bbejeck/chapter_7/transformer/StockPerformanceMetricsTransformer.java.

Now that you've seen how to view Kafka Streams metrics, let's move on to other useful techniques for observing what's going on in an application.

7.3 More Kafka Streams debugging techniques

We'll now look at some more ways you can observe and debug Kafka streaming applications. The previous section was more about performance; the techniques we'll look at in this section focus on getting notified about various states of the application and viewing the structure of the topology.

7.3.1 Viewing a representation of the application

After your application is up and running, you might run into situations where you need to debug it. You might like to get a second pair of eyes on the job, but for whatever reason, you can't share the code. Or you'd like to see the `TopicPartition` assigned to the tasks of the application.

The `Topology.describe()` method provides general information on the structure of the application. It prints out information regarding the structure of the program, including any internal topics created to support repartitioning. Figure 7.12 displays the results of calling `describe` on the `CoGroupingListeningExampleApplication` from chapter 7 (src/main/java/bbejeck/chapter_7/CoGroupingListeningExampleApplication.java).

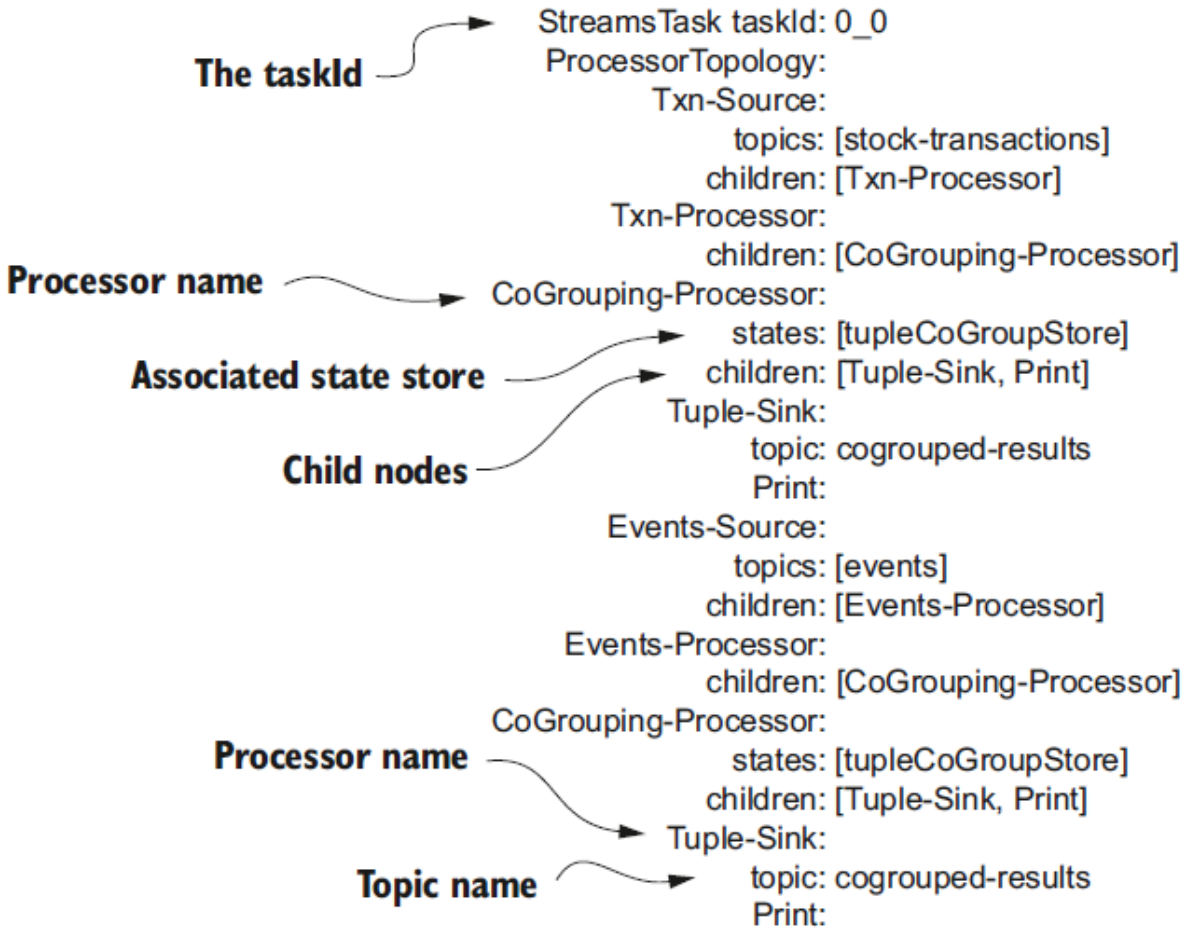


Figure 7.12 Displaying node names, associated child nodes, and other info

As you can see, the `Topology.describe()` method prints out a nice, concise view of the application structure. There are two things you should note here.

First, the `CoGroupingListeningExampleApplication` used the Processor API, so all the nodes in the topology have the names you chose. With the Kafka Streams API, the names of the nodes are a little more generic:

```
KSTREAM-SOURCE-0000000000:
    topics:       [transactions]
    children:    [KSTREAM-MAPVALUES-0000000001]
```

TIP When you're using the Kafka Streams DSL API, you don't directly use the `Topology` class, but it's easily accessed. If you want to print the physical topology of the application, use the `StreamsBuilder.build()` method, which returns a `Topology` object, and then you can call `Topology.describe()` as you just saw.

Getting information about the `StreamThread` objects, which shows runtime information,

in your application can be useful as well. To access the `StreamThread` info, use the `KafkaStreams.localThreadsMetadata()` method.

7.3.2 Getting notification on various states of the application

When you start your Kafka Streams application, it doesn't automatically begin processing data—some coordination has to happen first. The consumer needs to fetch metadata and subscription information; the application needs to start `StreamThread` instances and assign `TopicPartitions` to `StreamTasks`.

This process of assigning or redistributing tasks (workload) is called *rebalancing*. Rebalancing means Kafka Streams can autoscale up or down. This is a crucial strength—you can add new application instances *while an existing application is already running*, and the rebalancing process will redistribute the workload.

For example, suppose you have a Kafka Streams application with two source topics, and each topic has two partitions, resulting in four `TopicPartition` objects needing assignment. You initially start the application with one thread. Kafka Streams determines the number of tasks to create by taking the max partition size among all input topics. In this case, each topic has two partitions, so the max is two, and you end up with two tasks. The rebalance process then assigns the two tasks two `TopicPartition` objects each.

After running the app for a little while, you decide you want to process records more quickly. All you need to do is start another version of the application *with the same application ID*, and the rebalance process will distribute the load across the new application thread, resulting in the two tasks being assigned across both threads. You've just doubled the scale of your application while the original version is still running—there's no need to shut the initial application down.

Other causes of rebalancing include another Kafka Streams instance (with the same application ID) starting or stopping, adding partitions to a topic, or, in the case of a regex-defined source node, adding or removing topics matching the regex pattern.

During the rebalance phase, external interaction temporarily pauses until the application has completed the assignment of topic partitions to stream tasks, so you'd like to be aware of this point in the lifecycle of the application. For example, the queryable state stores are unavailable, so you'd like to be able to restrict requests to view the contents of the store until the stores are available again.

But how can you check whether your other applications are going through a rebalance?

Fortunately, Kafka Streams provides just such a mechanism, the `StateListener`, which we'll look at next.

7.3.3 Using the StateLstener

A Kafka Streams application can be in one of six states at any point in time. Figure 7.13 shows the possible valid states for a Kafka Streams application. As you can see, there are a few state-change scenarios we could discuss, but we're going to focus on the transition between running and rebalancing. The transition between these two states is the most frequent and has the most impact on performance, because during the rebalancing phase no processing occurs.

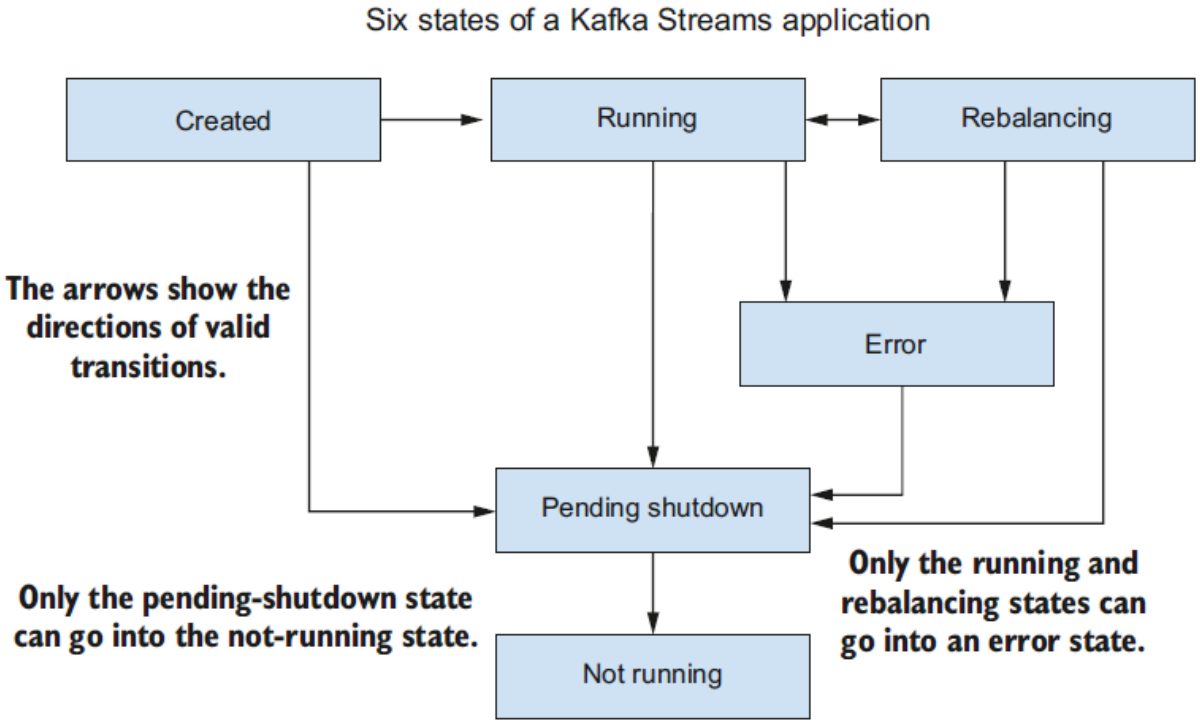


Figure 7.13 Possible states of a Kafka Streams application

To capture these state changes, you'll use the `KafkaStreams#setStateListener` method, which takes an instance of the `StateListener` interface. It's a single-method interface, so you can use Java 8 lambda syntax, as follows (found in src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java).

Listing 7.4 Adding a state listener

```
KafkaStreams.StateListener stateListener = (newState, oldState) ->; {
    if (newState == KafkaStreams.State.RUNNING &&
[CA]oldState == KafkaStreams.State.REBALANCING) {   (1)
      LOG.info("Application has gone from REBALANCING to RUNNING ");
      LOG.info("Topology Layout {}",
```

```
[CA]streamsBuilder.build().describe());   (1)
    }
 };
```

(1) Checks that you're transitioning from REBALANCING to RUNNING

(2) Prints out the structure of the topology

TIP Listing 7.4, running ZMartKafkaStreamsAdvancedReqsMetricsApp.java, involves viewing JMX metrics and the state-transition notification, so I've turned off printing the streaming results to the console. You're writing solely to Kafka topics. When you run the app, you should see the listener output in the console.

For your first `StateListener` implementation, you'll log the state change to the console. In section 7.3.17.15, when we discussed printing the topology structure, I spoke of needing to wait until the application has completed rebalancing. That's what you do in the listing 7.4: print out the structure once all tasks and assignments are complete.

Let's take this example a little further and show how to signal when the application is going into a rebalancing state. You can update your code to handle this additional state transition as follows (found in src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java).

Listing 7.5 Updating the state listener when `REBALANCING`

```
KafkaStreams.StateListener stateListener = (newState, oldState) ->; {
  if (newState == KafkaStreams.State.RUNNING &&
[CA}oldState == KafkaStreams.State.REBALANCING) {
      LOG.info("Application has gone from REBALANCING to RUNNING ");
      LOG.info("Topology Layout {}", streamsBuilder.build().describe());
  }

  if (newState == KafkaStreams.State.REBALANCING) {
      LOG.info("Application is entering REBALANCING phase");   (1)
  }
};
```

(1) Adds an action when entering the rebalancing phase

Even though you're using simple logging statements, it should be evident how you can add more-sophisticated logic to handle the state changes within your application.

NOTE	Because Kafka Streams is a library and not a framework, you can run a single instance on one server. If you do run multiple applications on different machines, you'll only see results from state changes on your local machine.

The critical point of this section is that you can hook into the current state of your Kafka Streams application, making it less of a black-box operation.

Next, we'll look at rebalancing in a little more in depth. Although the ability to automatically rebalance the workload is a strength of Kafka Streams, you'll likely want to keep the number of rebalances to a minimum. When a rebalance occurs, you're not processing data, and you'd like to have your application processing data as much as possible.

7.3.4 State restore listener

You learned in chapter 4 about state stores and the importance of having your state stores backed up, in case of a failure. In Kafka Streams, we use topics frequently called *changelogs* as the backup for the state stores.

The changelog records the updates to the state store as the update occurs. When a Kafka Streams application fails, or you restart it, the state store can recover from the local state files, as shown in figure 7.14.

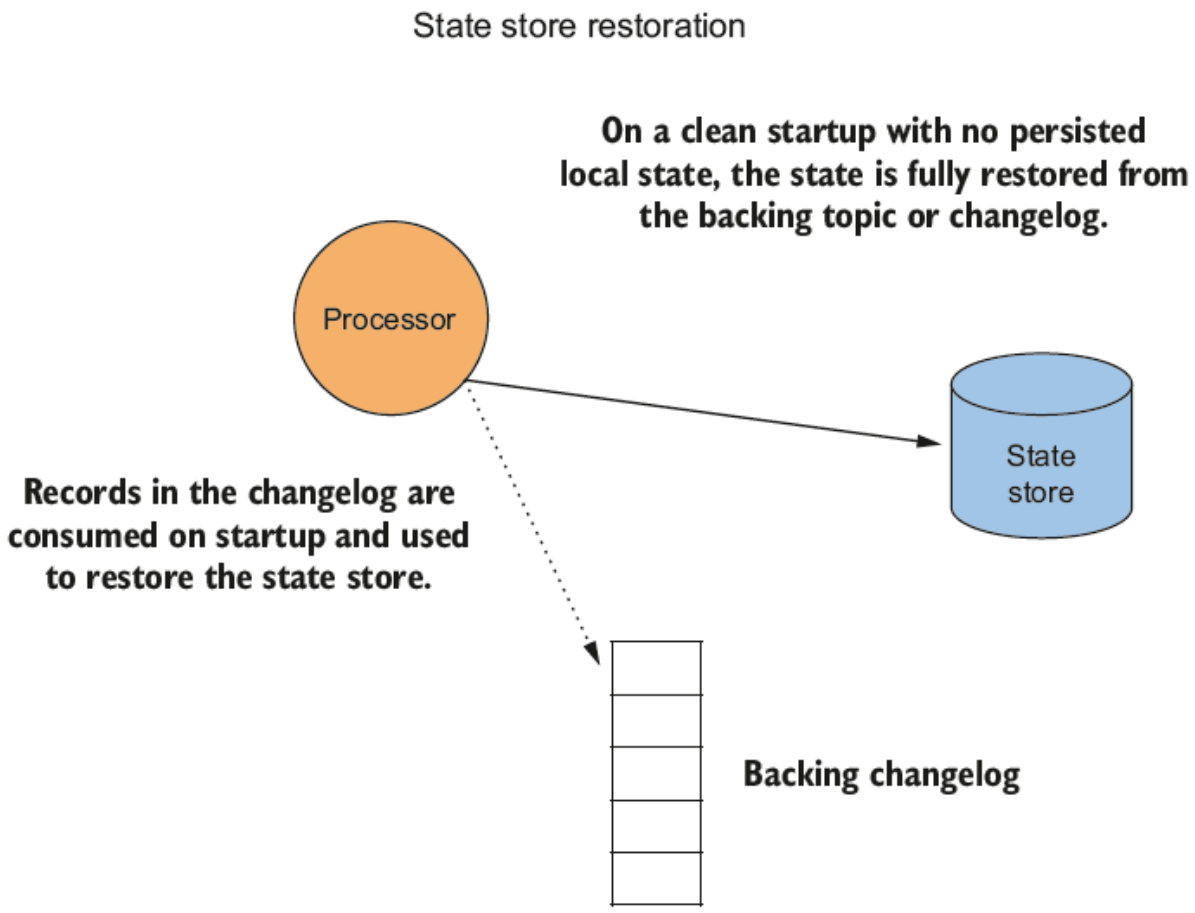


Figure 7.14 Restoring a state store from clean start/recovery

In some circumstances, however, you may need to do a full recovery of state store from the changelog, such as if you're running your Kafka Streams application in a stateless environment like Mesos, or if you encounter a severe failure and the files on local disk are wiped out. Depending on the amount of data you have to restore, this restoration process could take a non-trivial amount of time.

During this recovery period, any state stores you have exposed for querying are unavailable, so it would be nice to get an idea of how long this restoration process is likely to take and how progress is coming along. Additionally, if you have a custom state store, you'd like notification of when the restore is starting and ending so you can do any necessary setup or teardown tasks.

The `StateRestoreListener` interface, much like the `StateListener`, allows notification of what's going on inside the application. `StateRestoreListener` has three methods: `onRestoreStart`, `onBatchRestored`, and `onRestoreEnd`. The `KafkaStreams#setGlobalRestoreListener` method is used to specify the global restore listener to use.

NOTE The provided `StateRestoreListener` is shared application-wide and is expected to be stateless. If you need to keep track of any state in the listener, you'll need to provide the synchronization.

Let's walk through the listener code to get an idea of how this notification process can work. We'll start with the declaration of the variable and the `onRestoreStart` method (found in src/main/java/bbejeck/chapter_7/restore/LoggingStateRestoreListener.java).

Listing 7.6 A logging restore listener

```
public class LoggingStateRestoreListener implements StateRestoreListener {

    private static final Logger LOG =
[CA]LoggerFactory.getLogger(LoggingStateRestoreListener.class);
    private final Map<TopicPartition, Long>; totalToRestore =
[CA]new ConcurrentHashMap<>;();                                    (1)
    private final Map<TopicPartition, Long>; restoredSoFar =
[CA]new ConcurrentHashMap<>;();                                    (1)


    @Override
    public void onRestoreStart(TopicPartition topicPartition,
[CA]String store, long start, long end) {
        long toRestore = end - start;

        totalToRestore.put(topicPartition, toRestore);             (1)
        LOG.info("Starting restoration for {} on topic-partition {}
[CA]total to restore {}", store, topicPartition, toRestore);
    }

   // other methods left out for clarity covered below
}
```

(1) Creates ConcurrentHashMap instances for keeping track of restore progress

(2) Stores the total amount to restore for the given TopicPartition

Your first steps are to create two `ConcurrentHashMap` instances for keeping track of restoration progress. In the `onRestoreStart` method, you store the total number of records you need to restore and log the fact that you're starting.

Next, let's move on to the code that handles each batch restored (found in src/main/java/bbejeck/chapter_7/restore/LoggingStateRestoreListener.java).

Listing 7.7 Handling `onBatchRestored`

```
@Override
 public void onBatchRestored(TopicPartition topicPartition,
[CA]String store, long start, long batchCompleted) {
   NumberFormat formatter = new DecimalFormat("#.##");
```

```
   long currentProgress = batchCompleted +
[CA]restoredSoFar.getOrDefault(topicPartition, 0L);
   double percentComplete =
[CA](double) currentProgress / totalToRestore.get(topicPartition);

   LOG.info("Completed {} for {}% of total restoration for {} on {}",
                batchCompleted,
[CA]formatter.format(percentComplete * 100.00),
[CA]store, topicPartition);

   restoredSoFar.put(topicPartition, currentProgress);
 }
```

1. Calculates the total number of records restored
2. Determines the percentage of restoration completed
3. Logs the percent restored
4. Stores the number of records restored so far

The restoration process uses an internal consumer to read the changelog topic, so it follows that the application restores records in batches from each `consumer.poll()` method call. As a consequence, the maximum size of any batch will be equal to the `max.poll.records` setting.

The `onBatchRestored` method is called after the restore process has loaded the latest batch into the state store. First, you add the size of the current batch to the accumulated restore count. Then, you calculate the percentage of restoration completed and log the results. Finally, you store the new total number of records, computed earlier.

The last step we'll cover is when the restoration process completes (found in src/main/java/bbejeck/chapter_7/restore/LoggingStateRestoreListener.java).

Listing 7.8 Method called when restoration is completed

```
@Override
 public void onRestoreEnd(TopicPartition topicPartition,
[CA]String store, long totalRestored) {
    LOG.info("Restoration completed for {} on
[CA]topic-partition {}", store, topicPartition);
    restoredSoFar.put(topicPartition, 0L);
 }
```

1. Keeps track of restore progress for a TopicPartition

Once the application completes the recovery process, you make one final call to the listener with the total number of records restored. In this example, you log the finished

state and update the full restoration count map to 0.

Finally, you can use the `LoggingStateRestoreListener` in your application as follows (found in src/main/java/bbejeck/chapter_7/CoGroupingListeningExampleApplication.java).

Listing 7.9 Specifying the global restore listener

```
kafkaStreams.setGlobalStateRestoreListener(new LoggingStateRestoreListener());
```

This is an example of using a `StateRestoreListener`. In chapter 9, you'll see an example that includes a graphical representation of the restore progress.

TIP To view the log file generated by running the `CoGroupingListeningExampleApplication` example, look for a log file named logs/state_restore_listener.log in the root directory where you installed the source code.

7.3.5 Uncaught exception handler

I think it's fair to say that every developer, from time to time, has encountered an unaccounted-for `Exception` and the big stack trace in the console/log as your program suddenly quits. Although this situation doesn't quite fit into a "monitoring" example, the ability to get a notification and handle any cleanup when the unexpected occurs is good practice. Kafka Streams provides `KafkaStreams#setUncaughtExceptionHandler` for dealing with these unexpected errors (found in src/main/java/bbejeck/chapter_7/CoGroupingListeningExampleApplication.java).

Listing 7.10 Basic uncaught exception handler

```
kafkaStreams.setUncaughtExceptionHandler((thread, exception) ->; {
            CONSOLE_LOG.info("Thread [" + thread + "]
[CA]encountered [" + exception.getMessage() +"]");
          });
```

This is definitely a bare-bones implementation, but it serves to demonstrate where you can set a hook for dealing with unexpected errors, either by logging the error as shown here or by performing any required cleanup and shutting down the streams application.

That wraps up our coverage of monitoring Kafka Streams applications.

7.4 Summary

- To monitor Kafka Streams, you'll need to look at the Kafka brokers as well.
- You should enable metrics reporting from time to time to see the how the performance of the application is doing.
- Peeking under the hood is required, and sometimes you'll need to go to lower levels and use command-line tools included with Java, such as `jstack` (thread dumps) and `jmap`/`jhat` (for heap dumps) to understand what your application is doing.

In this chapter, we focused on *observing* behavior. In the next chapter, we'll shift our focus to making sure the application handles errors consistently and adequately. We'll also make sure that it provides *expected* behavior by doing regular testing.

8 Testing a Kafka Streams application

This chapter covers

- Testing a topology
- Testing individual processors and transformers
- Integration testing with an embedded Kafka cluster

So far, we've covered the essential building blocks for creating a Kafka Streams application. But there's one crucial part of application 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 a Kafka Streams application, because that makes your logic much more accessible to test. I expect you're aware of the importance of testing, but we'll review my 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 will execute. The only way to prove that the code works is by testing thoroughly, so you'll use testing to provide a good breadth of possible inputs and scenarios to make sure your code works appropriately under reasonable circumstances.

The second reason you need an excellent suite of tests is to deal with the inevitable changes in software. Having a good set of tests gives you immediate feedback when your new code has broken the expected set of behaviors. Additionally, when you do a major refactor or add new functionality, passing tests give you a level of confidence about the impact of your changes (provided you have good tests).

Even once you understand the importance of testing, testing a Kafka Streams application

isn't always straightforward. You still have the option of running a simple topology and observing the results, but there's one drawback with that approach. You'll want a suite of *repeatable* tests that you can run at any time, and as part of a build, so you'd like the ability to test your applications without a Kafka cluster and ZooKeeper ensemble.

That's what we'll cover in this chapter. First, you'll see how you can test a topology *without* Kafka running, so you can see the entire topology working from a unit test. You'll also learn how to test a processor or transformer independently and mock any required dependencies.

NOTE You likely have experience testing with mock objects, but if not, Wikipedia's article provides a good introduction: https://en.wikipedia.org/wiki/Mock_object.

Although unit testing is critical for repeatability and quick feedback, integration testing is important as well, because sometimes you'll need to see the moving parts of your application in action. For example, consider the case of rebalancing, an essential part of a Kafka Streams application. Getting rebalancing to work in a unit test is nearly impossible. Table 8.1 summarizes the differences between unit and integration testing.

Table 8.1 Testing approaches

Test type	Purpose	Testing speed	Level of use
Unit	Testing individual parts of functionality in isolation	Fast	Large majority
Integration	Testing integration points between whole systems	Longer to run	Small minority

You need to trigger an actual rebalance under realistic conditions to test it. For those situations, you'll need to run with a live instance of a Kafka cluster. But you don't want to rely on having an external cluster set up, so we'll look at how you can use an embedded Kafka and ZooKeeper for integration testing.

8.1 Testing a topology

The first topology we built, in chapter 3, was relatively complicated. Figure 8.1 shows it again, to refresh your memory.

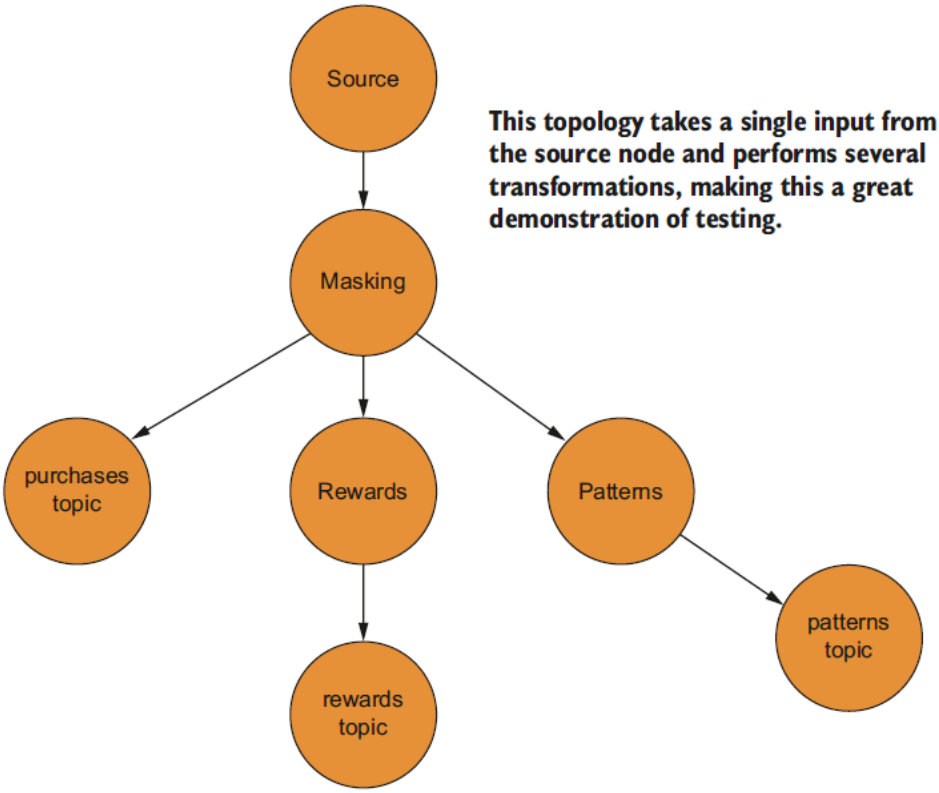


Figure 8.1 Initial complete topology for ZMart Kafka Streams program

The processing logic is pretty straightforward, but as you can see from the structure, it has several nodes. It also has one key thing that will help demonstrate testing: it takes one input—an initial purchase—and it performs several transformations. This will make testing somewhat easy, in that you only need to supply a single purchase value, and you can confirm that all the appropriate transformations take place.

TIP In most cases, you'll want to have your logic in separate classes so you can unit test the business logic separately from the topology. In the case of the ZMart topology, most of the logic is simple and represented as Java 8 lambda expressions, so in this case you'll test the topology flow.

You'll want a repeatable standalone test, so you'll use the `ProcessorTopologyTestDriver`, which will allow you to write such a test *without* needing Kafka to run the test. Remember, the ability to test your topology without a live Kafka instance makes testing faster and more lightweight, leading to a quicker development cycle. Also note that the `ProcessorTopologyTestDriver` is a generic testing framework that tests the individual Kafka Streams `Topology` objects you've created.

SIDEBAR

Using Kafka Streams' testing utilities

To use Kafka Streams' testing utilities, you'll need to update your build.gradle file with the following:

```
testCompile  group:'org.apache.kafka', name:'kafka-streams',
[CCA]version:'1.0.0', classifier:'test'

testCompile  group:'org.apache.kafka', name:'kafka-clients',
[CCA]version:'1.0.0', classifier:'test'
```

If you're using Maven, use this code:

```
<dependency>;
   <groupId>;org.apache.kafka</groupId>;
   <artifactId>;kafka-streams</artifactId>;
   <version>;1.0.0</version>;
   <scope>;test</scope>;
   <classifier>;test</classifier>;
 </dependency>;

<dependency>;
   <groupId>;org.apache.kafka</groupId>;
   <artifactId>;kafka-clients</artifactId>;
   <version>;1.0.0</version>;
   <scope>;test</scope>;
   <classifier>;test</classifier>;
 </dependency>;
```

TIP

If you write your own projects using Kafka and Kafka Streams testing code, it's best if you use the all the dependencies that come with the sample code.

When you initially built this topology, you did it entirely within the `ZMartKafkaStreamsApp#main` method, which was fine for quick development at the time, but it doesn't lend itself to being testable. What you'll do now is refactor the topology into a standalone class, which will enable you to test the topology.

The logic isn't changing, and you'll move the code as is, so I won't show the conversion here. Instead, I'll point you to src/main/java/bbejeck/chapter_8/ZMartTopology.java, where you can view it if you choose.

With the code transitioned, let's get on with constructing a test.

8.1.1 Building the test

Let's move on to building a unit test for the ZMart topology. You can use a standard JUnit test, and you'll have some setup work to do before running the test (found in src/main/java/bbejeck/chapter_8/ZMartTopologyTest.java).

Listing 8.1 Setup method for topology test

```
@Before
public  void setUp() {

 // properties construction left out for clarity
 StreamsConfig streamsConfig = new StreamsConfig(props);
 Topology topology = ZMartTopology.build();                    (1)

 topologyTestDriver =
[CCA]new ProcessorTopologyTestDriver(streamsConfig, topology);  (2)
}
```

(1) Refactored ZMart topology: now you can get the topology from the method call.

(2) Creates the ProcessorTopologyTestDriver

The critical point of listing 8.1 is the creation of the `ProcessorTopologyTestDriver`, which you'll use in the following listing when you run your test (found in src/main/java/bbejeck/chapter_8/ZMartTopologyTest.java).

Listing 8.2 Testing the topology

```
@Test
 public void testZMartTopology() {

 // serde creation left out for clarity

 Purchase purchase = DataGenerator.generatePurchase();          (1)

 topologyTestDriver.process("transactions",                     (2)
                            null,
                            purchase,
                            stringSerde.serializer(),
                            purchaseSerde.serializer());

 ProducerRecord<String, Purchase>; record =
 [CCA]topologyTestDriver.readOutput("purchases",                (2)
                                    stringSerde.deserializer(),
                                   purchaseSerde.deserializer());

 Purchase expectedPurchase =
 [CCA]Purchase.builder(purchase).maskCreditCard().build();      (3)
 assertThat(record.value(), equalTo(expectedPurchase));         (3)
```

1. Creates a test object; reuses the generation code from running the topology
2. Sends an initial record into the topology
3. Reads a record from the purchases topic
4. Converts the test object to the expected format
5. Verifies that the record from the topology matches the expected record

There are two critical sections in listing 8.2. Starting with `topologyTestDriver.process`, you feed a record into the `transactions` topic, because it's the source for the entire topology. With the topology loading completed, you can verify that the correct actions have taken place. In the following line, using the `topologyTestDriver.readOutput` method, you read the record from the `purchases` topic, with one of the sink nodes defined in the topology. In the second-to-last line, you create the expected output record, and on the final line, you assert that the results are what you expect.

There are two other sink nodes in the topology, so let's complete the test by verifying you get the correct output from them (found src/test/java/bbejeck/chapter_8/ZMartTopologyTest.java).

Listing 8.3 Testing the rest of the topology

```
@Test
 public void testZMartTopology() {

 // continuing test from the previous section

  RewardAccumulator expectedRewardAccumulator =
[CCA]RewardAccumulator.builder(expectedPurchase).build();

 ProducerRecord<String, RewardAccumulator>; accumulatorProducerRecord =
 [CCA]topologyTestDriver.readOutput("rewards",                    (1)
                            stringSerde.deserializer(),
                            rewardAccumulatorSerde.deserializer());

 assertThat(accumulatorProducerRecord.value(),
[CCA]equalTo(expectedRewardAccumulator));                        (2)

 PurchasePattern expectedPurchasePattern =
[CCA]PurchasePattern.builder(expectedPurchase).build();

 ProducerRecord<String, PurchasePattern>; purchasePatternProducerRecord =
[CCA]topologyTestDriver.readOutput("patterns",                   (2)
                                stringSerde.deserializer(),
                                purchasePatternSerde.deserializer());

 assertThat(purchasePatternProducerRecord.value(),
[CCA]equalTo(expectedPurchasePattern));                          (3)
}
```

1. Reads a record from the rewards topic
2. Verifies the rewards topic output matches expectations
3. Reads a record from the patterns topic
4. Verifies the patterns topic output matches expectations

As you add another processing node to the test, you'll see the same pattern as in listing 8.3. You read records from each topic and verify your expectations with an assert statement. The critical point to keep in mind with this test is that you now have a repeatable test running a record through your entire topology, without the overhead of running Kafka.

The `ProcessorTopologyTestDriver` also supports testing topologies with a state store, so let's look at how you'd accomplish that.

8.1.2 Testing a state store in the topology

To demonstrate testing a state store, you'll refactor another class, `StockPerformanceStreamsAndProcessorApplication`, to have the `Topology` returned from a method call. You'll find the class in src/main/java/bbejeck/chapter_8/StockPerformanceStreamsProcessorTopology.java. I haven't made any changes to the logic, so we won't review it here.

The test setup is the same as in the previous test, so I'll limit my explanations to the parts that are new (src/test/java/bbejeck/chapter_8/StockPerformanceStreamsProcessorTopologyTest.java).

Listing 8.4 Testing the state store

```
StockTransaction stockTransaction =
[CCA]DataGenerator.generateStockTransaction();  (1)

topologyTestDriver.process("stock-transactions",  (2)
                stockTransaction.getSymbol(),
                stockTransaction,
                stringSerde.serializer(),
                stockTransactionSerde.serializer());

KeyValueStore<String, StockPerformance>; store =
[CCA]topologyTestDriver.getKeyValueStore("stock-performance-store");  (2)

assertThat(store.get(stockTransaction.getSymbol()),
[CCA]notNullValue());  (3)
```

1. Generates a test record

2. Processes the record with the test driver
3. Retrieves the state store from the test topology
4. Asserts the store contains the expected value

As you can see, the last `assert` line quickly verifies that your code is using the state store as expected. You've seen the `ProcessorTopologyTestDriver` in action, and you've seen how you can achieve end-to-end testing of a topology. The topologies you test can be very simple, with one processing node, or very complex, consisting of several sub-topologies. Even though you're doing this testing without a Kafka broker, make no mistake: this is a full test of the topology that will exercise all parts, including serializing and deserializing records.

You've seen how you can do end-to-end testing of a topology. But you'll also want to test the internal logic of your `Processor` and `Transformer` objects. Testing an entire topology is great, but verifying the behavior inside each class requires a more fine-grained approach, which we'll cover next.

8.1.3 Testing processors and transformers

To verify the behavior inside a single class requires a true unit test, where there's only *one* class under test. Writing a unit test for a `Processor` or `Transformer` shouldn't be very challenging, but remember that both classes have a dependency on the `ProcessorContext` for obtaining any state stores and scheduling punctuation actions.

You don't want to create a real `ProcessorContext` object, but rather a *stand-in* you can use for testing purposes: a mock object. When it comes to using a mock object, you can follow two paths.

One option is to use a mock object framework such as Mockito (http://site.mockito.org) to generate mock objects in your test. Another option is to use the `MockProcessorContext` object found in the same test library as `ProcessorTopologyTestDriver`. Which one you use will depend on how you need to use them.

If you need the mock object strictly as a placeholder for a real dependency, concrete mocks (mocks not created from a framework) are a good choice. But if you want to verify the parameters passed to a mock, the value returned, or any other behavior, using a mock object generated by a framework is a good choice. Mock object frameworks (like Mockito) come with a rich API for setting expectations and verifying behavior, saving you development time and speeding up your testing process.

In listing 8.5, you'll use both types of mock objects. You'll use the Mockito framework to create the `ProcessorContext` mock, because you want to verify parameters during the `init` call as well as validate that you're forwarding the expected values from the `punctuate()` method. You'll also use a custom mock object for the key/value store, which you'll see in action as we step through the code example.

In this listing, you'll test a `Processor`, using mock objects. You'll start with a test for the `AggregatingMethodHandleProcessor` named `AggregatingMethodHandleProcessorTest`, located in src/test/java/bbejeck_chapter6/processor/cogrouping/. First, you want to verify the parameters used in the `init` method (see src/test/java/bbejeck/chapter_6/AggregatingMethodHandleProcessorTest.java).

Listing 8.5 Testing the `init` method

```
// some details left out for clarity
private ProcessorContext processorContext =
[CCA]mock(ProcessorContext.class);                    (1)
private MockKeyValueStore<String, Tuple<List<ClickEvent>;,
[CCA]List<StockTransaction>;>;>; keyValueStore =
[CCA]new MockKeyValueStore<>;();                      (1)

private AggregatingMethodHandleProcessor processor =
[CCA]new AggregatingMethodHandleProcessor();          (2)

 @Test
 @DisplayName("Processor should initialize correctly")
 public void testInitializeCorrectly() {
  processor.init(processorContext);                   (2)
  verify(processorContext).schedule(eq(15000L), eq(STREAM_TIME),
[CCA] isA(Punctuator.class));                         (3)
  verify(processorContext).getStateStore(TUPLE_STORE_NAME);   (6)
}
```

(1) Mocks the ProcessorContext with Mockito

(2) A mock KeyValueStore object

(3) The class under test

(4) Calls the init method on the processor, triggering method calls on ProcessorContext

(5) Verifies the parameters for the ProcessorContext.schedule method

(6) Verifies retrieving the state store

This first test is a simple one: you call the `init` method on the processor under test with

the mocked `ProcessorContext`. You then validate the parameters used to schedule the punctuate method, and that the state store is retrieved.

Next, let's test the `punctuate` method to verify that the records are forwarded as expected (found in src/test/java/bbejeck/chapter_6/AggregatingMethodHandleProcessorTest.java).

Listing 8.6 Testing the `punctuate` method

```
@Test
@DisplayName("Punctuate should forward records")
public void testPunctuateProcess(){
  when(processorContext.getStateStore(TUPLE_STORE_NAME))
                       .thenReturn(keyValueStore);  (1)

  processor.init(processorContext);                 (1)
  processor.process("ABC", Tuple.of(clickEvent, null));   (2)
  processor.process("ABC", Tuple.of(null, transaction));  (2)

  Tuple<List<ClickEvent>;,List<StockTransaction>;>; tuple =
[CCA]keyValueStore.innerStore().get("ABC");   (3)
  List<ClickEvent>; clickEvents = new ArrayList<>;(tuple._1);
  List<StockTransaction>; stockTransactions = new ArrayList<>;(tuple._2);

  processor.cogroup(124722348947L);             (3)

  verify(processorContext).forward("ABC",
[CCA]Tuple.of(clickEvents, stockTransactions));   (6)

  assertThat(tuple._1.size(), equalTo(0));      (7)
  assertThat(tuple._2.size(), equalTo(0));      (7)
}
```

(1) Sets mock behavior to return a KeyValueStore when called
(2) Calls init method on processor
(3) Processes a ClickEvent and a StockTransaction
(4) Extracts the entries put into the state store in the process method
(5) Calls the co-group method, which is the method used to schedule punctuate
(6) Validates that the ProcessorContext forwards the expected records
(7) Validates that the collections within the tuple are cleared out

This test is a little more involved, and it utilizes a mix of mock and real behavior. Let's take a brief walk through the test.

The first line specifies the behavior for the mock `ProcessorContext` to return the

stubbed-out `KeyValueStore` when the `ProcessorContext#getStateStore` method is called. This line alone is an interesting mix of generated mock versus a stubbed-out mock object.

I could easily have used Mockito to generate a mock `KeyValueStore`, but I chose not to for two reasons. First, a generated mock returning another generated mock seems a bit unnatural (in my opinion). Second, you want to verify and use the stored values in the `KeyValueStore` during the test instead of setting expectations with a canned response.

The next three lines, starting with `processor.init`, run the processor through its usual steps: first initializing and then processing records. The fourth step is where having a working `KeyValueStore` is important. Because the `KeyValueStore` is a simple stub, you use a `java.util.HashMap` underneath for the actual storage. In the three lines after setting processor expectations, you retrieve the contents from the store placed there by the `process()` method calls. You create new `ArrayList` objects with the contents of the `Tuple` (again, this is a custom class developed for the sample code in this book) pulled from the state store by the provided key.

Next, you drive the `punctuate` method of the processor. Because this is a unit test, you don't need to test how time is advanced—that would constitute testing the Kafka Streams API itself, which you don't want here. Your goal is to verify the behavior of the method you defined as your `Punctuator` (via a method reference).

Now, you verify the main point of the test: that the expected key and value are forwarded downstream via the `ProcessorContext#forward` method. This portion of the test demonstrates the usefulness of a generated mock object. Using the Mockito framework, you just need to tell the mock to expect a `forward` call with the given key and value, and verify that the test executed the code precisely in this manner. Finally, you verify that the processor cleared out the collections of `ClickEvent` and `StockTransaction` objects after forwarding them downstream.

As you can see from this test, you can isolate the class under test with a mix of generated and stubbed-out mock objects. As I stated earlier in this chapter, the bulk of the testing in your Kafka Streams API application should be unit tests on your business logic and on any individual `Processor` or `Transformer` objects. Kafka Streams itself is thoroughly tested, so you'll want to focus your efforts on new, untested code.

You probably won't want to wait to deploy your application to see how it interacts with a Kafka cluster. You'll want to sanity check your code, which will require integration testing. Let's look at how you can *locally* test against a real Kafka broker.

8.2 Integration testing

So far, you've seen how you can test an entire topology or an individual component in a unit test. For the most part, these types of tests are best, as they're quick to run, and they validate specific parts of your code.

But there are times where you'll need to test all the working parts together, end to end: in other words, an integration test. Usually, an integration test is required when you have some functionality that can't be covered in a unit test.

For an example, let's go back to our very first application, the Yelling App. Because you created the topology so long ago, take another look at it in figure 8.2.

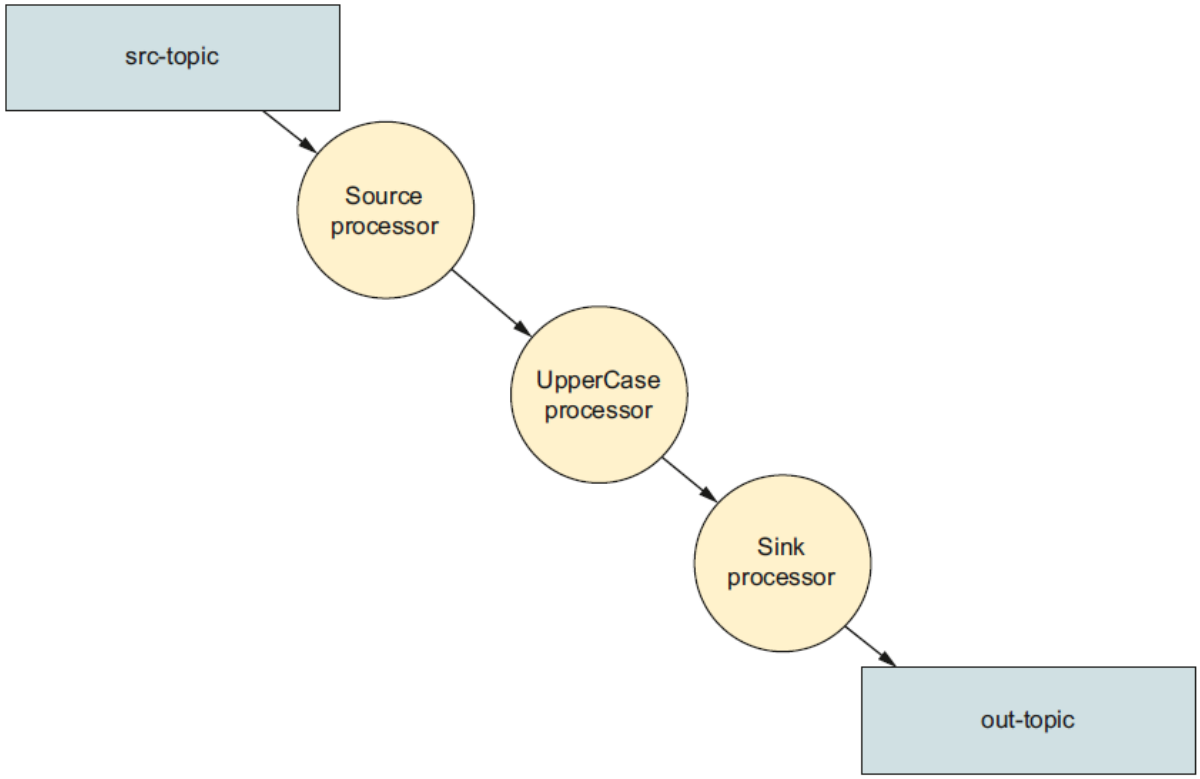


Figure 8.2 Another look at the Yelling App topology

Suppose you've decided to change the source from a single named topic to any topic matching a regex:

```
yell-[A-Za-z0-9-]`
```

As an example, you want to confirm that if a topic matching the pattern `yell-at-everyone` is added while your application is deployed and running, you'll start

reading information from that newly added topic.

You won't update the original Yelling App, since it's so small. Instead you'll use the following modified version directly in the test (found in src/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java).

Listing 8.7 Updating the Yelling App

```
streamsBuilder.<String,String>;stream(Pattern.compile("yell.*"))     (1)
                .mapValues(String::toUpperCase)                     (1)
                .to(OUT_TOPIC);                                     (1)
```

(1) Subscribes to any topic starting with "yell"

(2) Converts all text to uppercase

(3) Writes out to topic, or yells at people!

Because you add topics at the Kafka broker level, the only real way to test whether your application picks up a newly created topic is to add one while your application is running. Running this scenario is all but impossible with a unit test. But does this mean you need to deploy your updated app to test it?

Fortunately, the answer is no. You can use the *embedded* Kafka cluster available with Kafka test libraries.

By using the embedded Kafka cluster, you can run an integration test requiring a Kafka cluster on your machine at any point, either individually or as part of your entire battery of tests. This speeds up your development cycle. (I use the term *embedded* here to refer to running a large application like Kafka or ZooKeeper in local standalone mode, or "embedding" it in an existing application.) Let's move on to building an integration test.

8.2.1 Building an integration test

The first step to using the embedded Kafka server requires you to add three more testing dependencies—scala-library-2.12.4.jar, kafka_2.12-1.0.0-test.jar, and kafka_2.12-1.0.0.jar—to your build.gradle or pom.xml file. We've already covered the syntax for providing a test JAR in section 8.1, so I won't repeat it here.

While it may seem like the number of dependencies is starting to increase, remember that anything you add here is a *testing* dependency. Testing dependencies aren't packaged up and deployed with your application code; hence, they won't affect the size of your final

application.

Now that you've added the required dependencies, let's start defining the integration test with an embedded Kafka broker. You'll use a standard JUnit approach for creating the integration test.

ADDING THE EMBEDDEDKAFKACLUSTER

Adding the embedded Kafka broker to the test is a matter of adding one line, as shown in the following listing (found in src/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java).

Listing 8.8 Adding the embedded Kafka broker

```
private static final int NUM_BROKERS = 1;                    (1)

@ClassRule                                                   (1)
public static final EmbeddedKafkaCluster EMBEDDED_KAFKA
[CCA]= new EmbeddedKafkaCluster(NUM_BROKERS);                (2)
```

(1) Defines the number of brokers

(2) The JUnit ClassRule annotation

(3) Creates an instance of the EmbeddedKafkaCluster

In the second line listing 8.8, you create the `EmbeddedKafkaCluster` instance that serves as the cluster for running the tests in the class. The key point in this example is the `@ClassRule` annotation. A full description of testing frameworks and JUnit is beyond the scope of this book, but I'll take a minute here to explain the importance of `@ClassRule` and how it drives the test.

JUNIT RULES

JUnit introduced the concept of *rules* to apply some common logic JUnit tests. Here's a brief definition, from https://github.com/junit-team/junit4/wiki/Rules#rules: "Rules allow very flexible addition or redefinition of the behavior of each test method in a test class."

JUnit provides three types of rules, and the `EmbeddedKafkaCluster` class uses the `ExternalResource` rules (https://github.com/junit-team/junit4/wiki/Rules#externalresource-rules). You use `ExternalResource` rules for setting up and tearing down external resources, such as the `EmbeddedKafkaCluster` needed for a test.

JUnit provides the `ExternalResource` class, which has two no-op methods, `before()` and `after()`. Any class extending the `ExternalResource` must override the `before()` and `after()` methods for setting up and tearing down the external resource needed for testing.

Rules provide an excellent abstraction for using external resources in your tests. After you create your class extending `ExternalResource`, all you need to do is create a variable in your test and use the `@Rule` or `@ClassRule` annotation, and all the setup and teardown methods will be executed automatically.

The difference between `@Rule` and `@ClassRule` is how often `before()` and `after()` are called. The `@Rule` annotation executes `before()` and `after()` methods for *each individual* test in the class. `@ClassRule` executes the `before()` and `after()` methods *once*; `before()` is executed prior to any test execution, and `after()` is called when the last test in the class completes. Setting up an `EmbeddedKafkaCluster` is relatively resource intensive, so it makes sense that you'll only want to set it up once per test class.

Let's get back to building an integration test. You've created an `EmbeddedKafkaCluster`, so the next step is to create any topics you'll initially need.

CREATING TOPICS

Now that your embedded Kafka cluster is available, you can use it to create topics, as follows (src/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java).

Listing 8.9 Creating the topics for testing

```
@BeforeClass                                    (1)
public static void setUpAll() throws Exception {
  EMBEDDED_KAFKA.createTopic(YELL_A_TOPIC);     (2)
  EMBEDDED_KAFKA.createTopic(OUT_TOPIC);        (3)
}
```

(1) BeforeClass annotation

(2) Creates the first source topic

(3) Creates the output topic

Creating topics for the test is something you'll want to do only once for all tests, so you can use a `@BeforeClass` annotation, which creates the required topics before the execution of any tests. For this test, you only need topics with a single partition and a replication factor of 1, so you can use the convenience method

`EmbeddedKafkaCluster#createTopic(String name)`. If you needed more than one partition, a replication factor greater than 1 requires configurations different from the defaults. For that, you can use one of the following overloaded `createTopic` methods:

- `EmbeddedKafkaCluster#createTopic(String topic, int partitions, int replication)`
- `EmbeddedKafkaCluster#createTopic(String topic, int partitions, int replication, Properties topicConfig)`

With all the pieces in place for the embedded Kafka cluster to run, let's move on to testing the topology with the embedded broker.

TESTING THE TOPOLOGY

All the pieces are in place. Now you can follow these steps to execute the integration test:

1. Start the Kafka Streams application.
2. Write some records to the source topic and assert the correct results.
3. Create a new topic matching your pattern.
4. Write some additional records to the newly created topic and assert the correct results.

Let's start with the first two parts of the test (found in src/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java).

Listing 8.10 Starting the application and asserting the first set of values

```
// some setup code left out for clarity

kafkaStreams = new KafkaStreams(streamsBuilder.build(), streamsConfig);
kafkaStreams.start();                                                    (1)

List<String>; valuesToSendList =
[CCA]Arrays.asList("this", "should", "yell", "at", "you");               (2)
List<String>; expectedValuesList =
[CCA]valuesToSendList.stream()
                     .map(String::toUpperCase)
                     .collect(Collectors.toList());                      (3)


IntegrationTestUtils.produceValuesSynchronously(YELL_A_TOPIC,            (4)
                                                valuesToSendList,
                                                 producerConfig,
                                                 mockTime);
int expectedNumberOfRecords = 5;
List<String>; actualValues =
[CCA]IntegrationTestUtils.waitUntilMinValuesRecordsReceived(
[CCA]consumerConfig, OUT_TOPIC, expectedNumberOfRecords);                (5)

assertThat(actualValues, equalTo(expectedValuesList));                   (6)
```

(1)

0-1)) Starts the Kafka Streams application

2. 0-2)) Specifies the list of values to send
3. 0-3)) Creates the list of expected values
4. 0-4)) Produces the values to embedded Kafka
5. 0-5)) Consumes records from Kafka
6. 0-6)) Asserts the values read are equal to the expected values

This portion of the test is pretty standard testing code. You "seed" your streaming application by writing records to the source topic. The streaming application is already running, so it consumes, processes, and writes out records as part of its standard processing. To verify that the application is performing as you expect, the test consumes records from the sink-node topic and compares the expected values to the actual values.

Toward the end of listing 8.10 are two static utility methods, `IntegrationTestUtils.produceValuesSynchronously` and `IntegrationTestUtils.waitUntilMinValuesRecordsReceived`, making the construction of this integration test much more manageable. These producing and consuming utility methods are part of kafka-streams-test.jar. Let's discuss these methods briefly.

PRODUCING AND CONSUMING RECORDS IN A TEST

The `IntegrationTestUtils#produceValuesSynchronously` method creates a `ProducerRecord` for each item in the collection with a null key. This method is synchronous, as it takes the resulting `Future<RecordMetadata>;` from the `Producer#send` call and immediately calls `Future#get()`, which blocks until the produce request returns. Because this method is sending records synchronously, you know the records are available for consuming once the method returns. Another method, `IntegrationTestUtils#produceKeyValuesSynchronously`, takes a collection of `KeyValue<K,V>;` if you want to specify a value for the key.

For consuming records in listing 8.10, you use the `IntegrationTestUtils#waitUntilMinValuesRecordsReceived` method. As you can probably guess from the name, this method will attempt to consume the expected number of records from the given topic. By default, this method will wait up to 30 seconds, and if the expected number of records has not been consumed, an `AssertionError` is thrown, failing the test.

If you need to work with the consumed `KeyValue` instead of just the value, there's the

`IntegrationTestUtils#waitUntilMinKeyValueRecordsReceived` method, which works in the same manner but returns a `Collection` of `KeyValue` results. Additionally, there are overloaded versions of the consuming utility, where you can specify a custom amount of time to wait via a parameter of type `long`.

Now, let's finish describing the test.

DYNAMICALLY ADDING A TOPIC

You're at the point in the test where you want to test the dynamic behavior that you need a live Kafka broker for. The previous portion of the test was done to verify the starting point. Now, you're going to use the `EmbeddedKafkaCluster` to create a new topic, and you'll test that the application consumes from the new topic and processes records as expected (found in src/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java).

Listing 8.11 Starting the application and asserting values

```
EMBEDDED_KAFKA.createTopic(YELL_B_TOPIC);                                    (1)

valuesToSendList = Arrays.asList("yell", "at", "you", "too");               (2)

expectedValuesList = valuesToSendList.stream()                               (3)
                                     .map(String::toUpperCase)
                                     .collect(Collectors.toList());

IntegrationTestUtils.produceValuesSynchronously(YELL_B_TOPIC,                (4)
                                                valuesToSendList,
                                                producerConfig,
                                                mockTime);
expectedNumberOfRecords = 4;
List<String>; actualValues =
[CCA]IntegrationTestUtils.waitUntilMinValuesRecordsReceived(
[CCA]consumerConfig, OUT_TOPIC, expectedNumberOfRecords);                    (5)

assertThat(actualValues, equalTo(expectedValuesList));                       (6)
```

(1) Creates the new topic

(2) Specifies a new list of values to send

(3) Creates the expected values

(4) Produces the values to the source topic of the streaming application

(5) Consumes the results of the streaming application

(6) Asserts that the expected results match the actual results

You create a new topic matching the pattern for the source node of the streaming application. After that, you go through the same steps of populating the new topic with data, and consuming records from the topic backing the sink node of the streaming application. At the end of the test, you verify that the consumed results match the expected results.

You can run this test from inside your IDE, and you should see a successful result. You've completed your first integration test!

You won't want to use an integration test for everything, because unit tests are easier to write and maintain. But integration tests can be indispensable when the only way to verify the behavior of your code is working with a live Kafka broker.

NOTE It may be tempting to make all your tests using the `EmbeddedKafkaCluster`, but it's best if you don't. If you run the sample integration test you just built, you'll notice that it takes much longer to run than the unit tests. The few extra seconds taken by one test might not seem like much, but when you multiply that time by several hundred or a thousand or more tests, the time it takes to run your test suite is substantial. Additionally, you should always try to keep your tests small and focused on one specific piece of functionality instead of exercising all parts of the application chain.

8.3 Summary

- You should strive to keep business logic in standalone classes that are entirely independent of your Kafka Streams application. This makes them easy to unit test.
- It's useful to have at least one test using `ProcessorTopologyTestDriver` to test your topology from end to end. This type of test doesn't use a Kafka broker, so it's fast, and you can see end-to-end results.
- For testing individual `Processor` or `Transformer` instances, try to use a mock object framework only when you need to verify the behavior of some class in the Kafka Streams API.
- Integration tests with the `EmbeddedKafkaCluster` should be used sparingly, and only when you have interactive behavior that can only be verified with a live, running Kafka broker.

It's been a fun journey, and you've learned quite a lot about the Kafka Streams API and how you can use it to handle your data-processing needs. So to conclude your learning path, we'll now switch gears from student to master. The next and final chapter of this

book will be a capstone project based on everything you've learned so far, and in some cases extending out to write custom code that isn't in the Kafka Streams API. The result will be a live, end-to-end application using the core functionality presented in this book.

Part 4: Advanced Concepts with Kafka Streams

In this final part, you'll take everything you've learned and apply it to building advanced applications. You'll integrate Kafka Streams with Kafka Connect so that you can stream data even if it's being written to a relational database. Then, you'll learn how to use the power of interactive queries to display—in real time—information your application is building, directly from Kafka Streams, without needing an external tool. Finally, you'll learn about KSQL, a new tool introduced by Confluent (the company founded by the original developers of Kafka at LinkedIn), and how you can write SQL statements and run continuous queries on data coming into Kafka.

9 Advanced applications with Kafka Streams

This chapter covers

- Integrating outside data into Kafka Streams with Kafka Connect
- Kicking your database to the curb with interactive queries
- KSQL continuous queries in Kafka

You've come a long way in your journey to learn how to use Kafka Streams. We've covered a lot of ground, and now you should know how to build streaming applications. Up to this point, you've included the core functionality of Kafka Streams, but there's much more you can do. In this chapter, you'll use what you've learned to build two advanced applications that will allow you to work in real-world situations.

For example, in many organizations, when you bring in new technology, it must mesh with legacy technology or processes. It's not uncommon to see database tables as the main sink for incoming data. You learned in chapter 5 that tables are streams, so you know you should be able to treat database tables as streams of data.

The first advanced application we'll look at in this chapter will "convert" a physical database into a streaming application by integrating Kafka Connect with Kafka Streams. Kafka Connect will listen for new insertions into the table(s) and write those records to a topic in Kafka. This same topic will serve as the source for the Kafka Streams application so that you can turn your database table into a streaming application.

When you're working with legacy applications, even if data is captured in real time, it's typical to dump the data into a database to serve as the data source for dashboard applications. In this chapter's second advanced application, you'll see how to use interactive queries that expose the Kafka Streams state stores for direct queries. A

dashboard application can then pull directly from the state stores and show data as it's flowing through the streaming application, eliminating the need for a database.

We'll wrap up our coverage of advanced features by looking at a powerful new Kafka feature: KSQL. KSQL lets you write long-running SQL queries on data coming into Kafka. KSQL gives you all the power of Kafka Streams combined with the ease of writing a SQL query. When you use KSQL, it uses Kafka Streams under the covers to get the job done.

9.1 Integrating Kafka with other data sources

For the first advanced example application, let's suppose you work at an established financial services firm, Big Short Equity (BSE). BSE wants to migrate its legacy data operations into the modern era, and that plan includes using Kafka. The migration is part-way through, and you're tasked with updating the company's analytics. The goal is to display the latest equities transactions and associated information in real time, and Kafka Streams is the perfect fit.

BSE offers funds focused on different areas of the financial market. The company records fund transactions in real time in a relational database. Eventually, BSE plans to write transactions directly into Kafka, but in the short term, the database is the system of record.

Given that incoming data is fed into a relational database, how can you bridge the gap between the database and your emerging Kafka Streams application? The answer is to use Kafka Connect (https://kafka.apache.org/documentation/#connect), a framework that's included with Apache Kafka and that integrates Kafka with other systems. Once Kafka has the data, you're no longer concerned about the location of the source data; you just point your Kafka Streams application at the source topic as you've done with other Kafka Streams applications.

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 `KafkaConsumer` can use the imported data. Because this is a book about Kafka Streams, I emphasize how to integrate with a Kafka Streams application.

Figure 9.1 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, which is the source of your financial analysis application.

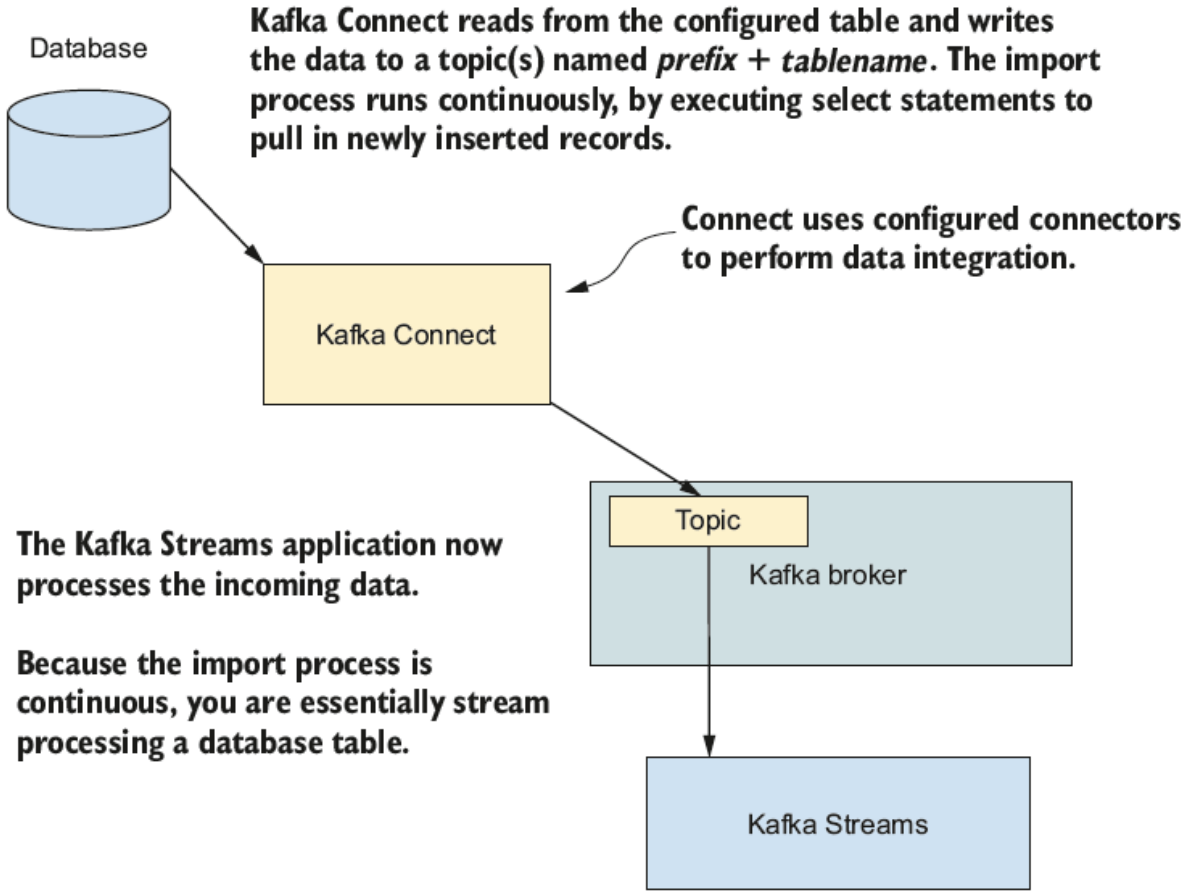


Figure 9.1 Kafka Connect integrating a database table and Kafka Streams

TIP	Because this is a book on Kafka Streams, this section is a whirlwind tour of Kafka Connect. For more in-depth information, see the Apache Kafka documentation (https://kafka.apache.org/documentation/#connect) and Kafka Connect Quick Start (https://docs.confluent.io/current/connect/quickstart.html).

9.1.1 Using Kafka Connect to integrate data

Kafka Connect is explicitly designed for streaming data from other systems into Kafka and for streaming data from Kafka into another data-storage application such as MongoDB (www.mongodb.com) or Elasticsearch (www.elastic.co). With Kafka Connect, it's possible to import entire databases into Kafka, or other data such as performance metrics.

Kafka Connect uses specific *connectors* to interact with outside data sources. Several connectors are available (www.confluent.io/product/connectors), many developed by the connector community, making integration between Kafka and almost any other storage system possible. If there isn't a connector available for your purposes, you can

implement a connector of your own (https://docs.confluent.io/current/connect/devguide.html).

9.1.2 Setting up Kafka Connect

Kafka Connect runs in two flavors: distributed mode 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. I'm assuming you'll run the examples on your local machine, so everything is configured in standalone mode.

The connectors that Kafka Connect uses to interact with outside data sources come in two types: source connectors and sink connectors. Figure 9.2 illustrates how Kafka Connect uses both types. As you can see, source connectors bring data into Kafka, and sink connectors receive data from Kafka for use by another system.

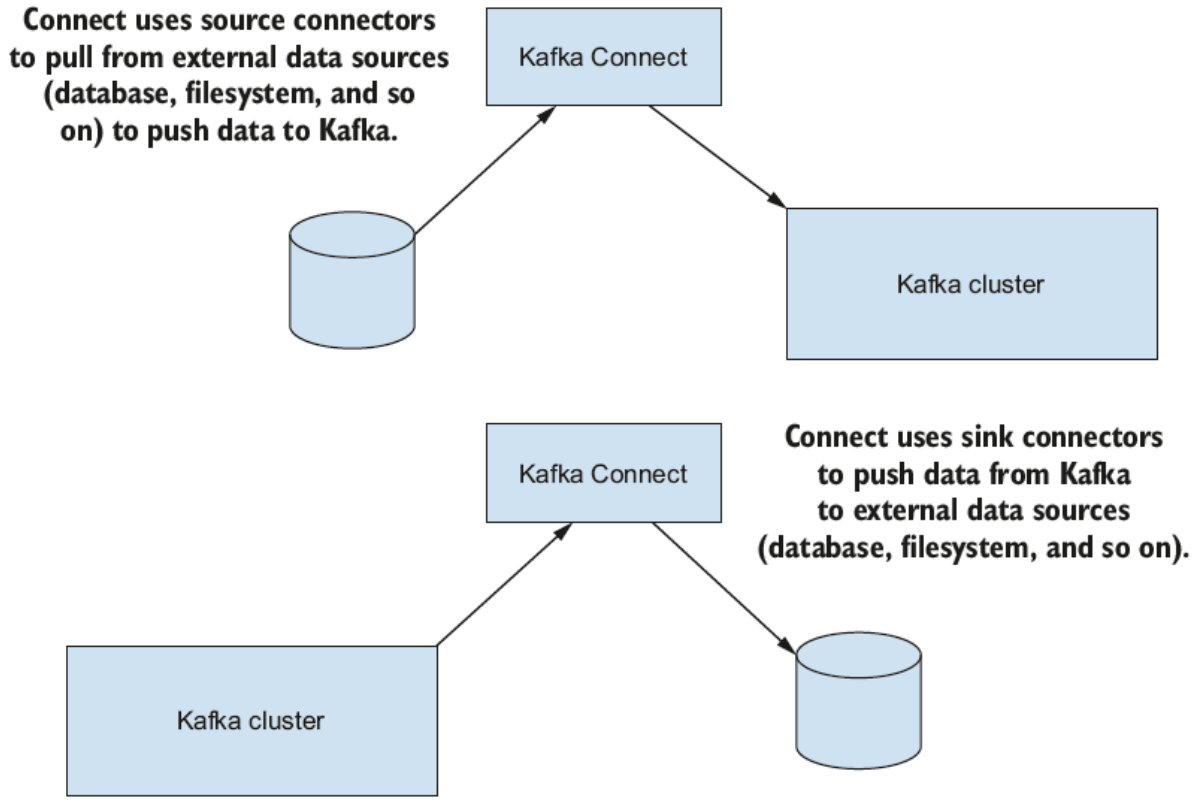


Figure 9.2 Kafka Connect source and sink connectors

For this example, you'll use the Kafka JDBC connector (https://github.com/confluentinc/kafka-connect-jdbc). It's available on GitHub and also packaged with the book's source code distribution as a convenience (https://manning.com/books/kafka-streams-in-action).

When using Kafka Connect, you'll need to do some minor configuration to Kafka Connect itself and to the individual connector you're using to import or export data. First, let's look at the configuration parameters you'll work with for Kafka Connect:

- `bootstrap.servers`—Comma-separated list of the Kafka broker(s) used by Connect.
- `key.converter`—Class of the converter that controls serialization of the key from Connect format to the format written to Kafka.
- `value.converter`—Class of the converter that controls 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`.
- `value.converter.schemas.enable`—`true` or `false`, specifying whether Connect should include the schema for the value. In this example, you'll set this value to `false`; I explain why in the next section.
- `plugin.path`—Tells Connect the location of a connector and its dependencies. This location can be a single, top-level directory containing an uber JAR file or multiple JAR files. You can also provide several paths in a comma-separated list of locations.
- `offset.storage.file.filename`—File containing the offsets stored by the Connect consumer.

You'll also need to provide some configuration for the JDBC connector. Let's review those settings:

- `name`—Name of the connector.
- `connector.class`—Class of the connector.
- `tasks.max`—The maximum number of tasks used by this connector.
- `connection.url`—URL used to connect to the database.
- `mode`—Method the JDBC source connector uses to detect changes.
- `incrementing.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 `incrementing.column.name`—in a little more detail, because they play an active role in how the connector runs. The JDBC source connector uses `mode` to detect which rows it needs to load. The example uses the `incrementing` setting, which relies on an auto-incrementing column where each insert increments the column value by 1. By tracking an incrementing column, you'll only pull in *new* records; updates will go unnoticed. Your Kafka Streams application is only concerned with pulling in the latest equities-product purchases, so this setting is ideal. `incrementing.column.name` is the name of the column containing the auto-incrementing value.

TIP	The source code for the book contains the nearly completed configuration for both Connect and the JDBC connector. The config files are located in the src/main/resources directory of the source code distribution (https://manning.com/books/kafka-streams-in-action). You'll need to provide some information about the path where you've extracted the source code repository. Be sure to look at the README.md file for full instructions.

This brings us to the end of our overview of Kafka Connect and the JDBC source connector. We have one more integration point to cover, which we'll discuss in the next section.

NOTE	You can find more information about the JDBC source connector in the Confluent documentation (http://mng.bz/01vh). Additionally, there are other incremental query modes you should look over (http://mng.bz/0pjP).

9.1.3 Transforming data

Before getting this new assignment, you had already developed a Kafka Streams application using similar data. As a result, you have an existing model and `Serde` objects (using Gson underneath for JSON serialization and deserialization). To keep your development velocity high, you'd prefer not to write any new code to support working with Connect. As you'll see in the next section, you'll be able to import data from Connect seamlessly.

TIP	Gson (https://github.com/google/gson) is an Apache licensed library developed by Google for the serialization and deserialization of Java objects into and from JSON. You can learn more from the user guide: http://mng.bz/JqV2.

To enable this seamless integration, you'll need to make some minor additional configuration changes to your JDBC connector's properties. Before you do, let's revisit section 9.1.2, where we discussed configuration settings. Specifically, I said you'd use `org.apache.kafka.connect.json.JsonConverter` with schemas disabled; hence, the value is converted into a simple JSON format.

Although JSON is what you want to consume in your Kafka Streams application, there are two issues. First, when converting the data into JSON format, the column names are used for the names of fields in the converted JSON. The names are all in an abbreviated BSE format that has no meaning outside the organization, so when your Gson serde is

converted from JSON to the expected model object, all the fields are null because the names don't match.

Second, the date and time are stored in the database as a timestamp, as expected. But the provided Gson serde hasn't defined a custom `TypeAdapter` (http://mng.bz/inzB) for the `Date` type, so all dates need to be a `String` formatted like this: `yyyy-MM-dd'T'HH:mm:ss.SSS-0400`. Fortunately, Kafka Connect provides a mechanism that allows you to handle these two issues with ease.

Kafka Connect has the concept of `Transformations` that let you perform *lightweight* transformations before Connect writes data to Kafka. Figure 9.3 shows where this transformation process takes place.

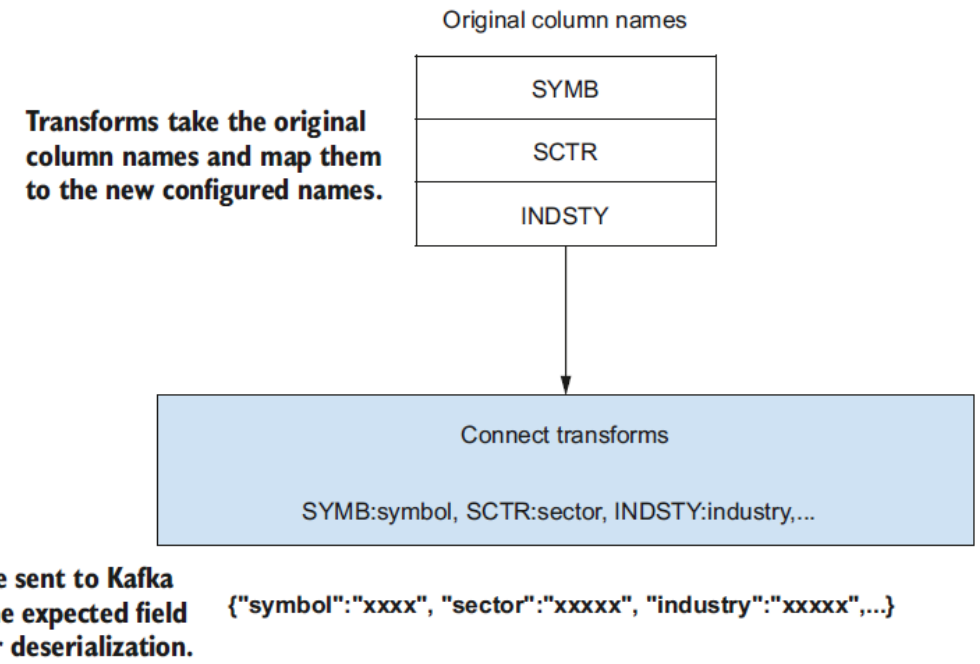


Figure 9.3 Transforming the names of the columns to match expected field names

In this example, you'll use two built-in transformations: `TimestampConverter` and `ReplaceField`. As I mentioned previously, to use these transformations, you need to add the following configuration lines to the connector-jdbc.properties file (see src/main/resources/conf/connector-jdbc.properties).

Listing 9.1 JDBC connector properties

```
transforms=ConvertDate,Rename                                    ❶
transforms.ConvertDate.type=
[CA]org.apache.kafka.connect.transforms.TimestampConverter$Value  ❷
transforms.ConvertDate.field=TXNTS                               ❸
```

```
transforms.ConvertDate.target.type=string                    4
transforms.ConvertDate.format=yyyy-MM-dd'T'HH:mm:ss.SSS-0400  5
transforms.Rename.type=
[CA]org.apache.kafka.connect.transforms.ReplaceField$Value   5
transforms.Rename.renames=SMBL:symbol, SCTR:sector,....       6
```

1. Aliases for the transformers
2. Type for the ConvertDate alias
3. Date field to convert
4. Output type of the converted date field
5. Format of the date
6. Type for the Rename alias
7. List of column names (truncated for clarity) to replace. The format is Original:Replacement.

These properties are relatively self descriptive, so we won't spend much time on them. As you can see, they provide you with exactly what you need for your Kafka Streams application to successfully deserialize messages imported into Kafka by Connect and the JDBC connector.

With all the Connect pieces in place, completing the integration between the database table and your Kafka Streams application is just a matter of using a topic with the prefix specified in the connector-jdbc.properties file (found in src/main/java/bbejeck/chapter_9/StockCountsStreamsConnectIntegrationApplication.java).

Listing 9.2 Kafka Streams source topic populated with data from Connect

```
Serde<StockTransaction>; stockTransactionSerde =
[CA]StreamsSerdes.StockTransactionSerde();   1
StreamsBuilder builder = new StreamsBuilder();
builder.stream("dbTxnTRANSACTIONS",
[CA]Consumed.with(stringSerde,stockTransactionSerde))   2
       .peek((k, v)->;
[CA]LOG.info("transactions from database key {}value {}", k, v));   2
```

1. Serde for the StockTransaction object
2. Uses the topic Connect writes to as the source for the stream
3. Prints messages out to the console

At this point, you're stream processing records from a database table in Kafka Streams,

but there's more to do. You're streaming stock-transaction data—to do any analysis, you need to group the transactions by their stock ticker symbol.

You've seen how to select a key and repartition the records, but it's more efficient if the records come into Kafka keyed; that way, your Kafka Streams application can skip the repartitioning step, which saves processing time and disk space. Let's revisit the configuration for Kafka Connect.

First, you can add a `ValueToKey` transformer that takes a list of field names in the value to extract and use for the key. Update the connector-jdbc.properties file as shown here (src/main/resources/conf/connector-jdbc.properties).

Listing 9.3 Updated JDBC connector properties

```
transforms=ConvertDate,Rename,ExtractKey                          (1)
transforms.ConvertDate.type=
[CA]org.apache.kafka.connect.transforms.TimestampConverter$Value
transforms.ConvertDate.field=TXNTS
transforms.ConvertDate.target.type=string
transforms.ConvertDate.format=yyyy-MM-dd'T'HH:mm:ss.SSS-0400
transforms.Rename.type=
[CA]org.apache.kafka.connect.transforms.ReplaceField$Value
transforms.Rename.renames=SMBL:symbol, SCTR:sector,....
transforms.ExtractKey.type=
[CA]org.apache.kafka.connect.transforms.ValueToKey                (2)
transforms.ExtractKey.fields=symbol                               (2)
```

(1) Adds the ExtractKey transform

(2) Specifies the class name of the ExtractKey transform

(3) Lists the field(s) to extract for the key

You add the `ExtractKey` transform and give Connect the name of the transformer class: `ValueToKey`. You also provide the name of the field to use for the key: `symbol`. This could consist of multiple comma-separated values, but in this case, you need only one value. Note that you use the renamed version for the field, because this transformer is executed *after* the `Rename` transformer.

The result of the `ExtractKey` field is a struct of one value. But you only want the value contained in the struct for the key—the stock ticker symbol. For this operation, you'll add a `FlattenStruct` transform to pull the ticker symbol out by itself (see src/main/resources/conf/connector-jdbc.properties).

Listing 9.4 Adding a transform

```
transforms=ConvertDate,Rename,ExtractKey,FlattenStruct                    (1)
transforms.ConvertDate.type=
[CA]org.apache.kafka.connect.transforms.TimestampConverter$Value
transforms.ConvertDate.field=TXNTS
transforms.ConvertDate.target.type=string
transforms.ConvertDate.format=yyyy-MM-dd'T'HH:mm:ss.SSS-0400
transforms.Rename.type=
[CA]org.apache.kafka.connect.transforms.ReplaceField$Value
transforms.Rename.renames=SMBL:symbol, SCTR:sector,....
transforms.ExtractKey.type=org.apache.kafka.connect.transforms.ValueToKey
transforms.ExtractKey.fields=symbol
transforms.FlattenStruct.type=
[CA]org.apache.kafka.connect.transforms.ExtractField$Key                  (2)
transforms.FlattenStruct.field=symbol                                     (2)
```

(1) Adds the last transform

(2) Specifies the class for the transform (ExtractField$Key)

(3) Name of the field to pull out

You add the final transform (`FlattenStruct`) and specify the `ExtractField$Key` class, which is used by Connect to extract the named field and only include that field in the results (in this case, the key). Finally, you provide the name of the field (`symbol`), which is the same as in the previous transform; this makes sense, because that's the field used to create the key struct.

With just a few extra lines of configuration, you can expand the previous Kafka Streams application to do more-advanced operations without the need to select a key and do the repartitioning step (found in src/main/java/bbejeck/chapter_9/StockCountsStreamsConnectIntegrationApplication.java).

Listing 9.5 Processing transactions from a table in Kafka Streams via Connect

```
Serde<StockTransaction>; stockTransactionSerde =
 [CA]StreamsSerdes.StockTransactionSerde();
 StreamsBuilder builder = new StreamsBuilder();
 builder.stream("dbTxnTRANSACTIONS",
 [CA]Consumed.with(stringSerde, stockTransactionSerde))
        .peek((k, v)->;
[CA]LOG.info("transactions from database key {}value {}", k, v))
        .groupByKey(
[CA]Serialized.with(stringSerde,stockTransactionSerde))                   (1)
        .aggregate(()->; 0L,(symb, stockTxn, numShares) ->;
[CA]numShares + stockTxn.getShares(),                                     (1)
                          Materialized.with(stringSerde, longSerde)).toStream()
        .peek((k,v) ->; LOG.info("Aggregated stock sales for {} {}",k, v))
        .to( "stock-counts", Produced.with(stringSerde, longSerde));
```

(1) Groups by key

(2) Performs an aggregation of the total number of shares sold

Because the data is coming in keyed, you can use `groupByKey`, which won't set the automatic repartitioning flag. From the group-by operation, you can directly go into an aggregation without performing a repartitioning step, which is important for performance reasons. The README.md file included with the source code contains instructions for running an embedded H2 database (www.h2database.com/html/main.html) and Kafka Connect to produce data for the `dbTxnTRANSACTIONS` topic to run the streaming application.

> **TIP** Although it might seem tempting to use `Transformations` to perform all the work when importing data into Kafka via Connect, remember that transformations are meant to be *lightweight*. For any transformations beyond the simple ones shown in the examples, it's better to pull the data into Kafka and then use Kafka Streams to do the heavy transformational work.

Now that you've seen how to use Kafka Connect to get data into Kafka for processing with Kafka Streams, let's turn our attention to how you can visualize the state of data in real time.

9.2 Kicking your database to the curb

In chapter 4, you learned how to add local state to a Kafka Streams application. Streaming applications need to use state to perform operations like aggregations, reduces, and joins. Unless your streaming application works exclusively with individual records, having local state is required.

Going back to the requirements from BSE, you've developed a Kafka Streams application that captures three categories of equity transactions:

- Total transactions by market sector
- Customer purchases of shares per session
- Total number of shares traded by stock symbol, in tumbling windows of 10 seconds

In the examples so far, you've either reviewed the results in the console or read them from a sink topic. Viewing data in the console is suitable for development, but the console isn't the best medium for displaying results. To do any analytic work or quickly understand what's going on, a dashboard application is a better medium to use.

In this section, you'll see how you can use interactive queries in Kafka Streams to develop a dashboard application for viewing analytics, *without* the need for a relational database to hold the state. You'll populate the dashboard application directly from Kafka

Streams, as the data streams. Hence, the app will continually update naturally.

In a typical architecture, data that's captured and operated on is pushed out to a relational database for viewing. Figure 9.4 shows this setup: pre-Kafka Streams, you ingested data with Kafka, fed an analysis engine, and then pushed the data to a relational database table used by a dashboard application.

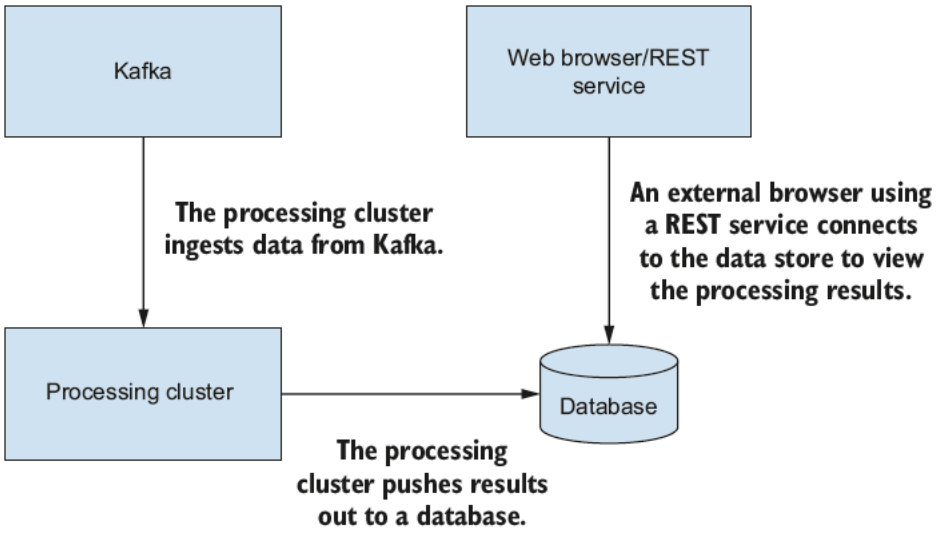


Figure 9.4 Architecture of applications viewing processed data prior to Kafka Streams

If you add Kafka Streams, using local state, the architecture changes slightly, as shown in figure 9.5. You can simplify the structure significantly by removing an entire cluster (not to mention that the deployment is much more manageable). Kafka Streams still writes back to Kafka, and the database is still the primary consumer of the transformed data.

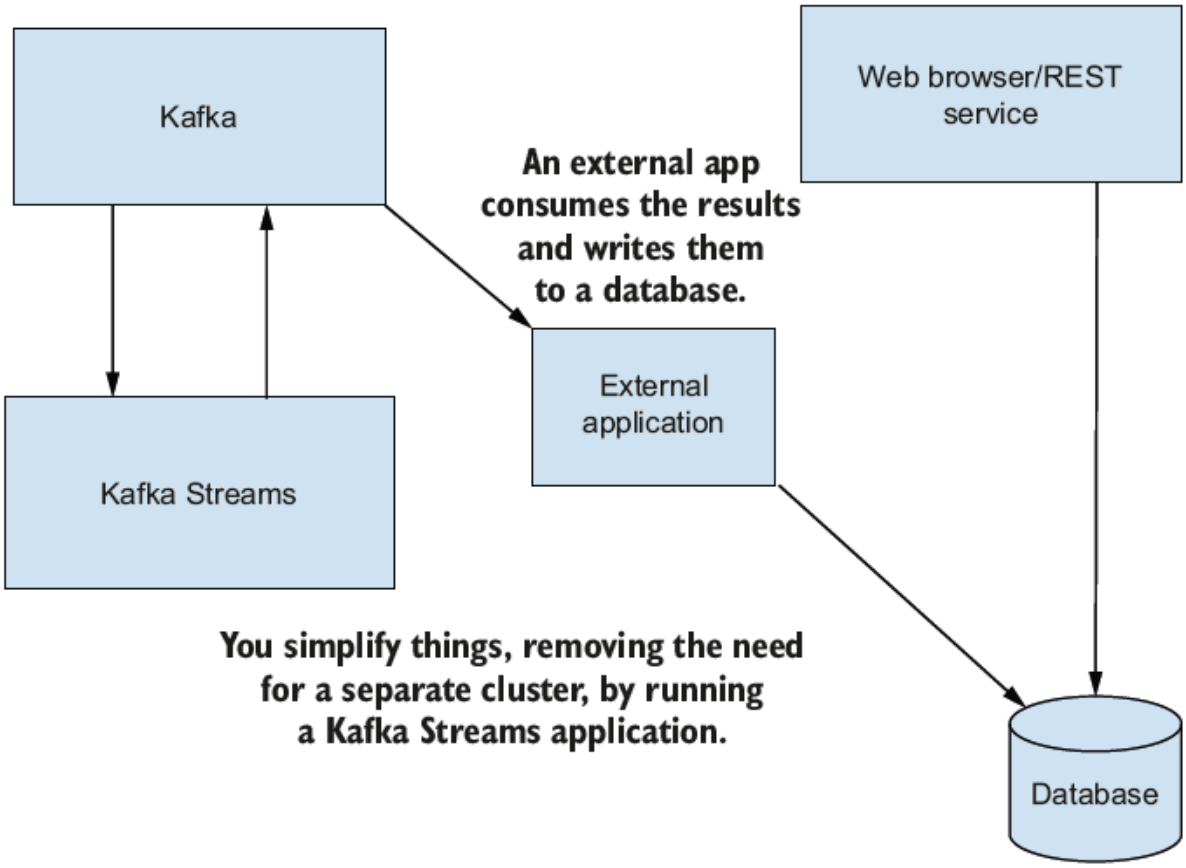


Figure 9.5 Architecture with Kafka Streams and state added

In chapter 5, I talked about interactive queries. Let's revisit the definition briefly: interactive queries let you directly view the data in the state store *without having to consume the data from Kafka*. In other words, the stream becomes the database as well.

Because a picture is worth a thousand words, let's take another look at figure 9.5, but adjusted in figure 9.6 to use interactive queries.

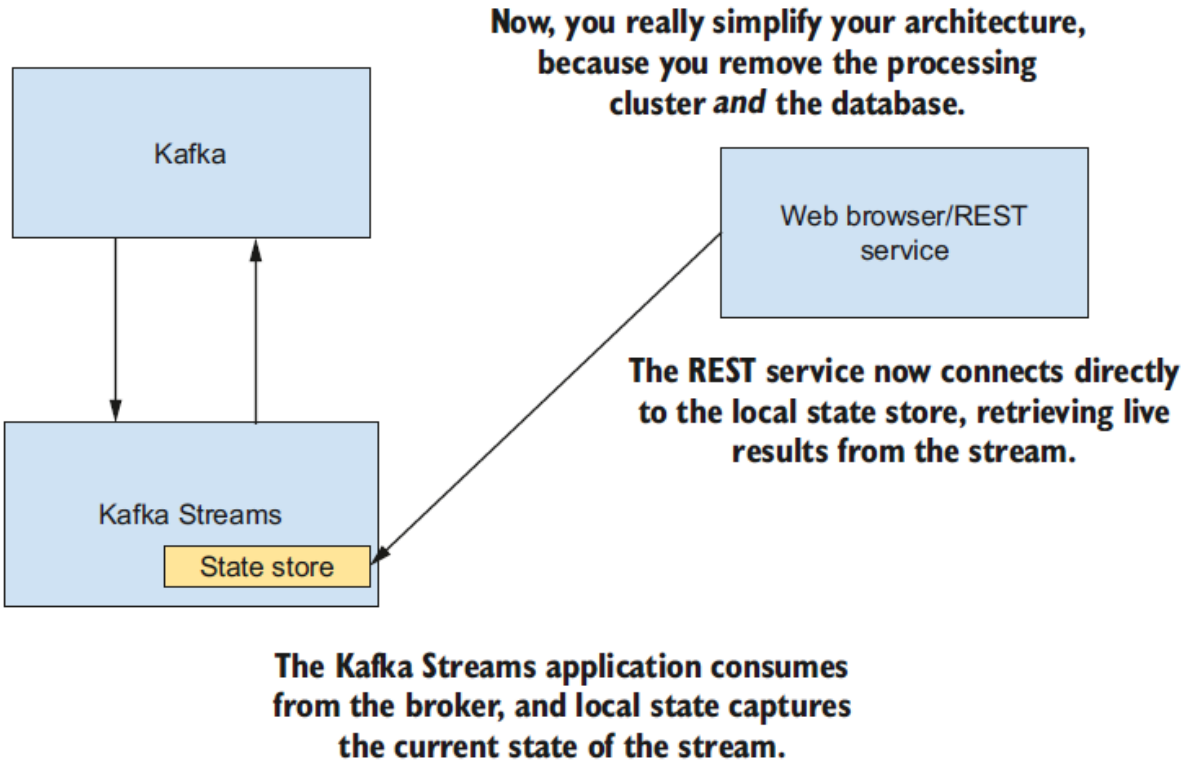


Figure 9.6 Architecture using interactive queries

The idea demonstrated here is simple but powerful. While state stores hold the state of the stream, Kafka Streams provides *read-only* access from outside the streaming application via a RESTful interface. It's worth stating again how powerful this concept is; you can view the running state of the stream without the need for an external database.

Now that you have an understanding of the impact of interactive queries, let's walk through how they work.

9.2.1 How interactive queries work

For interactive queries to work, Kafka Streams exposes state stores in a read-only wrapper. It's important to understand that while Kafka Streams makes the stores available for queries, there's no updating or modifying the state store in any way. Kafka Streams exposes state stores from the `KafkaStreams#store` method.

Here's how this method works:

```
ReadOnlyWindowStore readOnlyStore =
[CA]kafkaStreams.store(storeName, QueryableStoreTypes.windowStore());
```

This example retrieves a `WindowStore`. In addition, `QueryableStoreTypes` provides two other types:

- `QueryableStoreTypes.sessionStore()`
- `QueryableStoreTypes.keyValueStore()`

Once you have a reference to the read-only store, it's just a matter of exposing the store to a service (a RESTful service, for example) for users to query the state of the streaming data. But retrieving the state store is only part of the picture. The state store extracted here will only contain keys contained in the local store.

> **NOTE** Remember, Kafka Streams assigns a state store per task, and as long as you use the same application ID, a Kafka Streams application can consist of multiple instances. Additionally, those instances need not all be located on the same host. Thus, it's possible that the state store you query may contain only a subset of all the keys; other state stores (with the same name, located on other machine[s]) may contain another subset of the keys.

Let's use the analytics listed earlier to make this concept clear.

9.2.2 Distributing state stores

Consider the first analytic: aggregating stock trades per market sector. Because you're doing an aggregation, state stores come into play. You want to expose the state stores to provide a real-time view of the number of trades per sector, to gain insight into which segment of the market is seeing the most action at the moment.

Stock market activity generates significant data volume, but I'll only discuss using two partitions, to keep the details of the example clear. Additionally, let's say you're running two single-threaded instances on two separate machines, located in the same data center. Because of Kafka Streams' automatic load balancing, each application will have one task processing the data from each partition of the input topic.

Figure 9.7 shows the distribution of tasks and state stores. As you can see, instance A handles all records on partition 0, and instance B handles all records on partition 1. Figure 9.8 illustrates what happens when you have two records with the keys `"Energy"` and `"Finance"`.

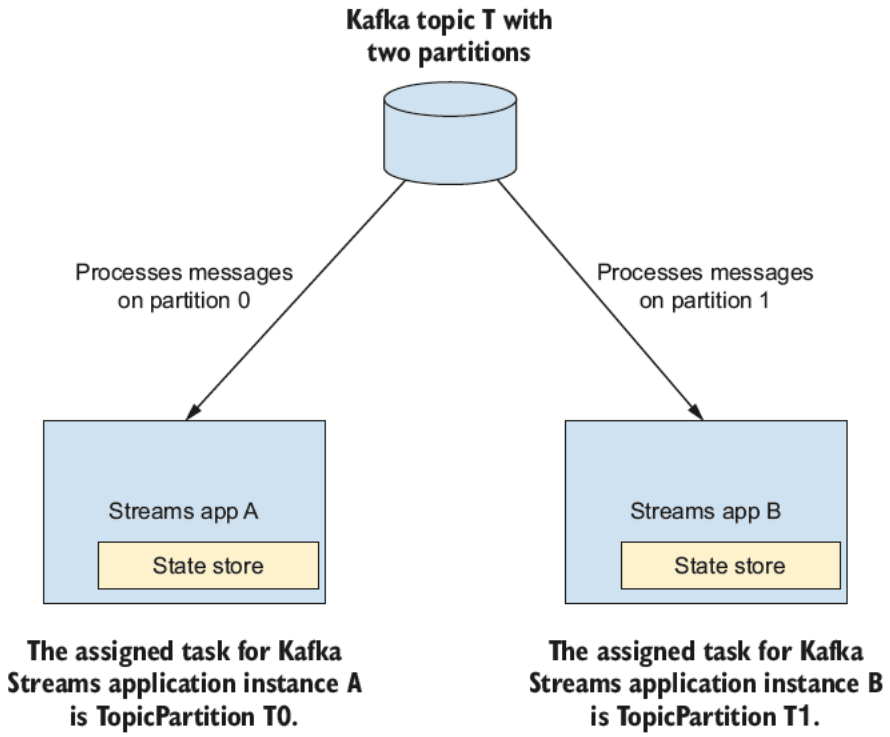


Figure 9.7 Task and state store distribution

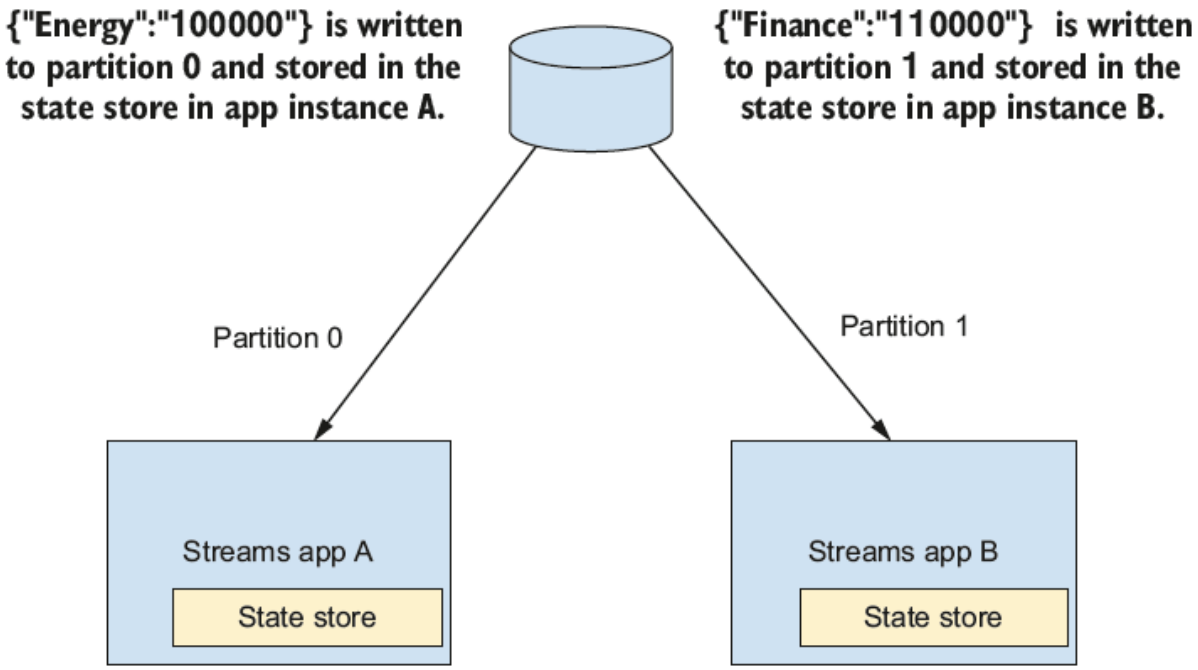


Figure 9.8 Key and value distribution in state stores

`"Energy":"100000"` lands in the state store on instance A, and `"Finance":"110000"` ends up in the state store on instance B. Going back to the example of exposing the state store for queries, you can clearly see that if you expose the state store on instance A to a web service or any external querying, you can only retrieve the value for the key `"Energy"`.

What's the solution to this issue? You certainly don't want to set up an individual web service to query each instance—that approach won't scale. Fortunately, you don't have to: the solution is as simple as setting a configuration.

9.2.3 Setting up and discovering a distributed state store

To enable interactive queries, you need to set the `StreamsConfig.APPLICATION_SERVER_CONFIG` parameter. It consists of the hostname of the Kafka Streams application and the port that a query service will listen on, in `hostname:port` format.

When a Kafka Streams instance receives a query for a given key, you'll need to find out whether the key is contained in the local store. More important, if it's not local, you'll want to find out which instance contains the key and query against that store.

Several methods on the `KafkaStreams` object allow for retrieving information for all *running* instances with the *same* application ID and defining the `APPLICATION_SERVER_CONFIG`. Table 9.1 lists the method names and descriptions.

Table 9.1 Methods for retrieving store metadata

Name	Parameter(s)	Usage
`allMetadata`	N/A	All instances, some possibly remote
`allMetadataForStore`	Store name	All instances (some remote) containing the named store
`allMetadataForKey`	Key, `Serializer`	All instances (some remote) with the store containing the key
`allMetadataForKey`	Key, `StreamPartitioner`	All instances (some remote) with the store containing the key

You can use `KafkaStreams.allMetadata` to obtain information for all instances that are eligible for interactive queries. I find that `KafkaStreams.allMetadataForKey` is the method I use most when writing interactive queries.

Next, let's take another look at the key/value distribution across Kafka Streams instances, adding the sequence of checking for the `"Finance"` key, which is found and returned from another instance (see figure 9.9). Each Kafka Streams instance has a lightweight embedded server listening to the port specified in `APPLICATION_SERVER_CONFIG`.

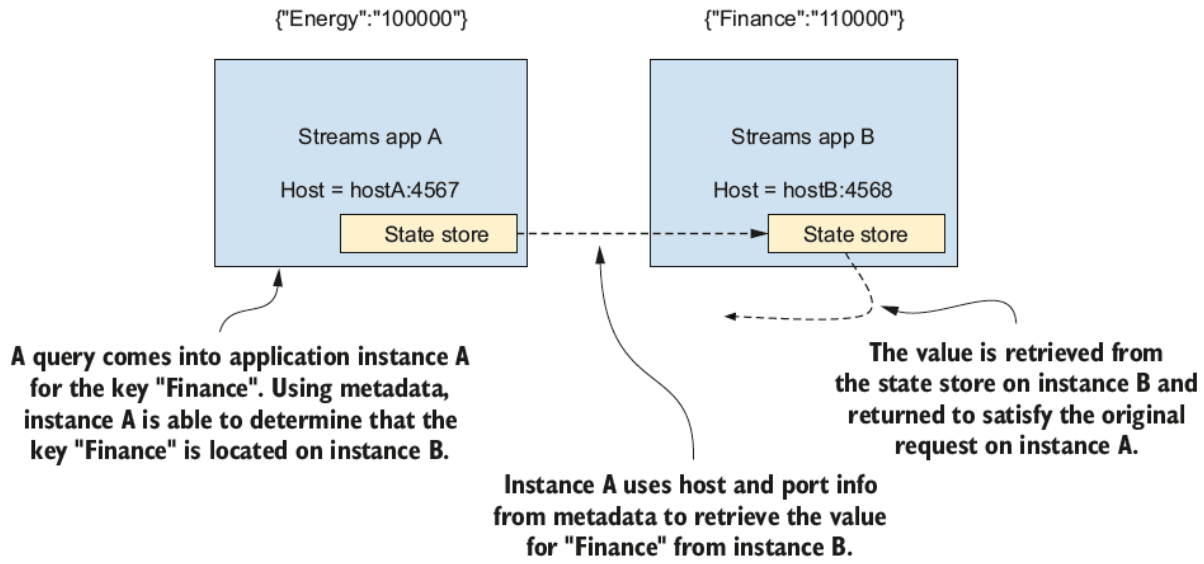


Figure 9.9 Key and value query-and-discovery process

It's important to point out that you'll need to query only *one* of the Kafka Streams instances, and which one you choose doesn't matter (assuming you've configured the application correctly). By using an RPC mechanism and metadata-retrieval methods, if the instance you've queried doesn't contain the data you're looking for, the queried Kafka Streams instance will find where it's located, pull the results, and return the results to the original query.

You can see this in action by tracing the flow of the calls in figure 9.9. Instance A doesn't contain the key `"Finance"` but discovers that instance B does contain the key. So, A issues a call to the embedded server on B, which retrieves the data and returns the result to the original caller.

NOTE Interactive queries will work on a single node out of the box, but an RPC mechanism isn't provided—you have to implement your own. This section offers one possible solution, but you're free to implement your own process, and I'm sure many of you will come up with something better. A great example of another RPC implementation is located in the Confluent kafka-streams-examples GitHub repo: http://mng.bz/Ogo3.

Let's move on to see interactive queries in action.

9.2.4 Coding interactive queries

The application you'll write for interactive queries will look very similar to the other apps you've written so far, with a couple of small changes. The first difference is that you need to pass in two arguments when launching the Kafka Streams application: the hostname and the port the embedded service will listen to (found in src/main/java/bbejeck/chapter_9/StockPerformanceInteractiveQueryApplication.java).

Listing 9.6 Setting the hostname and port

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

 if(args.length < 2){
   LOG.error("Need to specify host and port");
   System.exit(1);
 }

 String host = args[0];
 int port = Integer.parseInt(args[1]);

 final HostInfo hostInfo = new HostInfo(host, port);          (1)

Properties properties = getProperties();
properties.put(

[CA]StreamsConfig.APPLICATION_SERVER_CONFIG,host+":"+port);   (1)

// other details left out for clarity
```

(1) Creates a HostInfo object for later use in the application

(2) Sets the config for enabling interactive queries

Until this point, you've fired up the application without a second thought. Now, you need to provide two arguments (the host and the port), but this change has minimal impact.

You also embed the local server for performing actual queries: for this implementation, I've chosen to use the Spark web server (http://sparkjava.com). (Not *that* Spark—this is a book about Kafka Streams, after all!) My motivation for going with the Spark web server is its small footprint, its convention-over-configuration approach, and the fact that it's purpose-built for microservices—and a microservice is what you can provide by using interactive queries. If the Spark web server isn't to your liking, feel free to replace it with another web server.

NOTE I think most readers will be familiar with the term *microservice*, but here's the best definition I've seen, from http://microservices.io: "Microservices—also known as the microservice architecture—is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack."

Now, let's look at the point in the code where you embed the Spark server, and some of the supporting code used to manage it (found in src/main/java/bbejeck/chapter_9/StockPerformanceInteractiveQueryApplication.java).

Listing 9.7 Initializing the web server and setting its status

```
// details left out for clarity

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsConfig);
InteractiveQueryServer queryServer =
[CA]new InteractiveQueryServer(kafkaStreams, hostInfo);          (1)
queryServer.init();


 kafkaStreams.setStateListener(((newState, oldState) ->; {          (1)
   if (newState == KafkaStreams.State.RUNNING && oldState ==
[CA]KafkaStreams.State.REBALANCING) {
      LOG.info("Setting the query server to ready");
      queryServer.setReady(true);                                  (1)
   } else if (newState != KafkaStreams.State.RUNNING) {
      LOG.info("State not RUNNING, disabling the query server");
      queryServer.setReady(false);
   }
}));

kafkaStreams.setUncaughtExceptionHandler((t, e) ->; {
   LOG.error("Thread {} had a fatal error {}", t, e, e);
   shutdown(kafkaStreams, queryServer);                            (2)
});


Runtime.getRuntime().addShutdownHook(new Thread(() ->; {
  shutdown(kafkaStreams, queryServer);                             (3)
}));
```

(1) Creates the embedded web server (actually a wrapper class)

(2) Adds a StateListener to only enable queries to state stores until ready

(3) Enables queries to state stores once the Kafka Streams application is in a RUNNING state. Queries are disabled if the state isn't RUNNING.

(4) Sets an UncaughtExceptionHandler to log unexpected errors and close everything

down

5. Adds a shutdown hook to close everything down when the application exits normally

In this code, you create an instance of `InteractiveQueryServer`, which is a wrapper class containing the Spark web server and the code to manage the web service calls and start and stop the web server.

Chapter 7 discussed using a `StateListener` for notifications about various states of a Kafka Streams application. Here you can see an efficient use of this listener. Recall that when running an interactive query, you need to use an instance of `StreamsMetadata` to determine whether the data for the given key is local to the instance processing the query. You set the state of the query server to `true`, allowing access to the metadata needed for queries only if the application is in the `RUNNING` state.

A key point to keep in mind is that the metadata returned is a snapshot of the makeup of the Kafka Streams application. At any point in time, you can scale the application up (or down). When this occurs (or when any other qualifying event takes place, such as adding topics with a regex source node), the Kafka Streams application goes through a rebalancing phase and may change partition assignments. In this case, you're allowing queries only in the `RUNNING` state, but feel free to use whatever strategy you think is appropriate.

Next is another example of a concept covered in chapter 7: setting an `UncaughtExceptionHandler`. In this case, you log the error and shut down the application and the query server. Because this application runs indefinitely, you add a shutdown hook to close everything down once you stop the demo.

Now that you've seen how to instantiate and start the service, let's move on to the code for running the query server.

9.2.5 Inside the query server

When implementing your RESTful service, the first step is to map URL paths to the correct methods to execute (found in src/main/java/bbejeck/webserver/InteractiveQueryServer.java).

Listing 9.8 Mapping URL paths to methods

```
public void init() {
 LOG.info("Started the Interactive Query Web server");
```

```
 get("/kv/:store", (req, res) ->;  ready ?
 [CA]fetchAllFromKeyValueStore(req.params()) :
 [CA]STORES_NOT_ACCESSIBLE);                    (1)
 get("/session/:store/:key", (req, res) ->; ready ?
 [CA]fetchFromSessionStore(req.params()) :
 [CA]STORES_NOT_ACCESSIBLE);                    (1)
 get("/window/:store/:key", (req, res) ->; ready ?
 [CA]fetchFromWindowStore(req.params()) :
 [CA]STORES_NOT_ACCESSIBLE);                    (2)
 get("/window/:store/:key/:from/:to",(req, res) ->; ready ?
 [CA]fetchFromWindowStore(req.params()) :
 [CA]STORES_NOT_ACCESSIBLE);                    (2)
}
```

(1) Mapping to retrieve all values from a plain key/value store

(2) Mapping to return all sessions (from a session store) for a given key

(3) Mapping for a window store with no times specified

(4) Mapping for a window store with from and to times

This code highlights the decision to go with the Spark web server: you can concisely map URLs to a Java 8 lambda expression to handle the request. These mappings are straightforward, but notice that you map the retrieval from the window store twice. To retrieve values from a window store, you need to provide a *from* time and a *to* time.

In the URL mappings, notice the check for the `ready` Boolean value. This value is set in `StateListener`. If `ready` evaluates to `false`, you don't attempt to process the request, and you return a message that the stores aren't currently accessible. This makes sense because a window store is segmented by time, with the segment size established when you create the store. (We covered windowing in section 5.3.2.) But I'm cheating here and offering you a method that accepts only a key and a store and provides default *from* and *to* times that we'll explore in the next example.

NOTE There's a proposal (KIP-205, http://mng.bz/ll9Y) to extend `ReadOnlyWindowStore` to provide an `all()` method that retrieves all time segments by key, alleviating the need to specify *from* and *to* times. This functionality hasn't been implemented yet but should be included in a future release.

As an example of how the interactive query service works, let's walk through retrieving from a windowed store. Although we'll only look at one example, the source code contains instructions to run all types of queries.

CHECKING FOR THE STATE STORE LOCATION

You'll remember that you need to collect various metrics on BSE's securities sales to provide stock-transaction data analysis. You decide to first track sales of individual stocks, keeping a running total over 10-second windows to identify stocks that may trend up or down.

You'll use the following mapping to walk through the example, from reviewing the request to returning the response:

```
get("/window/:store/:key", (req, res) ->; ready ?
[CA]fetchFromWindowStore(req.params()) : STORES_NOT_ACCESSIBLE);
```

To help you keep your place in the query process, let's use figure 9.9 as a roadmap. You'll start by sending an HTTP get request `http://localhost:4567/window/NumberSharesPerPeriod/XXXX`, where `XXXX` represents the ticker symbol for a given stock (found in src/main/java/bbejeck/webserver/InteractiveQueryServer.java).

Listing 9.9 Mapping the request and checking for the key location

```
private String fetchFromWindowStore(Map<String, String>; params) {
   String store = params.get(STORE_PARAM);
   String key = params.get(KEY_PARAM);
   String fromStr = params.get(FROM_PARAM);
   String toStr = params.get(TO_PARAM);                          (1)

   HostInfo storeHostInfo = getHostInfo(store, key);             (2)

   if(storeHostInfo.host().equals("unknown")){                   (3)
      return STORES_NOT_ACCESSIBLE;
   }

   if(dataNotLocal(storeHostInfo)){                              (4)
      LOG.info("{} located in state store on another instance", key);
      return fetchRemote(storeHostInfo,"window", params);
   }
```

1. Extracts the request parameters
2. Gets the HostInfo for the key
3. If the hostname is "unknown", returns an appropriate message
4. Checks whether the returned hostname matches the host of this instance

The request is mapped to the `fetchFromWindowStore` method. The first step is to pull out the store name and key (stock symbol) from the request-parameters map. You fetch

the `HostInfo` object for the key in the request, and you use the hostname to determine whether the key is located on this instance or a remote one.

Next, you check whether the Kafka Streams instance is (re)initializing, which is indicated by the `host()` method returning `"unknown"`. If so, you stop processing the request and return a `"not accessible"` message.

Finally, you check whether the hostname matches the hostname for the current instance. If the hostname doesn't match, you get the data from the instance containing the key and return the results.

Next, let's look at how you retrieve and format the results (found in src/main/java/bbejeck/webserver/InteractiveQueryServer.java).

Listing 9.10 Retrieving and formatting the results

```
Instant instant = Instant.now();

long now = instant.toEpochMilli();                                    (1)
long from =  fromStr !=

[CA]null ? Long.parseLong(fromStr) : now - 60000;                     (2)

long to =  toStr != null ? Long.parseLong(toStr) : now;               (3)

List<Integer>; results = new ArrayList<>;();

ReadOnlyWindowStore<String, Integer>; readOnlyWindowStore =
[CA]kafkaStreams.store(store,

[CA]QueryableStoreTypes.windowStore());                               (4)
try(WindowStoreIterator<Integer>; iterator =

[CA]readOnlyWindowStore.fetch(key, from , to)){                       (5)
   while (iterator.hasNext()) {

     results.add(iterator.next().value);                              (6)
   }
 }

return gson.toJson(results);                                          (7)
```

(1) 0-1)) Gets the current time in milliseconds

(2) 0-2)) Sets the window segment start time or, if not provided, the time as of one minute ago

(3) 0-3)) Sets the window segment ending time or, if not provided, the current time

(4) 0-4)) Retrieves the ReadOnlyWindowStore

(5) 0-5)) Fetches the window segments

(6) 0-6)) Builds up the response

(7) 0-7)) Converts the results to JSON and returns to the requestor

I mentioned earlier that you'll cheat on the window store query if the *from* and *to* parameters aren't provided in the query. If the user doesn't specify a range, by default you return the last minute of results from the window store. Because you've defined a window of 10 seconds, you'll return six-windowed results. After you fetch the window segments from the store, you iterate over them, building a response that indicates the number of shares purchased for each 10-second interval over the last minute.

RUNNING THE INTERACTIVE QUERY EXAMPLE

To observe the results of this example, you need to run three commands:

- `./gradlew runProducerInteractiveQueries` produces the data needed for the examples.
- `./gradlew runInteractiveQueryApplicationOne` starts a Kafka Streams application with `HostInfo` using port 4567.
- `./gradlew runInteractiveQueryApplicationTwo` starts a Kafka Streams application with `HostInfo` using port 4568.

Then, point your browser to http://localhost:4568/window/NumberSharesPerPeriod/AEBB. Click Refresh a few time to see different results. Here's a static list of company symbols for this example: `AEBB, VABC, ALBC, EABC, BWBC, BNBC, MASH, BARX, WNBC, WKRP`.

RUNNING A DASHBOARD APPLICATION FOR INTERACTIVE QUERIES

A better example is a mini-dashboard web application that updates automatically (via Ajax) and displays the results from four different Kafka Streams aggregation operations. By running the commands listed in the previous subsection, you have everything set up; point your browser to localhost:4568/iq or localhost:4567/iq to run the dashboard application. By going to either instance, you'll see how Kafka Stream's interactive queries handle getting results from all instances with the same application ID. Look in the README file in the source code for full instructions on how to set up and start the dashboard application.

As you can see from observing the web application, you can view live results of the stream in a dashboard-like application. Previously, this type of application required a relational database; but here, Kafka Streams provides the information as needed.

We've wrapped up our coverage of interactive queries. Let's move on to KSQL: an exciting new tool that Confluent (the company founded by the original developers of Kafka at LinkedIn) recently released, which allows you to specify long-running queries against records streaming into Kafka without code, but using SQL.

9.3 KSQL

Imagine you're working with business analysts at BSE. The analysts are interested in your ability to quickly write applications in Kafka Streams to perform real-time data analysis. This interest puts you in a bind.

You want to work with the analysts and write applications for their requests, but you also have your normal workload—the additional work makes it hard to keep up with everything. The analysts understand the added work they're creating, but they can't write code, so they depend on you to write their analytics.

The analysts are experts on working with relational databases and thus are comfortable with SQL queries. If there were some way to give the analysts a SQL layer over Kafka Streams, everyone's productivity would increase. Well, now there is.

In August 2017, Confluent unveiled a powerful new tool for stream processing: KSQL (https://github.com/confluentinc/ksql#-ksql). KSQL is a streaming SQL engine for Apache Kafka, providing an interactive SQL interface that you can use to write powerful stream-processing queries *without* writing code. KSQL is especially adept at fraud detection and real-time applications.

> **NOTE** KSQL is a big topic and could take a chapter or two if not an entire book on its own. So, the coverage here will be concise. Fortunately, you've already learned the core concepts underpinning KSQL, because it uses Kafka Streams under the covers. For more information, see the KSQL documentation (http://mng.bz/zw3F).

KSQL provides scalable, distributed stream processing, including aggregations, joins, windowing, and more. Additionally, unlike SQL run against a database or a batch-processing system, the results of a KSQL query are *continuous*. Before we dive into writing streaming queries, let's take a minute to review some fundamental concepts of KSQL.

9.3.1 KSQL streams and tables

Section 5.1.3 discussed the concept of an *event stream* versus an *update stream*. An event stream is an unbounded stream of individual *independent* events, where an update or record stream is a stream of updates to previous records with the same key.

KSQL has a similar concept of querying from a `Stream` or a `Table`. A `Stream` is an

infinite series of immutable events or facts, but with a query on a `Table`, the facts are updatable or can even be deleted.

Although some of the terminology is different, the concepts are pretty much the same. If you're comfortable with Kafka Streams, you'll feel right at home with KSQL.

9.3.2 KSQL architecture

KSQL uses Kafka Streams under the covers to build and fetch the results of queries. KSQL is made up of two components: a CLI and a server. Users of standard SQL tools such as MySQL, Oracle, and even Hive will feel right at home with the CLI when writing queries in KSQL. Best of all, KSQL is open source (Apache 2.0 licensed).

The CLI is also the client connecting to the KSQL server. The KSQL server is responsible for processing the queries and retrieving data from Kafka as well as writing results into Kafka.

KSQL runs in two modes: *standalone*, which is useful for prototyping and development; and *distributed*, which of course is how you'd use KSQL when working in a more realistic-size data environment. Figure 9.10 shows how KSQL works in local mode. As you can see, the KSQL CLI, REST server, and KSQL engine are all located on the same JVM, which is ideal when running on your laptop.

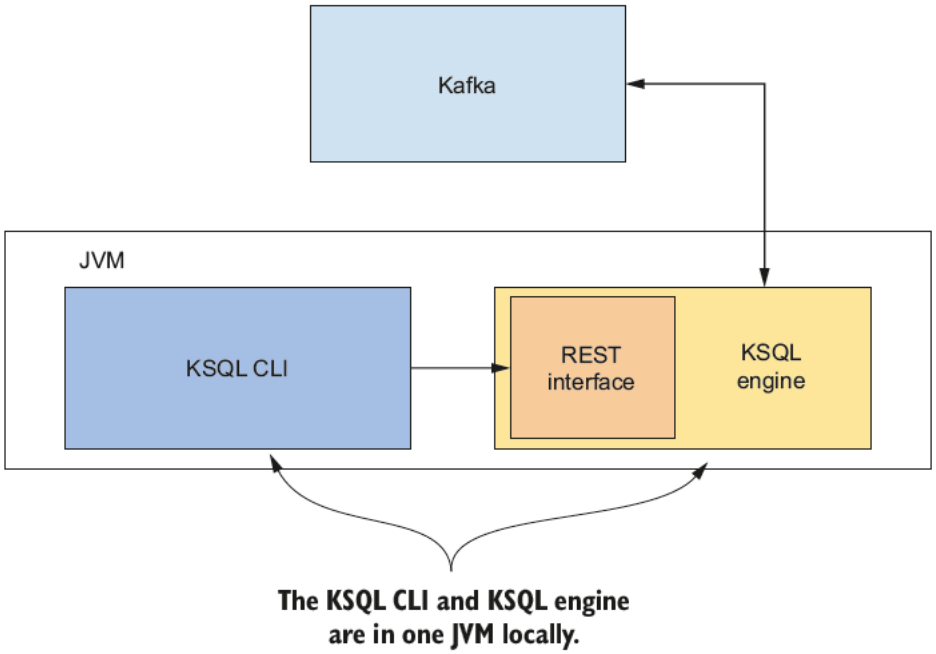


Figure 9.10 KSQL in local mode

Now, let's look at KSQL in distributed mode; see figure 9.11. The KSQL CLI is by itself,

and it will connect to one of the remote KSQL servers (we'll cover starting and connections in the next section). A key point is that although you only explicitly connect to one of the remote KSQL servers, all servers pointing to the same Kafka cluster will share in the workload of the submitted query.

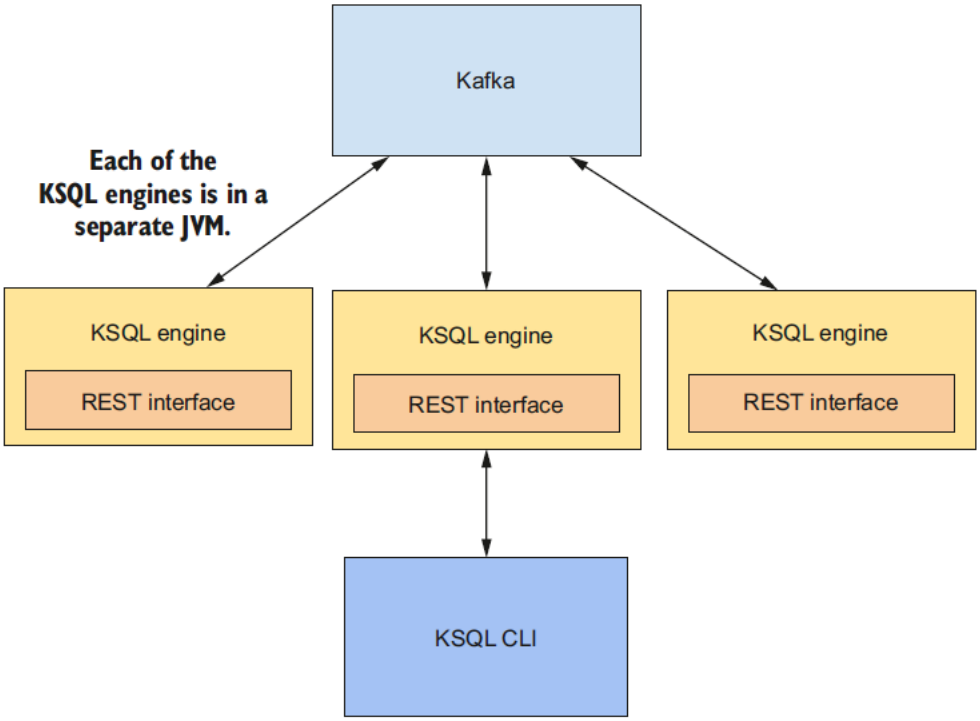


Figure 9.11 KSQL in distributed mode

Note that the KSQL servers are using Kafka Streams to execute the queries. This means that if you need more processing power, you can stand up another KSQL server, even during live operations (just like you can spin up another Kafka Streams application). The opposite case works just as well: if you have excess capacity, you can stop any number of KSQL servers, with the assumption that you'll leave at least one server operational. Otherwise, your queries will stop running!

Next, let's see how you get KSQL installed and running.

9.3.3 Installing and running KSQL

To install KSQL, you'll clone the KSQL repo with the command `git clone git@github.com:confluentinc/ksql.git` and then `cd` into the ksql directory and execute `mvn clean package` to build the entire KSQL project. If you don't have `git` installed or don't want to build from source, you can download the KSQL release from http://mng.bz/765U.

TIP KSQL is an Apache Maven–based project, so you'll need Maven installed to build KSQL. If you don't have Maven installed and you're on a Mac and have Homebrew installed, run `brew install maven`. Otherwise, you can head over to https://maven.apache.org/download.cgi and download Maven directly; installation instructions are at https://maven.apache.org/install.html.

Make sure you're in the base directory of the KSQL project before going any further. The next step is to start KSQL in local mode:

```
./bin/ksql-cli local
```

Note that you'll be using KSQL in local mode for all the examples, but we'll still cover how to run KSQL in distributed mode.

After running the previous command, you should see something like figure 9.12 in your console. Congratulations—you've successfully installed and launched KSQL! Next, let's start writing some queries.

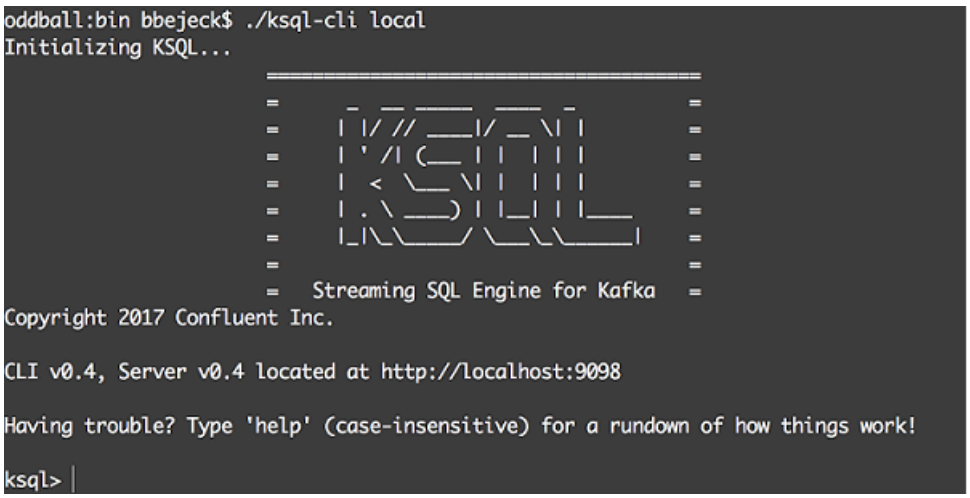


Figure 9.12 KSQL successful launch result

9.3.4 Creating a KSQL stream

Getting back to your work at BSE, you've been approached by an analyst who is interested in one of the applications you've written and would like to make some tweaks to it. But instead of this request resulting in more work, you spin up a KSQL console and turn the analyst loose to reconstruct your application as a SQL statement!

The example you're going to convert is the last windowed stream from the interactive queries example found in

src/main/java/bbejeck/chapter_9/StockPerformanceInteractiveQueryApplication.java, lines 96-103. In that application, you track the number of shares sold every 10 seconds, by company ticker symbol.

You already have the topic defined (the topic maps to a database table) and a model object, `StockTransaction`, where the fields on the object map to columns in a table. Even though the topic is defined, you need to register this information with KSQL by using a `CREATE STREAM` statement in src/main/resources/ksql/create_stream.txt.

Listing 9.11 Creating a stream

```
CREATE STREAM stock_txn_stream (symbol VARCHAR, sector VARCHAR, \     (1)
   industry VARCHAR, shares BIGINT, sharePrice DOUBLE, \              (2)
   customerId VARCHAR, transactionTimestamp STRING, purchase BOOLEAN) \
   WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'stock-transactions');  (3)
```

(1) CREATE STREAM statement to create a stream named stock_txn_stream

(2) Registers the fields of the StockTransaction object as columns

(3) Specifies the data format and the Kafka topic serving as the source of the stream (both required parameters)

With this one statement, you create a KSQL Streams instance that you can issue queries against. The `WITH` clause has two required parameters: `VALUE_FORMAT`, telling KSQL the format of the data, and `KAFKA_TOPIC`, telling KSQL where to pull the data from. There are two additional parameters you can use in the `WITH` clause when creating a stream. The first is `TIMESTAMP`, which associates the message timestamp with a column in the KSQL stream. Operations requiring a timestamp, such as windowing, will use this column to process the record. The other parameter is `KEY`, which associates the key of the message with a column on the defined stream. In this case, the message key for the `stock-transactions` topic matches the `symbol` field in the JSON value, so you don't need to specify the key. Had this not been the case, you would have needed to map the key to a named column, because you always need a key to perform grouping operations. You'll see this when you execute the stream SQL.

TIP The KSQL command `list topics;` shows a list of topics on the broker the KSQL CLI is pointing to and whether the topics are registered.

You can view all streams and verify that KSQL created the new stream as expected with the following commands:

```
show streams;
describe stock_txn_stream;
```

The results are shown in figure 9.13. Notice that KSQL inserted two extra columns: `ROWTIME` and `ROWKEY`. The `ROWTIME` column is the timestamp placed on the message (either from the producer or by the broker), and `ROWKEY` is the key (if any) of the message.

```
ksql> show streams;

 Stream Name      | Kafka Topic        | Format
-----------------------------------------------
 STOCK_TXN_STREAM | stock-transactions | JSON
ksql> describe stock_txn_stream;

 Field                | Type
-----------------------------------------
 ROWTIME              | BIGINT
 ROWKEY               | VARCHAR(STRING)
 SYMBOL               | VARCHAR(STRING)
 SECTOR               | VARCHAR(STRING)
 INDUSTRY             | VARCHAR(STRING)
 SHARES               | BIGINT
 SHAREPRICE           | DOUBLE
 CUSTOMERID           | VARCHAR(STRING)
 TRANSACTIONTIMESTAMP | VARCHAR(STRING)
 PURCHASE             | BOOLEAN
```

Figure 9.13 Listing streams, and describing your newly created stream

Now, let's run the query on this stream.

> **NOTE** You'll need to run `./gradlew runProducerInteractiveQueries` to provide data for the KSQL examples

9.3.5 Writing a KSQL query

The SQL query for performing the stock analysis is as follows:

```
SELECT symbol, sum(shares) FROM stock_txn_stream
[CA]WINDOW TUMBLING (SIZE 10 SECONDS) GROUP BY symbol;
```

Run this query, and you'll see results similar to those shown in figure 9.14. The column on the left is the ticker symbol, and the number is the number of shares traded for that

symbol over the last 10 seconds. With this query, you specify a tumbling window of 10 seconds, but KSQL supports session and hopping windows, as well, as we discussed in section 5.3.2.

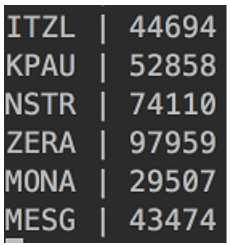


Figure 9.14 Results of the tumbling window query

You've built a streaming application without writing any code—quite an achievement. For a comparison, let's look at the corresponding application written in the Kafka Streams API.

Listing 9.12 Stock analysis application written in Kafka Streams

```
KStream<String, StockTransaction>; stockTransactionKStream =
[CA]builder.stream(MockDataProducer.STOCK_TRANSACTIONS_TOPIC,
                   Consumed.with(stringSerde, stockTransactionSerde)
             .withOffsetResetPolicy(Topology.AutoOffsetReset.EARLIEST));

Aggregator<String, StockTransaction, Integer>; sharesAggregator =
[CA](k, v, i) ->; v.getShares() + i;

stockTransactionKStream.groupByKey()
                .windowedBy(TimeWindows.of(10000))
                .aggregate(() ->; 0, sharesAggregator,
                        Materialized.<String, Integer,
                        WindowStore<Bytes,
                        byte[]>;>;as("NumberSharesPerPeriod")
                                .withKeySerde(stringSerde)
                                .withValueSerde(Serdes.Integer()))
                .toStream().
[CA]peek((k,v)->;LOG.info("key is {} value is{}", k, v));
```

Even though the Kafka Streams API is concise, the equivalent you wrote in KSQL is *one* -liner query.

Before we wrap up our coverage of KSQL, let's discuss some additional features of KSQL.

9.3.6 Creating a KSQL table

So far, we've demonstrated creating a KSQL stream. Now, let's see how to create a KSQL table, using the `stock-transactions` topic as the source, for familiarity (found in src/main/resources/ksql/create_table.txt).

Listing 9.13 Creating a KSQL table

```
CREATE TABLE stock_txn_table (symbol VARCHAR, sector VARCHAR, \
                              industry VARCHAR, shares BIGINT, \
                              sharePrice DOUBLE, \
                              customerId VARCHAR, transactionTimestamp \
                              STRING, purchase BOOLEAN) \
                              WITH (KEY='symbol', VALUE_FORMAT = 'JSON', \
                              KAFKA_TOPIC = 'stock-transactions');
```

Once you've created the table, you can execute queries against it. Keep in mind that the table will contain updates for each transaction by `symbol`, because the `stock-transactions` topic is keyed by the ticker symbol.

A useful experiment is to pick a ticker symbol from the streaming stock-performance query, and then run the following queries in the KSQL console, and notice the difference in output:

```
select * from stock_txn_stream  where symbol='CCLU';
select * from stock_txn_table where symbol='CCLU';
```

The first query produces several results, because it's a stream of individual events. But the table query returns far fewer results (one record, when I ran the experiment). These results are the expected behavior, because a table represents updates to facts, whereas a stream represents a series of unbounded events.

9.3.7 Configuring KSQL

KSQL offers the familiar SQL syntax and the ability to write powerful streaming applications quickly, but you may have noticed the lack of configuration. This is not to say you can't configure KSQL. You're free to override any settings as needed, and any of the stream, consumer, and producer configs that you can set for a Kafka Streams application are available. To view the properties that are currently set, run the `show properties;` command.

As an example of setting a property, here's how you can change `auto.offset.reset` to `earliest`:

```
SET 'auto.offset.reset'='earliest';
```

This is the approach you use to set any property in the KSQL shell. But if you need to set several configurations, typing each one into the console isn't convenient. Instead, you can specify a configuration file on startup:

```
./bin/ksql-cli local --properties-file /path/to/configs.properties
```

This has been a quick tour of KSQL, but I hope you can see the power and flexibility it gives you for creating streaming applications on Kafka.

9.4 Summary

- By using Kafka Connect, you can incorporate other data sources into your Kafka Streams applications.
- Interactive queries are a potent tool: they allow you to see data in a stream as it flows through your Kafka Streams application, without the need for a relational database.
- The KSQL language lets you quickly build powerful streaming applications without code. KSQL promises to deliver the power and flexibility of Kafka Streams to workers who aren’t developers.

A Additional configuration information

This appendix covers common and not-so-common configuration options for a Kafka Streams application. During the course of the book, you've seen several examples of configuring a Kafka Streams application, but the configurations usually included only the required (application ID, bootstrap servers) and a handful of other configs (key and value serdes). In this appendix, I'll show you some other settings that, although not required, will help you keep your Kafka Streams applications running smoothly. These options will be presented in somewhat of a cookbook fashion.

A.1 Limiting the number of rebalances on startup

When starting up a Kafka Streams application, if you have multiple instances, the first instance gets all the topic partitions assigned from the `GroupCoordinator` on the broker. If you start another instance, a rebalance occurs, removing current `TopicPartition` assignments and reassigning all `TopicPartitions` across both Kafka Streams instances. This process is repeated until you've started all Kafka Streams applications that have the same application ID.

This is normal operation for a Kafka Streams application. But during a rebalance, processing of records is paused until the rebalance is completed; thus, you'd like to limit the number of rebalances when starting up, if possible.

With the release of Kafka 0.11.0, a new broker configuration, `group.initial.rebalance.delay.ms`, was introduced. This configuration delays the initial consumer rebalance from the `GroupCoordinator` when a new consumer joins the group by the amount specified in the `group.initial.rebalance.delay.ms` configuration. The default setting is 3 seconds. As other consumers join the group, the rebalance is continually delayed by the configured amount (up to a limit of `max.poll.interval.ms`). This benefits Kafka Streams because as you start new

instances, the rebalance is delayed until all instances have come online (assuming you're starting them up one after another). For example, if you start four instances with the appropriate rebalance-delay setting, you should have only one rebalance after all four instances come online—meaning you'll start processing data more quickly.

A.2 Resilience to broker outages

To keep your Kafka Streams application resilient in the face of broker failures, here are some recommended settings (see listing A.1):

- Set `Producer.NUM_RETRIES` to `Integer.MAX_VALUE`.
- Set `Producer.REQUEST_TIMEOUT` to `305000` (5 minutes).
- Set `Producer.BLOCK_MS_CONFIG` to `Integer.MAX_VALUE`.
- Set `Consumer.MAX_POLL_CONFIG` to `Integer.MAX_VALUE`.

Listing A.1 Setting properties for resilience to broker outages

```
Properties props = new Properties();
props.put(StreamsConfig.producerPrefix(
[CA]ProducerConfig.RETRIES_CONFIG), Integer.MAX_VALUE);
props.put(StreamsConfig.producerPrefix(
[CA]ProducerConfig.MAX_BLOCK_MS_CONFIG), Integer.MAX_VALUE);
props.put(StreamsConfig.REQUEST_TIMEOUT_MS_CONFIG, 305000);
props.put(StreamsConfig.consumerPrefix(
[CA]ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG), Integer.MAX_VALUE);
```

Setting these values should help ensure that if all brokers in the Kafka cluster go down, your Kafka Streams application will stay up and be ready to resume working once the brokers are back online.

A.3 Handling deserialization errors

Kafka works with byte arrays for keys and values, and you need to deserialize the keys and values to work with them. This is why you need to provide serdes for all source and sink processors. It wouldn't be unexpected to have some malformed data during record processing. Kafka Streams provides the configurations `default.deserialization.exception.handler` and `StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG` to specify how you want to handle these deserialization errors.

The default setting is `org.apache.kafka.streams.errors.LogAndFailExceptionHandler`, which, as the name implies, logs the error. Your Kafka Streams application instance will fail (shut down) due to this deserialization exception. Another class,

`org.apache.kafka.streams.errors.LogAndContinueExceptionHandler`, logs the error, but your Kafka Streams application will continue to run.

You can implement your own deserialization exception handler by creating a class implementing the `DeserializationExceptionHandler` interface.

Listing A.2 Setting a deserialization handler

```
Properties props = new Properties();
props.put(StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_
[CA]CONFIG, LogAndContinueExceptionHandler.class);
```

I only show how to set the `LogAndContinueExceptionHandler` handler, because the log-and-fail version is the default setting.

A.4 Scaling up your application

In all the examples in the book, the Kafka Streams applications run with one stream thread. That's fine for development, but in practice you'll most likely need to run with more than one stream thread. The question is how many threads and how many Kafka Streams instances to use. There are no concrete answers, because only you know your circumstances well enough to address those questions, but we can go over some basic calculations to give you a good idea.

You'll remember from chapter 3 that Kafka Streams creates a `StreamsTask` per partition of the input topic(s). For our first example, we'll consider a single input topic with 12 partitions, to keep the discussion straightforward.

With 12 input partitions, Kafka Streams creates 12 tasks. For the moment, let's assume you want to have 1 task per thread. You could have 1 instance with 12 threads, but that approach has a drawback: if the machine hosting your Kafka Streams application were to go down, all stream processing would stop.

But if you start instances with 4 threads each, then each instance will process 4 input partitions. The benefit of this approach is that if one of the Kafka Streams instances goes down, a rebalance will be triggered, and the 4 tasks from the non-running instance will be assigned to the other 2 instances; thus, the remaining applications will process 6 tasks each. Additionally, when the stopped instance resumes running, another rebalance will occur, and all 3 instances will go back to processing 4 tasks.

One important consideration is that when determining the number of tasks to create,

Kafka Streams takes the maximum number of partitions from *all* input topics. If you have 1 topic with 12 partitions, you end up with 12 tasks; but if the number of source topics is 4, with 3 partitions each, you'll have 3 tasks, each of which is responsible for processing 4 partitions.

Keep in mind that any stream threads beyond the number of tasks will be idle. Going back to the example of 3 Kafka Streams instances, if you stand up a fourth instance of 4 threads, then after rebalancing you'll have 4 idle stream threads among your applications (16 threads, but only 12 tasks).

This is a key component of Kafka Streams that I mentioned earlier in the book. This dynamic scaling up or down doesn't involve taking your application offline—it happens automatically. This feature is helpful because if you have an uneven flow of data into the application, you can spin up additional instances to handle the load and then take some offline when the volume drops off.

Do you always want a single thread per task? Maybe, but it's hard to say, because it depends on the demands of your application.

A.5 RocksDB configuration

For stateful operations, Kafka Streams uses RocksDB (http://rocksdb.org) under the covers as the persistence mechanism. RocksDB is a fast, highly configurable key/value store. There are too many options to make specific recommendations here, but Kafka Streams provides a way to override the default settings with the `RocksDBConfigSetter` interface.

To set custom RocksDB settings, create a class implementing the `RocksDBConfigSetter` interface, and then provide the class name when configuring your Kafka Streams application via the `StreamsConfig.ROCKSDB_CONFIG_SETTER_CLASS_CONFIG` setting. To get an idea of what you can adjust with RocksDB, I encourage you to read the RocksDB Tuning Guide at http://mng.bz/I88k.

A.6 Creating repartitioning topics ahead of time

In Kafka Streams, any time you perform an operation that may potentially change the `map` key—`transform` or `groupBy`, for example—an internal flag is set in the `StreamsBuilder` class, indicating that repartitioning will be required. Now, performing a `map` or `transform` won't automatically force the creation of a repartitioning topic and the repartitioning operation; but as soon as you add an operation using the updated key, a repartitioning operation will be triggered.

Although this is a required step (covered in chapter 4), in some cases, it's better to repartition the data yourself ahead of time. Consider the following (abbreviated) example:

```
KStream<String, String>; mappedStream =
[CA]streamsBuilder.stream("inputTopic").map(....);                    (1)
KTable<Windowed<String>;, Long>; ktable1 =
[CA]mappedStream.groupByKey().windowedBy...count()                    (2)
KTable<Windowed<String>;, Long>; ktable2 =
[CA]mappedStream.groupByKey().windowedBy...count()                    (3)
KTable<Windowed<String>;, Long>; ktable3 =
[CA]mappedStream.groupByKey().windowedBy...count()                    (4)
```

(1) Maps the original input stream to create a new key

(2) Windowed count option 1

(3) Windowed count option 2

(4) Windowed count option 3

Here, you map the original stream to create a new key to group by. You want to perform three counts with three different windowing options—a legitimate use case. But because you mapped to a new key, *each* windowed count operation creates a new repartition topic. Again, the need for a repartition topic makes sense due to the changed key, but having three repartition topics duplicates data when you need only one repartition topic.

The solution to this issue is simple: after your `map` call, you immediately use a `through` operation to partition the data. Then, the subsequent `groupByKey` calls won't trigger repartitioning, because the `groupByKey` operator *does not* set the repartition-needed flag. Here's the revised code:

```
KStream<String, String>; mappedStream =
[CA]streamsBuilder.stream("inputTopic").map(....).through(...);       (1)
```

❶ Maps the original input stream to create a new key, and repartitions

By adding the `through` processor and repartitioning manually, you have one repartition topic instead of three.

A.7 Configuring internal topics

When building a topology, depending on the processors you add, Kafka Streams may create several internal topics. These internal topics can be changelogs for backing up state stores, or repartition topics. Depending on your data volume, these internal topics can consume a large amount of space. Additionally, even though changelog topics are by default created with a cleanup policy of `"compact"`, if you have many unique keys, these compacted topics can grow in size. With this in mind, it's a good idea to configure your internal topics to keep their size manageable.

You have two options for managing internal topics. First, you can provide configs directly when creating state stores, using either `StoreBuilder.withLoggingEnabled` or `Materialized.withLoggingEnabled`. Which method you use depends on how you create the state store. Both methods take a `Map<String, String>;` containing the topic properties. You can see an example in src/main/java/bbejeck/chapter_7/CoGroupingListeningExampleApplication.

The other option for managing internal topics is to provide configurations for them when configuring your Kafka Streams application:

```
Properties props = new Properties();
// other properties set here
props.put(StreamsConfig.topicPrefix("retention.bytes"), 1024 * 1024);
props.put(StreamsConfig.topicPrefix("retention.ms"), 3600000);
```

When using the `StreamsConfig.topicPrefix` approach, the provided settings are applied globally to all internal topics. Any topic settings provided when creating a state store will take precedence over the settings provided with `StreamsConfig`.

I can't give you much advice regarding what settings to use, because that depends on your particular use case. But keep in mind that the default size of a topic is unlimited and the default retention time is one week, so you should adjust the `retention.bytes` and `retention.ms` settings. In addition, for changelogs backing state stores with many unique keys, you can set `cleanup.policy` to `compact,delete` to ensure that the topic size stays manageable.

A.8 Reseting your Kafka Streams application

At some point, you may need to start a Kafka Streams application over and reprocess data, either in development or after a code update. To do this, Kafka Streams provides a `kafka-streams-application-reset.sh` script in the bin directory of the Kafka installation.

The script has one *required* parameter: the application ID of the Kafka Streams application. The script offers several options, but in a nutshell, it can reset input topics to the earliest available offset, reset intermediate topics to the latest offset, and delete any internal topics. Note that you'll need to call `KafkaStreams.cleanUp` the next time you start your application, to delete any local state from previous runs.

A.9 Cleaning up local state

Chapter 4 discussed how Kafka Streams stores local state per task on the local filesystem. During development or testing, or when migrating to a new instance, you may want to clean out all previous local state.

To clean up any previous state, you can use `KafkaStreams.cleanUp` either before you call `KafkaStreams.start` or after `KafkaStreams.stop`. Using the `cleanUp` method at any other time will result in an error.

B Exactly once semantics

Kafka achieved a major milestone with the release of version 0.11.0: *exactly once semantics*. Prior to this release of Kafka, the delivery semantics of Kafka could have been described as *at-least-once* or *at-most-once*, depending on the producer.

In the case of at-least-once delivery, a broker can persist a message but experience an error before sending the acknowledgment back to the producer, assuming the producer is configured with `asks="all"` and times out waiting for the acknowledgment. If the producer is configured with retries greater than zero, it will resend the message, unaware that the previous message was successfully persisted. In this scenario (although rare), a duplicate message is delivered to consumers—hence, the phrase *at least once*.

For the at-most-once condition, consider the case where a producer is configured with retries set to zero. In the previous example, the message in question would be delivered only once, because there are no retries. But if the broker experiences an error before it can persist the message, the message won't be sent. In this case, you've traded receiving all messages for not receiving any duplicate messages.

With exactly once semantics, even in situations where a producer resends a message that was previously persisted to a topic, consumers will receive the message exactly once. To enable transactions or exactly once processing with a `KafkaProducer`, you add a configuration `transactional.id` and a couple of method calls, as shown in the following example. Otherwise, producing messages with transactions looks familiar. Note that this excerpt doesn't stand alone—it's provided to highlight what's required to produce and consume messages with the transactional API:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "transactional-id");
```

```
Producer<String, String>; producer =
[CA]new KafkaProducer<>;(props, new StringSerializer(), new StringSerializer());

producer.initTransactions();  (1)


try {
      // called right before sending any records
      producer.beginTransaction();

     ...sending some messages

     // when done sending, commit the transaction
     producer.commitTransaction();

 } catch (ProducerFencedException | OutOfOrderSequenceException |
 [CA]AuthorizationException e) {

      producer.close();  (2)
 } catch (KafkaException e) {

     producer.abortTransaction();  (3)
 }
```

1. When setting transactional.id, you need to call this method before any others.
2. The only option for any of the non-recoverable exceptions is to close the producer.
3. For any other exception, abort and retry.

To use `KafkaConsumer` with transactions, you need to add only one configuration:

```
props.put("isolation.level", "read_committed");
```

In `read_committed` mode, `KafkaConsumer` only reads successfully committed transactional messages. The default setting is `read_uncommitted`, which returns all messages. Non-transactional messages are always retrieved in either configuration setting.

The impact of exactly once semantics is a big win for Kafka Streams. With exactly once, or transactions, you're guaranteed to process records through a topology exactly once.

To enable exactly once processing with Kafka Streams, set `StreamsConfig.PROCESSING_GUARANTEE_CONFIG` to `exactly_once`. The default setting for `PROCESSING_GUARANTEE_CONFIG` is `at_least_once`, or non-transactional processing. With that simple configuration setting, Kafka Streams handles all required steps for performing transactional processing.

This has been a quick overview of Kafka's transactional API. For more information,

check out the following resources:

- Dylan Scott, *Kafka in Action* (Manning, forthcoming), www.manning.com/books/kafka-in-action
- Neha Narkhede, “Exactly-once Semantics Are Possible: Here’s How Kafka Does It,” *Confluent*, June 30, 2017, http://mng.bz/t9rO
- Apurva Mehta and Jason Gustafson, “Transactions in Apache Kafka,” *Confluent*, November 17, 2017, http://mng.bz/YKqf
- Guozhang Wang, “Enabling Exactly-Once in Kafka Streams,” *Confluent*, December 13, 2017, http://mng.bz/2A32