# Kafka Streams IN ACTION

Bill Bejeck

MEAP

MANNING

**MEAP Edition**
**Manning Early Access Program**
# Kafka Streams in Action
**Version 4**



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 is what I use to improve the data flow downstream, and 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 data flow.

Please remember, these chapters are still works in progress, and will definitely get more polished by the time the book is complete. And 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

# 1 Welcome to Kafka Streams

In this chapter

- The "Big Data" movement and how it changed the programming landscape
- The Need for Stream Processing
- How Stream Processing works
- Introducing Kafka Streams
- What Problems Kafka Streams solves

## 1.1 The "Big Data" movement and how it changed the programming landscape

If you take a look around at the modern programming landscape, it has exploded with 'Big Data' frameworks and technologies. Sure client side development has undergone transformations of its own and mobile device applications have exploded as well. But no matter how big the mobile device market gets or how client side technologies evolve, there is one constant; we need to process more and more data every day. As the amount of data grows, the need to analyze and leverage the benefit from that data grows at the same rate.

But having the ability to process large quantities of data in bulk amounts (batch processing) is not always enough. Increasingly, organizations are finding there is a need to process data as it is 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. By per-event processing, I mean the ability to work on data as it arrives. Per-event processing means you process each single record as soon as it is available; there is no grouping data together in small batches, no 'micro-batching,' required.

**SIDEBAR** **What is Micro Batching?**

When the need for processing data as it arrives became more and more apparent a new strategy developed, that of 'micro' batching. As the name implies micro-batching is nothing more that batch processing, but in smaller quantities of data. In some cases, by reducing the size of the 'batch' one can indeed process results quicker, but micro-batching is still batch processing, although at faster intervals, and does not give you real per-event processing

In this book, I'll teach you how to use Kafka Streams to solve your streaming application needs. From basic ETL work, complex stateful transformations to joining records we'll cover the components of Kafka Streams to enable you to solve these kinds of challenges in your streaming applications.

Before we dive into Kafka Streams, I want to take a few minutes to explore the history of big data processing. I feel its useful to understand the history of big data processing, because as we explore the problems and solutions we'll clearly see how the need for Kafka and then Kafka Streams evolved. In the next several sections I will elaborate on my views and experiences of how the 'Big Data' era got started and what led us to the Kafka Streams solution.

### *1.1.1 The Genesis of Big Data*

While one could argue that the internet started back in the late 1960's to early 1970's it was mainly in the hands of the U.S Department of Defense and university researchers at the time. It wasn't until the mid-1990's that the internet started to have a real impact on our daily lives. 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.

There was an unexpected byproduct of all this connectivity. The Internet also has helped to generate massive amounts of data. But in my mind, I'm going to 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 works like this: an algorithm rates websites by counting the number and quality of links pointing to that site. The assumption is the more important or relevant a web page is, more sites will refer to it. Here is a graphical representation of the PageRank algorithm:

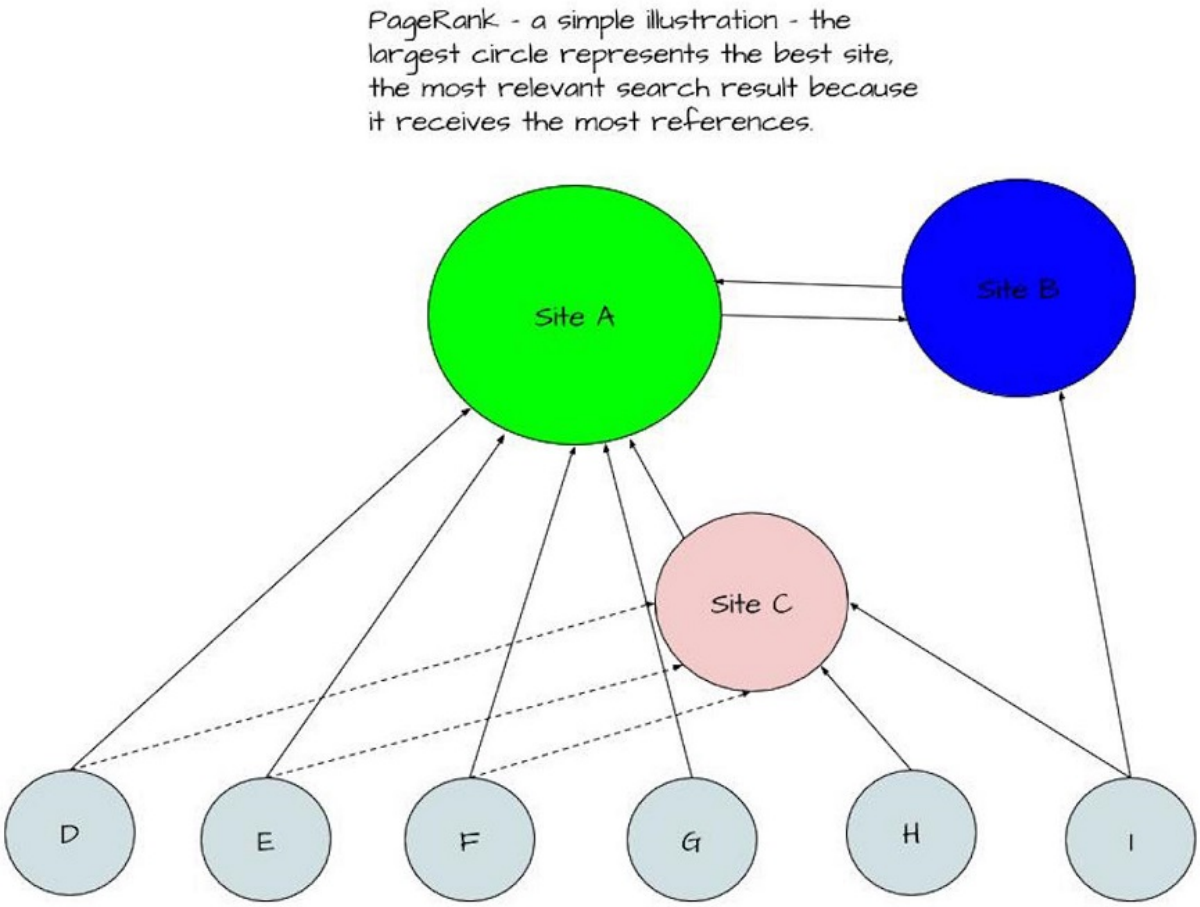



**Figure 1.1 The PageRank algorithm in action. The circles represent websites, and the larger ones represent sites having a high number of links pointing to it from other sites.**

Let's take a quick moment to explain the mass of arrows and circles represented in the graphic.

- Site A is the most important because it has the most references pointing to it.
- Site B is somewhat important, although it does not have as many references, it does have a site pointing 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 less.
- All the rest of the sites at the bottom (small circles D through I) have no references pointing to them. This makes them the least valuable.

While the image is an oversimplification of the PageRank algorithm, it does serve to give you the big picture of how it works.

### *1.1.2 How to Implement PageRank*

At the time this was a revolutionary approach because, until Google's approach, page rank 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 actual quality of the content. To perform the PageRank algorithm on *all internet content* the traditional approaches to working with data weren't going to be effective for Google. The traditional methods of working with data at the time would take too long, so another approach was required.

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, otherwise it would be game over for Google. What Google developed for processing all that data was another revolutionary approach, the MapReduce paradigm. Not only did MapReduce enables Google to do the work it needed to as a company; it inadvertently spawned an entire new industry in computing.

### 1.1.3 Important Concepts from Map Reduce

To be clear, map and reduce functions were not new concepts at the time. What was unique about Google's approach was to apply 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.

**Listing 1.1 Simple Map Function**

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

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

On the other hand, a "reduce" function takes N number of parameters and "reduces" them down into a singular, or at least smaller, value.

Taking a collection of numbers and adding them together is a good example. 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 first 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 reaching the last value, producing a single number. Here are the steps to reduce a `List<Integer>` containing the values 1,2,3:

**Listing 1.2 Reduce walkthrough**

```
0 + 1 = 1   (1)
1 + 2 = 3   (2)
3 + 3 = 6   (3)
```

(1) Adding seed value to the first number.

(2) Taking result from step one and adding to the second number in the list.

(3) Adding the sum of step 2 with the third and last number.

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

Now, let's take a look at implementing such a simple reduce function using a Java 8 lambda:

**Listing 1.3 Sample Reduce Function**

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

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

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

1. How to distribute data across a cluster to achieve scale in processing.
2. The use of key-value pairs and partitions to group distributed data together.
3. Instead of avoiding failure, you embrace failure by using replication.

Next, we are going to discuss these concepts in general terms, but pay attention, because as we go through the book, you'll see them coming up again.

### *1.1.4 Distributing data across a cluster to achieve scale in processing*

If you have 5TB (5000GB) of data, that could be an overwhelming amount of data to work with on one machine at one time. But if you can split the data up to more machines to the point where each server is processing a manageable amount, your problem of a large amount of data is minimized. Table 1 illustrates this point very clearly:

**Table 1.1 How Splitting Up 5TB Helps Processing**

| Number of Machines | Amount of Data Processed Per Server |
|---|---|
| 10 | 500GB |
| 100 | 50GB |
| 1000 | 5GB |
| 5000 | 1GB |

As you can see from the table, you start out with an unwieldy amount data to process. But by just spreading the load out to more servers, the processing of data is no longer an issue. By looking at the last entry in the table, 1GB of data is something a laptop could trivially handle. So the first key concept to understand from map-reduce is by spreading the load across a cluster of machines, we can make what was originally an imposing problem concerning the size of data, into a manageable one.

### 1.1.5 Using key-value pairs and partitions to group distributed data together

The key-value pair is a simple data structure with powerful implications. In the previous section, you saw how to process a massive amount of data; you spread that data over a cluster of machines. Distributing your data solves the processing problem, but now you have the problem of getting data back together during your processing. To address the issue of grouping distributed data you use the keys from the key-value pairs as a partition of the data. By using keys as a partition, grouping distributed data is now an easy problem to solve using the following formula:

**Listing 1.4 Determine the partition with a hashing function**

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

Here is a graphical depiction of how to apply a hashing function so you can group distributed data together on the same partition.

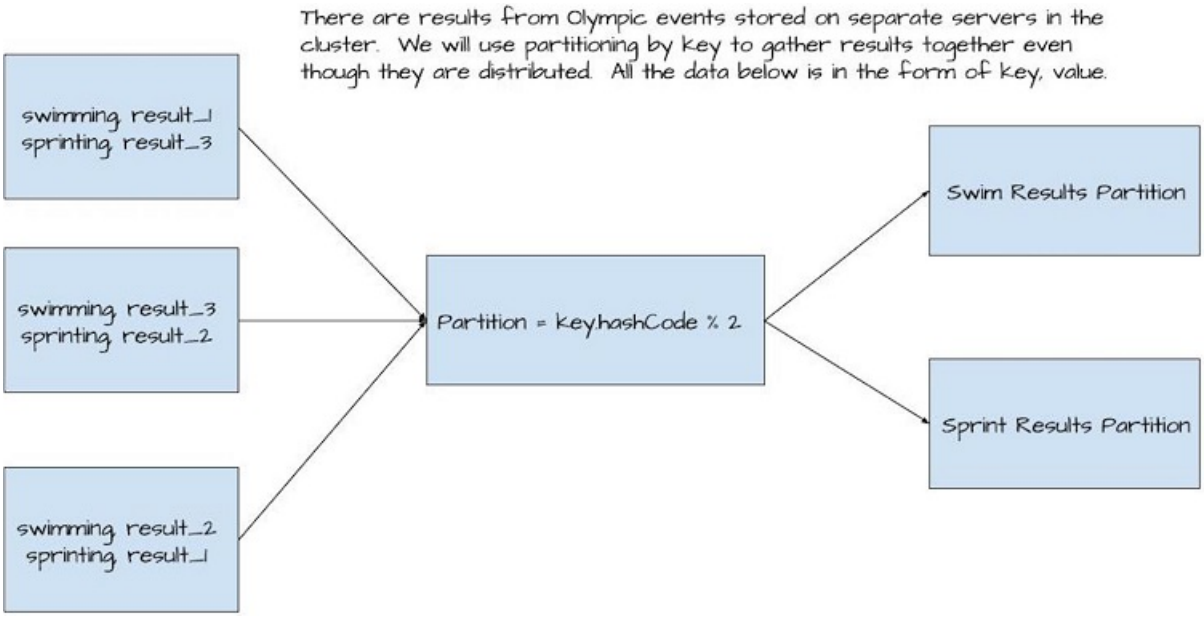



**Figure 1.2 Here we see grouping records by key on the same partition. Even though the records started out on separate servers, they arrive in the same location as a result of using partitions.**

Partitioning is an important concept, and we will see detailed examples in later chapters.

### 1.1.6 Embracing failure by using replication.

Another key component of Google's MapReduce was the Google File System (GFS). In conjunction with Hadoop being the open source implementation of MapReduce HDFS (Hadoop File System) was the open source implementation of GFS.

At a very high level, GFS/HDFS split data up into blocks and distributed those blocks of across the 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 using

replication. The blocks of data are replicated 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 in giving distributed applications fault tolerance, which is essential for a distributed application to be successful.

The key point of this section is the replication of data across a cluster gives you resilience in the face of the disk and machine failures. We'll see later on how partitions and replication work in Kafka Streams.

Now, let's take a look at how we evolved from batch processing to the need for stream processing (again based on my personal experiences and journey through programming).

### 1.1.7 Batch Processing is Not Enough

Hadoop caught on with the computing world like wildfire. Now you can process vast amounts of data, have fault tolerance while using commodity hardware (cost savings). But Hadoop/MapReduce is a batch-oriented process. Being batch oriented means, you collect large amounts of data, process it, then store the output for later use.

Batch processing is a perfect fit for something like PageRank; you can't make determinations of what is a valuable resource across the entire internet by watching user clicks in real-time. But business came under ever increasing pressure to respond to important questions more quickly. Questions like:

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

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

Every business, government or organization today has one thing in common, and that is the byproduct of conducting your business is the generation of data, lots, and lots of data. For example, there are estimates that a single flight of a commercial airplane can generate up to half a terabyte of data *per flight*. When one stops to consider the number of domestic flights every day in the United States alone, you can see this data volume amounts to a staggering amount of data to be analyzed.

For all that data to be of any value, people and or machines need to review it and provide assessments. All that data can be collected and used for making decisions to cut costs, improve the performance or any number of decision points.

Now that we have established the need for stream processing let's go on to define what it means.

## 1.2 Introducing Stream Processing

While there are varying definitions of stream processing, in this book we are going to define stream processing as working with data as it is arriving into your system. We can further refine the definition to say that stream processing is the ability to work with an infinite stream of data with continuous computation, as it flows, no need to collect or store the data to act on it. This simple diagram below represents a stream of data each circle on the line represents data at a point in time. Data is continuously flowing, as data in stream processing is unbounded, think of the number line we learned in school, you can stop to look at or work with a single number, but the line keeps going.

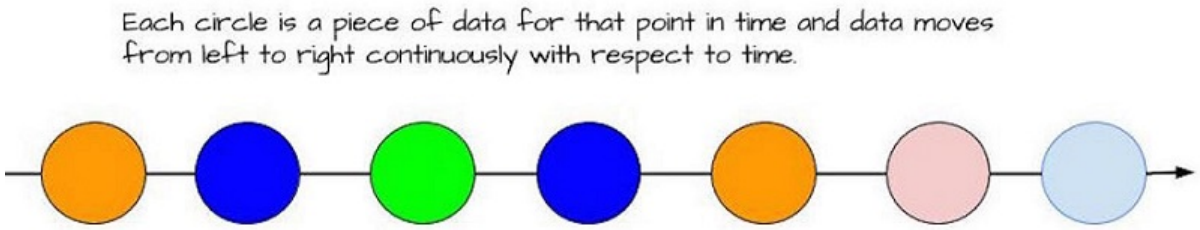



**Figure 1.3 This marble diagram is a simple representation of stream processing. Each circle represents some information or event occurring at a particular point in time. The number of events is unbounded and move continually from left to right just like a number line.**

So who need to use stream processing? Well anyone who needs feedback very quickly from any observable event. Let's discuss some examples below.

### 1.2.1 When to use Stream Processing and when not to use it

Like any technical solution, stream-processing is not a "one size fits all" solution. So let's look at some scenarios where stream processing is a fit first.

The need to quickly respond or report on incoming data is a good use case for stream-processing.

A good illustration is credit card fraud. The owner may not notice a stolen credit card, 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. Another good example would be intrusion detection into a network. While analyzing application log files after a breach has occurred may be helpful to prevent future attacks or where to improve security, the ability to monitor aberrant behavior in real-time is critical.

Imagine a large race such as the NYC Marathon. Almost all runners will have a chip on their shoe, and when runners go by sensors spread out across the course, you can use that information to track the position of runners. By using the sensor data, we can determine the leaders, spot potential cheating as well as detecting if a runner has potentially run into problems.

In terms of time sensitivity for data, the financial industry is probably at the top of the list. The ability to track market prices and direction in real time is essential for brokers or consumers to make effective decisions regarding when to sell or buy stocks or other

financial instruments.

On the other hand, stream-processing is not a solution for all problem domains, so let's take a quick look at where stream-processing might not be appropriate.

To effectively make forecasts of future behavior to predict the outcome of future events, one needs to use a large amount of data. Only by using vast quantities of data accumulated over time can one eliminate anomalies and start to identify patterns and trends of problem domain you are measuring.

Economic forecasting is a good example. Information is collected on many variables over an extended period in an attempt to make an accurate forecast, interest rates in the housing market for example. Another good example is assessing the effect of implementing a change in curriculum in schools. It's only after one or two testing cycles that school administrators can measure if the changes made are achieving their goals.

In this section, we have discussed areas where stream-processing is a good fit and some others where isn't appropriate.

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 candidate for you. On the contrary, if you need to perform in-depth analysis or are only compiling a large repository of data for later analysis a stream-processing approach may not be a fit.

Since this is a book about Kafka Streams, a stream-processing technology, our next step is to walk through a concrete example where we have a need to do stream-processing.

## 1.3 Handling a Purchase Transaction

Our first step is to apply a general stream-processing approach to an example from the world of retail sales. After we have walked through the general solution, we are going to show how we can use Kafka Streams specifically to implement the stream processing application.

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

Jane has no cash (does anyone use cash anymore?) so when the total is rung up Jane hands the cashier her debit card. The cashier swipes the card and gives Jane the receipt, and she leaves. As Jane is walking out of the store, she checks her email, and there is a message from Zmart thanking her for her patronage with various coupons attached for discounts on Jane's next visit.

While the transaction above is a normal occurrence that a customer would not give a second thought about, a while back you recognized it for what it is; a wealth of information that can help ZMart run more efficiently and serve the customer base in a better fashion. Let's go back in time a little bit to see how you were able to make this situation a reality.

### 1.3.1 Weighing the Stream Processing Option

You are the lead developer for ZMart's streaming data team. ZMart is a big box retail store with several locations across the country. ZMart does a great business with total sales for any given year upwards of $1B. You would 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 data to work with from ZMart's daily sales, so whatever technology you implement, it will need to be able to work fast and scale to handle this vast volume of data.

You decide to use stream processing because there are business decisions and opportunities that can be leveraged as each transaction occurs, no need to wait to gather data and make decisions hours later. You get together with management and your team to come up with four primary requirements that are necessary for the stream processing initiative to succeed.

So our next step is to go over each one quickly.

### 1.3.2 The Business Requirements for ZMart's Purchase Transaction Tracking Program

- First and foremost you value your relationship with your customers. With all the privacy concerns of today, your first goal is to protect their privacy, with protecting their credit cards numbers as the highest priority. So however we use the information from the transaction above, Jane's credit card information should never at risk of exposure.
- There is a new customer rewards program in place where customers earn bonus points for the amount of money they spend on certain items. The goal is to notify customers very quickly once they have 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.
- The company would like to refine further 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 specific items that are best sellers in a given area of the country.
- All purchase records need to be saved in an off-site storage center for historical and ad-hoc analysis.

The requirements you have are straight forward enough on their own, but how would you go about implementing those various requirements against a single purchase transaction like the one outlined above?

### 1.3.3 Deconstructing the Requirements into a Graph

If we stop and look at the requirements outlined above, we can quickly recast them into a graph, namely a directed acyclic graph or DAG. The point where the customer completes the transaction at the register is the source node for the entire graph. Viewing the purchase transaction as the node of origin, then the list of requirements simply become child nodes of the main source node.

Here's the view of the business requirements recast as a DAG.

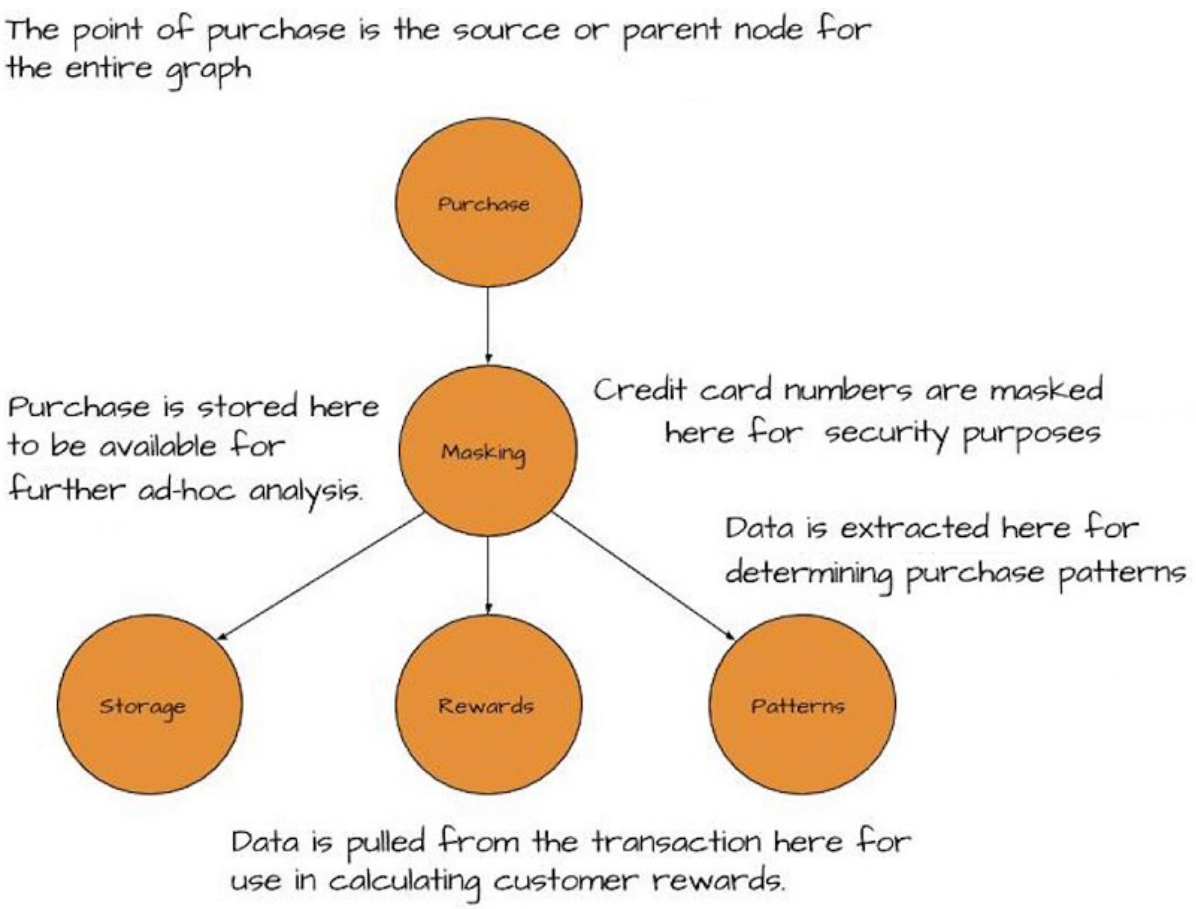



**Figure 1.4 Here we are recasting the business requirements for your streaming application as a directed acyclic graph (DAG). Each vertex represents a requirement, and the edges show the flow of data through the graph.**

Next, we will show how to map a purchase transaction to our requirements graph.

## 1.4 Changing the Perspective of a Purchase Transaction

In this section, we will walk through every step of the purchase to show how we apply it to the requirements graph. This walkthrough is essentially how Kafka Streams works. For now, we will stick with generic graph flow, but in the next section, we will demonstrate how we will specifically apply Kafka Streams to this issue. Also, we will use very high-level language as we map the purchase to the requirements graph.

### 1.4.1 Source Node

The source node of the graph is where the application consumes the purchase transaction.


**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. For the credit-card masking operation, we make a copy of the data then convert all the digits on the credit card number to an "x" except for the last four digits.

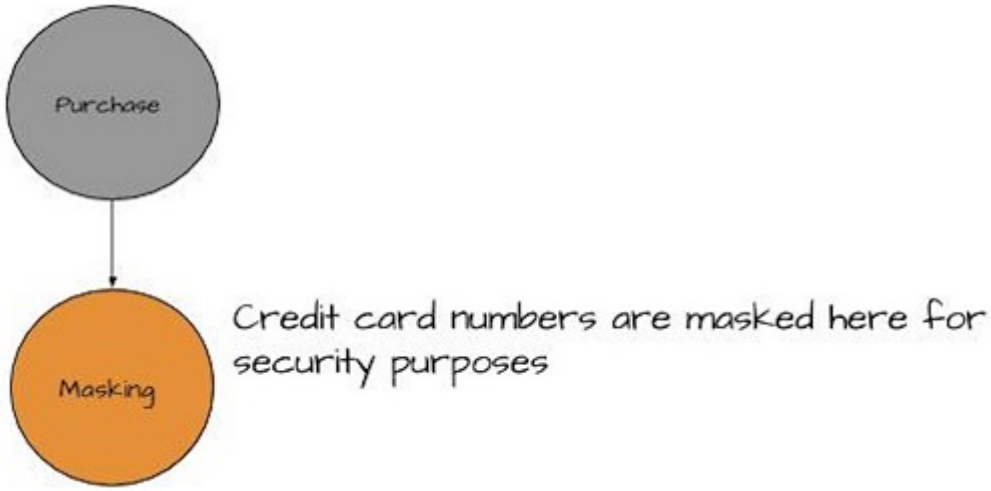



**Figure 1.6 The first vertex or node in the graph representing the business requirements. This node is responsible for masking credit card numbers and is the only node that receives the raw sales data from the source node effectively making this node the source for all other nodes connected to it.**

This node is the only direct child of the source, all other nodes in the graph are children of the credit-card masking node, so data flowing through the rest of the graph will have the credit-card field converted to the 'xxxx-xxxx-xxxx-1122' format. The remaining nodes that are

- The Purchase patterns node
- The Rewards accumulator node
- The Storage node that writes the sales transaction to storage for ad-hoc analysis later.

### 1.4.3 Patterns Node

The purchase patterns node extracts the relevant information to establish where customers purchase products throughout the country. I

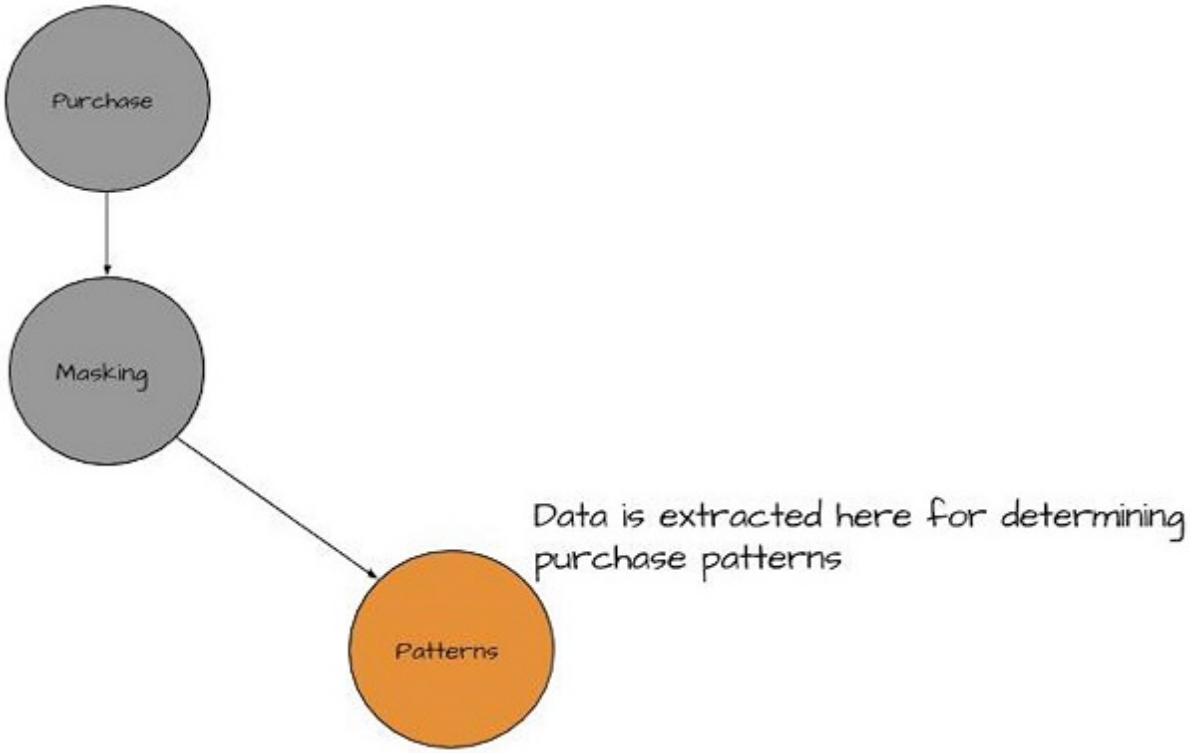



**Figure 1.7 Here we are adding the patterns node in the graph. It consumes purchase information from the masking node and converts it into a record showing when a customer purchased an item and the zip code where the customer completed the transaction.**

Instead of making a copy of the data; the patterns node will retrieve the item, date and the zip code where the item was purchased and create a new object containing those fields. Now, let's move on to the next node in our graph.

### *1.4.4 Rewards Node*

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

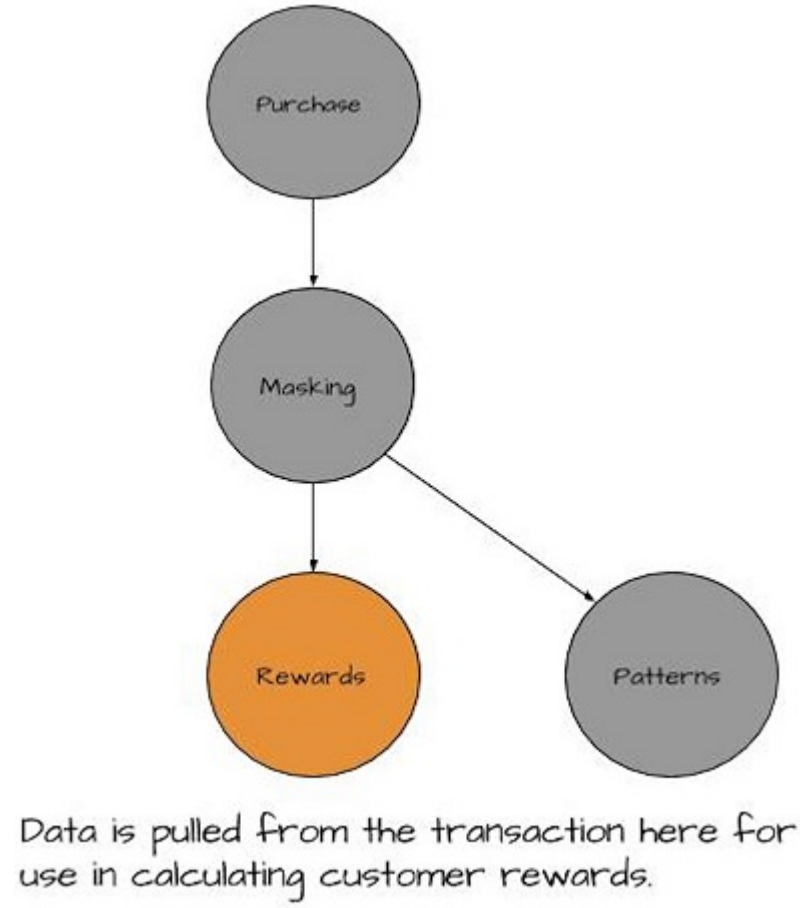



**Figure 1.8 This is the rewards node in the graph and is responsible for consuming sales records from the masking node and converting them into records containing the total of the purchase and the customer id.**

As with the patterns node, this is where we will stop with the rewards node in the graph, but when we are applying Kafka Streams, we will be adding an extra node as well.

### *1.4.5 Storage Node*

The final child node writes the entire purchase data out to storage for further ad-hoc analysis.

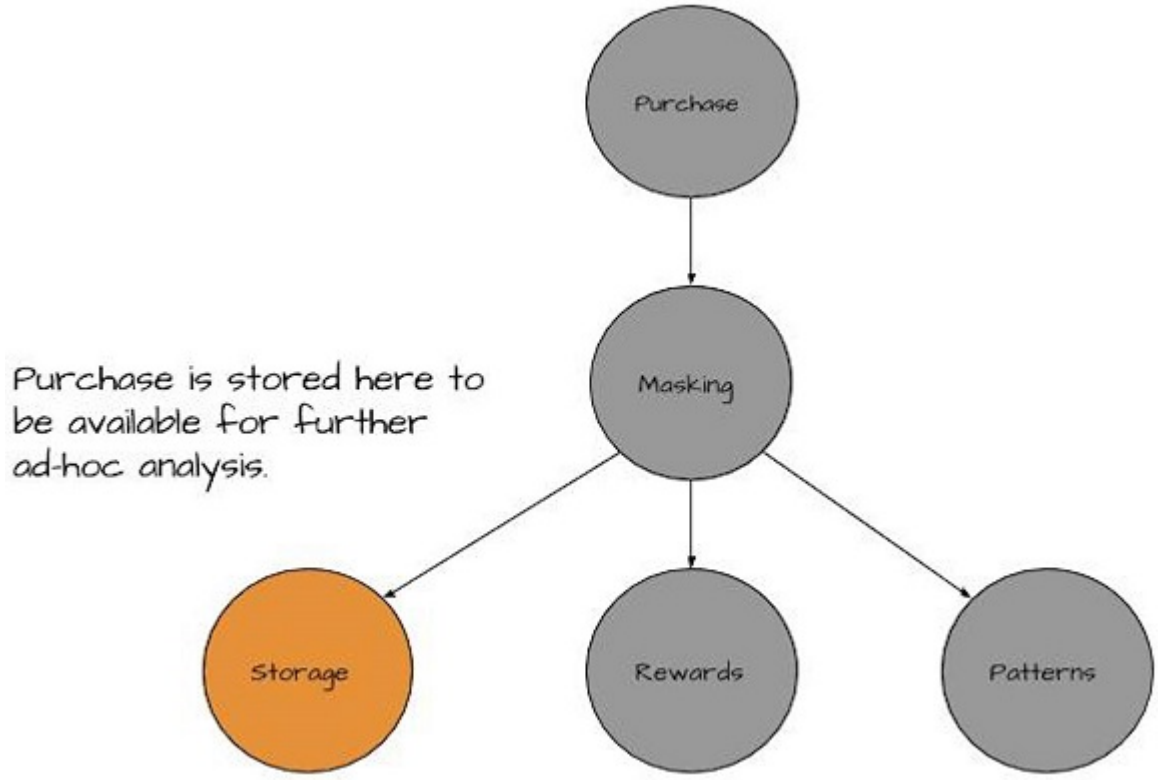



**Figure 1.9 Here is the last node in the graph and consumes records from the masking node as well. These records aren't converted into any other format but are stored in a NoSql data store for ad-hoc analysis later.**

We have now finished tracking the results of the example purchase transaction through the graph of your requirements. So the next step is, what will we use to convert this graph into live, functional streaming application? In the next section, we'll see exactly how we can use Kafka Streams to build a solution to meet our goals.

## *1.5 Applying Kafka Streams to the Purchase Processing Graph*

Before we get into the technical details of implementing Kafka Streams to our issue, let's start with a 'plain-speak' definition.

Kafka Streams is a library that allows you to perform per-event processing of records. What do we mean by per-event processing? You have the ability to work on data as it arrives, no grouping data together in small batches, you process each single record *as soon as* it is available no 'micro-batching' required.

Most of the goals here you have time sensitive in that you want to be able to take action as soon as possible. Preferably you will be able to collect information as it occurs. Additionally, there are several locations across the country, so you need all transaction records to funnel down into a single, constant flow or "stream" of data for your analysis. For these reasons, Kafka Streams is a perfect fit. Kafka Streams will allow you to process records as they arrive and give you the low latency processing you require.

## 1.6 Kafka Streams as a Graph of Processing Nodes

In Kafka Streams, you define a topology of processing "nodes". One or more nodes will have Kafka topic(s) as its source, and you add additional processing nodes considered child nodes. Each child node itself can define other child nodes (we will use the terms processor and node interchangeably) or processors. Each processor node performs its assigned task then forwards the message to each of its child nodes. This process of performing work then forwarding data to any child nodes continues until every child node has executed its function.

Does this process sound familiar? It should as this is the exact transformation performed with your business requirements, we transformed them into a graph of processing nodes. Traversing a graph is how Kafka Streams works; it is a directed acyclic graph (DAG) or topology of processing nodes. You start with a source or parent node, and it 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 their own and so on. Messages flow through the graph in a depth-first manner. This depth-first approach has significant implications. Each record (a key-value pair) is processed *in full* by the entire graph before accepting another one for processing. Having each record processed depth-first through the whole DAG eliminates the need to have back pressure built into Kafka Streams.

**SIDEBAR** **What is Back Pressure?**

While there are varying definitions of back pressure, here I define backpressure as the need to restrict the flow of data by buffering or some blocking mechanism. Back pressure is necessary when a "source" is producing data faster than a "sink" can receive and process the data.

By being able to connect or chain together multiple processors in this manner, you can quickly build up complex processing logic, while at the same time having each component itself remain relatively straightforward and easy to keep the details straight. It is this composition of processors where the power and complexity come into play.

**SIDEBAR** **What is a Topology?**

A topology is a way you arrange parts of an entire system and have connected them with each other. When we say Kafka Streams has a topology, we are referring to transforming data by running through one or more processors.

**NOTE** **Kafka Streams and Kafka**

As you might have guessed from the name, Kafka Streams runs on top of Kafka. In this introductory chapter knowledge of Kafka itself won't be strictly necessary, as we will focus more how Kafka Streams works conceptually. While there may be a few Kafka specific terms mentioned, for the most part, we will be concentrating on the stream processing aspects of Kafka Streams. But, knowledge of Kafka is essential to work effectively with Kafka Streams. So for readers who are new to Kafka or are unfamiliar with Kafka altogether, we will teach you what you need to know about Kafka in chapter 2.

## 1.7 Applying Kafka Streams to the Purchase Transaction Flow

Now let's walk through building a processing graph again, but this time we will create a Kafka Streams program. To refresh our memory, here is the requirements graph that we just constructed:

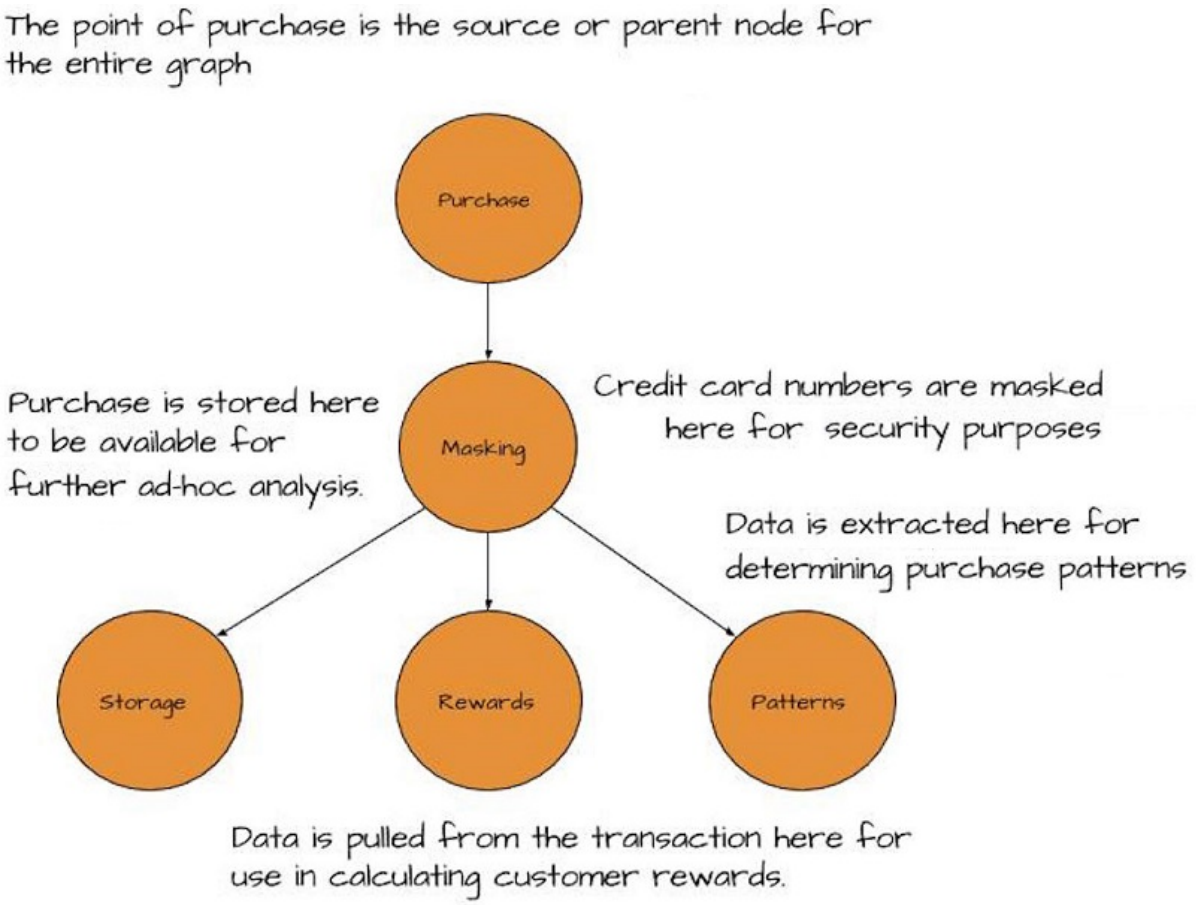



**Figure 1.10 Here's the view of our business requirements again cast a graph. Remember vertexes are processing nodes that handle data and the edges show the flow of data.**

While we will be building a Kafka Streams program as we build this graph, this will still be a relatively high-level approach, and some details will be left out. We will go into

more detail later in the book when we take a look at the actual code.

Once our Kafka Streams program starts to consume records, we convert the raw messages into `Transaction` objects. Specifically here is the pieces information that will make up the transaction object:

1. Your ZMart customer id (you scanned your member card).
2. The total dollar amount spent.
3. The item(s) you purchased.
4. The zip code of the store where the purchase took place.
5. The date and time of the transaction.
6. Your debit/credit card number.

### 1.7.1 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:

1. A single topic
2. Multiple topics in a comma separated list
3. A regex that can match one or more topics

**NOTE** **Terminology**
While my goal is to keep the description on at a high level here, I may mention some Kafka specific terms. Not to worry we will cover these terms in Chapter 2.

In this case, it will be the "transaction" topic. A Kafka Streams program runs one or more `StreamThread` (the number if Stream threads are user defined) instances. Each `StreamThread` has an embedded `Consumer` and `Producer` that handles all reading from and writing to Kafka. In addition to specifying source topic(s), you also provide `Serdes` for the keys and values. A `Serdes` instance contains the serializer and deserializer needed to convert objects to byte arrays/byte arrays to objects, respectively.

**SIDEBAR** **What is a Serdes?**
Serdes is a class containing a Serializer/Deserializer pair for a given object.

In this case, we will work with the `Purchase` object, as it defines methods that will aid in the processing work. You provide key serdes, value serdes and source topic(s) to a `KStreamBuilder` which returns a `KStream` instance. You use the `KStream` object throughout the rest of the program. The value deserializer automatically converts the incoming byte arrays into to `Purchase` objects as messages continue to flow into the stream.

It is important to note that to Kafka itself, the Kafka Streams program appears as any other combination of consumers and producers. There could be any other number of

applications reading from the same topic in conjunction with your streaming program.

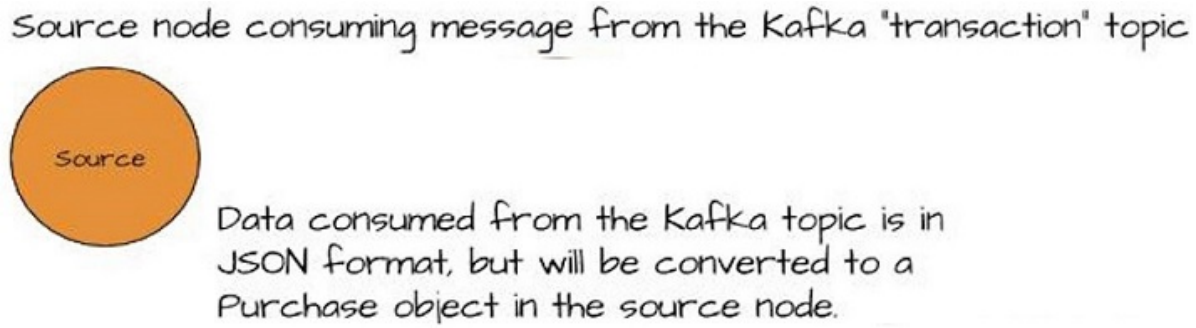



**Figure 1.11 Here is the source node, but instead using the vague term "source" we'll call this what it is - a Kafka topic.**

### *1.7.2 The First Processor - Masking Credit Card Numbers*

Now you have your source defined, and we can start creating processors that will do the work on the data. The first goal from above is to mask the credit-card numbers recorded in the incoming purchase records. The first processor will be used to convert credit-card numbers from 1234-5678-9123-2233 to xxxx-xxxx-xxxx-2233. The `KStream.mapValues` method performs the masking. The `KStream.mapValues` method returns a new `KStream` instance that changes the values as specified by the given `ValueMapper` as records flow through the stream. This particular `KStream` instance will be the parent processor for any other processors you define. Our new parent processor provides any downstream processors with `Purchase` objects with the credit-card number already masked.

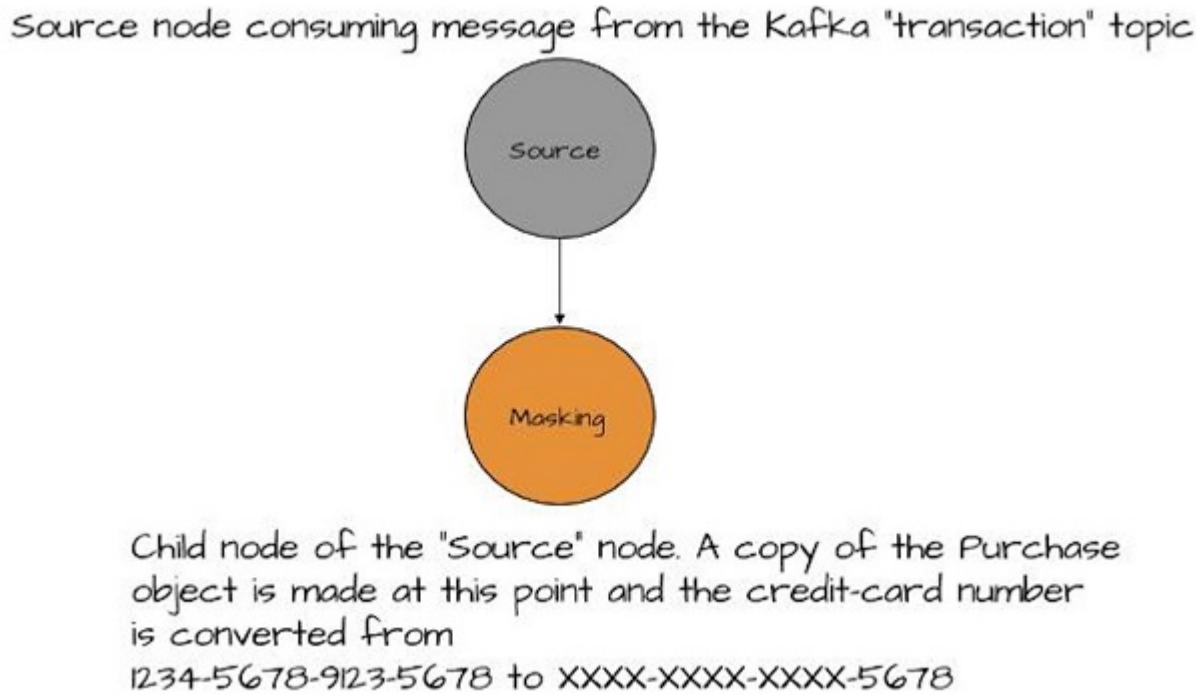



**Figure 1.12 This is the masking processor and is a child of the main source node. It receives all the raw sales transactions and emits new records with the credit card number masked. All other processors receive their records from this processor.**

### CREATING PROCESSOR TOPOLOGIES

A few words on creating processor topologies. Each time you create a new `KStream` instance by using a transformation method, you are in essence building a new processor. That new processor is connected to the other processors already created. By composing processors, we can use Kafka Streams to create complex data flows elegantly.

It is important to note that by calling a method that returns a new `KStreams` instance does not cause the original one to stop consuming messages. When using a transforming method, a new processor is created and added to the existing processor topology. The updated topology is then used as a parameter to create the new `KStream` instance. The new `KStream` object starts receiving consumed messages from the point of its creation.

It is very likely that you will build new `KStream` instances to perform additional transformations while retaining the original stream to continue consuming incoming messages for an entirely different purpose. You will see an example of this when we define the second and third processors.

While it is possible to have a `ValueMapper` convert the incoming value to an entirely new type, in this case, it will just return an updated copy of the `Purchase` object. Using a "mapper" to update an object is a pattern we will see over and over in KStreams. Each transforming method returns a new instance of a `KStream` that will apply some function to all incoming data at that point.

Now you should have a clear image of how we can build up our processor "pipeline" to transform and output data. Next, we move on to creating the next two processors using the first processor built as a parent processor.

### 1.7.3 The Second Processor - Purchase Patterns

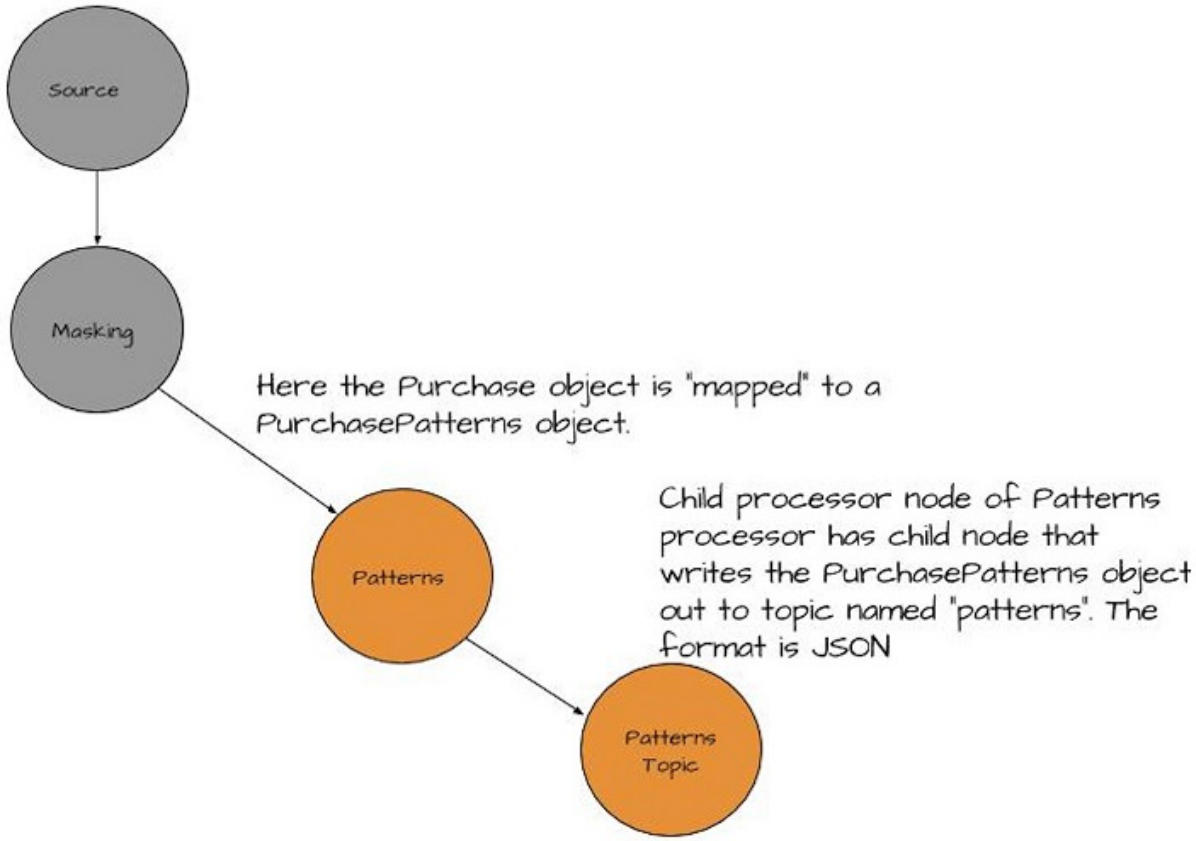



**Figure 1.13 Here is the purchase patterns processor. It's responsible for taking Purchase objects and converting them into PurchasePattern objects containing the item purchases and the zip code of the area of the country where the transaction took place. We've added a new processor here that takes records from the patterns processor and writes them out to a Kafka topic.**

Next in the list of processors to create is one that can capture information necessary for determining purchase patterns in different regions of the country. To do this, we will add a child processing node to the first processor/KStream we created. Let's call this first KStream instance purchaseKStream from here on out. The purchaseKStream produces `Purchase` objects with the credit-card number masked.

The purchase patterns processor receives a `Purchase` object from its parent node and "maps" the object of a new type, a `PurchasePattern` object. The mapping process extracts the actual item purchased (toothpaste) and the zip code it was bought in and uses that information to create the `PurchasePattern` object. We will go over exactly how this mapping process occurs in chapter 3.

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

### 1.7.4 The Third Processor - Customer Rewards

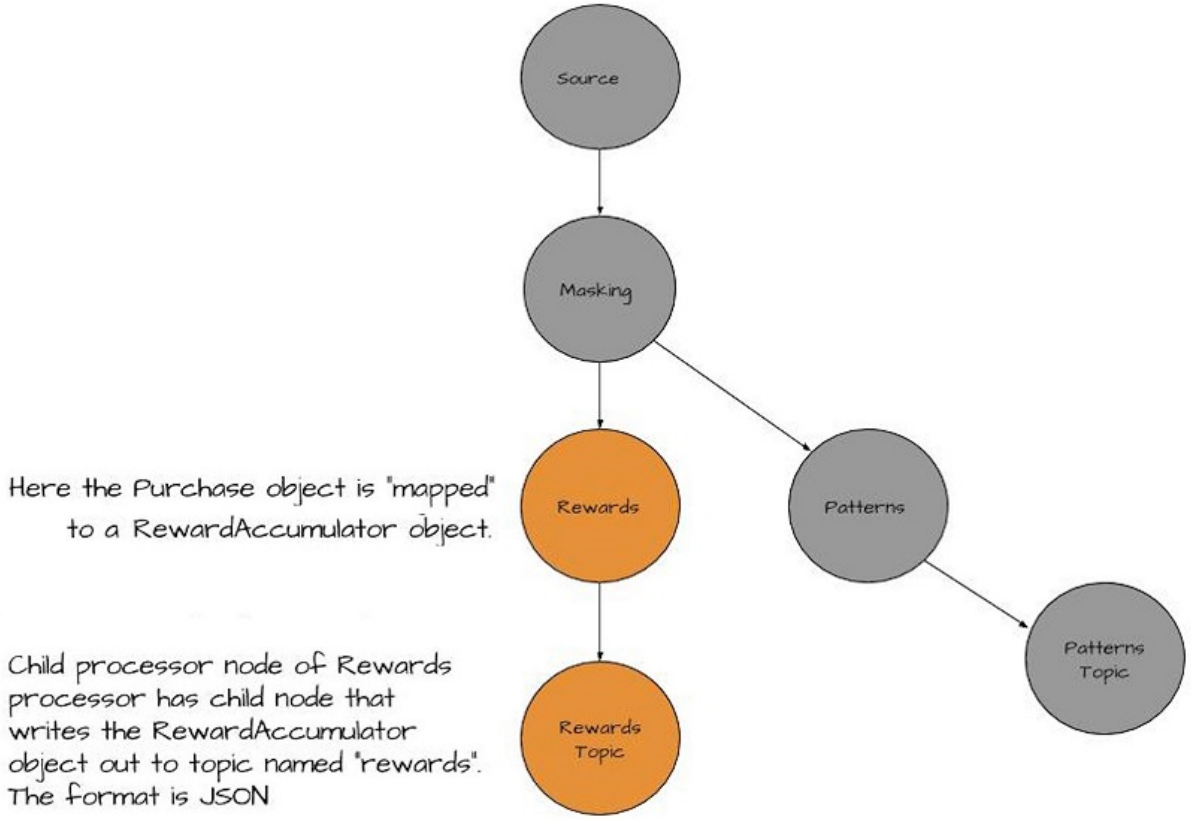



**Figure 1.14 Here is the customer rewards processor responsible for transforming Purchase objects into a RewardAccumulator object containing the customer id, date and total dollar amount of the transaction. We've also added another child processor here to write the Rewards objects to another Kafka topic.**

The third processor created will extract the required information for the customer rewards program. This processor is also a child node of the purchaseKStream processor. The rewards processor, similar to the purchase patterns processor, receives the `Purchase` object and "maps" it into another type the `RewardAccumulator` object.

The reward processor also adds a child processing node to write the `RewardAccumulator` object out to a Kafka topic, "rewards". It's from consuming records from the "rewards" topic that another application would determine any rewards for ZMart customers, the email that Jane Doe received from the purchase scenario for example.

### 1.7.5 Fourth Processor - Writing Purchase Records

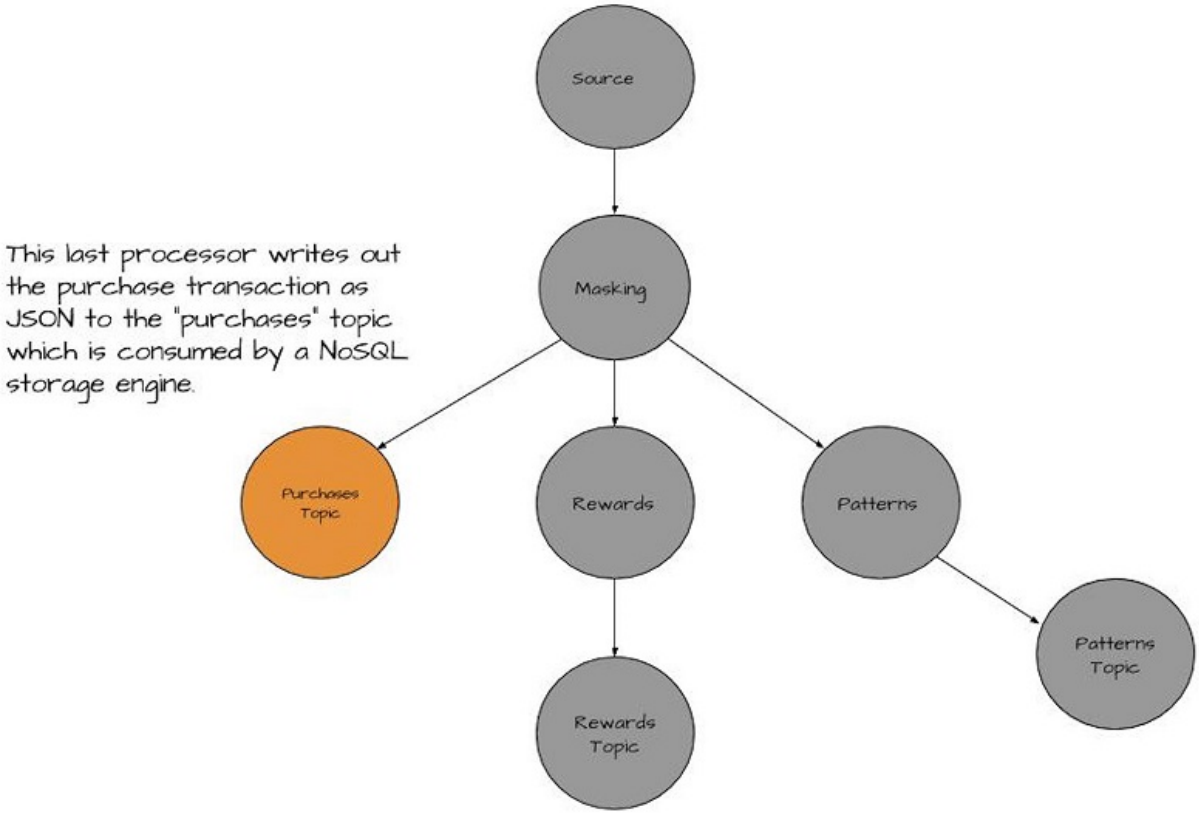



**Figure 1.15 This is the final processor in the topology. This processor is responsible for writing out the entire Purchase object out to another Kafka topic. The consumer for this topic will store the results in a NoSql store such as MongoDB**

The last processor and third child node of the masking processor node write the purchase record out to a topic "purchases". The purchases 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 ad-hoc analysis later. This record contains all the information from the purchase example above.

We have taken the first processor that masked the credit-card number and have used it to feed three other processors; 2 that further refined or transformed the data and one that wrote the masked results out to a topic for further use by other consumers. You can see now that by using Kafka Streams, we can build up a powerful processing "graph" of connected nodes to perform data integration tasks in a stream processing manner.

## 1.8 Summary

We have covered a lot of information in this chapter. Let's do a quick review of the key points you should remember:

- Kafka Streams is a graph of processing nodes that when combined provide powerful and complex stream processing.
- How batch processing is very powerful but is not enough to satisfy real-time needs for working with data.
- How distributing data, key-value pairs, partitioning and data replication are critical for

distributed applications.

To understand Kafka Streams, you really should know some Kafka. If you don't we'll cover the essentials so you can get started with Kafka Streams:

- Installing Kafka and sending a message.
- A deep dive into Kafka's architecture, what is a distributed log.
- What are Topics and how are they used in Kafka
- Understanding how producers and consumers work and how to write them effectively.

However, if you are already comfortable with Kafka, feel free to go straight to Chapter 3 where we'll build a Kafka Streams application to tackle the concrete example discussed in this chapter.

# 2 Kafka Quickly

In this chapter

- Kafka Architecture
- Sending Messages with Producers
- Reading Messages with Consumers
- Installing and Running Kafka

While this is a book about Kafka Streams, it's impossible to have a conversation about Kafka Streams without discussing Kafka itself. After all Kafka Streams is a library that runs on Kafka. Kafka Streams is designed very well so strictly speaking it is possible to get up and running with minimal to no Kafka experience. However, your progress and ability to fine tune are going to be limited. Having a good fundamental knowledge of Kafka is essential to get the most out of Kafka Streams.

## 2.1 Who Should Read This Chapter

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

**NOTE** **Kafka Is A Big Topic**
Kafka is too large a topic to cover in it's *entirety* in one chapter. We'll cover enough information to give you a good understanding how Kafka works and a few of the core configuration settings you'll need to know.

## 2.2 The Data Problem

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

Let's make a list of the various requirements we'd like to have in our data management solution.

1. What we need is a way to send data to a central storage quickly.
2. Because machines fail all the time, we also need the ability to have our data replicated, so those inevitable failures don't cause downtime and data loss.
3. Finally, we need the potential to scale to any number of consumers of data without having to keep track of different applications. We 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.

## 2.3 Using Kafka to Handle Data

In chapter 1 we were introduced to the large retail company ZMart. At that point ZMart wanted a streaming platform to leverage their sales data to offer better customer service and improve sales overall. But if we went back six months before that point, we would have seen ZMart looking to get a handle on its data situation. ZMart had a custom solution that worked well initially but is now becoming unmanageable for reasons we will soon see.

### 2.3.1 ZMarts Original Data Platform

Originally ZMart was a small company and had retail sales data flowing into its system by separate applications. The initial approach worked fine initially, but over time it was evident that a new approach would be needed. Data from sales in one department is not an isolated concern for that department alone. Several areas of the company are interested, and each area has a different take on what's important and how they want to the data structured. Here's how ZMart's data platform looked initially:

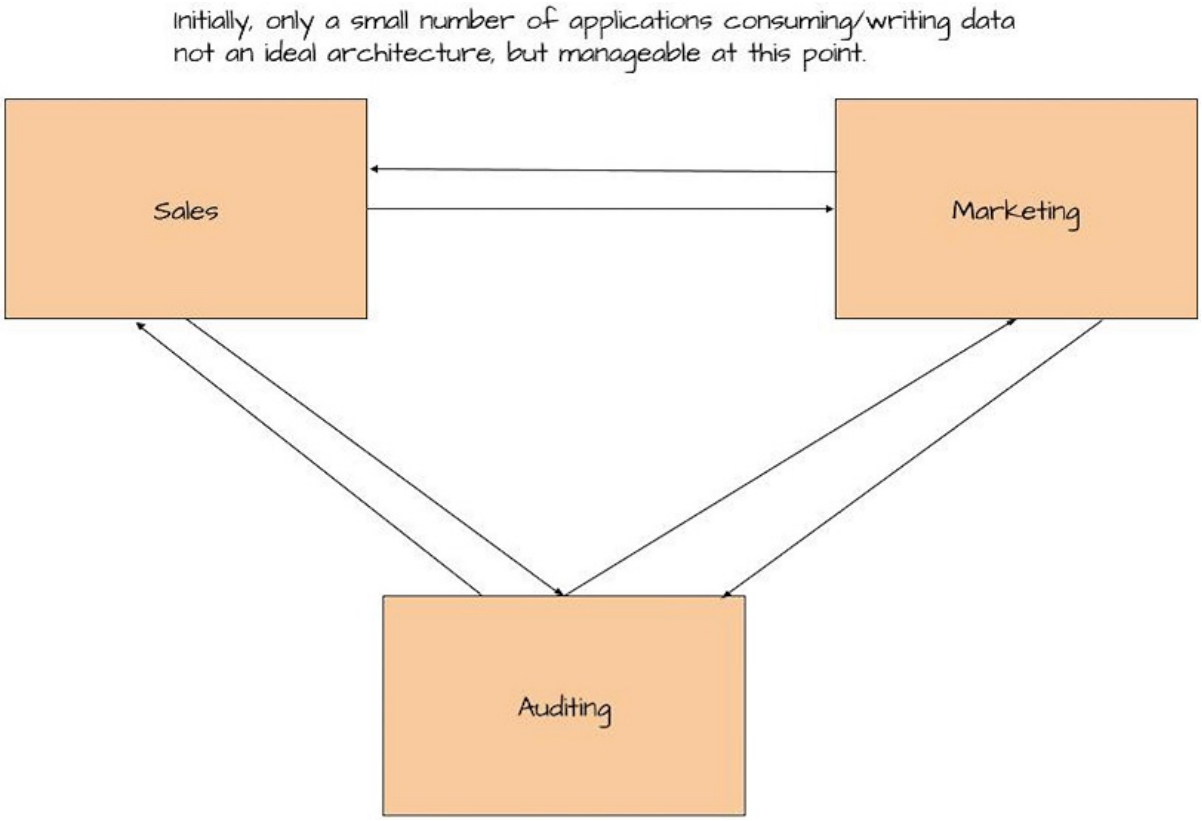



**Figure 2.1 This is the original data architecture for ZMart. In the beginning, it was simple enough to have information flowing to and from each source of information.**

Over time ZMart has continued to grow by acquiring other companies and expanding their offerings in existing stores. With each addition, the "web" of connected applications continued to become more complicated. What initially started out with a handful of applications communicating with each other, has turned into a veritable pile of spaghetti trying to make all these existing applications aware of each other to share information.

As you can see from the graphic below even with just three applications, the amount connections needed is cumbersome and confusing.

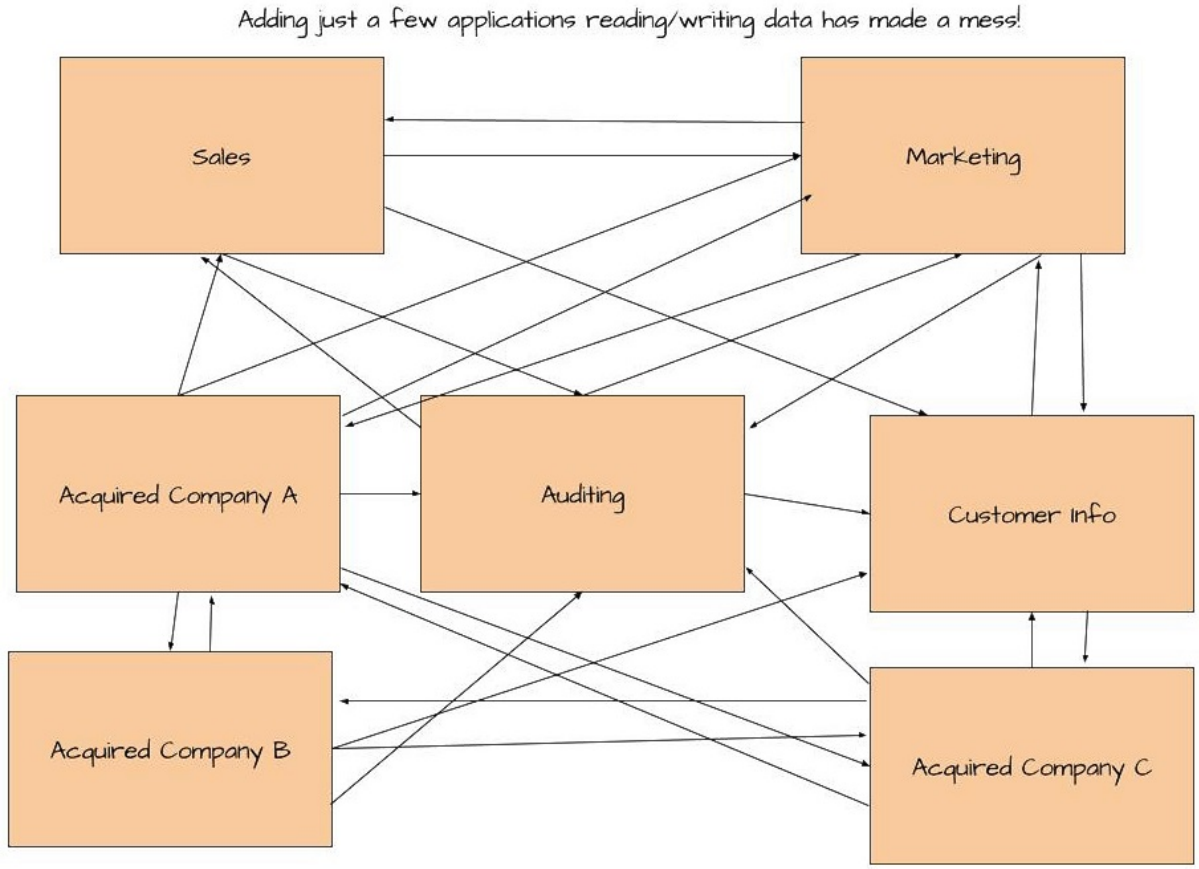



**Figure 2.2 With more machines added over time, the level of complexity trying to connect all these information sources became somewhat of a mess.**

You should be able to see how adding any additional applications will make this data architecture unmanageable over time.

### 2.3.2 Sales Transaction Data Hub

What's needed is one intake process to hold all transaction data, a transaction data hub. Additionally, this transaction data hub should be stateless, in that it accepts transaction data and stores it in such a fashion that any consuming application can pull the information it needs and it's up to the consuming application to keep track of what it has seen. Here's an image showing how you could envision Kafka as the sales transaction data hub.

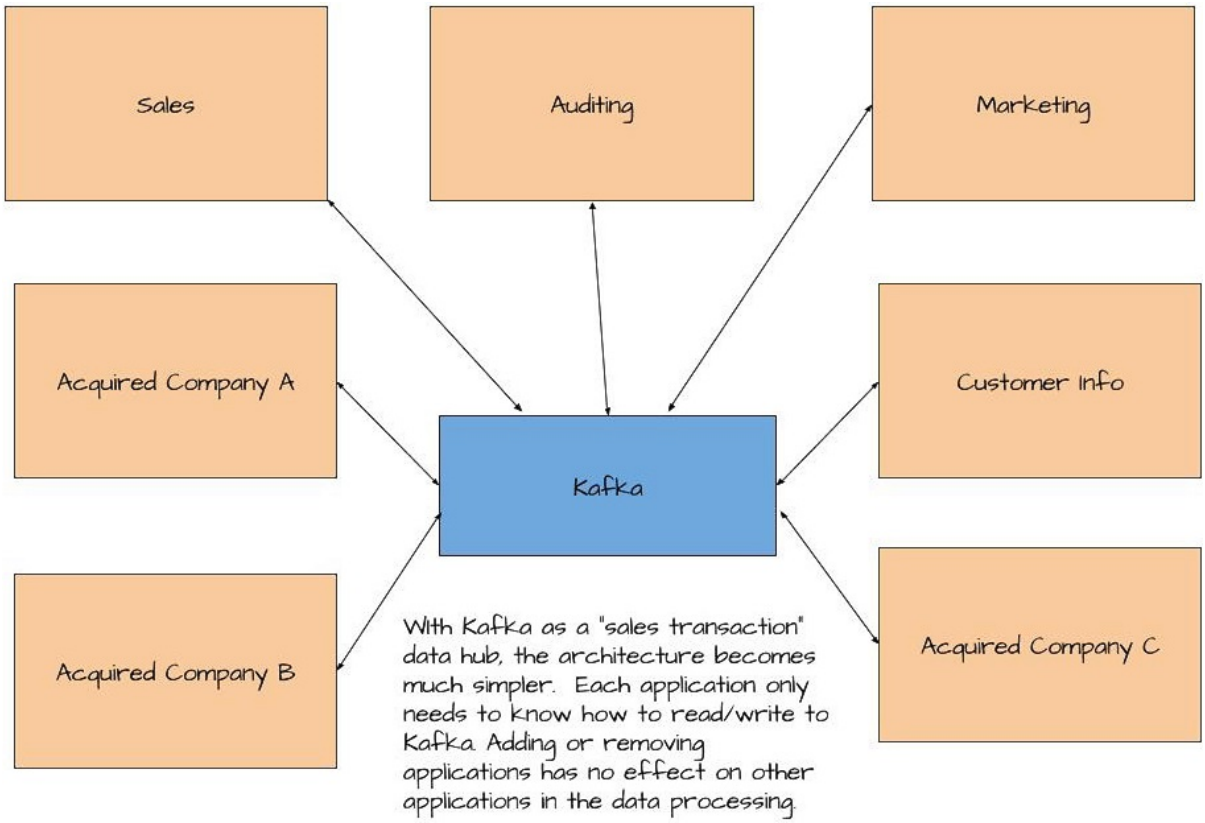



**Figure 2.3 Here we can see how using Kafka as a sales transaction hub simplifies the ZMart data architecture significantly. Now each machine does not need to know about every other source of information. All that is needed is how to read from and write to Kafka.**

The transaction data hub will only maintain how long it's been holding any transaction data and when that data should be rolled off or deleted.

### 2.3.3 Enter Kafka

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 other than development on your local machine, you will have more than one Kafka servers that make up a cluster. Kafka stores messages are written by `Producers` in topics. `Consumers` subscribe to topics and contact Kafka to see if there are messages available in those subscribed topics(s).

The preceding section is an overview of Kafka from 50,000 feet now let's take a deeper look in the following sections.

## 2.4 Kafka Architecture

In this section, we will take a much deeper look into how Kafka works and key parts of its architecture.

| NOTE | **For the Impatient**<br>In the next several sections we will be discussing the architecture of Kafka in depth. If you are interested in kicking the tires on Kafka sooner rather than later feel free to skip ahead to the section titled "Installing Kafka." After going through the install process come back to this point to continue learning about Kafka. |
|---|---|

### *2.4.1 Kafka Is A Message Broker*

Earlier we stated that Kafka is a publish/subscribe system, but to be more precise Kafka acts a message broker. A broker acts as an intermediary bringing two parties together (that don't necessarily know each other) for a mutually beneficial exchange or deal. In figure 4 below we see the evolution of the ZMart data infrastructure from what we covered in sections 2 and 3. In this image, we've added the producer(s) and consumer(s) to show how the individual parts communicate with Kafka, but not directly with each other.

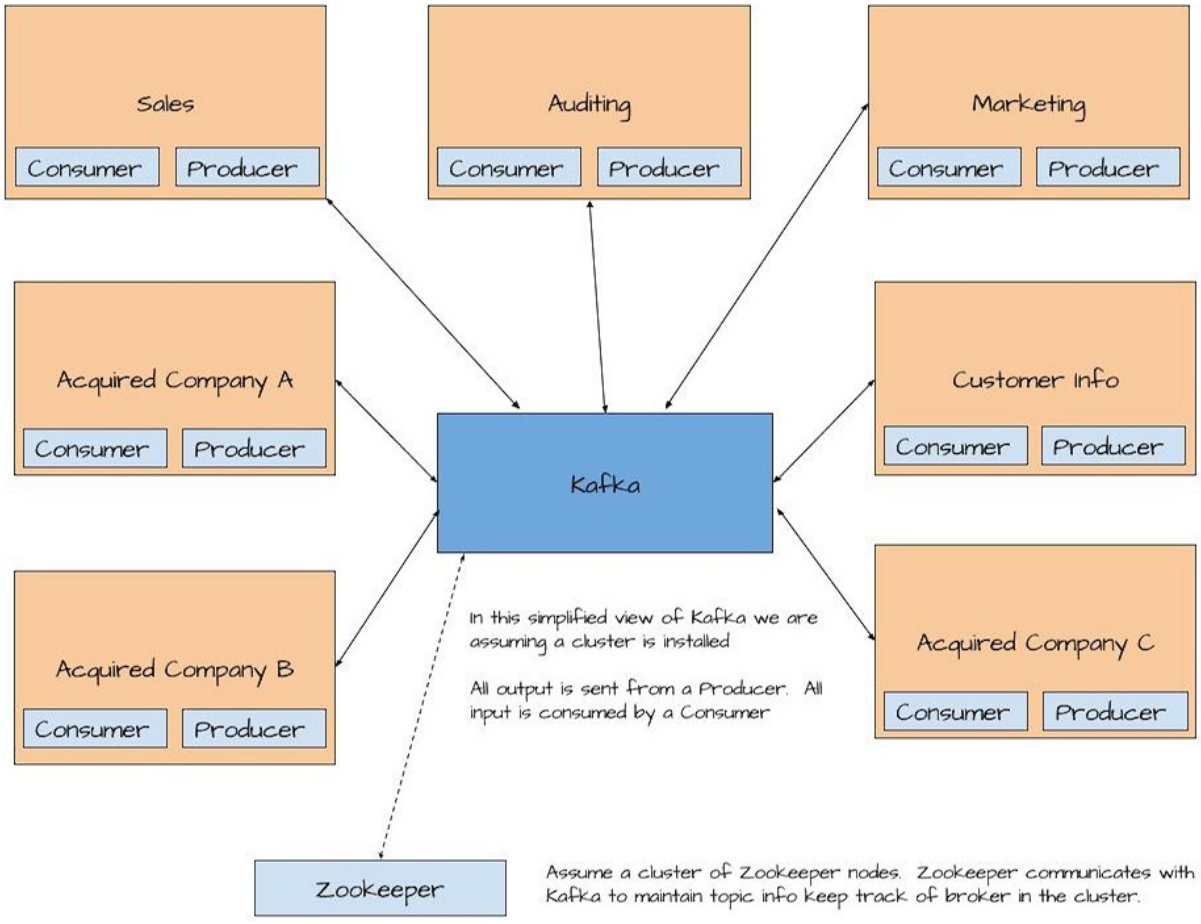



**Figure 2.4 Kafka is a message broker. We can see here how producers send messages to Kafka and those messages are stored and made available to consumers via subscriptions to topics.**

Kafka stores messages in topics and retrieves messages from topics. There is no connection at all between the producers and the consumers of the messages. Additionally,

Kafka does not keep any state regarding the producers and consumers; it acts solely as a message "clearing house."

The underlying technology of a Kafka topic is a "log". We'll learn what a log is in the next section, but for now, we can say that it is a file where Kafka appends incoming records to the end. To help manage the load of messages coming into a topic Kafka uses partitions. We covered partitions in Chapter 1 and if you recall one use of partitions is to bring data located on different machines together on the same server. We'll discuss partitions in detail in the section after we discuss logs.

### 2.4.2 Kafka Is A Log

The mechanism underlying Kafka is the notion of a log. Now, most software engineers have a familiarity with logs regarding tracking what an application is doing. Having performance issues or errors in your application? The first place to check is the application logs. But, that is not the definition of a log we are using here. The definition of a log in the context of Kafka (or any other distributed system) is "It is an append-only, totally ordered sequence of records ordered by time" [1]. Here's an image of what a log looks like:

---

Footnote 1 Jay Kreps
engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-ur

---

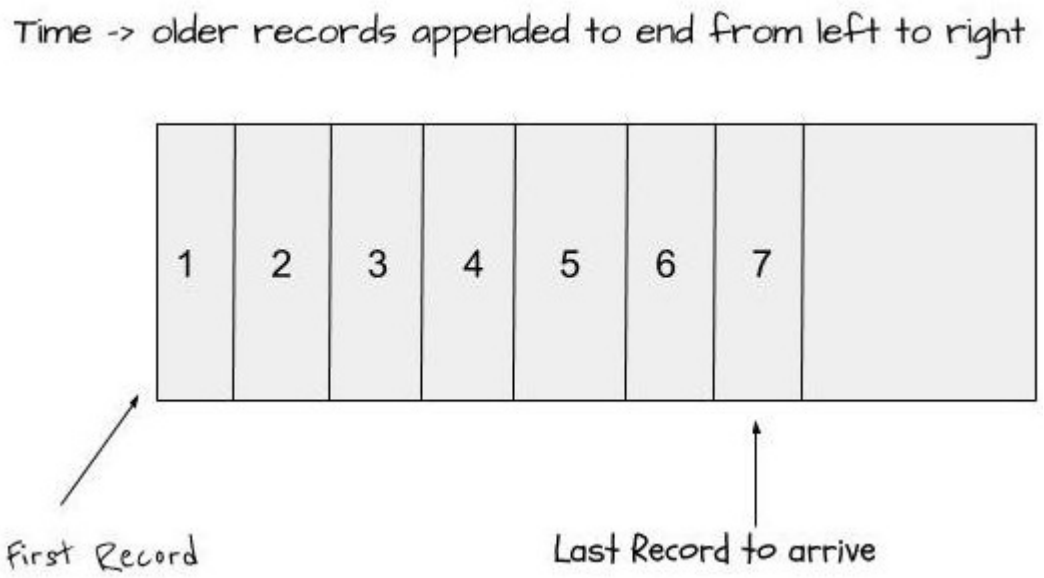



**Figure 2.5 A log is a file with incoming records are always appended to the end of the last record received. This process gives time ordering of records contained in the file.**

An application appends records to the end of the log as they arrive. Records have an implied ordering by time since the earliest records are to the left and the last record to arrive is at the end to the right, even though there might not be a timestamp associated with each record. Logs are a very simple data abstraction with very powerful

implications. If you have records in complete order with respect to time, making conflict resolution or determining which update to apply to different machines becomes straight forward, the latest record wins.

Now to take the next step, topics in Kafka are simply logs that are segregated by the topic name. We could almost think of topics as "labeled" logs.

If the log is replicated among a cluster of machines, if a single machine goes down, it becomes easier 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. Hopefully, now the concept of a "distributed commit log" is a little more clear at this point. We have only scratched the surface of a very deep topic when it comes to distributed applications and data consistency, but what we have covered so far should be enough to get an understanding of what is going on under the covers with Kafka. When we start to discuss how Kafka works, all the information presented here will be very apparent.

### 2.4.3 How logs work in Kafka

When installing Kafka, one of the configuration settings is the "logs" directory. This configuration parameter specifies where Kafka stores messages in the "log" that just discussed. Each topic maps to a directory under the specified log directory location. If the topic is partitioned, then there are as many directories as partitions with a number appended to the end (we cover partitions in detail in the next section). Inside each directory is the log file where incoming messages get added to the end. Once the log files reach a certain size (either number of records or size on disk), or when a configured time difference between message timestamps is met, the log file is "rolled", and Kafka appends incoming messages to a new log. Here's a quick graphical view:

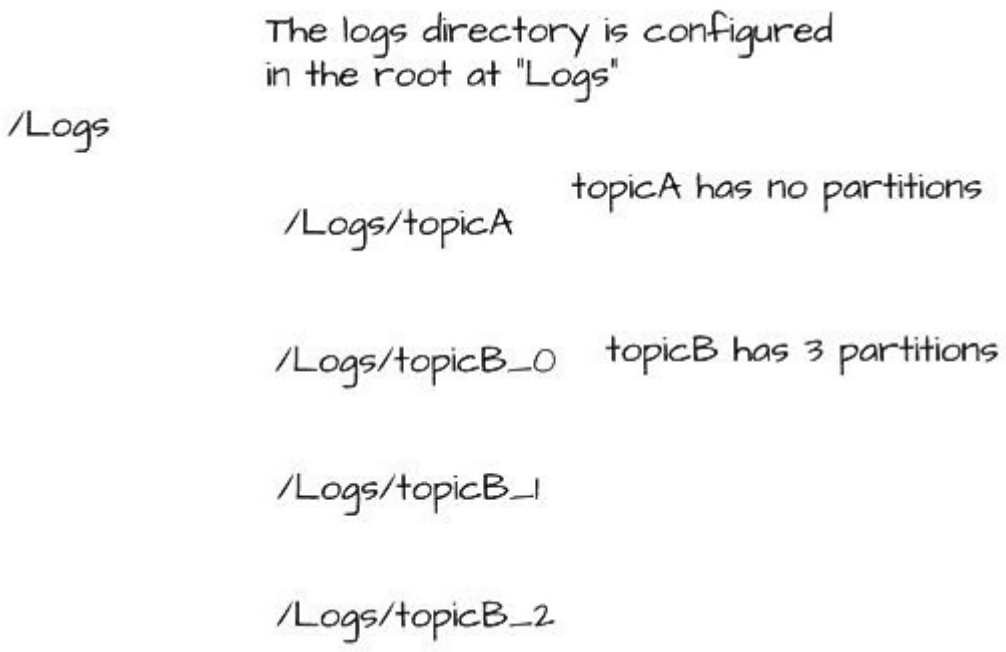



**Figure 2.6 The logs directory is base storage for messages. Each directory under /logs represents a topic. If the topic has partitions, the directory name has an underscore and a number representing the partition following it.**

At this point, we can see that a log and topic are highly connected concepts. We could say that a topic is a log or represents a log and the topic name gives us a good handle on which log we need to store the messages sent to Kafka via producers.

Now that we have covered the concept of a log let's discuss another fundamental concept in Kafka, partitions.

### 2.4.4 Kafka and Partitions

Partitions are a critical part of Kafka's design as they are essential for performance and guarantee data with the same keys will be sent to the same `Consumer` and in order. In figure 7, we dissect a topic to give us a clear idea of the role partitions play.

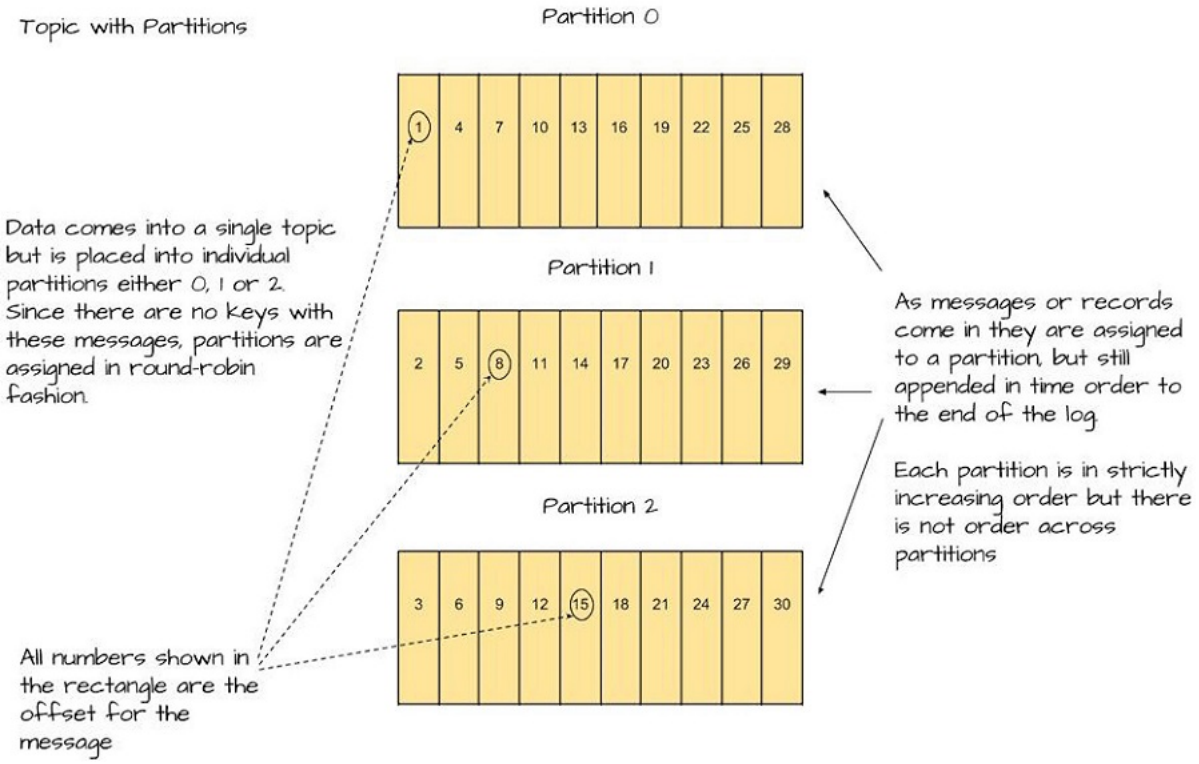



**Figure 2.7 Kafka uses partitions to achieve high throughput and spread the amount of messages sent to an individual topic across several machines in the cluster.**

Partitioning a topic is essentially splitting the data forwarded to a topic across parallel streams. Partitioning is key to how Kafka can achieve its tremendous throughput. While we discussed that a topic is a distributed log, each partition is a log unto itself and follows the same rules. Kafka appends each incoming message to the end of the log, and all messages are strict time ordered. Each message has an offset number assigned to it. Order of messages across partitions is not guaranteed, but the order of each partition guaranteed.

Partitioning serves another purpose aside from increasing throughput. Partitioning allows for spreading topic messages across several machines, so as to not restrict the capacity of a given topic to the available disk space on one server. Now that we've seen the role partitions play in helping throughput, next we'll take a look at another critical part partitions play; ensuring messages with the same keys end up together.

### 2.4.5 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 in a round-robin fashion. Here's a graphical representation of how round-robin partition assignment operates:

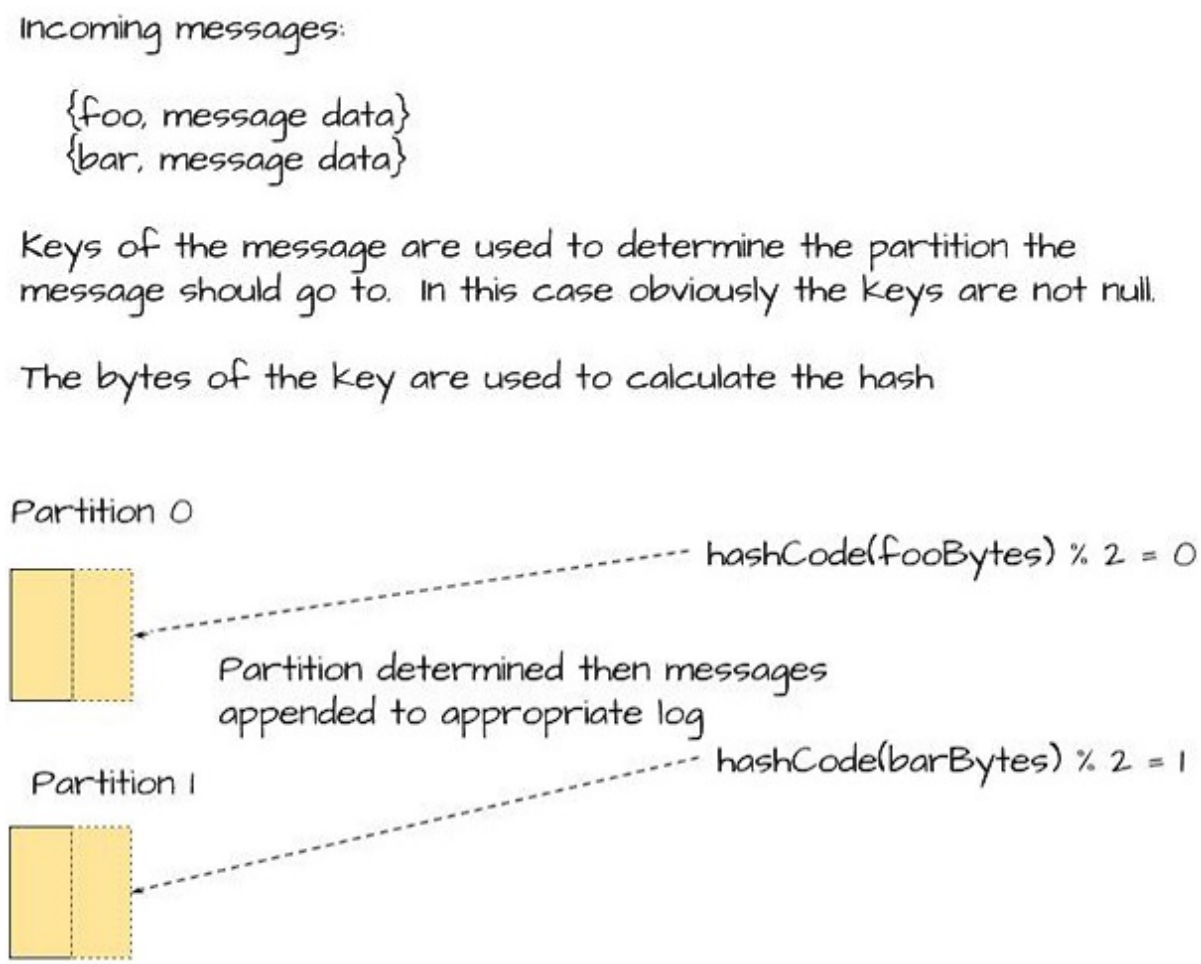



**Figure 2.8 Here we can see that "foo" is sent to partition 0 and "bar" gets sent to partition 1. You obtain the partition by hashing the bytes of the key modulus the number of partitions.**

However, if the data has populated the keys Kafka uses the following formula (in pseudo code) to determine which partition to send the key-value pair to:

**Listing 2.1 Determining Partitions**

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

By using a deterministic approach to finding the partition to use, records with the same key will *always* be sent to the same partition and in order. We should note the `DefaultPartitioner` uses this approach and should you need a different strategy to select partitions; you have the ability to provide your custom partitioner.

### 2.4.6 Writing a Custom Partitioner

Why would you want to write a custom partitioner? While there are several possibilities, we will provide one simple use case here; that is the use of composite keys.

For example, let's say 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 the customer-id, so taking the hash of the customer-id and the purchase date will not work.

In this case, we will need to write a custom partitioner that "knows" the correct part of our composite key to use for determining which partition to use.

Here the composite key looks like this:

**Listing 2.2 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;
    }
}
```

But when it comes to partitioning, we need to ensure that all customer transactions go to the same partition. By using the key in its entirety, this won't happen, though. Since purchases are across many dates, including the date makes the key non-unique placing the transactions across random partitions. We need to ensure we send transactions with the same customer id to the same partition. The only way to do that only uses the customer id when determining the partition.

Here we present an example of a custom partitioner that does what's required. The `PurchaseKeyPartitioner` extracts the customer-id from the key for determining the partition to use.

**Listing 2.3 Custom Partitioner Code**

```
public class PurchaseKeyPartitioner extends DefaultPartitioner {

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

        if (key != null) {
            PurchaseKey purchaseKey = (PurchaseKey) key;

            key = purchaseKey.getCustomerId();  (1)
```

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

❶ If the Key isn't null extract customer id

❷ Set the key bytes to the new value

❸ Return the partition with the updated key, delegating to the superclass.

With our custom partitioner here we have chosen to extend the `DefaultPartitioner`. While we could have implemented the `Partitioner` interface directly, there is existing logic in the `DefaultPartitioner` that we want to leverage. Keep in mind when creating a custom partitioner you are not limited to using only the key. Using the value alone or in combination with the key is valid as well.

**NOTE** **Partitioner Interface**
The Kafka API provides a Partitioner interface should you require to roll your partitioner. We won't be covering writing your partitioner from scratch, but the principals would be the same as presented in the example.

We've just seen how to construct a custom partitioner; next is how to wire the partitioner up with Kafka.

### 2.4.7 Specifying a Custom Partitioner

Now that we have written a custom partitioner we need to tell Kafka we want to use it over the default partitioner. Although we haven't covered producers yet, you specify a different partitioner when configuring the Kafka `Producer`.

**Listing 2.4 Partitioner Config Setting**

```
partitioner.class=bbejeck.partitioner.PurchaseKeyPartitioner
```

We'll go over producer configuration in detail when we cover using Kafka producers, but as you may have surmised, by setting a partitioner per producer instance we are free to choose to use any partitioner class per producer.

**WARNING** **Choosing Keys**
You must exercise some caution when choosing keys to use and when using parts of a key/value to parttion on. Make sure that the choice of the key to use has a fair distrubution across all of your data. Otherwise you'll end up with a "data skew" problem as most of your data is centered on just few of your partitions.

### 2.4.8 Determining the Correct Number of Partitions

The number of partitions to use when creating a topic is part art and part science. One of the key considerations for the number of partitions 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 tradeoffs for everything.

Increasing the number of partitions increases the number of TCP connections and open file handles. Additionally, depending on 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 some, but ultimately the slower processing will hinder performance. [2].

Footnote 2 Jun Rao
www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/

### 2.4.9 The Distributed Log

Up to this point, we have discussed the concept of a log and partitioned topics. Let's take a minute to explain those two concepts together to help us understand a distributed log.

So far, all of our discussions on Kafka have centered on logs and topics on one Kafka server or broker. But typically a Kafka production cluster environment includes several machines. We have intentionally kept the discussion centered around one node as it is easier to understand the concepts introduced when considering one node. But in practice, we will always be working with a cluster of machines in Kafka. The following image gives you a pictorial representation of message distribution to partitions across a cluster.

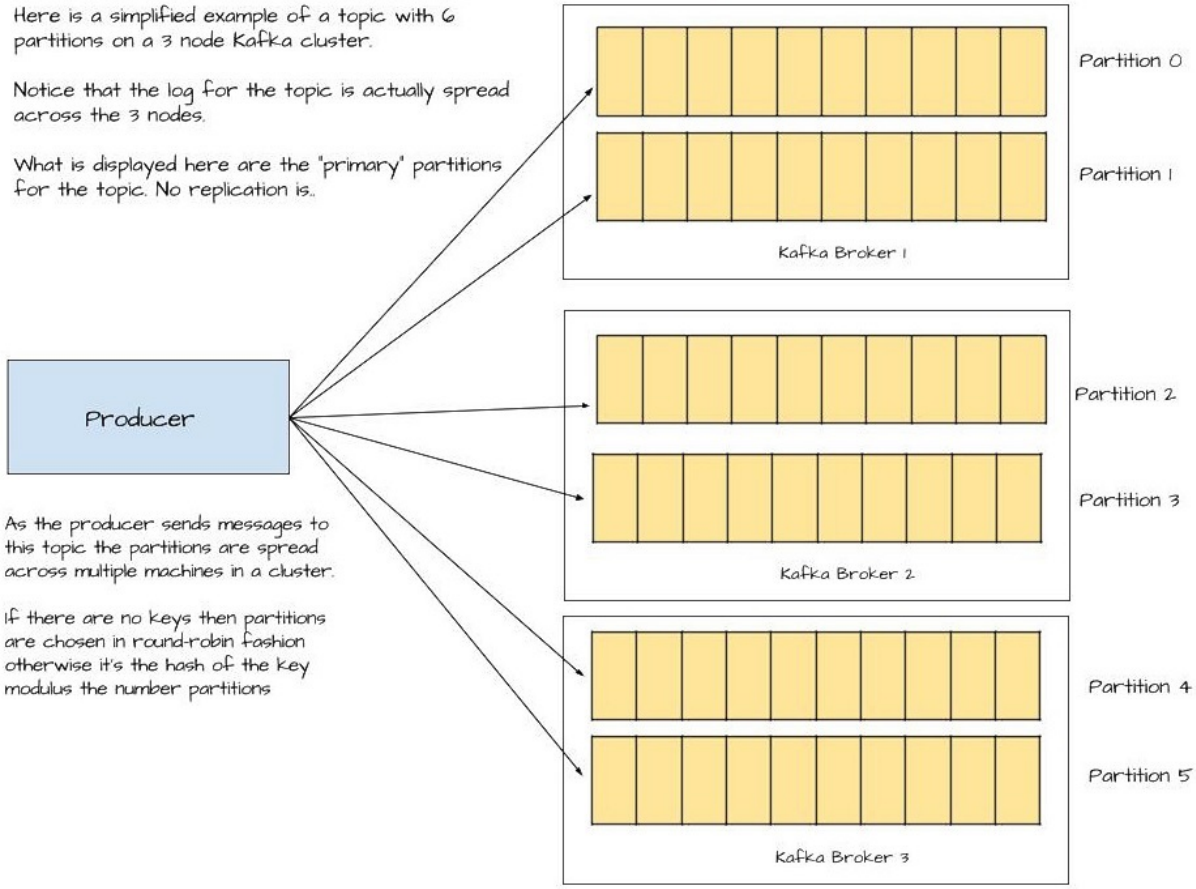



**Figure 2.9 A producer is responsible for allocation of messages to partitions of a topic. If there is no key associated with the message the producer chooses one in a round-robin fashion. Otherwise the hash of the key modulo the number of partitions is used.**

When a topic is partitioned Kafka does not allocate those partitions on one machine, Kafka spreads them across other machines in the cluster. As Kafka appends records to the log, Kafka is distributing those records across several machines by partition. If you take another look at figure 9, you can view this process in action. Let's walk through a quick example using the image above as a guide. For this example, we will assume one topic and null keys, so the producer assigns partitions in a round-robin manner.

The producer sends it's 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.

While that might sound like your data is at risk, 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).

### 2.4.10 Zookeeper - Leaders, Followers, and Replication

Up to this point, we have discussed the role topics in Kafka and how and why topics are partitioned. We discussed how partitions are not co-located on one machine but are spread out on brokers throughout the cluster. Now we have come to another crucial juncture in Kafka's architecture and how Kafka provides data availability in the face of machine failures.

Kafka has the notion of leader and follower brokers. In Kafka, 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 following brokers. Just as Kafka allocates partitions for a topic across the cluster, Kafka also places replication of the partitions across machines.

But before we go into details on how leaders, followers and replication work we need to discuss the technology Kafka uses to achieve this.

### 2.4.11 Apache Zookeeper

For the 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 tracking the replication of topics.

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

Given that Kafka is a distributed application, it should start becoming clearer how ZooKeeper is involved in Kafka's architecture. (For our discussion here we will only consider Kafka installations where there are 2 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.

### 2.4.12 Electing A Leader

Kafka uses Zookeeper for the election process for the controller broker. Discussing consensus algorithms are way beyond the scope of this book, so we'll take the 50,000-foot view by stating Zookeeper elects a broker out of the cluster to be the controller (hopefully with less contention than 2016).

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 or ISR). The brokers that make up this set are dynamic, and

zookeeper only recognizes brokers included in this set for election as leader. [3]

Footnote 3 kafka.apache.org/documentation/#design_replicatedlog

### 2.4.13 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 (leading and following) are reassigned by the controller broker. Figure 10 below gives us a brief example of a controller broker in action:

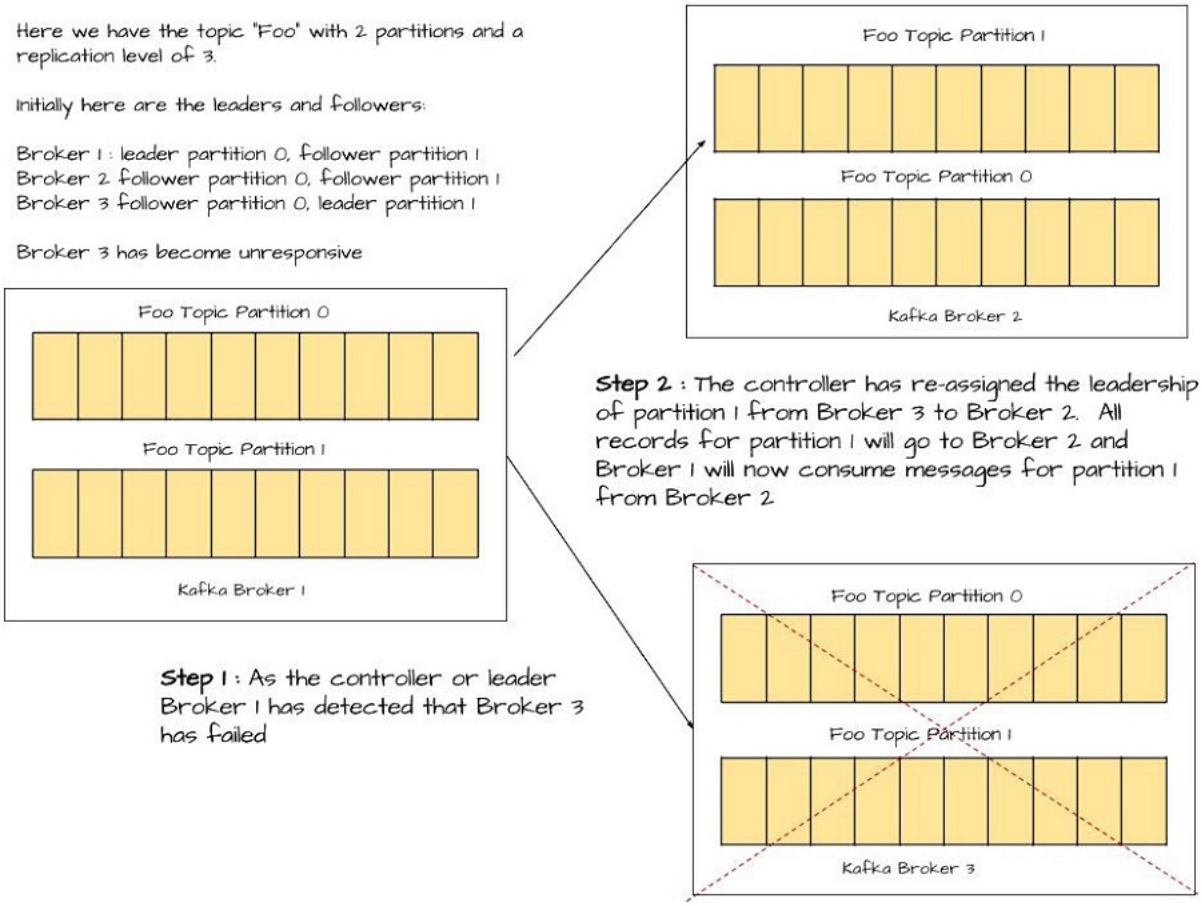



**Figure 2.10 The controller broker is responsible for making assignments to other brokers to be the lead broker for some topics/partitions and a follower for other topics/partitions. When a broker becomes unavailable, the controller broker will re-assign the failed broker's assignments to other brokers in the cluster.**

In the image above we see the demonstration of a simple failure scenario. In step one the controller broker detects that broker 3 is not available. In step two, the controller broker re-assigns leadership of the partitions on broker-3 to broker-2.

Zookeeper is also involved in the following aspects of Kafka operations:

- Cluster Membership - Joining the cluster and maintaining membership in the cluster. If a broker becomes unavailable, Zookeeper removes the broker from cluster membership.
- Topic configuration - what are the topics in the cluster, which broker is the leader for a topic, what are the number of partitions for a topic and what are the specific configuration overrides for a given topic.
- Access Control - who can read and write from particular topics.

4

Footnote 4 This section derived information from answers given by Gwen Shapira on www.quora.com/What-is-the-actual-role-of-ZooKeeper-in-Kafka/answer/Gwen-Shapira

In this section, we covered why Kafka has a dependency on Apache Zookeeper. It's important for you to come away with the idea that its 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 reassignments in the case of when a member broker fails. In the next section, we'll discuss the replication process itself.

### 2.4.14 Replication

Replication is how Kafka ensures data availability in the loss of a broker in the cluster. The level of replication can be set individually for each topic, as we saw in our example for publishing and consuming a message above. The following image (figure 11) demonstrates the replication flow between brokers.

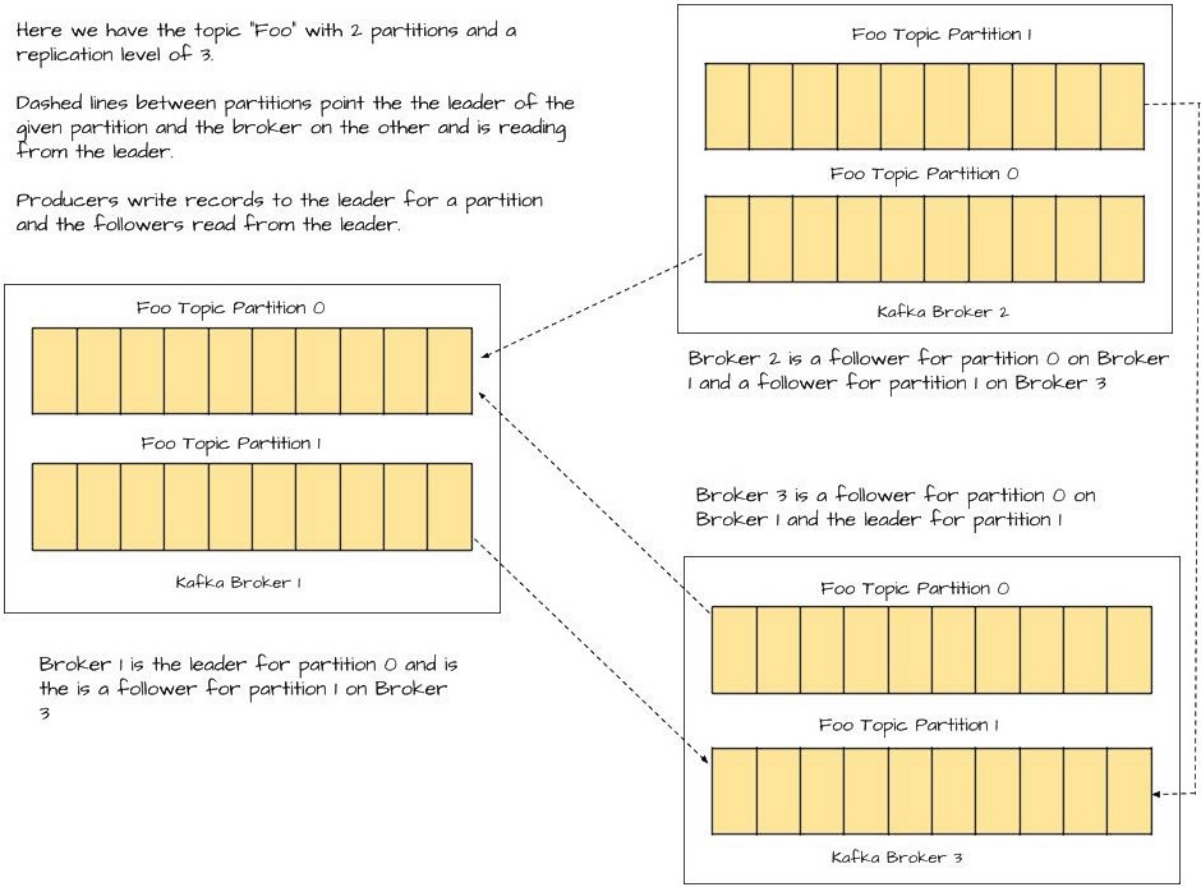



**Figure 2.11 Each broker in this image is a leader for one topic-partition and a follower for another. Replication is the process of the following broker copying data from the lead broker. The dashed lines represent where the following broker will copy data from.**

The replication process is relatively straight forward in Kafka. Following brokers of a given partition just consume messages from the leader for that partition. The following brokers then append the records to their log. As we discussed in the section "Electing A

Leader," followers that are caught up with the primary are considered in-sync replicas (ISR) and are available to be elected leader should the current controller fail or become unavailable [5].

Footnote 5 kafka.apache.org/documentation/#replication

### 2.4.15 Log Management

While we have covered appending messages, we haven't talked about how to manage the logs as they continue to grow. The amount of space on spinning disks in a cluster is a finite resource, so it is important that Kafka removes messages over time.

When it comes to handling the removal of old data in Kafka, there are two approaches, the "traditional" deletion approach of log files and compaction.

### 2.4.16 Deleting Logs

The strategy of deletion is a two-phased approach, rolling the log into segments then eventually removing the older ones.

Since appending messages is a constant process, the log will continue to grow. To manage the increasing size of the files, Kafka "rolls" the log into segments. Log rolling is configured to occur based on timestamps embedded in the message. The time threshold is considered met when a new message is received, and its timestamp is greater than the first message in the log plus the `log.roll.ms` configuration. At that point, the log is rolled thereby creating a new segment that serves as the active log. The previous current segment is still used to retrieve messages for consumers. Log rolling is a configuration set when standing up a Kafka broker [6]. There are two options for log rolling:

Footnote 6 kafka.apache.org/documentation/#brokerconfigs

**Listing 2.5 Log Rolling Configuration**

```
log.roll.ms - primary configuration, but no default value
log.roll.hours - secondary configuration,
    only used of log.role.ms is not set, default value 168
```

Over time the number 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. In figure 12 you see the process of "log rolling," and the gray section represents a recently deleted segment.

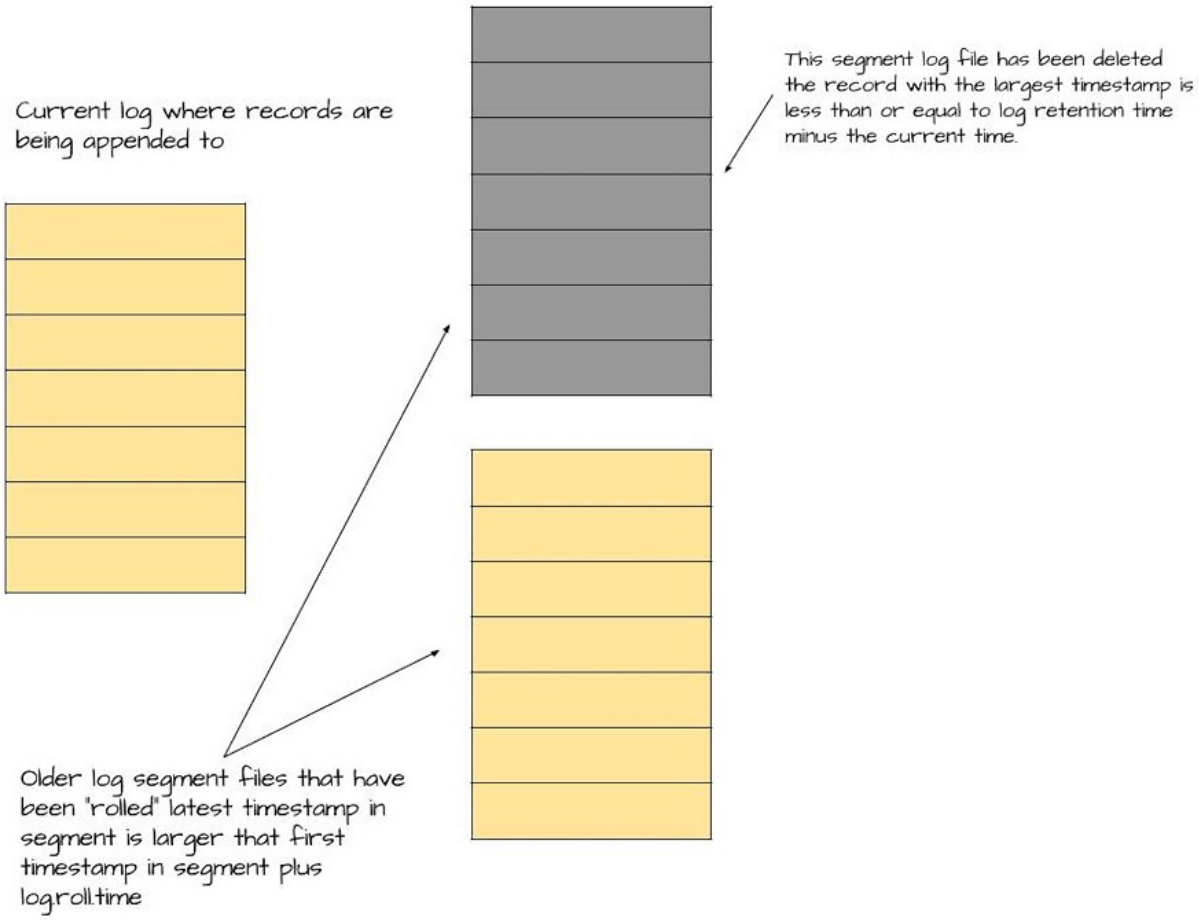



**Figure 2.12 Here on the left side of the image we have the current log segments. The gray portion accounts for a deleted log segment, and the one below represents a recently rolled one still in use.**

Like the log rolling removal of segments is based off timestamps in the message and not just wallclock time or when the file was last modified. Log segment deletion uses the largest timestamp in the log. We will list three settings in order of priority meaning that configurations earlier in the list are used first if present.

**Listing 2.6 Log Segment Retention**

```
log.retention.ms - how long to keep a log file in milliseconds.
log.retention.minutes - how long to keep a log file only in minutes.
log.retention.hours - log file retention in hours.
```

I presented these settings with the implicit assumption of very high volume topics, wherein a given period you are guaranteed to reach this maximum file size. Another configuration setting `log.retention.bytes` could be specified with a higher rolling time threshold to keep additional I/O operations down. Finally, to guard against the case of a significant spike in volume when there are relatively large roll settings, there is the `log.segment.bytes` setting which governs how large an individual log segment can be.

We've covered deletion of logs in this section. While this approach works well for non-keyed records or records that stand alone, if you have keyed data and expected updates there is another method that will suit your needs better.

### 2.4.17 Compacted Logs

Consider the case where you have keyed data, where you will receive updates for that key over time, meaning a new record with the same key, is an update of the previous value. Think of a simple case of stock ticker symbol as the key and the price per share as the value. If you are using that information to display some dashboard application in the case of a crash or restart, you will need to be able to start back up with the latest data for each key. [7]

Footnote 7 kafka.apache.org/documentation/#compaction

But under the deletion policy, a segment could get removed in between the time the next update comes, and the application crashes or has a restart. You will not have all the records you received for a given key on startup. It would be better to be able to maintain the last known value for a given key, treating the new record with the same key as we would if it were an update to a database table.

Updating records by key is the behavior that compacted topics (logs) deliver. Instead of a course grained approach of deleting entire segments by 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, removing records where there is an occurrence later in the log with the same key. Here in figure 13, we can see how log compaction works leaving the most recent message for a given key.

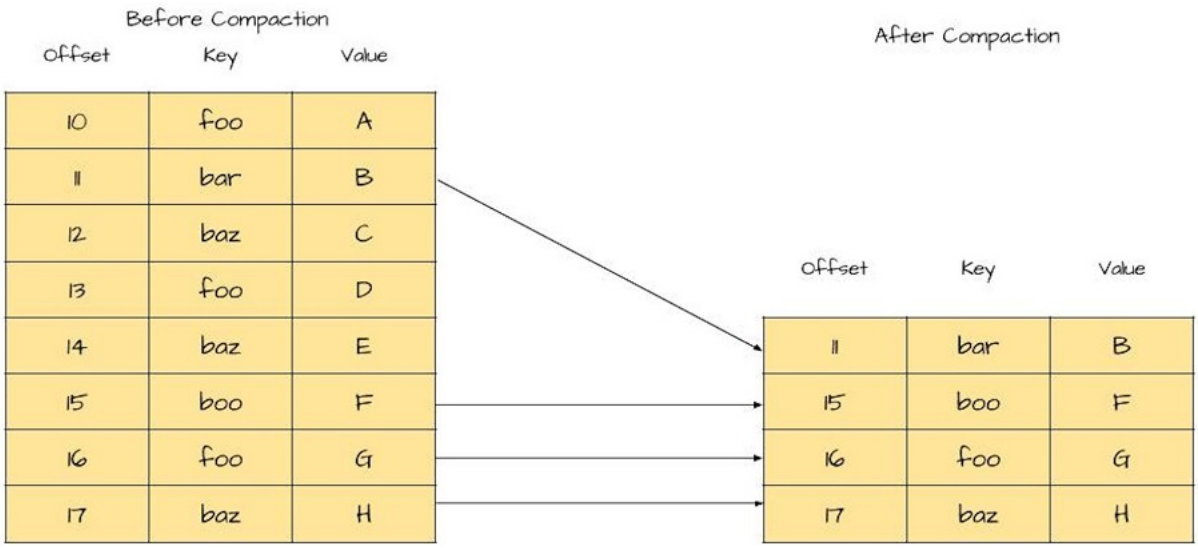



**Figure 2.13 On the left is a log before compaction. You'll notice duplicate keys with different values which are essentially updates for the given key. On the right is the log after compaction. Notice the latest value for each key is retained but the log is smaller in size.**

Using this approach guarantees we have last seen 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 that use time-based retention and other topics using compaction.

By default the log cleaner is enabled and to utilize compaction for a topic, you will need to set this property `log.cleanup.policy=compact` when creating the topic in

question. We should note at this point that compaction is used in Kafka Streams when using State Stores but you will not be creating those logs/topics yourself; the framework handles that task. But it's important to have an understanding of how compaction works at any rate.

Log compaction is a broad subject which could take almost an entire chapter by itself. For more information, the reader is encouraged to view the compaction documentation.

The key takeaway from this section is this: if you have independent, stand-alone events/messages use log deletion otherwise if you have updates to events/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 send messages to Kafka with producers and read messages from Kafka with consumers.

## 2.5 Sending Messages with Producers

Going back to ZMart's need for a centralized sales transaction data hub, we now have Kafka in place. Now let's get specific on how we send purchase transactions into Kafka. In Kafka, the `Producer` is the client used for sending messages. Below we revisit the view of ZMart's data architecture and have highlighted the producers to emphasize where they fit into the data flow.

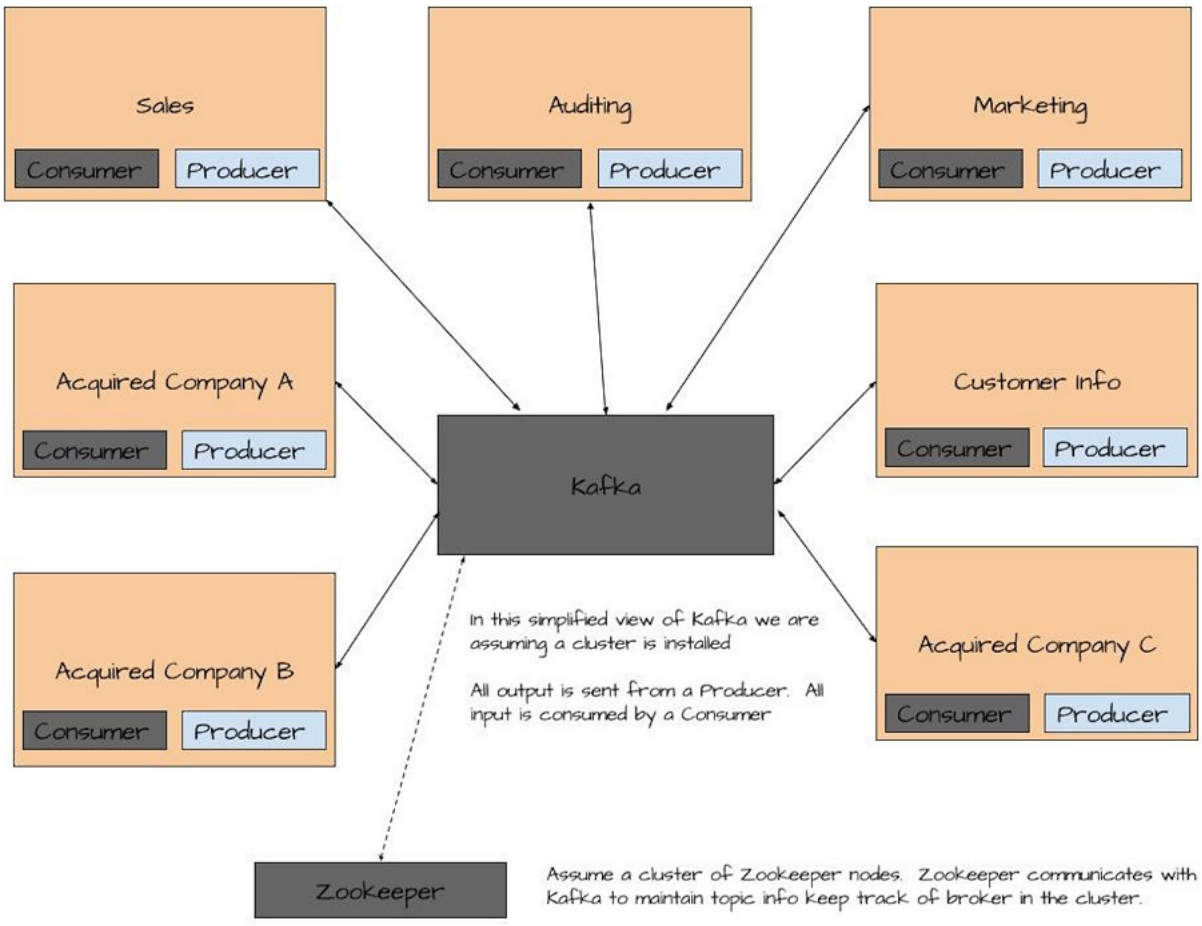



**Figure 2.14 Pictured here are the Producers used to send messages to Kafka. Keep in mind that each producer has no idea about which consumer or when the consumers read the messages. A publisher could almost be considered "fire" and forget.**

Although ZMart has a lot of sales transactions for simplicity, we are going to consider the purchase of a single item 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}" (we've escaped the double quotes so we can represent the JSON as a String literal in Java). With the data in this format we use a `Producer` to send the data to our Kafka cluster:

**Listing 2.7 Producer 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", "org.apache.kafka.common.serialization.StringSerializer");
properties.put("acks", "1");
properties.put("retries", "3");
properties.put("compression.type", "snappy");
properties.put("partitioner.class", PurchaseKeyPartitioner.class.getName());   (1)


Producer<String, String> producer = new KafkaProducer<>(properties);   (2)
ProducerRecord<String, String>  record = new ProducerRecord<>("transactions",
    "123447777", "{\"item\":\"book\",\"price\":10.99}");   (3)

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

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

(1) Properties for configuring a producer.

(2) Creating the KafkaProducer.

(3) Instantiating the ProducerRecord.

(4) Building a Callback.

(5) Sending the record and setting the returned Future to a variable.

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

The `Producer.send` method depicted here takes a `Callback` instance. Once the lead broker acknowledges the record, the producer fires `Callback.onComplete` method.

Only one of the arguments will be non-null. In this case, we are only concerned with printing out the stacktrace in the event of error. The returned `Future` yields a `RecordMetadata` object once the server acknowledges the record.

**NOTE** **What's a Future?**
In the previous example, we discussed how the `Producer.send` method returns a `Future` object rather casually. A `Future` object represents the result of an asynchronous operation. More importantly, a `Future` gives you the option lazily retrieve asynchronous results vs. waiting for completion. For more information look here - link:https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.htr

When creating the `KafkaProducer` instance above, we passed a `java.util.Properties` parameter containing the configuration for the producer. While the configuration of a `KafkaProducer` is not complicated, there key properties to consider when setting up the producer. It's in these settings where we would specify a custom partitioner like the one demonstrated before.

There are too many properties to cover entirely here, so we will include the ones presented from the example above. You are encouraged to review the full producer configurations.

### 2.5.1 Producer Properties

**Listing 2.8 Bootstrap Servers**

```
bootstrap.servers is a comma separated list of host:port values.
```

Eventually, the producer will use all brokers in the cluster, this initial list for initially connecting to the cluster.

**Listing 2.9 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 we need to provide Kafka the correct serializers to convert objects to byte arrays before sending across the wire.

**Listing 2.10 Acks**

```
acks are 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 or 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 does not wait for any followers to acknowledge committing the record. Using a 0 means the producer will not wait for any acknowledgment, mostly fire and forget.

**Listing 2.11 Retries**

```
retries if sending a batch results in a failure, the number of times
to attempt to re-send
```

Retries specify how many times to attempt to send a record when met with a failure. 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 then a failed record sent after a re-try.

**Listing 2.12 Compression Type**

```
compression.type is used to determine what compression algorithm to apply if any
```

If set, the "compression.type" instructs the producer to compress a batch before sending. Note that it's the entire batch that is compressed and not individual records.

**Listing 2.13 Partitioner Class**

```
partitioner.class name of class implementing Partitioner interface
```

The `partitioner.class` included for completeness and related to our custom partitioner discussion in the section on partitions in Kafka.

### 2.5.2 Specifying Partitions and Timestamps

When creating a `ProducerRecord`, we have the option of specifying a partition and or a timestamp. Instantiating the `ProducerRecord` in the above example is just one of 4 overloaded constructors. We could have used other constructors that allow for setting a partition and timestamp or just a partition.

**Listing 2.14 ProducerRecord Constructors**

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

### 2.5.3 Specifying a Partition

In a previous section, we discussed the importance of partitions in Kafka. We also discussed how the `DefaultPartitioner` works and how we can supply our custom partitioner. So why would you explicitly set the partition? While there are a variety of business reasons why you would do so, here's one example.

You have keyed data coming in, but it does n't matter that the records go to the same partition as the consumers have logic to handle any values that the key might contain. Additionally, the distribution of the keys might not be even, so you want to ensure all partitions receive roughly the same amount of data. Here's a rough implementation of how you could do this:

**Listing 2.15 Manually Setting the Partition**

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

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

Here we are using the `Math.abs` call, so we don't have to keep track of the value of our integer if it goes beyond `Integer.MAX_VALUE`;

**NOTE** **What's an AtomicInteger?**
An `AtomicInteger` belongs to the java.util.concurrent.atomic package which comprises classes supporting lock-free thread-safe operations on single variables. For more information can be found here - docs.oracle.com/javase/8/../atomic/package-summary.html

### 2.5.4 Timestamps in Kafka

Kafka version 0.10 added timestamps to records. You set the timestamp when creating a `ProducerRecord` via the overloaded constructor call `ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)`. If you don't set one, then the `Producer` sets a timestamp (current wallclock time) before sending the record to the Kafka broker.

Accompanying the addition of timestamps is the broker configuration setting, `log.message.timestamp.type`. This configuration can be set to either `CreateTime` or `LogAppendTime` (`CreateTime` is the default). Like many other broker settings, the value configured on the broker applies to all topics as a default value. But when creating a topic, you can specify a different value applying only to that topic.

If you specify `LogAppendTime`, the broker will overwrite the timestamp to 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? Using the setting `LogAppendTime` is considered "processing time" and a setting of `CreateTime` is considered "event time". So it depends on your business requirements. You'll need to decide does it matter more to you when Kafka processed the record, or when the actual event occurred.

We will see the important role timestamps have regarding controlling data flow in

Kafka Streams in later chapters. Now we will discuss the other side of the equation, consuming messages.

## 2.6 Reading Messages with Consumers

Now let's move to the "other side" of producing, consuming messages. Now we in an internal application at ZMart and we are building a prototype application to show the latest sales statistics. Again even though ZMart has a large sales volume, we are just going to consider consuming the message we sent in the `Producer` example above. Since this prototype is in it's earliest stages, all we are going to do at this point is consume the message and print out the information to the console.

The `KafkaConsumer` is the client used to consume messages from Kafka. The `KafkaConsumer` class is straight forward to use, but there are a few operational considerations to take into account. Since Kafka Streams requires Kafka version 0.10 or higher, we will only discuss the new consumer that was part of the Kafka 0.9 release. Below we have updated our ZMart architecture diagram to highlight the where the consumer plays a role in the data flow.

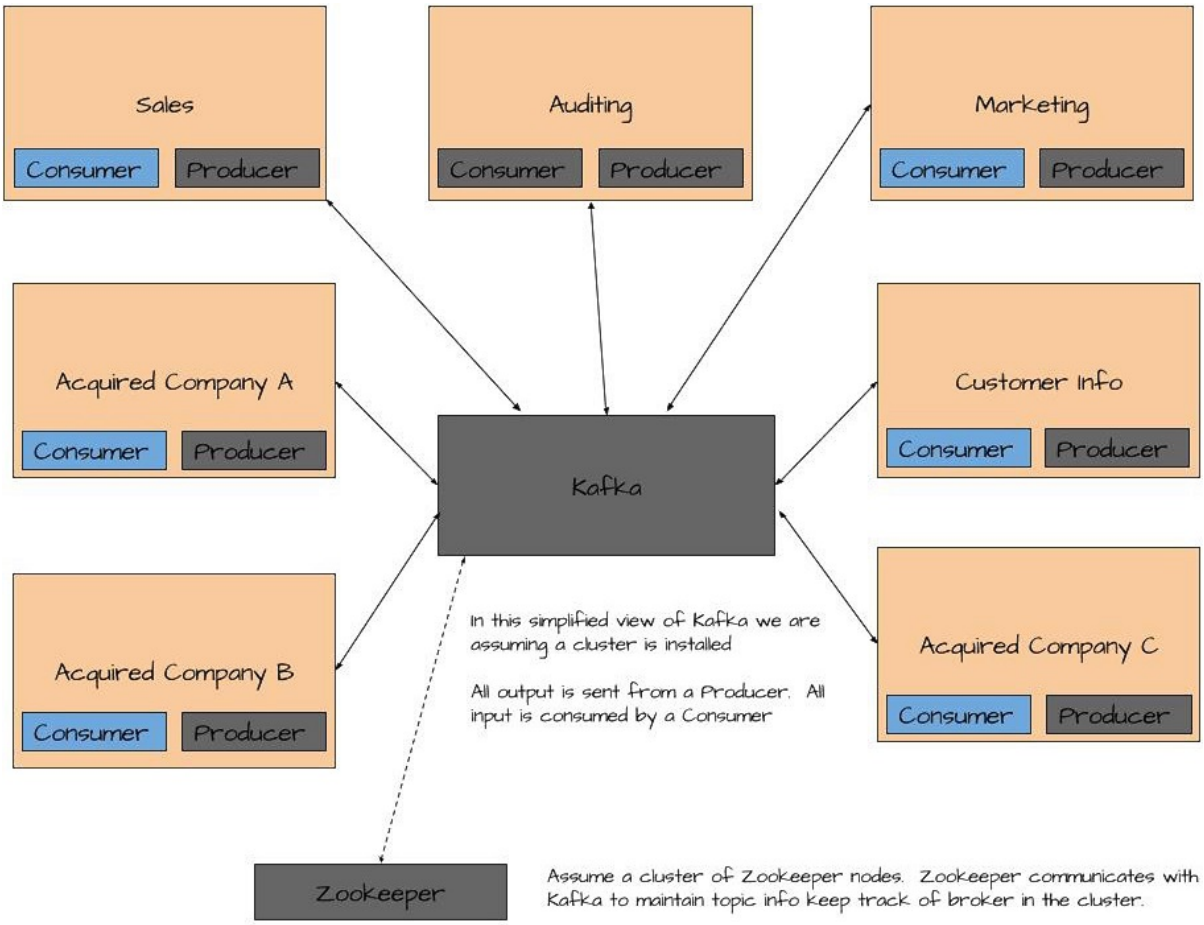



**Figure 2.15 Here are the consumers that read messages from Kafka. Just as producers have no knowledge of the consumer, consumers read messages from Kafka with no knowledge of who produced the message.**

### 2.6.1 Managing Offsets

While the `KafkaProducer` is essentially stateless, the `KafkaConsumer` manages some state by periodically committing the offests of messages consumed from Kafka. You'll recall from our previous conversation that offsets uniquely identify a message and represent its starting position in the log. Consumers will need to commit the offsets of messages they have received periodically. Committing an offset has two implications for a consumer:

1. Committing implies the consumer has fully processed the message
2. It also represents the starting point for that consumer in the case of failure or a re-start.

If you have a new consumer instance or some failure has occurred, and the last committed offset is not available, where the consumer will start from depends on your configuration.

- `auto.offset.reset="earliest"` retrieves messages starting at the earliest available offset, implying any messages that have not yet been removed by the log management process.
- `auto.offset.reset="latest"` starts retrieving messages from the latest offset, essentially only consuming messages from the point of joining the cluster.
- `auto.offset.reset="none"` no reset strategy specified, the broker throws an exception back to the consumer.

In the image below you can see the impact of choosing an auto offset reset setting. By selecting "earliest," you receive messages starting at offset one. If you choose "latest", you'll get a message starting at offset 11.

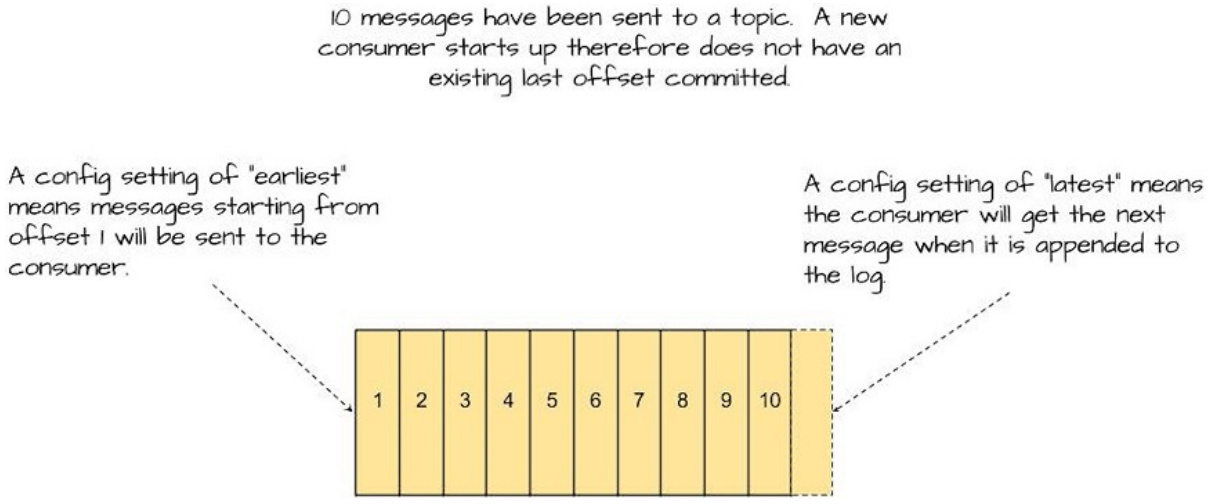



**Figure 2.16 Here is the graphical representation of "earliest" vs. "latest". As you can see, a setting of "earliest" will give you everything you have missed (any message not yet deleted), and "latest" means you will wait for the next available message to arrive.**

Next, we need to discuss the options we have for committing offsets, either automatically or manually.

### 2.6.2 Automatic Offset Commits

Automatic offset commits are enabled by default and is represented by the `enable.auto.commit` property. There is a companion configuration option `auto.commit.interval.ms` which specifies how often the consumer commits offsets (default value is 5 seconds). You should take care when adjusting this value, too small a value increases network traffic and to large a value could lead to large amounts of repeated data in the case of a failure before a commit occurs.

### 2.6.3 Manual Offset Commits

Within the category of manually committing offsets, there are two types: synchronous and asynchronous.

**Listing 2.16 Synchronous Commits**

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

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

#### ASYNCHRONOUS COMMITS

There are analogous `consumer.commitAsync()` methods that as the name implies are completely asynchronous and return immediately. There is an option with 2 of the `consumer.commitAsync` methods to provide an `OffsetCommitCallback` object, called when the commit had concluded either successfully or with an error. Providing a callback instance allows for asynchronous processing/error handling.

The advantage of using manual commits is that it gives you direct control over when a record is considered processed.

### 2.6.4 Creating the Consumer

We create a `Consumer` in a similar way to creating the producer. We supply configurations in the form a `java.util.Properties` object and get back a `KafkaConsumer` instance. The consumer instance then subscribes to topic from a supplied list of topic names or a regular expression. Typically we run the `Consumer` in a loop where we poll for a period specified in milliseconds.

A `ConsumerRecords<K, V>` object is the result of the polling. `ConsumerRecords` implements the `Iterable` interface and the object returned from each call to `next()` returns a `ConsumerRecord` object, containing metadata about the message in addition to the actual key and value.

After we have exhausted all of the `ConsumerRecord` objects returned from the last call to `poll` we 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 needed to be shut down and restarted (this is where committed offsets come into play as on reboot the consumer will pick up where it left off).

### 2.6.5 Consumers and Partitions

What we just described is how one `Consumer` works, but remember we may have multiple partitions per topic. While it is possible to have one `Consumer` reading from multiple partitions, it is standard practice to have a thread pool where the number of threads is equal to the number of partitions. The only change from the scenario above is that for each thread in the pool a `Consumer` instance is created, assigning each `Consumer` to one partition, maximizing throughput.

If a `Consumer` fails, the lead broker assigns its partitions to another active consumer.

**NOTE** In this example, we show subscribing a `Consumer` one topic. But this is for demonstration purposes only; you can subscribe a `Consumer` to an arbitrary number of topics.

The lead 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. It is probably preferred to have your consumers spread out across a few machines. In the case of a failure of one machine, the lead broker assigns topic-partitions to consumers on good machines.

### 2.6.6 Rebalancing

What we have described in the previous section is termed "rebalancing." Topic-partition assignments to a `Consumer` are not static, but very dynamic in nature. As you add consumers to the group, some of the current topic-partition assignments are taken from active consumers and given to the new consumers. And as we discussed just a moment ago, consumers leaving the group for whatever reason will have their topic-partition assignments reassigned to other consumers.

### 2.6.7 Finer Grained Consumer Assignment

In the previous section "Consumers and Partitions" we described the situation of using a thread pool and subscribing multiple consumers (belonging to the same consumer group) to the same topic(s). Although Kafka will balance the load of topic-partitions across all consumers, the assignment of the topic and which partition is not deterministic. We don't know what topic-partition coupling(s) each consumer will receive.

The `KafkaConsumer` has methods that allow for subscribing to a particular topic and partition only.

**Listing 2.17 Subscibing to Individual 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 tradeoffs to consider with using manual topic-partition assignment:

- Any failures will not result in topic-partitions to be re-assigned even for consumers with the same group id. Any needed changes in assignment require another call to `consumer.assign`.
- The group specified by the consumer is still used for committing. However, since each consumer will be acting independently, it is a good idea to give each consumer a unique group id when doing the manual topic-partition assignment.

### 2.6.8 Consumer Example

Here is the `Consumer` code for our ZMart prototype that consumes transactions and prints them to the console:

**Listing 2.18 ThreadedConsumerExample**

```
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("test-topic"));   (2)
                while (!doneConsuming) {
                    ConsumerRecords<String, String> records = consumer.poll(5000);   (3)
                    for (ConsumerRecord<String, String> record : records) {
                        String message = String.format("Consumed:
                            key = %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) Building consumer thread.

(2)

Subscribing to the topic.

3. Polling for 5 seconds.
4. Printing formatted message.
5. Closing the consumer regardless, will leak resources otherwise.

This example has left out other sections of the class for clarity; this example is not meant to stand on its own. You'll find the full example in the accompanying source code with the book.

## 2.7 Installing and Running Kafka

You will be using the .10.3.0 version of Kafka (the most recent as of the writing of this book). As Kafka is a Scala project, each release comes in two versions; one for Scala 2.11 and another for Scala 2.12. We will be using the 2.11 Scala version of Kafka in this book. Since the examples and features I present are from the 0.10.3 version of Kafka (unreleased at the time of writing), I have included in the source code repository the 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 source code repo in the "kafka_dist" folder somewhere on your machine.

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

### 2.7.1 Kafka Local Configuration

To run locally on your machine, Kafka requires minimal configuration if you choose the accept the default values. By default Kafka will use port 9092 and zookeeper will use port 2181. Assuming you have no applications already using those ports, you are all set. Kafka will write it's logs to /tmp/Kafka-logs and zookeeper uses /tmp/zookeeper for its log storage. Depending on your machine you may need to change permission/ownership of those directories or 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 would rather use. In the same directory open the zookeeper.properties file and change the "dataDir" entry. Later on configuring Kafka will be covered in detail, but this is all the configuration you will do for now. Keep in mind these "logs" are the actual data used by Kafka and zookeeper to their work, and not the application level logs that track the application's behavior. The application logs are found in the <kafka-install-dir>/logs directory.

### 2.7.2 Running Kafka

Kafka is very simple to get started. Since Zookeeper is essential for the Kafka cluster to function properly (zookeeper determines the lead broker, holds topic information, performs health checks on cluster members, etc.), you will need to start zookeeper before starting Kafka.

> **NOTE** **Kafka Directories**
> From now on all directories referenced will assume you are working in directory <kafka-install-dir>/ If you are 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 will see a lot of information run on the screen, and it should end up looking something like this:

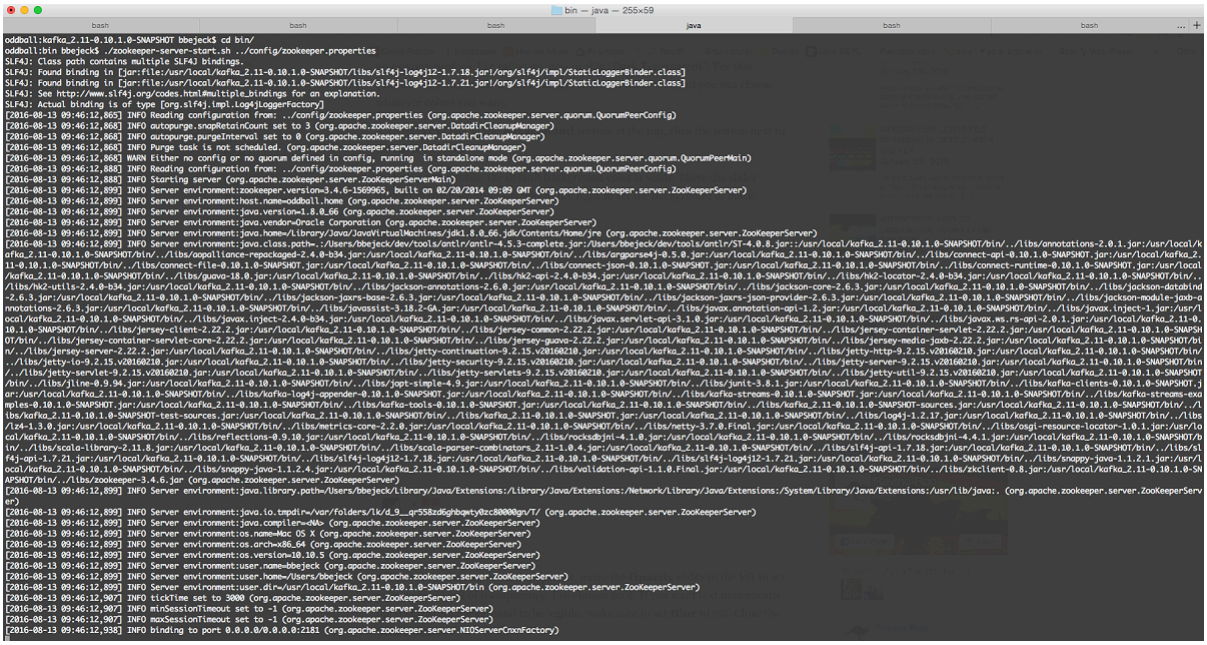



**Figure 2.17 Output visible on the console when starting Zookeeper starts up.**

#### STARTING KAFKA

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

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

again you will see text scroll by on the screen, and when Kafka has fully started you should see something similar to this:

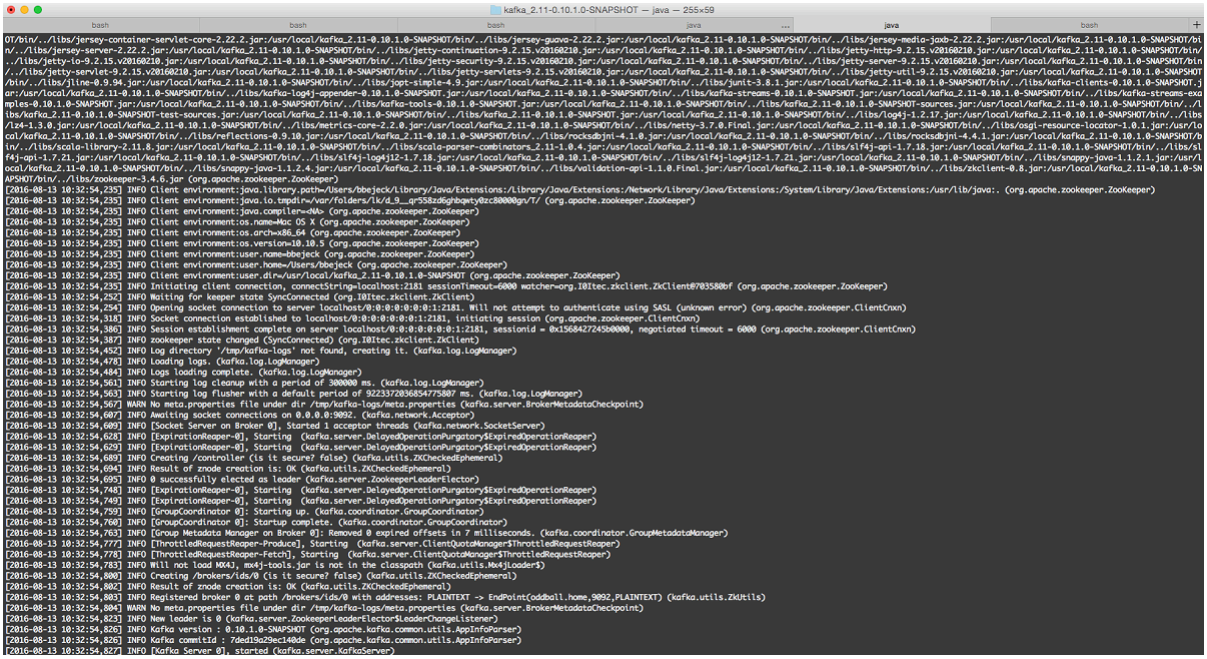



**Figure 2.18 Out put from Kafka when starting up.**

> **NOTE** **Reverse the Order on Shutdown**
>
> We mentioned why Zookeeper is essential for Kafka to run. Therefore it is important to reverse the order when shutting down; that is, stop Kafka first then stop zookeeper. To stop Kafka, you can enter control-c from the terminal Kafka is running in or run Kafka-server-stop.sh from another terminal. The same goes for zookeeper except for the script to shutdown zookeeper is zookeeper-server-stop.sh

## *2.7.3 Sending Your First Message*

Now that you have Kafka up and running, it is time to use Kafka what is meant to do, sending and receiving messages. Before you send a message, you will need to define a topic for a producer to send a message.

### YOUR FIRST TOPIC

Creating a topic in Kafka is very simple; it is just a matter of running a script with some configuration parameters. Although the configuration is an easy task, the settings you provide have broad performance implications. By default Kafka is configured to auto-create topics, meaning if you attempt to send or read a non-existent topic, the Kafka broker will create one for you (using default configurations in the server.properties file). However, this is rarely a good practice to rely on the broker to create topics, even in development. The issue is the first produce/consume attempt will fail, as it takes time for the metadata about the topic's existence to propagate. So be sure always proactively to create your topics.

### CREATING A TOPIC

To create a topic, you will 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
    --partitions 1 --zookeeper localhost:2181
```

When the script executes, you should see something similar to this in your terminal:

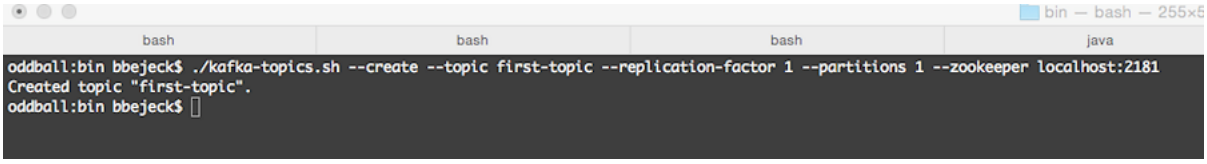



**Figure 2.19 Here are the results from creating a topic. It's important to create your topics ahead of time so you can supply your topic specific configurations. Otherwise, auto created topics use default configurations or the configuration in the server.properties file.**

Most of the configuration flags in the above command are obvious, but let us quickly review two of them, replication-factor and partitions. The replication-factor determines how many copies of the message the lead broker distributes in the cluster. In this case, there will be no copies 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 almost always want a replication factor of 2-3 to provide data availability in the face of machine failures. Partitions are the number of partitions that the topic will use. Again just one partition is fine here, but depending on the load, you will certainly want more partitions. Determining the correct number of partitions is not an exact science and strategies, and pros/cons for determining the number of partitions will be covered later in this chapter.

### SENDING A MESSAGE

Sending a message in Kafka requires writing a `Producer` client, but fortunately, Kafka comes with a handy script called kafka-console-producer that allows you to send a message from a terminal window. You use the console producer in this example, but we cover the `Producer` client in some detail in section 5 of this book. To send your first message run the following command:

```
bin/kafka-console-producer.sh --topic first-topic
    --bootstrap-server localhost:9092
```

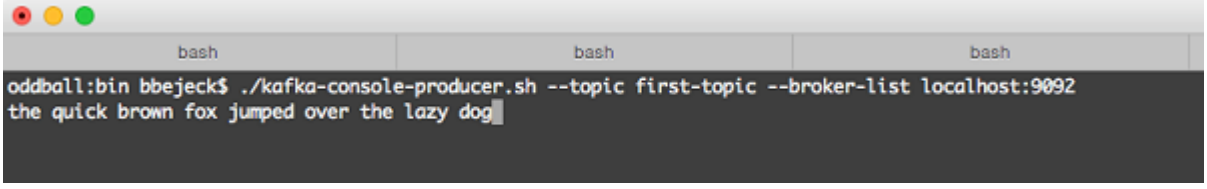



**Figure 2.20 The Console Producer is a great tool to quickly test your configuration and to ensure the end to end functionality.**

There are several options for configuring the console producer, but for now, you will 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 above command, you just need to enter some text and hit enter. You can send as many messages as you like, but for demo purposes here you will type a single message "the quick brown fox jumped over the lazy dog", hit the enter key and then enter control-c to exit the producer.

#### READING A MESSAGE

Kafka also provides a console consumer for reading messages from the command line. Both the console producer and consumer are very handy tools for troubleshooting or doing quick prototyping. Running the console consumer works in similar fashion, once started it will keep reading messages from a particular topic until the script is stopped by you (again control-c). To launch the console-consumer run this command:

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

After starting the console consumer, you should see this in your terminal:

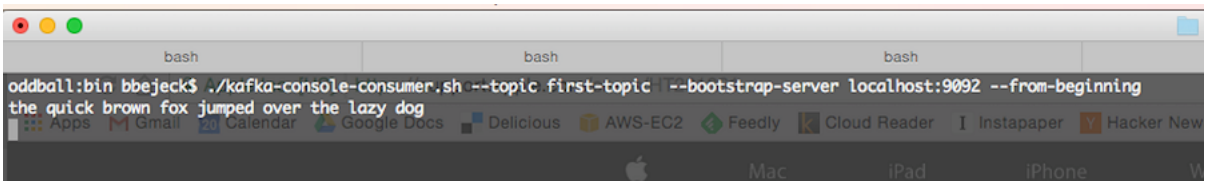



**Figure 2.21 The Console Consumer is a super handy tool to quickly get a feel if data is flowing and messages contain the expected information.**

The "--from-beginning" parameter specifies you will receive any message not deleted from that topic. Since the console consumer will not have any committed offsets, without the "--from-beginning" setting, you will only get messages sent after the console consumer has started.

You have just completed a whirlwind install of Kafka and producing/consuming your first message. Now it's time go back where you left off in this chapter to learn the details of how Kafka works!

## 2.8 Conclusion

We have covered a lot of ground in this chapter. As I stated before, since this book is about Kafka Streams this chapter is not intended to be a thorough treatment of Kafka itself, but a solid introduction to give those readers new to Kafka an understanding of how Kafka works. However, there are key points that we learned in this chapter that you need to take with you on your journey through the rest of this book:

- Kafka is message broker that receives messages and stores them in a way that makes is 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 provide a means for grouping messages with the same key 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 modulo the number of partitions for partition assignment.
- Consumers are what you use to read messages from Kafka.
- Consumers that are part of the same consumer group are given topic-partition allocations in an attempt to distribute messages evenly.

In the next chapter, we start looking at Kafka Streams with a concrete example from the world of retail sales. While 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.

# 3 Developing Kafka Streams

In this chapter

- Hello World Kafka Streams
- Introducing the KStreams API
- The ZMart Kafka Stream application in depth
- Splitting an incoming stream into multiple streams
- Advanced KStreams API methods

In the first chapter, learned about the Kafka Streams library. We talked about building a topology of processing nodes, or a graph used to transform data as it is streaming into Kafka. In this chapter, we will show you how to create this processing topology by using the KSteams API. The KStreams API is important to learn because it is the core tool with which you build Kafka Streams applications. By learning the KStream API you will gain the knowledge necessary to assemble Kafka Streams programs, but more importantly, you will gain a deeper understanding of how the components work together and how they can be used to achieve your stream processing goals.

## 3.1 KStreams API

The KStreams API is a high-level DSL that enables you to build Kafka Streams applications quickly. The API is very well thought out, and there are methods to handle most stream processing needs out of the box, so creating a sophisticated processing program can be achieved without much effort. At the heart of the API is the `KStream` object which is used to represent the streaming key-value pair records.

Most of the methods in the KStreams API 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 comprised of single method interfaces allowing for using Java 8 lambda expressions. Taking the previous two factors into account and you can see the simplicity and ease with which you can build a Kafka Streams program.

**NOTE** **What's a Fluent Interface?**
Back in 2005 Martin Fowler and Eric Evans developed the concept of the fluent interface (martinfowler.com/bliki/FluentInterface.html). A fluent interface is one where the return value of a method call is the same instance which originally called the method. This approach is very useful when constructing objects with several parameters, for example, Person.builder().firstName("Beth").withLastName("Smith").withOccupation('
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 is also a lower-level API, the Processor API that is not as succinct as the KStreams API but allows for more control. We will cover the Processor API in a later chapter. With the introduction out of the way, let's dive into the requisite hello world program for Kafka Streams.

## 3.2 Kafka Streams Hello World

For our first example, we will deviate from the problem outlined in chapter 1 to a more simple use case. I did this so you can quickly get off the ground and see how Kafka Streams works. We will get back to the problem statement from chapter 1 later for more realistic, concrete examples.

Your first program will be a toy application that takes incoming messages and upper-cases the text of the message, effectively yelling at anyone who reads the message. The title of this application is called the "Yelling Application."

Before diving into the code let's take a look at the processing topology you will assemble for the first application. We will follow the same pattern from chapter 1, where we built up a processing graph topology, where each node in the graph has a particular function. The main difference will be in the simplicity of the graph. The following image shows the topology we will build.

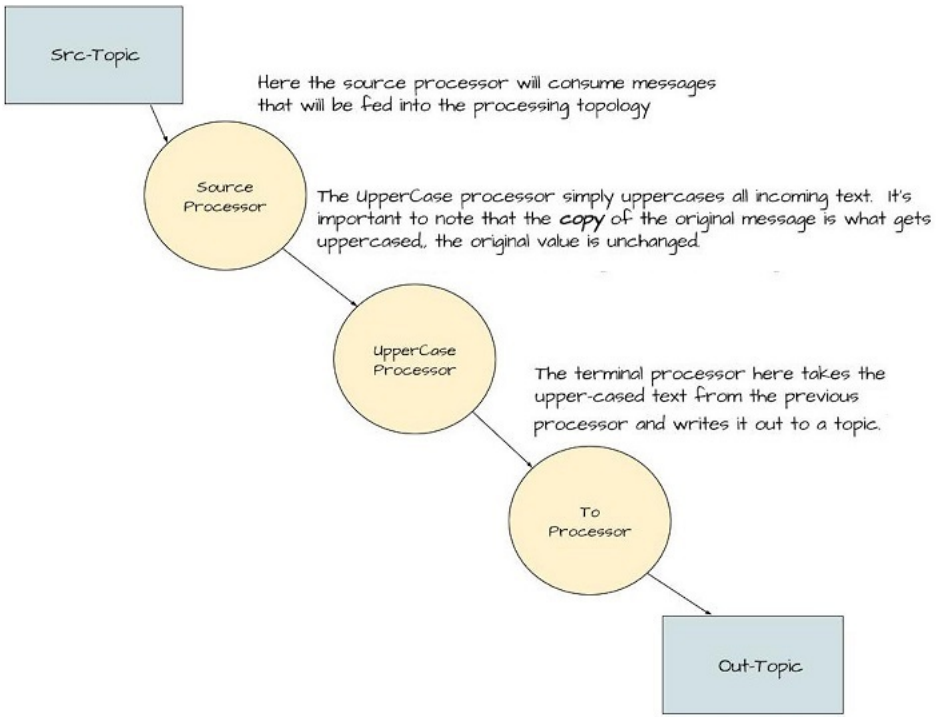



**Figure 3.1 Graph or Topology the our first Kafka Streams application, the 'Yelling' App**

As you can see from the image, we are building a very simple processing graph. So simple in fact it resembles a linked list of nodes more than the typical tree-like structure of a graph. But there is enough here to give us strong clues as to what expect in the code. There will be a source node, a processor node transforming incoming text to all uppercase, and finally, a processor writing results out to a topic.

While this is a trivial example, the code shown here is representative of what you will see in our Kafka Streams programs. For most of our examples you will see a familiar structure:

1. Define the configuration items
2. Create any Serdes instances either custom or pre-defined.
3. Build the processor topology.
4. Create a KStreams instance and start the KStreams running.

When we get into more advanced examples, the principal difference will be in the complexity of processor topology. With that in mind let's take a walk through building our 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 records from a topic which will flow through the application.

We'll present the graph with the relevant point highlighted, so it will be easier to follow how the code relates to the processing topology. Here in this image, we've added the source node indicated by the highlighted portion of the graph.

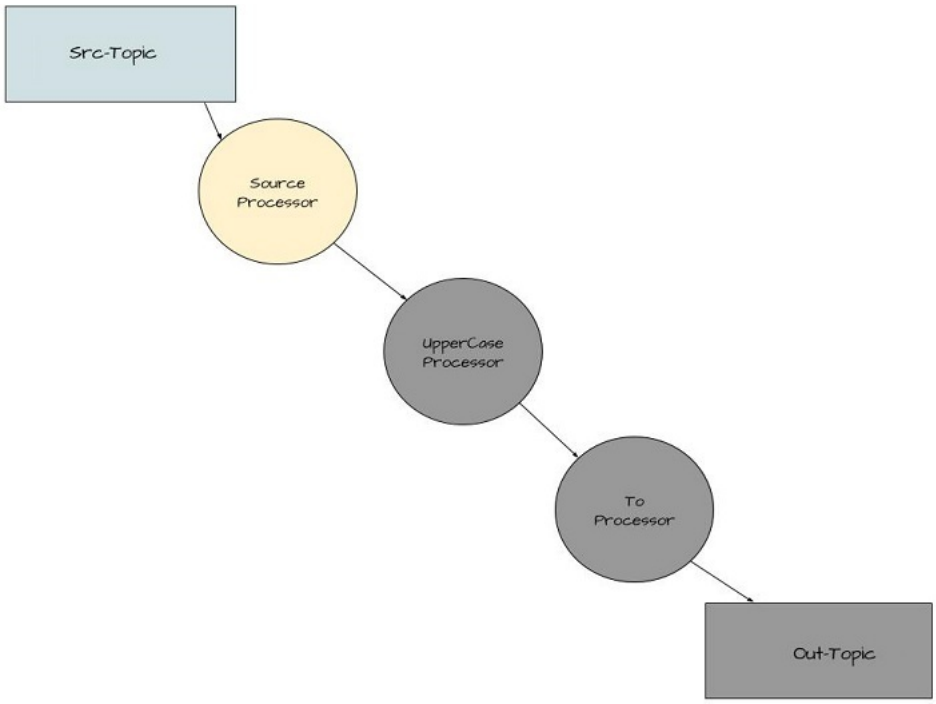



**Figure 3.2 Creating the Source Node of the 'Yelling' App**

The code we use behind the graph image follows:

**Listing 3.1 Defining the Source for the Stream**

```
KStream<String,String> simpleFirstStream =
    kStreamBuilder.stream(stringSerde, stringSerde, "src-topic");
```

With this line of code, you are creating the source or parent node of the graph. The source node of a topology consumes records from a given topic or topics. In this case, it's aptly named "src-topic." The KStream instance `simpleFirstStream` will start consuming messages written to the "src-topic" topic.

We have a source node for our application, but now we need to attach a processing node to make use of the data. Our next step is to add a processing node as shown in the following image:

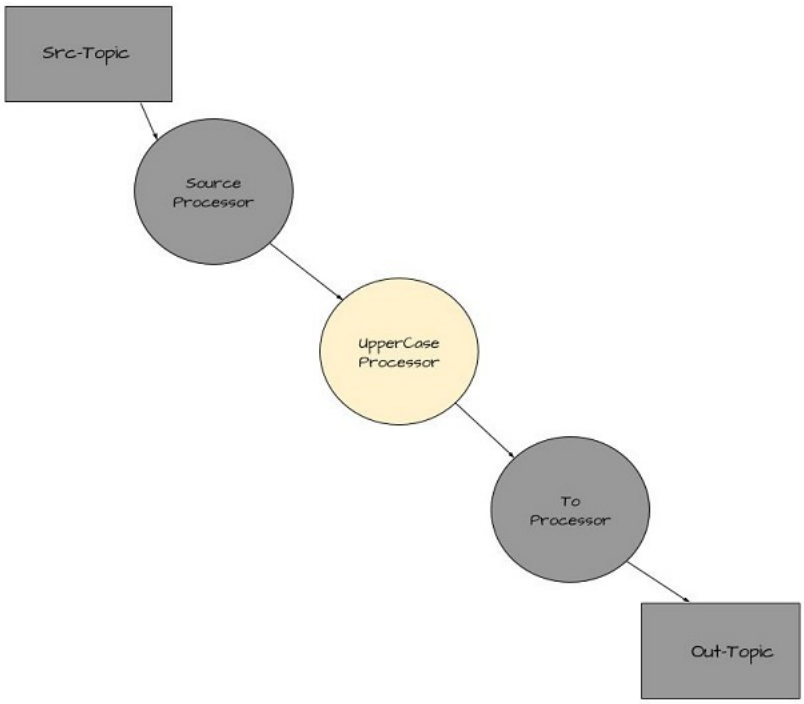



**Figure 3.3 Adding the UpperCase Processor to the 'Yelling' App**

The code used to attach the processor (a child node of the source node) looks like this:

**Listing 3.2 Mapping Incoming Text to Uppercase**

```
KStream<String, String> upperStream = simpleFirstStream.mapValues(String::toUpperCase);
```

With this line, you are creating another KStream instance that is a child node of the parent node. By calling the `KStream.mapValues' function you are creating a new processing node whose inputs are the results of going through the `mapValues` call. It's important to remember that `mapValues` does not modify the original values. The `upperStream` instance receives transformed copies of the initial value from the `simpleFirstStream.mapValues` call. In this case, it's upper-case 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. And this is what we've done here with `String::toUpperCase`, which is a method reference, an even shorter form of a Java 8 lambda expression.

**NOTE** While there many Java 8 tutorials available for lambda expressions and method references good starting points are https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html and https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html respectively.

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

Using lambda expressions vs. concrete implementations is a pattern we will see over and over with the KStreams API. Since most of the methods expect types that are single method interfaces, you can easily use Java 8 lambdas.

So far our Kafka Streams application is consuming records and then transforming them to upper-case. The final step for us to take is adding a processor/node that writes the results out to a topic. The following image shows us where we are in the construction of the topology:

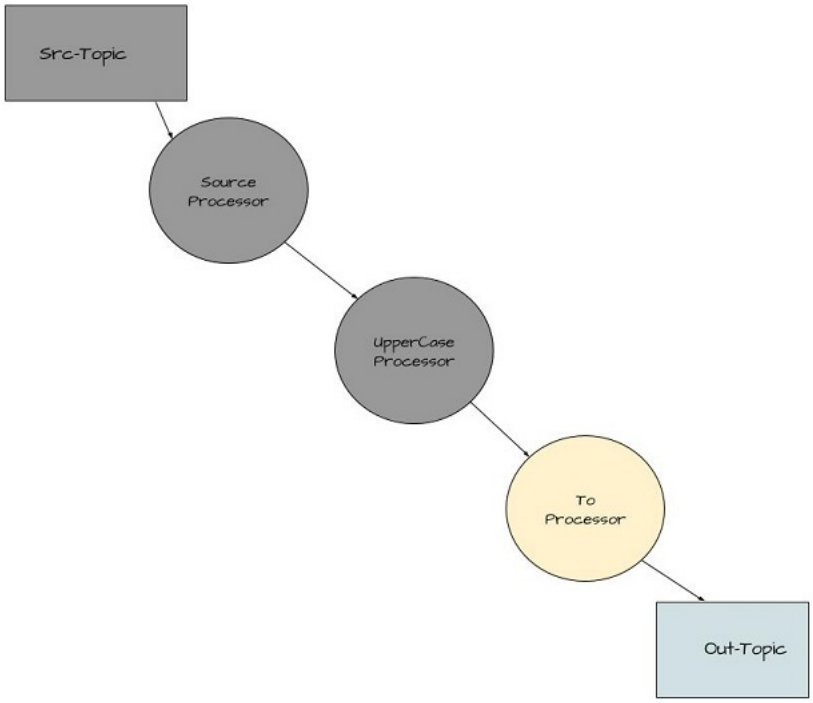



**Figure 3.4 Adding Processor for Writing 'Yelling' App Results**

**Listing 3.3 Creating a Sink Node**

```
upperStream.to(stringSerde, stringSerde, "out-topic");
```

Now we are adding the last processor in the graph. By calling the `KStream.to` method you are creating a terminal node in the topology. To be more precise, you are creating a sink node where the `upperStream` processor writes the key-value pairs to a topic named "out-topic." Any KStream method that has a return type of void is a terminal node and will end stream processing at that point, but it may not necessarily be a sink node.

Again we see the pattern of providing the required serde instances needed for serialization/deserialization when reading from or writing to a Kafka topic.

**NOTE** One note about serde usage in Kafka Streams. There are several overloaded methods in the KStream API where you don't need to specify the key/value serdes. In those cases, the application will use the serializer/deserializer specified in the configuration instead.

In our example above we used three lines to build the topology.

**Listing 3.4 Building Processor Topology in Stages**

```
KStream<String,String> simpleFirstStream = kStreamBuilder.stream(stringSerde,
    stringSerde, "src-topic");
KStream<String, String> upperStream = simpleFirstStream.mapValues(String::toUpperCase);
    upperStream.to(stringSerde, stringSerde, "out-topic");
```

Having each step on an individual line was used to demonstrate each stage of the building process. However, all methods in the KStream API that don't create terminal nodes return a new KStream instance. Returning a `KStream` object allows for using the "fluent interface" style of programming referenced earlier. To demonstrate this idea, here's another way you could have constructed the yelling application topology:

**Listing 3.5 Building Processor Topology Using Fluent Style**

```
kStreamBuilder.stream(stringSerde,stringSerde,"src-topic")
    .mapValues(String::toUpperCase).to(stringSerde,stringSerde,"out-topic");
```

We just shortened our program from three lines to one without losing any clarity or purpose. From this point forward all of our examples will be written using the "fluent" interface style unless doing so will make the clarity of the program suffer.

We've built our first Kafka Streams topology. But we've glossed over the important steps of configuration and Serde creation which we'll address now.

### *3.2.2 Configuration*

While Kafka Streams is highly configurable with several properties you can adjust for your specific needs; our first example uses two configuration settings. Both of these are required as there are no default values provided. Attempting to start a Kafka Streams program without these two properties defined results in throwing a `ConfigException`.

**Listing 3.6 Setting Configuration Items**

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

- StreamsConfig.APPLICATION_ID_CONFIG : This property identifies your Kafka Streams application and must be a unique value for the entire cluster. The application id config 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 used to identify

clients connecting to Kafka uniquely. The group-id is used to manage the membership of a group of consumers all reading from the same topic and assures that many consumers can effectively read subscribed topics in the group.

- StreamsConfig.BOOTSTRAP_SERVERS_CONFIG : The bootstrap servers configuration can be a single hostname: port setting or multiple values comma separated. The value of this setting should be very apparent as it points the Kafka Streams application to the Kafka cluster.

We will cover several more configuration items as we explore more examples throughout the book.

### 3.2.3 Serde Creation

**Listing 3.7 Instantiating a String Serde**

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

This line from point 3 above is where you create the `Serde` instance required for serialization/deserialization of the key/value pair of every message and store it in a variable for use later on. The `Serdes` class provides default implementations for the following types:

- Strings
- Byte arrays
- Long
- Integer
- Double

The `Serde` class is extremely useful as it is used to contain the serializer and deserializer, which eliminates the repetition for 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 our next example, we will show how to create your `Serde` implementation to handle serialization/deserialization of more complex types.

Now let's take a look at the whole program you just put together.

**Listing 3.8 Hello World - The Yelling Application**

```
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();                    ❸

  KStreamBuilder kStreamBuilder = new KStreamBuilder();           ❹

  KStream<String,String> simpleFirstStream = kStreamBuilder.stream(stringSerde,
    stringSerde, "src-topic");                                    ❺

  KStream<String, String> upperStream =
    simpleFirstStream.mapValues(String::toUpperCase);             ❻

  upperStream.to(stringSerde, stringSerde, "out-topic");          ❼

  KafkaStreams kafkaStreams = new KafkaStreams(kStreamBuilder, streamingConfig);

  kafkaStreams.start();                                           ❽

  }
}
```

❶ Properties for configuring the Kafka Streams program,

❷ Creating the StreamsConfig with the given properties.

❸ Creating the Serdes used to serialize/deserialize keys and values.

❹ Creating the KStreamBuilder instance used to construct our processor topology.

❺ Creating the actual stream with a source topic to read from (Parent node in the graph).

❻ A processor using a Java 8 method handle (first child node in the graph).

❼ Writing the transformed output to another topic (last/terminal node in the graph).

❽ Kicking off the Kafka Stream thread(s).

In this section, you have constructed your first Kafka Streams application. Let's quickly review the steps taken as it's a general pattern we'll see in most of our Kafka Streams applications. Specifically, you took the following steps:

1. Created a StreamsConfig instance.
2. Then you created a Serde object.
3. You constructed a processing topology.
4. After putting all the pieces into place, you started the Kafka Streams program.

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

Next, we'll move on to a more complex example that will allow us to explore more of the Kafka Streams API. The example will be new, but the scenario is one you are already familiar with, the solution to the ZMart data processing goals.

## 3.3 Working with Customer Data

In chapter one, we discussed the new customer data processing requirements you have to help ZMart do business more efficiently. We demonstrated how to build a topology of processors that will work on purchase records as they come streaming in from transactions in ZMart stores. Here we present the completed graph again for review:

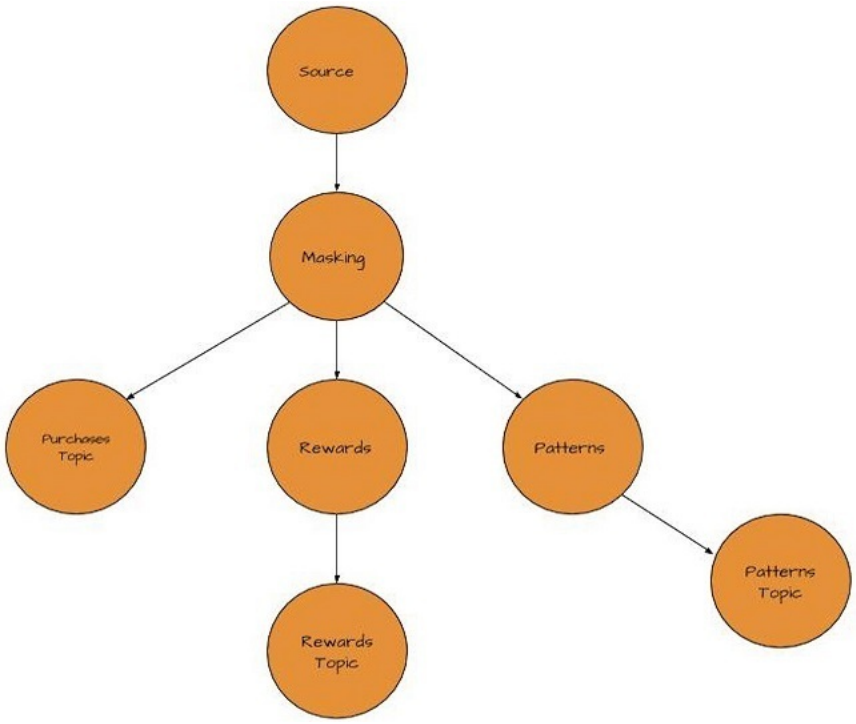



**Figure 3.5 Initial Complete Topology for ZMart Kafka Streams program**

While we're at it, let's briefly go over the requirements for our streaming program which also serves as a good description of what our program will do:

- All records need to have credit-card numbers protected, in this case masking the first 12 digits
- Extract the item(s) purchased and the zip code for determining purchase patterns in the company. This data will be written out to a topic.
- Capture the customer ZMart member number and amount spent and written to a topic. Consumers of the topic will use the data to determine rewards.
- Write the entire transaction out to topic to be consumed by a storage engine for ad-hoc analysis.

As in our first example, we will see the fluent interface approach combined with Java 8 lambdas when building the application. As this application is more involved than our first example, the use of these two techniques is pervasive. While sometimes it's clear the return type of a method call is a `KStream` object other times it may not be so clear. A key point to remember is that the majority of the methods in the KStream API return *new* `KStream` instances.

Now let's get to work building the streaming application that will satisfy your business requirements.

### 3.3.1 Building The Application

### 3.3.2 Constructing the Processors

Now let's dive into building the processing topology. To help us make the connection between the code and the processing topology graph we've created in the previous chapter, we will highlight the part of the graph that we are currently working on.

#### BUILDING THE SOURCE NODE

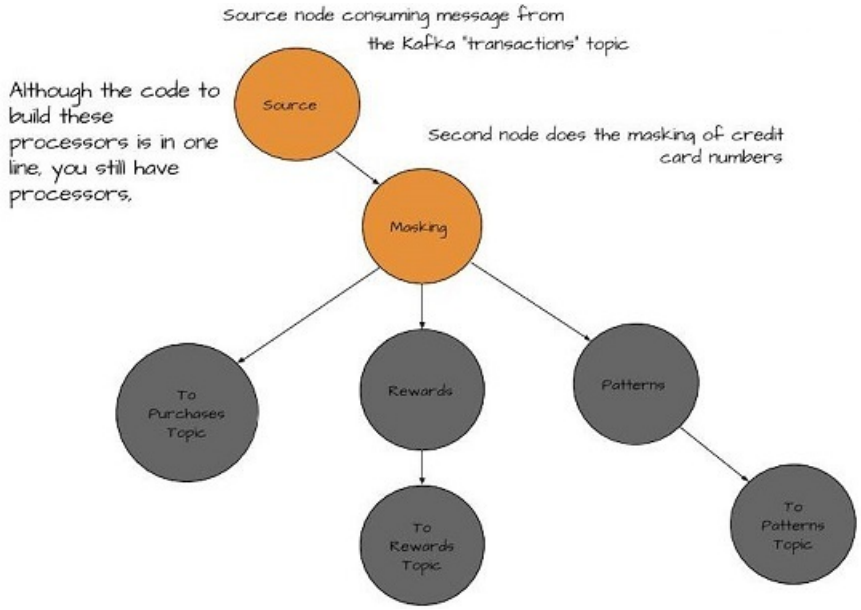



**Figure 3.6 The source processor consumes from a Kafka topic, but it feeds the masking processor exclusively, actually making it the source for the rest of the topology.**

**Listing 3.9 Building the Source Node and First Processor**

```
KStream<String,Purchase> purchaseKStream =
  kStreamBuilder.stream(stringSerde,purchaseSerde,"purchases")
              .mapValues(p -> Purchase.builder(p).maskCreditCard().build());
```

Here we are building the source node and first processor of the topology by chaining two calls of the KStream API together. While it should be fairly obvious what the role of the origin node is, the first processor in the topology is responsible for masking credit-card numbers to protect customer privacy.

You create the source node with a call to the `KStreamBuilder.stream` method using a default String Serde, a custom Serde for `Purchase` objects and the name of the topic that is the source of messages for the stream. In this case, we are only specifying one topic, but we could have provided a comma separated list of names or a regular expression to match topic names instead.

Here we chose to provide the serde instances, but we could have used an overloaded version only providing the topic name(s) 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 a to a new value, possibly of another type. In the example here the `KStreams.mapValues` returns an object of the same type (`Purchase`), but with a masked credit-card number.

You should take note that when using the `KStream.mapValues` method the original key is unchanged and is not factored into mapping a new value. If we had wanted to generate a new key-value pair or include the key in producing a new value, we would use the `KStream.map` method that takes a `KeyValueMapper<K, V, KeyValue<K1, V1>>` instance.

### HINTS OF FUNCTIONAL PROGRAMMING

An important concept to keep in mind with the 'map' or 'mapValues' functions is that they are expected to operate without side-effects. Meaning the functions do not modify the object or value presented as a parameter.

It's at this point we can get a sense of functional programming aspects in the KStreams API. While functional programming is a deep topic and a full discussion is beyond the scope of this book, we can briefly discuss two central principles of functional programming here.

The first would be to avoid state modification. If an object requires a change or update, we pass the object to a function, and a copy or entirely new instance is made containing the desired changes or updates. We have just seen an example of this by using `KStream.mapValues` to update the `Purchase` object having 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 we will frequently see working with the KStream API.

**SIDEBAR** **What is Functional Programming**

While there may be other definitions of functional programming, we will define it here in the following manner. Functional programming is a programming approach where functions are first class objects themselves. Furthermore, functions are expected to avoid providing side-effects, that is the modification of state or mutable objects.

### BUILDING THE SECOND PROCESSOR

Now we are going to go over building the second processor, listing number 3 from our example. The second processor is responsible for extracting pattern data sent to a topic used by ZMart to determine purchase patterns in regions of the country.

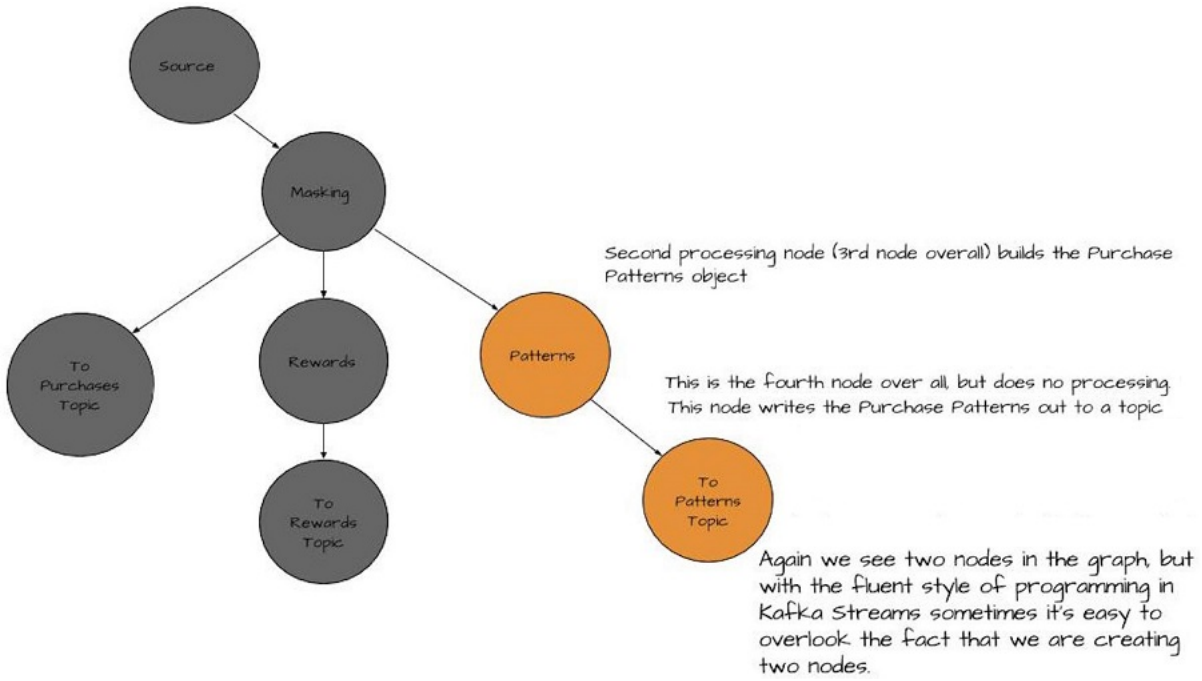



**Figure 3.7 Here we are adding the second processor, which build purchase pattern information. Additionally, we are adding a terminal node which writes the PurchasePattern object out to Kafka.**

**Listing 3.10 Code for the Second Processor and a Terminal node to write to Kafka**

```
purchaseKStream.mapValues(purchase ->
    PurchasePattern.builder(purchase).build()).to(stringSerde,
    purchasePatternSerde, "patterns");
```

Here we see the `purchaseKStream` processor using the familiar `mapValues` call to create a new `KStream` instance. Here this new `KStream` will start to receive `PurchasePattern` objects created as a result of the `mapValues` call.

One thing to notice here is that we didn't declare a variable to hold the reference of the new `KStream` instance as it isn't being used later in the topology. Here the purchase patterns processor forwards the records it receives to a child node of its own, defined by the method call `KStream.to` and using the previously built serdes and the name of the topic.

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 which is used to realize the data contained in a particular `KStream` instance to a Kafka topic. The `KStream.to` method also provides overloaded versions in which you can leave out the Serde parameters and use the default serdes defined in the configuration. There is one additional parameter the `KStream.to` method accepts; a `StreamPartitioner` that we will discuss next.

### BUILDING THE THIRD PROCESSOR

The third processor in the topology is the customer rewards accumulator node used that will be used to track purchases made by members of ZMart's preferred customer club. More specifically, the reward accumulator sends data to a topic consumed by applications at ZMart HQ to determine rewards as customers complete purchases.

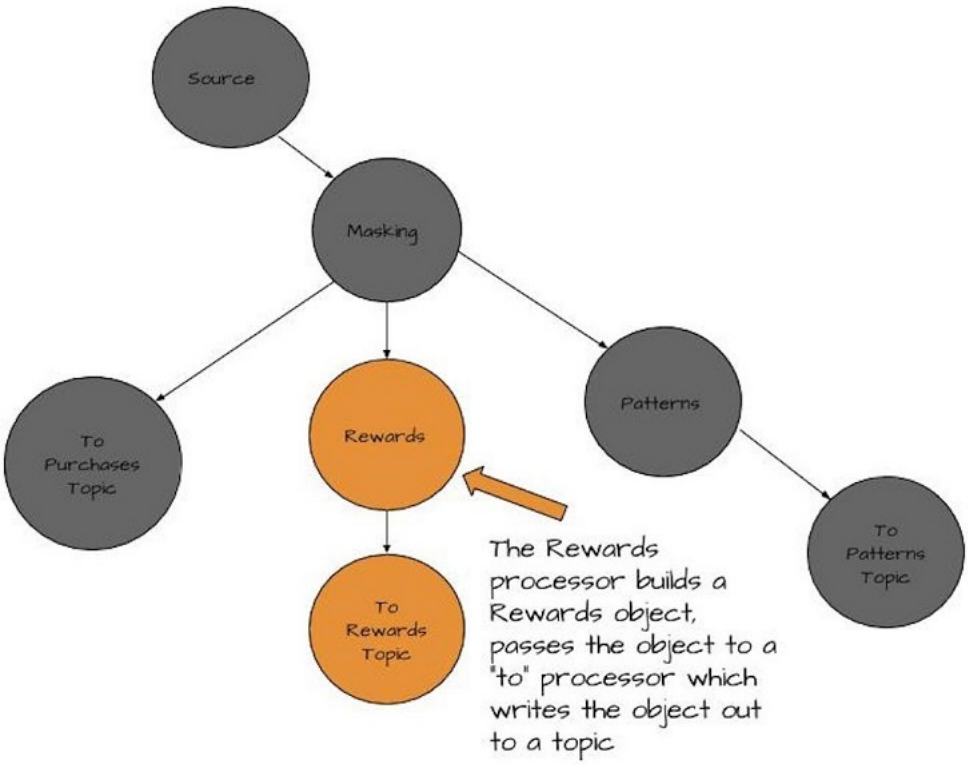



**Figure 3.8 At this point we are building the third processor which creates the RewardAccumulator object from the purchase data. We also are adding a terminal node to write the results out to Kafka. Each terminal node in the graph writes to separate topic.**

**Listing 3.11 Code for the Third Processor and Terminal node which writes to Kafka**

```
purchaseKStream.mapValues(purchase ->
    RewardAccumulator.builder(purchase).build()).to(stringSerde, rewardAccumulatorSerde,
    "rewards");
```

At this point in the topology, you are building 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 value to a new object type. We also attach a sink node to the reward accumulator so the results of the rewards KStream can be realized to a topic and used for determining customer reward levels.

### LAST PROCESSOR

Finally, we take the first KStream we created, `purchaseKStream`, and attach a sink node to write out the raw purchase records (credit-cards masked of course) out to a topic called "purchases". The purchases topic will be used to feed into a NoSql store such as Cassandra or Presto [8] to perform ad-hoc analysis.

Footnote 8 prestodb.io/

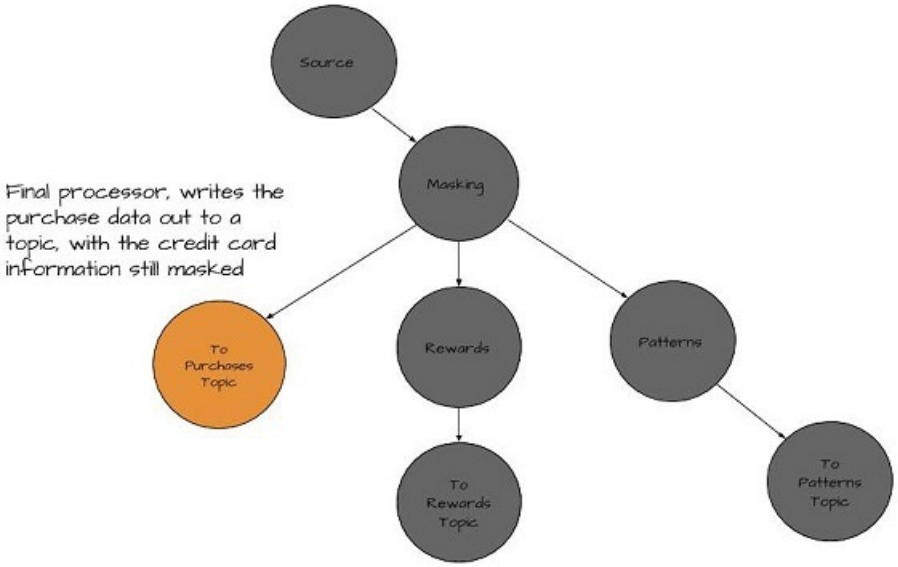



**Figure 3.9 This is the last node we're adding to our topology (at least at this point). This node writes out the entire purchase transaction out to a topic whose consumer is a NoSql data store such as MongoDB. The following graphic depicts where we are at this point in our application.**

## Listing 3.12 Final Processor

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

Now that we've built our application piece by piece let's look at the entire application. You'll quickly notice it's more complicated than our "hello world" example.

## Listing 3.13 ZMart Customer Purchase KStreams Program

```
public class ZMartKafkaStreamsApp {

    public static void main(String[] args) {

        StreamsConfig streamsConfig = new StreamsConfig(getProperties());

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

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

        KStreamBuilder kStreamBuilder = new KStreamBuilder();

        KStream<String,Purchase> purchaseKStream =
            kStreamBuilder.stream(stringSerde,purchaseSerde,"transactions")    (2)
                      .mapValues(p -> Purchase.builder(p).maskCreditCard().build());

        purchaseKStream.mapValues(purchase ->
            PurchasePattern.builder(purchase).build()).to(stringSerde,
            purchasePatternSerde, "patterns");                         (3)
```

```
        purchaseKStream.mapValues(purchase ->
            RewardAccumulator.builder(purchase).build()).to(stringSerde,

            rewardAccumulatorSerde, "rewards");                    4


        purchaseKStream.to(stringSerde,  purchaseSerde, "purchases");    5

        KafkaStreams kafkaStreams = new KafkaStreams(kStreamBuilder,streamsConfig);
        kafkaStreams.start();

    }
```

1. Creating Serde(s), the data format is in JSON.
2. Building the source and first processor
3. Building the PurchasePatterns processor
4. Building the RewardAccumultor processor
5. Building the storage sink, the topic used by the storage consumer.

**NOTE** We have left out some details here for clarity. Throughout the book code examples aren't necessarily meant to stand on their own. For the full the example, the reader is left to look at the accompanying source code.

As you can see this is a little more involved than our original introductory application, but it has a similar flow. Specifically, you still performed the following steps:

- A StreamsConfig instance.
- You build one or more Serde instances.
- The processing topology is constructed.
- Assemble all the component then start the KafkaStreams program.

While building the application we've mentioned using a `Serde` while not taking any time to explain why or how we create them. Let's take some time now to discuss the role of the `Serde` in a Kafka Streams application.

### 3.3.3 Creating a Custom Serde

Kafka transfers data in byte array format. Since the data format is JSON, we'll need to tell Kafka how to convert an object first into JSON then into a byte array when sending data to a topic. Conversely, we'll need to specify how to convert consumed byte arrays into JSON then into the object type our processors will use. This conversion of data to/from different formats is why we need to have a `Serde`. While there are some Serdes provided out of the box (String, Long, Integer, etc.) we'll need to create custom Serdes for our objects.

In our first example, we only needed a serializer/deserializer for Strings, and there is an implementation provided by the `Serdes.String()` factory method. In this example, however, we need to create custom Serde instances, as the type of the objects is arbitrary.

We will walk through building the `Serde` for the `Purchase` class. We won't cover making the other `Serde` instances as they follow the same pattern, just with different types.

Building a `Serde` requires implementations of the `Deserializer<T>` and `Serializer<T>` interfaces. Here are the implementations of the `Serializer` and `Deserializer` that we will use throughout the examples. Also, we have chosen to use the Gson library from Google to convert objects to/from JSON.

**Listing 3.14 The 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) Creating the Gson object

(2) Serializing an object to bytes

For serialization, we will first convert an object to JSON, then get the bytes from the string. To handle the conversions from/to JSON we are using Gson [9].

Footnote 9 github.com/google/gson

For our deserializing process we take different steps; create a new String from a byte array then use Gson to convert the JSON string into a Java object.

**Listing 3.15 The 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("deserializedClass");
        }
    }

    @Override
    public T deserialize(String s, byte[] bytes) {
         if(bytes == null){
             return null;
         }

         return gson.fromJson(new String(bytes),deserializedClass);  (3)

    }

    @Override
    public void close() {

    }
}
```

(1) Creating the Gson object.

(2) Instance variable of Class to deserialize.

(3) Deserializing bytes to an instance of expected Class.

Now going back to the lines at listing 1 :

**Listing 3.16 Creating a Serde**

```
JsonDeserializer<Purchase> purchaseJsonDeserializer =
   new JsonDeserializer<>(Purchase.class);  (1)
JsonSerializer<Purchase> purchaseJsonSerializer = new JsonSerializer<>();  (2)
Serde<Purchase> purchaseSerde =
   Serdes.serdeFrom(purchaseJsonSerializer,purchaseJsonDeserializer);  (3)
```

(1) Creating the Deserializer for the Purchase class

(2) Creating the Serializer for the Purchase class

(3) Creating the Serde for Purchase objects

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

We've covered a lot of ground so far in developing a Kafka Streams application. While we still have much more to cover, let's pause for a moment and talk about the development process itself and one way we can make life easier for us while developing a Kafka Streams application.

## 3.4 Interactive Development

So we have covered in depth the graph that has been built to process purchase records from ZMart in a streaming fashion. The purchase data is now flowing through the topology and so far so good as ZMart prepares to start taking advantage of your streaming application.

But there is one challenge here that may not be obvious at first. You have four processors, and 3 of those write out to individual topics. While it is certainly possible to have a console consumer running to view results as you develop, it 'd be good to have a more convenient solution, like the ability to watch data flowing through the topology in the console:

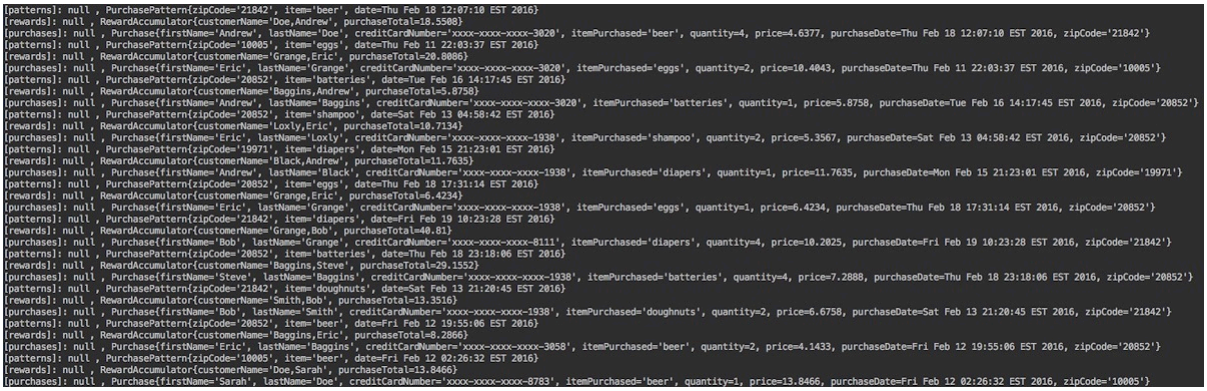



**Figure 3.10 A great tool while you are developing is the capacity to print data output from each node to the console. To enable printing to the console just replace any of the to methods with a call to print.**

There are two methods on the `KStream` interface that can be very useful during development. They are the `KStream.print` and `KStream.writeAsText` methods. The 'print' function writes records to stdout, and the 'writeAsText' function writes records out to a specified file. There are a few overloaded versions of both methods, but the ones you want to take notice of first are the `KStream.print(String streamName)` and the `KStream.writeAsText(String filePath, String streamName)` versions respectively. The ability to supply a unique name for an individual stream is a big plus when trying to disambiguate output on the console. Let's take a look at how we can simplify your development to view your progress as you go.

Since both methods only differ in where the final output of the records are, we will show how to use the `KStream.print` function.

**Listing 3.17 Using KStream Print**

```
purchaseKStream.mapValues(purchase -> PurchasePattern.builder(purchase).build())
                         .print(stringSerde,purchasePatternSerde,"patterns");  (1)

purchaseKStream.mapValues(purchase -> RewardAccumulator.builder(purchase).build())
                         .print(stringSerde,rewardAccumulatorSerde,"rewards");  (2)

purchaseKStream.print(stringSerde,purchaseSerde,"purchases");  (3)
```

1. Setting up to print the PurchasePattern transformation to the console.
2. Setting up to print the RewardAccumulator transformation to the console.
3. Printing the purchase data to the console.

Let's take a quick second to review the output we see on the screen and how it can up us during development.

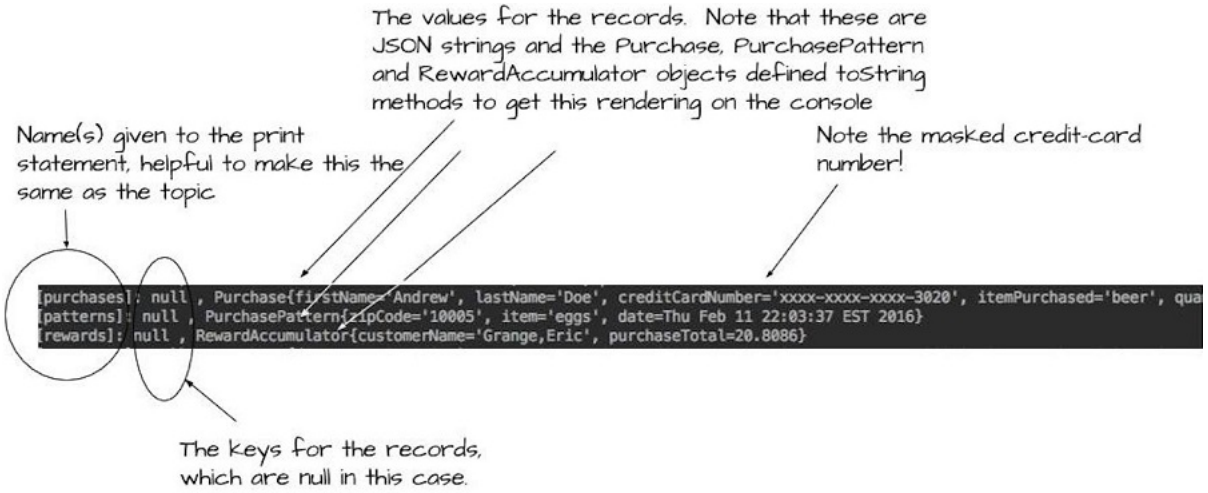



**Figure 3.11 This a detailed view of the data on the screen. You'll quickly see if your processors are working correctly with printing to the console enabled.**

When developing your Kafka Streams application, you can replace the `to` method calls with `print` method calls. Now you can run the Kafka Streams application directly from your IDE as you make changes, stop and start the application and confirm the output is what you expect. While this is no substitution for unit and integration tests, viewing streaming results directly as you develop is a great tool.

There is one small advantage to using the `print` method over `writeAsText`. The processor topology is a graph, and each child node gets processed or "visited" in a depth-first manner. As a result, you will see the output from each processor interleaved in their defined order in the topology. Processing in this manner allows you to confirm the processors are transforming records as expected. One final note on the `print` and `writeAsText` methods, for types other than String or Integer/Double/Float you need to have a good `toStirng` method defined on your object to get meaningful results.

## 3.5 Next Steps

At this point, you have your Kafka Streams purchase analysis program running well. Other applications have been developed to consume the messages written to the "purchase-patterns", "rewards" and "purchases" topics and the results for ZMart have been good. But alas, no good deed goes unpunished, and 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 have new requirements for each of the three categories of results you are producing so far. The good news is you still use the same source data. You are being asked to refine, and some cases further break down the data you are providing. The new requirements may apply to current topics or may require your to create entirely new topics.

- Purchases under a certain dollar amount need to be filtered out. Upper management is not that interested in the small item purchases for general daily articles.
- ZMart has expanded and has bought an electronics store chain and a popular coffee house chain. All purchases from these new stores will be flowing through the streaming application you have setup. You have been asked to send the purchases from these new subsidiaries to their topics.
- The NoSql solution you have chosen stores items in key-value format. Although Kafka uses key-value pairs also, the records coming into your Kafka cluster don't have any keys defined. So you'll need to generate a key for each record before the topology forwards it to the "purchases" topic.
- Unfortunately there seems to be the possibility of fraud in one of the stores. You have been asked to filter any purchases with a particular employee id and if possible, due to the sensitivity of the situation, write the results to a relational database instead of a topic.

While you are confident that more requirements will be coming your way, you start to work on the current set of new requirements now. You start to look through the KStream API and are 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 will be pared down to the essential parts of the discussion. Unless there is something new to introduce, you can assume the configuration and set up code remain the same. These truncated examples aren't meant to stand alone but are trimmed down to maximize clarity.

#### FILTERING PURCHASES

You decide to tackle the requirements in order, so you start with filtering out purchases that don't hit the minimum threshold. To remove low dollar purchases, you will need to insert a "filter" processing node between the `KStream` instance and the `to` processor. We are updating our processor topology graph in the following manner:

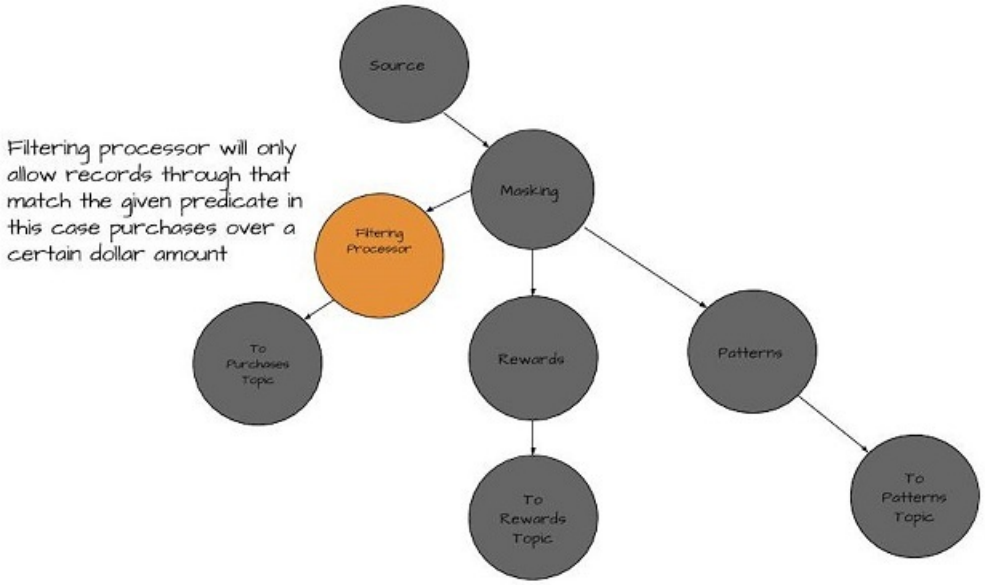



**Figure 3.12 Here we are placing a processor between the masking processor and the terminal node that writes to Kafka. This filtering processor will drop purchases under a given dollar amount.**

### Listing 3.18 Filtering on KStream

```
purchaseKStream.filter((key, purchase) -> purchase.getPrice()
   > 5.00).to(stringSerde, purchaseSerde, "purchases");  (1)

//or for local development purposes

purchaseKStream.filter((key, purchase) -> purchase.getPrice()
   > 5.00).print(stringSerde, purchaseSerde, "purchases");  (2)
```

(1) Writing the filtered results out to the "purchases" topic

(2) Using the "print" method to write the filtered purchase data to the console.

Here you decide to use the `KStream.filter` method which takes a `Predicate<K,V>` instance as a parameter. Although we are chaining method calls together here, we are creating a new processing node in the topology. The `Predicate` interface has one method defined, `test` which takes two parameters, the key, and the value, although at this point we only need to use the value. Again we see how we can use a Java 8 lambda in place of a concrete type defined in the KStreams API.

**SIDEBAR** **What is a Predicate**

If you are familiar with functional programming, then you should feel right at home with the Predicate interface. If the term 'predicate' is new to you, it is nothing more that a given statement such as 'x < 100' and an object either matches the predicate statement or not.

There is a mirror image function `KStream.filterNot` that performs the same functionality but in reverse. Only records that don't match the given predicate are

processed further in the topology.

### SPLITTING/BRANCHING THE STREAM

Now when it comes to splitting the stream of purchases into separate streams that can write to different topics has you scratching your head. Then you happen to find the `KStream.branch` method, 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 to `KStream.branch`. Where with the previous change we were modifying an existing "leaf" on the processing topology, with this requirement to branch the stream we will be creating brand new "leaf" nodes to our graph of processing nodes or topology.

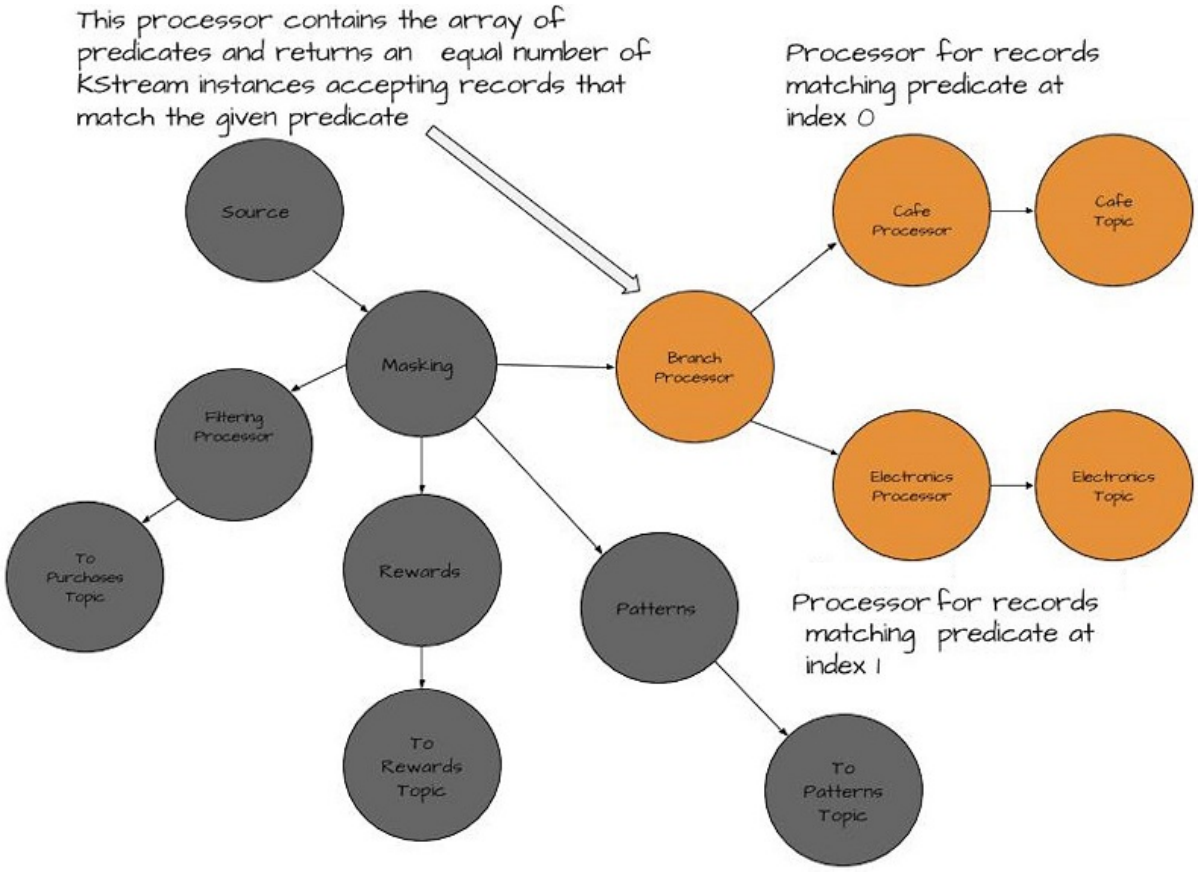



**Figure 3.13 The branch processor is splitting the stream at this point into two streams; one stream consists of purchases from the cafe the other stream contains purchases from the electronics store.**

As records from the original stream flow through the branch processor, each record gets matched against the supplied predicates in the same order as provided. Matching is 'short-circuit,' the processor assigns records to stream on the first match; no attempts are made to match additional predicates.

The branch processor drops records if they do not match any of the given predicates. The order of the streams in the returned array match the order of the predicates provided to the branch method. You start to think that a separate topic for each department may not be the only approach, but you stick with what you have come up with for now and mark this task down as one requirement you will revisit later.

**Listing 3.19 Splitting The Stream**

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

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

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

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

1. Creating the predicates, as Java 8 lambdas.
2. Labeling the expected indices of the returned array.
3. Calling branch to split the original stream into two streams.
4. Writing the results of each stream out to a topic.

**WARNING** **Topic Creation**

In the above examples, we are sending records to several different topics. Although Kafka can be configured to automatically create topics when attempting to produce/consume for the first time from non-existent topics, it's not a good idea to rely on this mechanism. If you rely on auto-creating topics, topic configuration uses 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 will need, the level of partitions and replication factor ahead of time and create them before running your Kafka Streams application.

In this example, we define the predicates ahead of time, as passing four lambda expression parameters would be a little unwieldy to view. For the same reason, we have labeled the indices of the returned array to maximize our readability here.

**SIDEBAR** **Splitting vs Partitioning Streams**

While "splitting" and "partitioning" may seem like similar ideas, regarding Kafka/Kafka Streams they are unrelated. Splitting a stream with the `KStream.branch` method is 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 is the principal means of achieving high throughput in Kafka.

This example demonstrates the power and flexibility of Kafka Streams. You have

been able to take the original stream of purchase transactions and split them into four streams, with very few lines of code. Also, you are starting to build up a more complex processing topology all while reusing the same source processor.

So far so good, you have been able to meet two of the three new requirements with ease. Now it's time to go to 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. However, no requirements are stating that keys can't be null. In practice, if there is no need, having a null key will reduce the overall amount of data that travels the network. All of the records flowing into the ZMart Kafka Streams application have null keys.

That has been fine, up until the point you realize you NoSql storage solution stores data in key-value format. So you need a way to create a key from the `Purchase` data before it gets written out to the "purchases" topic. While you certainly could use `KStream.map` to generate a key and return a new key-value pair (where only the key would be 'new'), there is the 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 the filtering in that we will add a processing node between the "filter" and "to" processor.

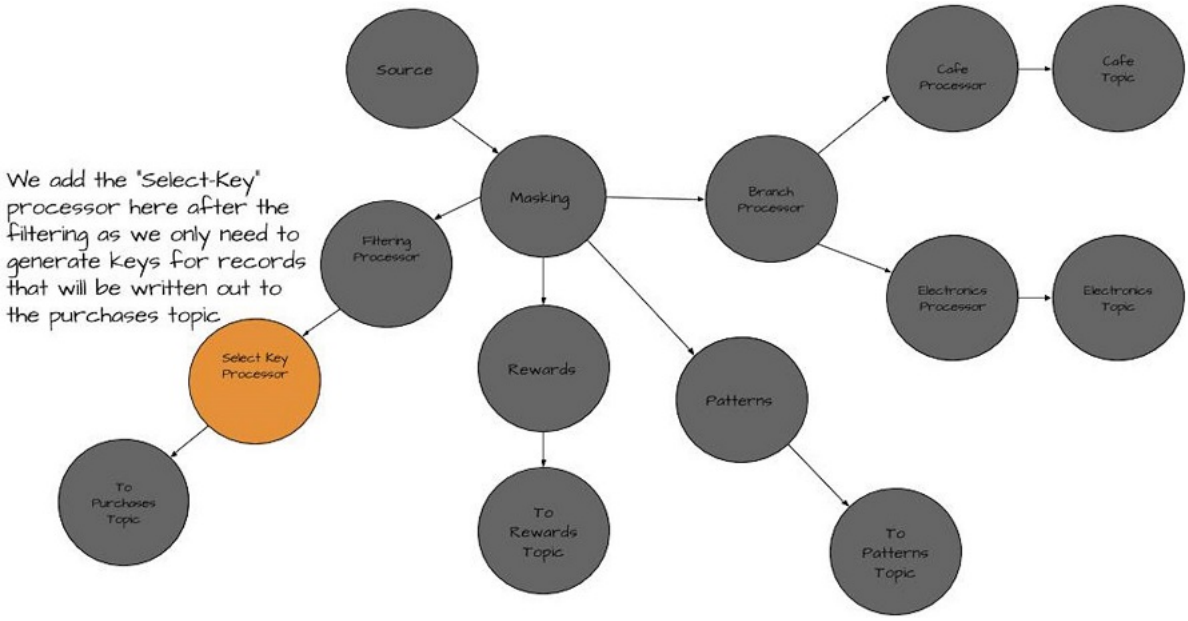



**Figure 3.14 The NoSql data store we are using will use the purchase date as a key for the data it stores. Here we're adding a selectKey processor to extract the purchase date to use as a key right before we write the data to Kafka.**

**Listing 3.20 Generating A New Key**

```
KeyValueMapper<String, Purchase, Long> purchaseDateAsKey =
    (key, purchase) -> purchase.getPurchaseDate().getTime();   (1)
```

```
purchaseKStream.filter((key, purchase) -> purchase.getPrice()
    > 5.00).selectKey(purchaseDateAsKey).to(Serdes.Long(), purchaseSerde, "storage");
```

❶ The KeyValueMapper extracting the purchase date and converted to a long

To create the new key, we are taking the purchase date and converting it to a long. Although we could have passed a lambda expression, we have assigned it to a variable to help with readability. Also, note that we needed to change the serde type used in the `KStream.to` method, as we have changed the type of the key.

The previous example is a very simple display of mapping to a new key. Later in another example, we will show how to select keys to enable joining separate streams. Also, all of our examples up until this point have been stateless, but there are several options available to us to do stateful transformations as well, which will see a little later on.

### 3.5.2 Writing Records Outside of Kafka

The security department at ZMartYou has approached you. Apparently, in one of the stores, there is of fraud. There have been reports that one of the managers of the store is entering invalid discount c purchases. Security is not sure of what is going on, but they are asking for your help.

The security folks don't want this information to go into a topic. You talk to them about securing Kafka, and access controls and 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 has results filtered down to a single employee id. Even though you have a large amount of data flow through your topology, this will reduce the volume to a tiny amount.

Here we are going to use `KStream.filter` with the predicate looking to match a specific employee id. This filter is completely separate from the previous filter and is attached to the source `KStream` instance. Although it is entirely possible to chain filters, this is not the case here; we want full access to data in the stream for this filter. As you go through the API, there is a method that catches your eye, the `KStream.foreach` method. The method predictable takes a `ForeachAction<K, V>` instance, and is a terminal node. The `KStream.foreach` is a simple processor and as the name implies `KStream.foreach` uses the provided `ForeachAction` instance to perform an action on each record it receives.

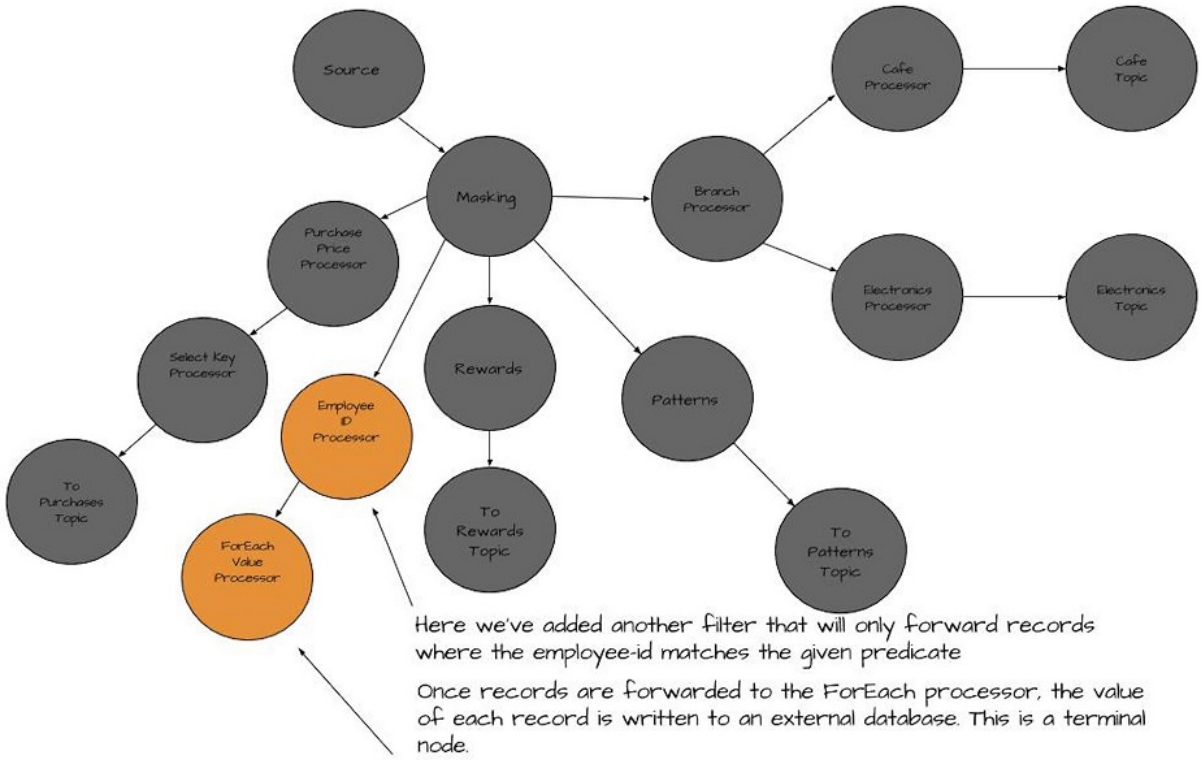



**Figure 3.15 To write purchases involving a given employee outside of our Kafka Streams application we'll first add a filter processor to extract purchases by employee id then use a foreach operator to write each record out to an external relational database.**

## Listing 3.21 Foreach Operations

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

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

Here we created the `ForeachAction` using a Java 8 lambda (again!) and have stored it in a variable, although it's an extra line of code, the clarity gained by doing so more than makes up for it.

On the next line, we have created another `KStream` instance that will send the filtered results to the `ForeachAction` defined directly above it.

We should mention here, that `KStream.foreach` is stateless and if you need state to perform some action per record you will use the `KStream.process` method. The `KStream.process` method is discussed in the next chapter when we cover adding state to a Kafka Streams application.

If you step back and take a look at what you have accomplished at this point, it's pretty impressive considering the lines of code written. Don't get too comfortable, though, because the upper management team at ZMart has taken notice of your productivity and are excited about the results. No good deed goes unpunished, so this means more changes and refinements to purchases streaming analysis program are coming.

## 3.6 Summary

Well, we have certainly covered a lot of ground in this chapter. You learned about the KStream API, and we were able to see how quickly we could develop an application that not only met your original goals but also allowed you to adapt to ever evolving changes and requirements quickly.

Specifically, we learned:

- How to use the `KStream.mapValues` function to map incoming record values to a new value, possibly of a different type. You also learned that these mapping changes should not modify the original object. There is another method `KStream.map` that performs the same action but can be used to map 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 if that object matches a given condition. We were able to use predicates in the filter function to remove records from being forwarded in the topology for that don't match the given predicate.
- The `KStream.branch` method uses predicates to split records into new streams when a record matches a given predicate. The processor assigns a record to a stream on the first match and dropping unmatched records.
- How to select or modify a key using the `KStream.selectKey` method. You also learned how the `KStream.through` method can be used to ensure streams that have modified keys place records together on the same partition when the keys are equal.

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

# 4 Streams and State

In this chapter:

- 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 head first into the KStreams API and built a processing topology to handle streaming requirements from purchases made at ZMart locations. While we built a non-trivial processing topology, it was one-dimensional in that all transformations and operations were stateless. We 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, we only dealt with individual streams, ignoring any possibility of gaining additional insight by joining streams together.

Now we want to extract the maximum amount information from our Kafka Streams application. To get this level of maximum information, we are going to need to use state. State is nothing more than the ability to recall information we've seen before and use it to connect to current information. We can utilize state in different ways. We'll see one example when we explore the stateful operations, such as accumulation of values, provided by the KStreams API.

Another example of state we'll discuss is the joining of streams. Joining streams is closely related to the joins performed in database operations, joining records from the employee and department tables to generate a meaningful report on who staffs which departments in a company.

We'll also define what our state needs to look like and the requirements around using state when we discuss StateStores in Kafka Streams. Finally, we'll weigh the importance

of timestamps and how they direct decisions for working with stateful operations from ensuring we only work with events occurring within a given timeframe to helping us work with data arriving out of order.

## 4.1 Thinking of Events

When it comes to event processing, sometimes an event by itself requires no further information or context. Other times we can understand an event in a literal sense, but without some added context we might miss the actual meaning of what is occurring or think of the situation in a whole new light given some additional information.

As an example of an event that does not require additional information, consider the attempted use of stolen credit card. The transaction is canceled immediately once the stolen card's attempted use is detected. We don't need any additional information to be sure of the decision.

But in some case, a singular event in and of itself doesn't give you enough information to make impactful decisions. Consider a series of stock purchases by three individual investors within a short period. On the face of it, there isn't anything inherently odd about those transactions that would give us pause. Investors buying shares of the same stock is something that happens every day on Wall Street.

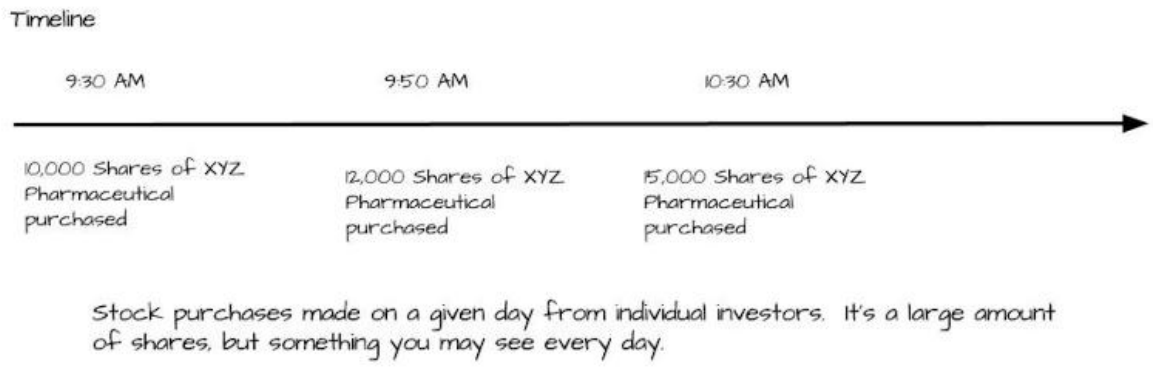



**Figure 4.1 A stock transaction without any extra information doesn't look like anything out of the ordinary.**

Now let us add some context; within a short period after the individual stock purchases, XYZ Pharmaceutical announced government approval for a new drug highly anticipated by the public, which sent the stock price to historic highs. Additionally, those three investors had close ties to XYZ Pharmaceutical. Now the transactions in question can be view in a whole new light.

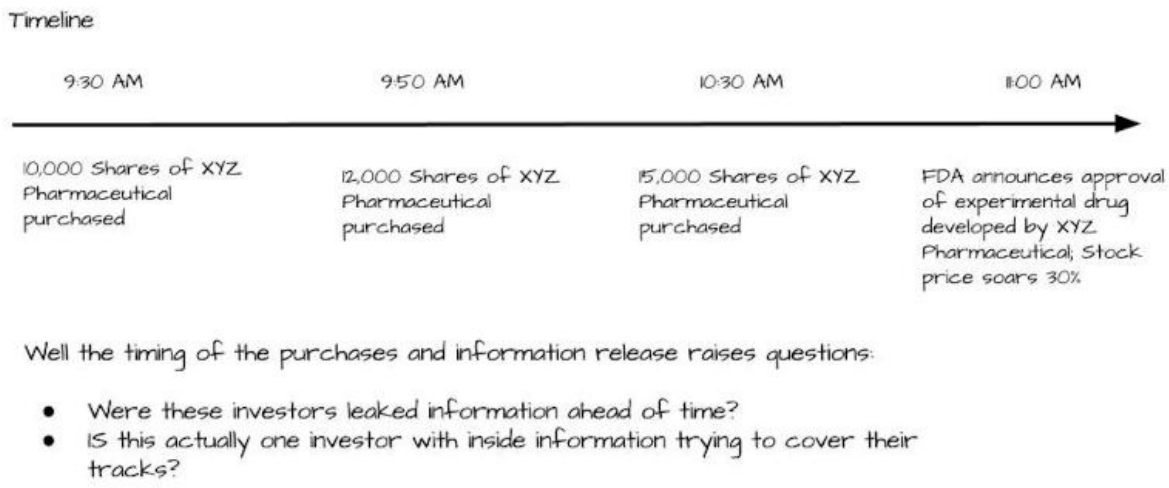



**Figure 4.2 However when we add some additional context about the timing of the stock purchase, and we'll see it in an entirely new light.**

### 4.1.1 Streams Need State

What the fictional scenario example from above shows us is something that most of us already know instinctively. Sometimes it's easy to reason about what's going on, but most of the time we need some context to make good decisions. When it comes to stream processing, we call that added context "state."

At first glance, the notion of state and stream processing may seem to be at odds with each other. Stream processing infers 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 conjure images of a static resource such as a database table.

In actuality we could view these two as one in the same, it's just that the rate of change in a stream is potentially much faster and frequent than a database table [10].

Footnote 10 www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing

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

## 4.2 Applying Stateful Operations to Kafka Streams

In this section, we are going to demonstrate how we can add a stateful operation to an existing non-stateful one to improve the information collected by our application. We are going to modify the original topology we built in Chapter 3 pictured below to refresh your memory.

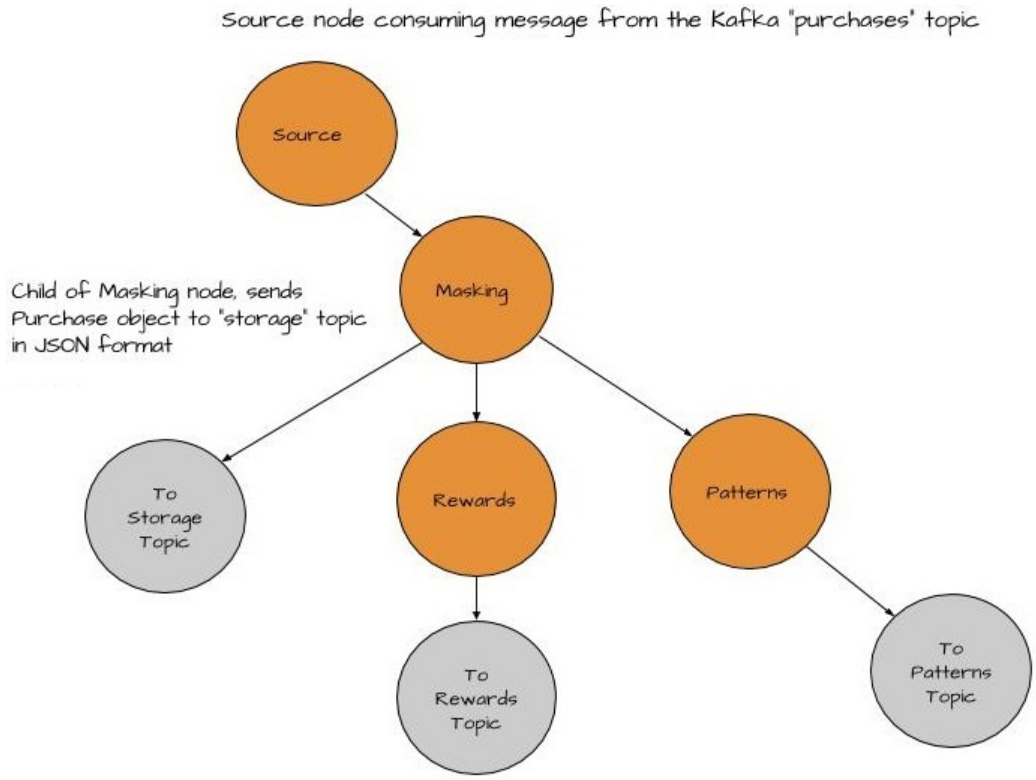



**Figure 4.3 Here's another look at the original topology you built in Chapter three.**

In the topology picture above, we produced a stream of purchase transaction events. One of the processing nodes in the topology, named "Rewards," calculated reward points for customers based on the amount of the sale. But in that processor, we were just calculating the total number of points for a single transaction and forwarding the results.

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

Now that we have discussed how and why we need state to work with Kafka Streams (or any other streaming application) let's go over some concrete examples of how we can apply stateful operations; starting with transforming the stateless rewards processor into a stateful processor using `TransformValues`.

### 4.2.1 Transform Values Processor

The most basic of the stateful functions is `KStream.transformValues`. The image below is a visual guide to how the `KStream.transformValues` method operates.

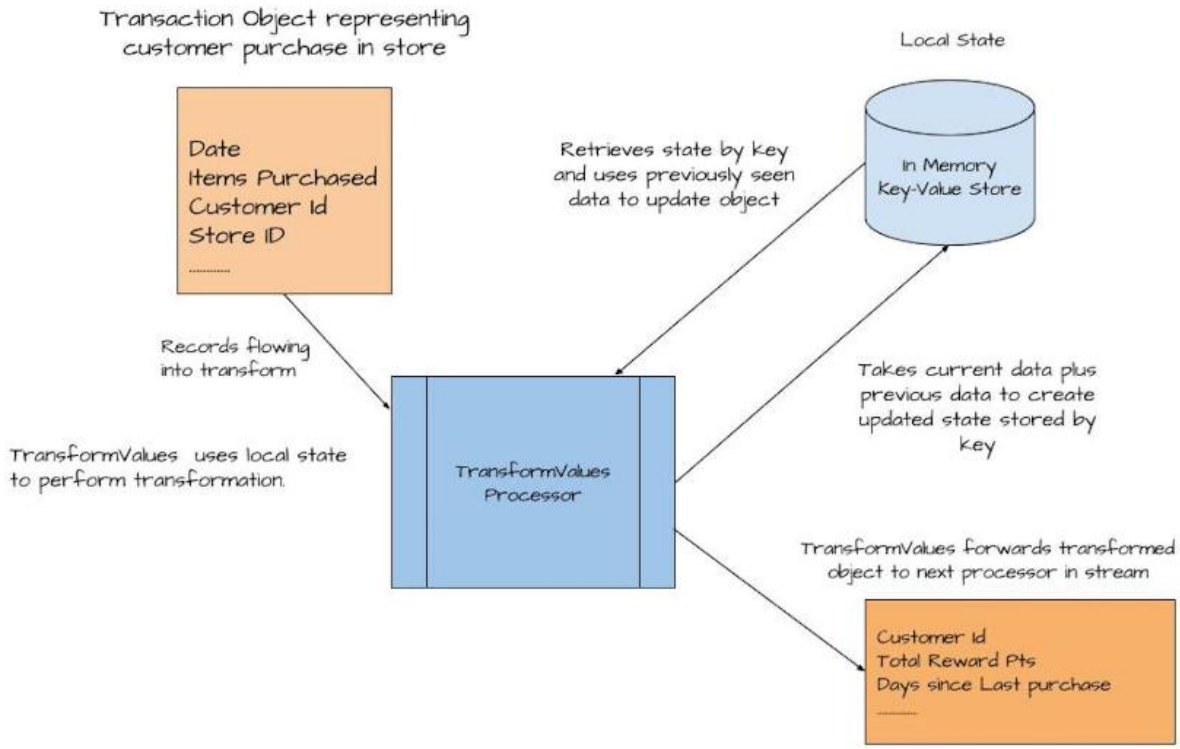



**Figure 4.4 The transformValues processor uses information stored in local state to update incoming records. In this case, the customer id is the key used to retrieve and store the state for a given record.**

This method is semantically the same as `KStream.mapValues` with a few exceptions. One difference is `transformValues` has access to a `StateStore` instance to accomplish its given task. The other difference is the 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.

Next, we are going to re-work the original stateless rewards process to a stateful one, keeping track of the total bonus points achieved so far and the amount of time between purchases to provide more information to downstream consumers of your application.

### 4.2.2 Stateful Customer Rewards

In chapter 3 we built a streaming topology for ZMart, and one of the processors in the topology extracts information for customers belonging to ZMart's rewards program. Here's the correspondent node in the topology:

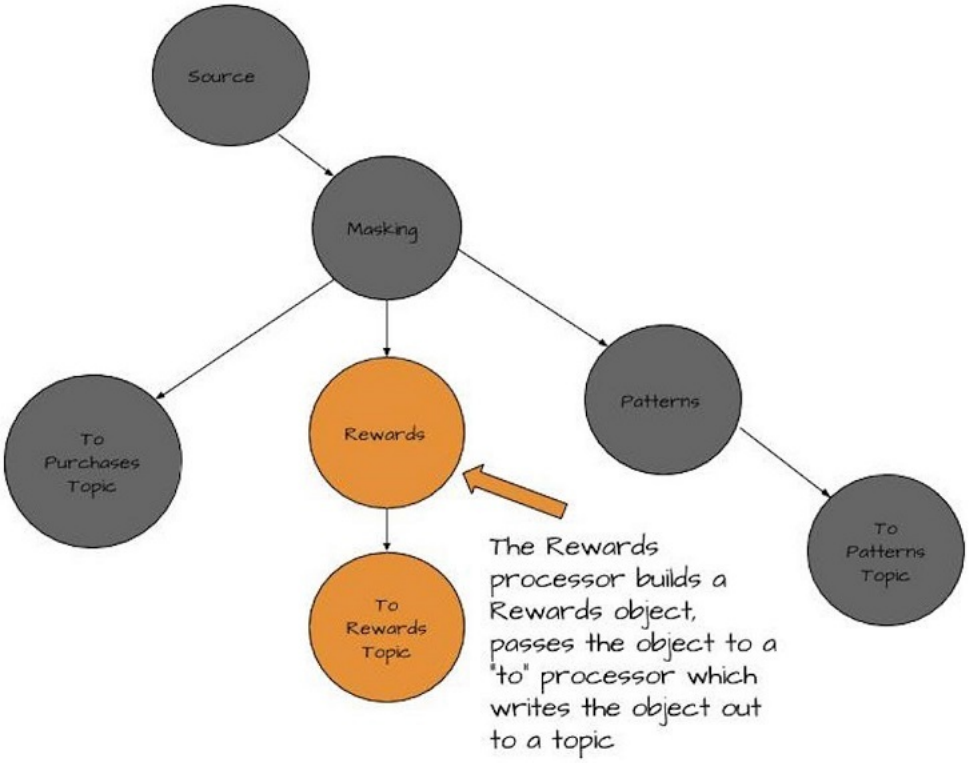



**Figure 4.5 This is the original rewards processor you added to the topology in Chapter three. This processor is going to get an upgrade and go from making "stateless" transformations to "stateful" ones.**

Initially, the rewards processor used the `KStream.mapValues` method to map the incoming `Transaction` object into a `Rewards` object. The `Rewards` object originally consisted of just two fields, the customerID and the purchase total for the transaction.

**Listing 4.1 RewardsAccumulator Object**

```
public class RewardAccumulator {

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

  //details left out for clarity
}
```

(1) The customer id.

(2) Total Dollar amount of purchase.

(3) The current number of reward points.

Now the requirements have changed some, and now there are points associated with the ZMart rewards program. Where before there was an application reading from the "rewards" topic calculating customer achievements, there is a desire from management to

have the point system maintained and calculated by the streaming application instead. Additionally, you need to capture the amount of time between the current and the last purchase the customer made.

When the application reads records from the "rewards" topic, the consuming application will only need to check if the total points are above the given threshold to distribute an award. To meet this new goal, you add the `totalPoints` and `daysSinceLastPurchase` fields to the `RewardAccumulator` object and use the local state to keep track of accumulated points and the last date of purchase.

As a result here is the refactored `RewardsAccumulator` code needed to support the changes.

**Listing 4.2 Refactored RewardsAccumulator Object**

```
public class RewardAccumulator {

    private String customerId;
    private double purchaseTotal;
    private int currentPoints;
    private int daysSinceLastPurchase;
    private long totalRewardPoints;    (1)


  //details left out for clarity
}
```

(1) 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 up 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 the `KStream.transformValues`. Semantically these two methods operate the same way in that we still map the `Purchase` object into a `RewardAccumulator` object. The difference lies in the ability to use local state to perform the transformation. Specifically, there are two main two steps we'll take:

1. Initialize the Value Transformer.
2. Map the `Purchase` object to a `RewardsAccumulator` using state.

The `KStream.transformValues` takes a `ValueTransformerSupplier<V, R>` object which supplies an instance of the `ValueTransformer<V, R>` interface.

Our implementation of the `ValueTransformer` is `PurchaseRewardTransformer<Purchase, RewardsAccumulator>`. For the sake of clarity, we won't reproduce the entire class here in the text. Instead, we'll walk through the important methods to cover the list of performed steps from above. Also note that these code snippets aren't meant to stand alone, and some details will be left out for clarity. Now let's move on with our first step, initializing the processor.

### 4.2.3 Initialize the Value Transformer

Your first step is to set up/create any instance variables in the transformer `init` method:

**Listing 4.3 Init Method**

```
private KeyValueStore<String, Integer> stateStore;     (1)

private final String storeName;
private ProcessorContext context;

public void init(ProcessorContext context) {
    this.context = context;                            (2)

    stateStore = (KeyValueStore) this.context.getStateStore(storeName);   (3)
}
```

(1) Instance variables.

(2) Setting local reference to ProcessorContext.

(3) Retrieving the StateStore instance by storeName variable.

In the `init` method we retrieve the state store created when building the processing topology. Inside our transformer class, we cast to a `KeyValueStore` type as inside our transformer we are not concerned with the implementation (more on state store implementation types in the next section) at this point, just that we can retrieve values by key. There are other methods (`punctuate`, `close`) not listed here that belong to the `ValueTransformer` interface. We discuss The `punctuate` and `close` methods when covering the Processor API in Chapter six.

### 4.2.4 Map the Purchase object to a RewardsAccumulator using state

Now that we have initialized our processor we move on to the main part of the action transforming a `Purchase` object using state. There a few simple steps we take performing the transformation:

1. Check for points accumulated so far by customer id.
2. Sum the points for the current transaction and present 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 4.4 Transforming the Purchase using State**

```
public RewardAccumulator transform(PurchaseTransaction value) {
    RewardAccumulator rewardAccumulator = RewardAccumulator.builder(value).build();   (1)
    Integer accumulatedSoFar = stateStore.get(rewardAccumulator.getCustomerId());     (2)

    if (accumulatedSoFar != null) {
         rewardAccumulator.addRewardPoints(accumulatedSoFar);                         (3)
    }
    stateStore.put(rewardAccumulator.getCustomerId(),
```

```
            rewardAccumulator.getTotalRewardPoints());   ❹

        return rewardAccumulator;   ❺
    }
```

❶ Building the Reward object from the Transaction.

❷ Retrieving latest count by customer id.

❸ If an accumulated number exists, add it to the current total.

❹ Store the new total points in the StateStore.

❺ Return the new accumulated rewards points.

In the `transform` method we first do the mapping of a `Purchase` object into the `Rewards` accumulator and is the same operation used in the `mapValues` method. On the next few lines are where state gets involved in the transformation process. We do a lookup by key (customer id) and adding any points accumulated so far to the points from the current purchase. Then we place the new total in the state store until another it's needed again.

All that's left is to update our rewards processor. But before we update the processor, we have overlooked the fact that we need access to all sales by customer id. Gathering information per sale for a given customer implies all transactions for that customer are on the same partition.

However, because our transactions come into the application without a key, the producer assigns the transaction to partitions in a round-robin fashion. We covered round-robin partition assignment in Chapter 2, but it's worth reviewing the image here again:

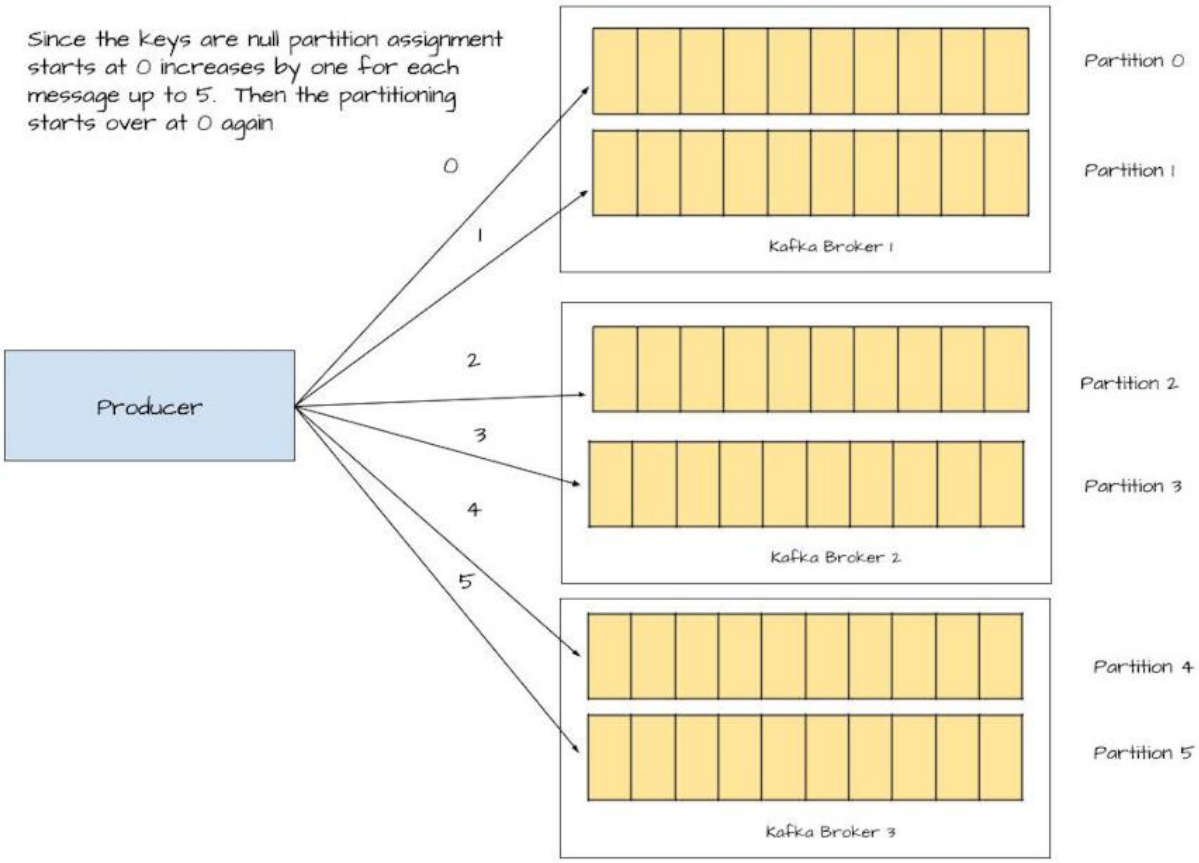


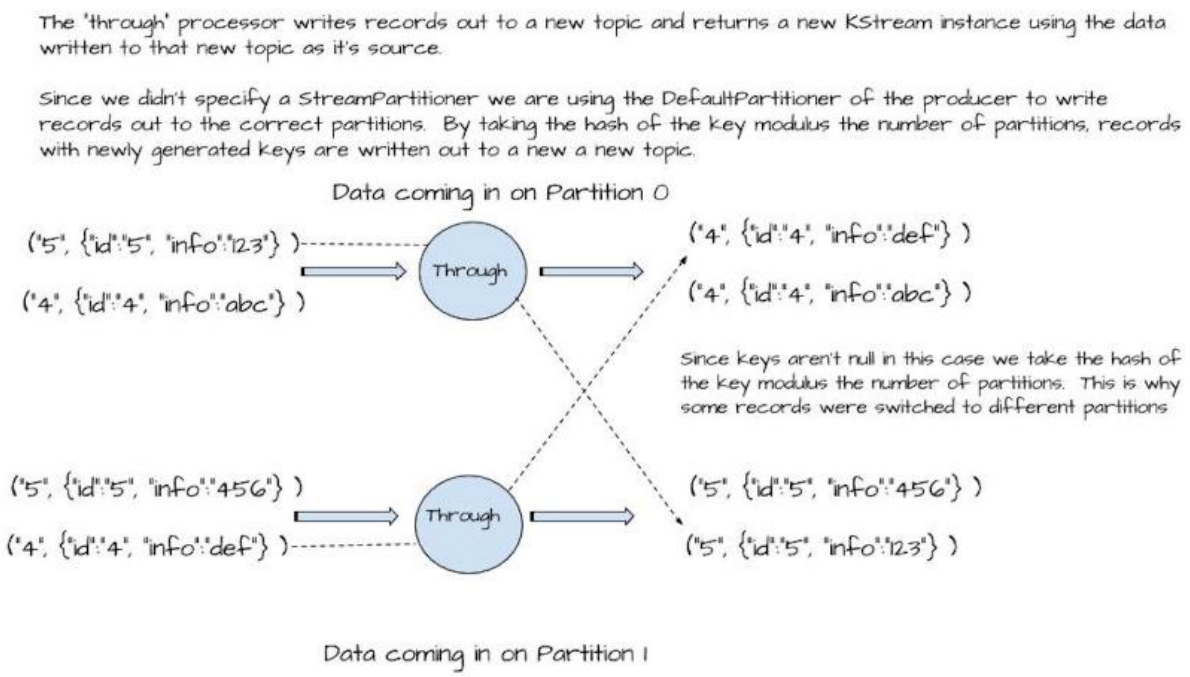

**Figure 4.6 Here's another look at how a Kafka Producer distributes records evenly (round-robin) when the keys are null.**

We'll have an issue here (unless you are using topics with one partition). Because of not having a populated key, round-robin assignment means all transactions for a given customer won't land on the same partition. Having customer transactions with the same id placed on the same partition is important because we look up records by id in the state store.

Otherwise, we'll have customers with the same id across different partitions, meaning lookups for the same customer in multiple state stores. The way for us to solve this problem is to repartition the data by customer id. Let's take a look at how we'll repartition our date in the next section.

### REPARTITIONING THE DATA

To repartition the data we write our records out to an intermediate topic then immediately read the records back into our toplogy. The following image gives us a good idea of how repartitioning works:




Now we've discussed what repartitioning is and why we need to do it, let's demonstrate how. Fortunately, it's a trivial operation accomplished by using the `KStream.through` method. Here's an image depicting how the `KStream.through` method works:

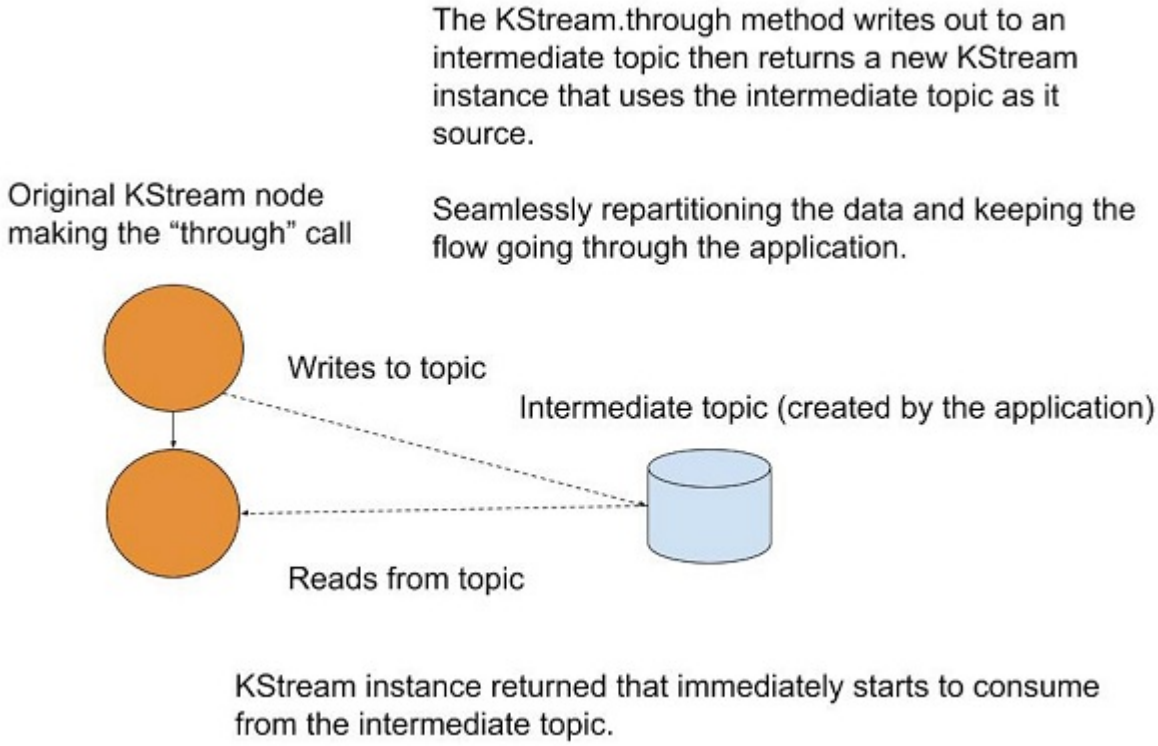



**Figure 4.7 Writing out to intermediate topic the reading from it in a new KStream instance**

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 we can seamlessly repartition our data and continue the data flow our Kafka Streams application. The code for the `KStream.through` method looks like this:

Using KStream.through Method

```
RewardsStreamPartitioner streamPartitioner =
    new RewardsStreamPartitioner();                                    (1)
KStream<String, Purchase> transByCustomerStream = purchaseKStream.through(stringSerde,
    purchaseSerde, streamPartitioner, "customer_transactions");        (2)
```

(1) Instantiating our concrete StreamPartitioner instance.

(2) Creating a new KStream with KStream.through method. The KStream.through method takes four parameters: key serde, value serde, stream partitioner and the topic name.

We could have taken the approach of generating new keys. But since we don't need keys beyond this point and the value contains the customer id, we've decided to use the value to determine the partition.

Since we've mentioned using a `StreamPartitioner` and it's essential for repartitioning to work, let's take a quick look at how we'd create one.

### USING A STREAMPARTITIONER

A `StreamPartitoner` is a class used to return a number representing the partition assignment. The partition assignment is calculated by taking the hash of an object modulo the number of partitions. Now partition assignments are determined by the value of the customer-id, and we know that data for a given customer is in the same state store.

Now let's look at our StreamPartitioner implementation:

**Listing 4.5 RewardsStreamPartitioner**

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

    @Override
    public Integer partition(String key, Purchase value, int numPartitions) {
        return value.getCustomerId().hashCode() % numPartitions;       (1)
    }
}
```

(1) Determine partition by customer id.

Notice that we have not generated a new key and we have used a property of the value to determine the correct partition.

The key point to take away from this quick "detour" is when using state to update and modify data by key it's important for those records to be on the same partition.

Use repartitioning with care.

Don't mistake the ease of our repartitioning demonstration as a recommended best practice you can be cavalier with. While sometimes repartitioning is necessary it comes

at the cost of duplication of your data. So my advice here is to use map/transform operations on values whenever possible and to use repartitioning logic sparingly.

Now let's pick up where we left off and go back to making changes in the rewards processor node to support stateful transformation.

### 4.2.5 Updating The Rewards Processor

Up to this point, we've created a new processing node that writes purchase objects out to a topic using the customer id as the key. This new topic also becomes the new source for our soon to be updated rewards processor.

We did this to ensure all purchases for a given customer are written to the same partition, allowing us to have consistent results. We've updated our processing topology below by inserting the new processor between the credit card making node (source for all purchase transactions) and the rewards processor. Now we'll move on to updating the rewards processor in the topology to use our new stateful processor. We'll call our new processor "Transform Values" as indicated below:

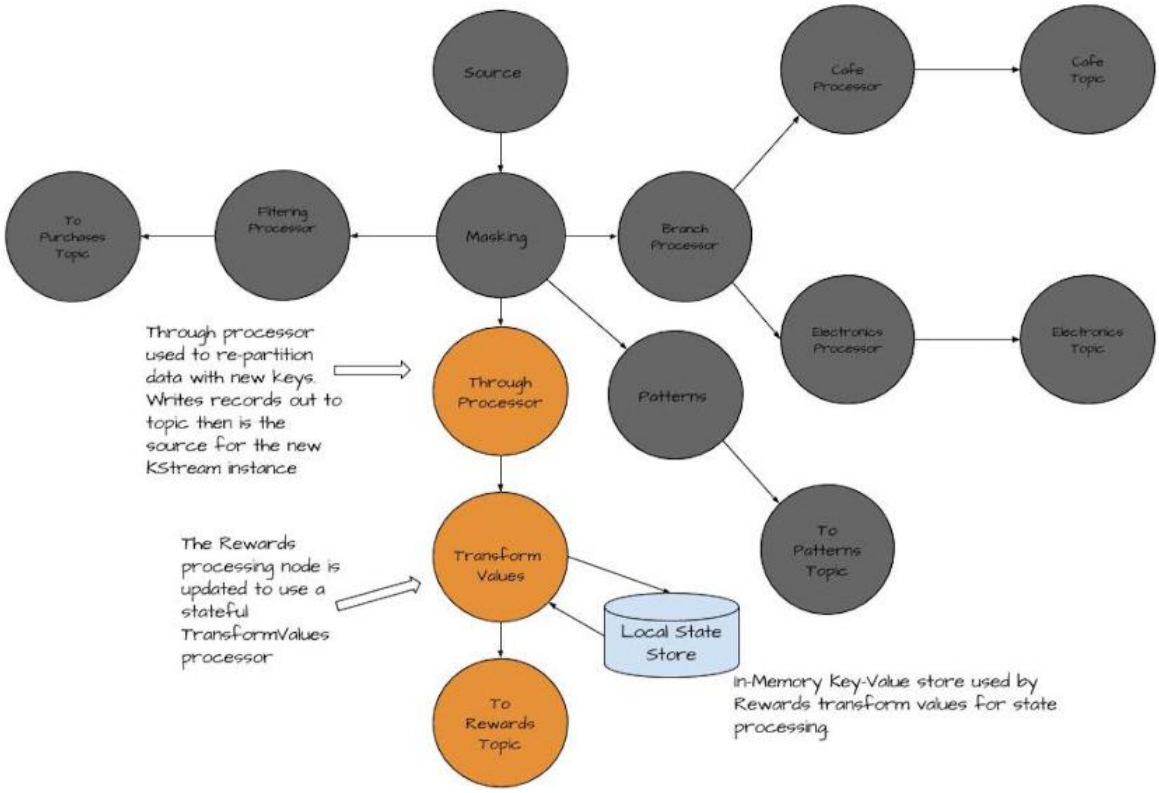



**Figure 4.8 Here we've added the through processor ensuring we send purchases to partitions by customer id, which allows the transformValues processor to make the right updates using local state.**

Now we use our new `KStream` instance (created above using the `KStream.through` method) to update our processor and use the stateful transform approach with the following line of code:

**Listing 4.6 Changing Rewards Processor to Use Stateful Transformation**

```
PurchaseRewardTransformer transformer =
   new PurchaseRewardTransformer(rewardsStateStoreName);   (1)
transByCustomerStream.transformValues(() -> transformer,
   rewardsStateStoreName).to(stringSerde, rewardAccumulatorSerde, "rewards");   (2)
```

(1) Creating the PurchaseRewardTransformer object.

(2) Using a stateful transformation and writing the results out to a topic.

Although the `KStreams.transformValues` method takes a `ValueTransformerSupplier<V, R>` object, in a pattern, we have seen before; we use a Java 8 lambda expression instead of a concrete instance of the `ValueTransformerSupplier`.

In this section, we added stateful processing to a stateless node. By adding state to the processor, ZMart takes action sooner after a customer makes a reward qualifying purchase. We've seen how to use a state store and the benefits using one provides.

But we've glossed over important details that we'll need to understand how state can impact our applications entirely. With this in mind, the next section will specify which type of state store to use, what requirements we'll need to make state efficient and how we add the stores to Kafka Streams program.

## 4.3 Using State Stores for Lookups and Seen Data

In this section, we are going to learn the essentials of using state stores in Kafka Streams and the key factors of using state in streaming applications in general. We'll do this so we can make useful choices when using state in our Kafka Streams applications.

So far we've discussed the need for using state with streams and have seen an example of one of the more basic stateful operations available in Kafka Streams. Before we describe the idea of state stores in Kafka Streams, let's briefly describe two important attributes of state that we'll need

1. Data locality
2. Failure Recovery

### 4.3.1 Data Locality

Our first key point is data locality. Data locality is critical for performance. While key lookups are typically very fast, at scale the latency introduced by using remote storage becomes a bottleneck. In the image below we show the principal behind data locality.

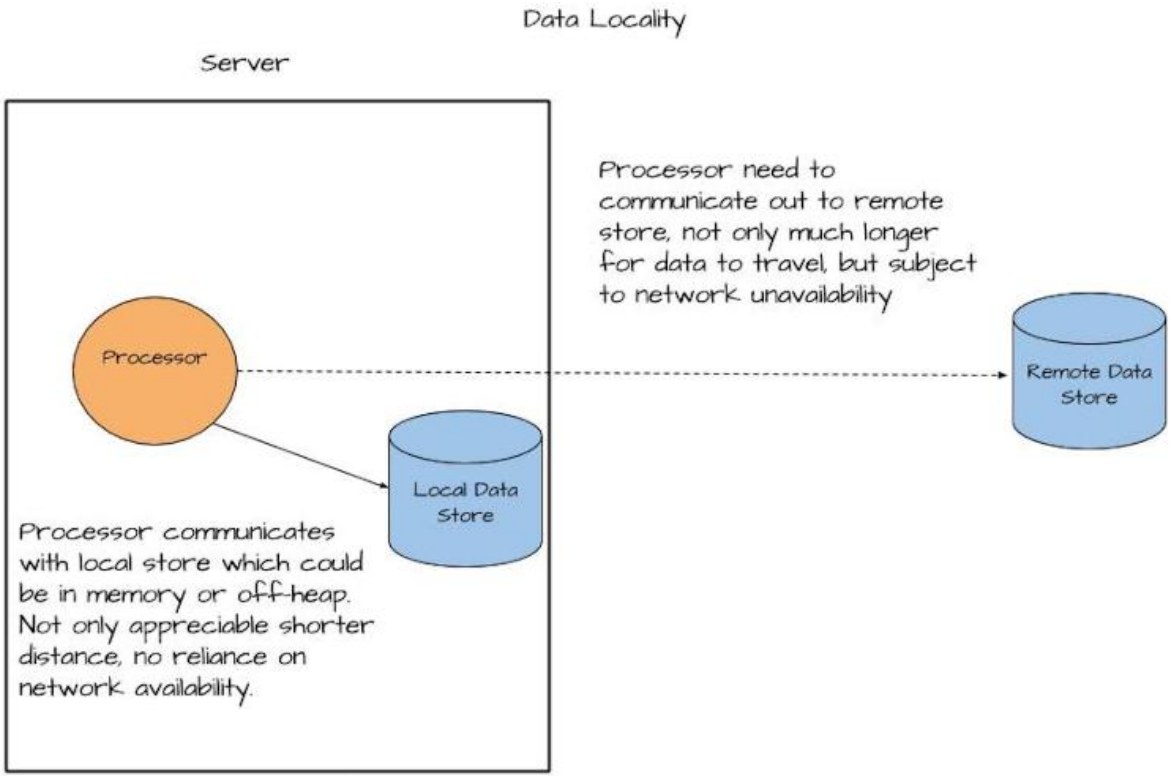



**Figure 4.9 Data locality is necessary for stream processing. First of all traveling across the network to retrieve data incurs a cost that at high data volume is noticeable, secondly it provides some isolation due to the fact if a process goes down other processes aren't affected.**

The dashed line represents a network call to retrieve data from a remote database hosted on another server. The solid line depicts a call to an in-memory data store co-located on the same server. While the image above is not drawn to scale, it's pretty clear that making a call to get data locally is more efficient than making a call across a network to a remote database.

The key point here is not the amount of latency per-record retrieval by itself, which may be minimal. The important factor here is you will potentially process millions to billions of records through a streaming application. When multiplied by a factor that large even a small amount of network latency has a huge impact.

Data locality also means that the store is local to each processing node and there is no sharing across processes or threads. By not sharing a store, if a particular process fails, it shouldn't have an impact on the other stream processing processes/threads.

The key point here is while streaming applications will sometimes require state, it needs to be local to where the processing occurs, and each server/node in the application should have an individual data store.

### *4.3.2 Failure Recovery/Fault Tolerance*

Failure in applications is inevitable, especially when we are talking about distributed applications. We need to shift our focus from preventing failure to how to recover quickly from failure or even restarts. The following image depicts the principal of data locality and fault tolerance. Each processor has its local data store, and a topic is used to back up the state store.

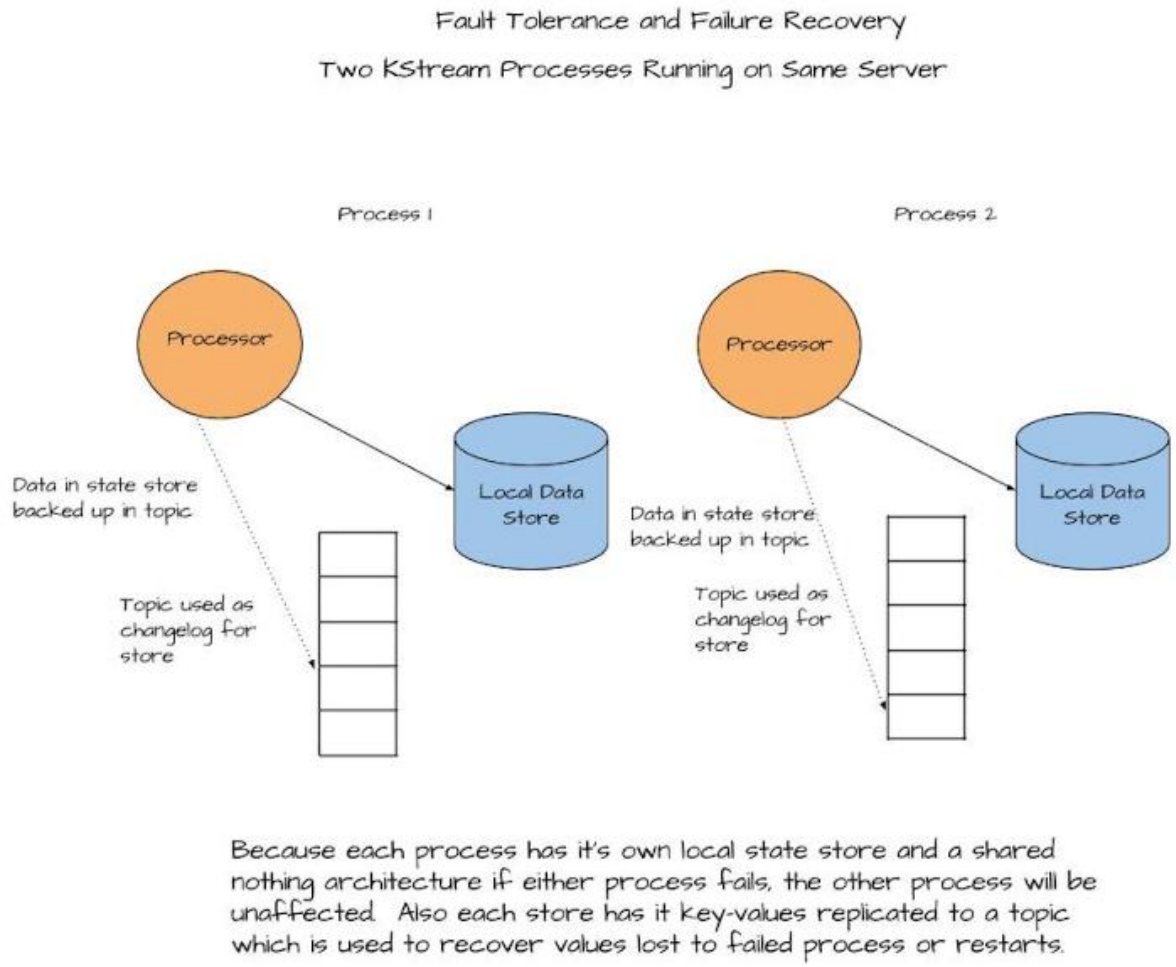



**Figure 4.10 The ability to recover from failure is important for stream processing applications. Kafka Streams persists data from the local in-memory stores to an internal topic so when resuming operations either from a failure or a re-start the data is re-populated.**

The state stores provided by Kafka Streams meets both the locality and fault-tolerance requirements. State stores are local to the defined processors and don't share access across processes or threads. State stores use topics as a backup and quick recovery, as shown in the image above.

Over the past two sections, we've covered the requirements we need for using state with a streaming application. The next step is to show how we enable the usage of state in our Kafka Streams applications.

### 4.3.3 Using State Stores in Kafka Streams

Adding a state store is a matter of creating a `StateStoreSupplier`. While there are concrete implementations we can directly instantiate, there is a builder available utilizing the "fluent" style we've seen in building a `KafkaStreams` instance. Please note that we will be using the term `StateStoreSupplier` throughout this section, but it is meant in synonymous terms with a `StateStore` since the supplier provides an instance of a `StateStore`.

**Listing 4.7 Adding A StateStore**

```
String rewardsStateStoreName = "rewardsPointsStore";
StateStoreSupplier stateStoreSupplier =
Stores.create(rewardsStateStoreName).withStringKeys().withIntegerValues().inMemory().build();
 (1)


kStreamBuilder.addStateStore(stateStoreSupplier);  (2)
```

(1) Creating the StateStore supplier.

(2) Adding State Store to the topology.

There are several chained method calls here so let's walk through each one:

- `create` - creates the initial `StoreFactory` with the given name. The name provided here is the same name we'll give to processors and when adding the state store to the topology.
- `withStringKeys` - method for specifying the key type (obviously String in this case) there are corresponding "withXKeys" for all default types Integer, Double, Long, ByteArray, and ByteBuffer.
- `withIntegerValues` - comparable method for specifying the value type, Integer, in this case, there are corresponding "withXValues" for all default Serde types listed above.
- `inMemory` - Specifying that this will be an in-memory key-value store.
- `build` - returns a `StateStoreSupplier` used to create a `StateStore` instance. Calling `StateStoreSupplier.get()` on any of the `StateStoreSupplier` instances returned from the `Stores` class always returns a new instance of a `StateStore`. Retrieving a new StateStore not be the case with a custom `StateStoreSupplier`, but in practice, a new instance per invocation is the expected behavior.

Here we created an in-memory key-value store with String keys and Integer values and have added the store to our application with the `KStreamBuilder.addStateStore` method. As a result, we now have the ability to use state in our processors by using a name for the state store.

We have shown one example of building a state store, but we have options at our disposal to create different types of `StateStore` instances, let's take a look at those options.

### 4.3.4 Key and Value Options

From the example above the `withStringKeys` and `withIntegerValues` are convenience methods for `withKeys(Serdes.String()` and `withValues(Serdes.Integer())` respectively. So if we want a `StateStore` of any arbitrary type, we will need to provide a `Serde` for the keys and values. The `Serde` comes into play when placing/retrieving keys and values from the `StateStore's` changelog (topic used for fault tolerance as it stores the entries for the `StateStore`). Persistent stores internally store keys and values as byte arrays in an off-heap local database, so a `Serde` is required as well.

There are four different implementations of a `StateStoreSupplier` available, two

in-memory types and two persistent types. We'll cover `StateStore` types next.

### 4.3.5 InMemoryKeyValueStoreSupplier

The first option we have is the `InMemoryKeyValueStoreSupllier`. As the name indicates, it provides an in-memory key-value store.

**Listing 4.8 Create An InMemoryKeyValueStoreSupplier**

```
Stores.create(rewardsStateStoreName).withStringKeys().withIntegerValues().inMemory().build();
```

The code example above is the same one introduced in the previous section "Adding a StateStore." Next, let's move on to another option of a state store supplier.

### 4.3.6 InMemoryLRUCacheStoreSupplier

The `InMemoryLRUCacheStoreSupplier` is an in-memory store that keeps a specified number of the most recently used key-value pairs. To create and `InMemoryLRUCacheStoreSupplier` you would add the `maxEntries(N)` (where N is between 1 and Integer.MAX_VALUE - 1) method call after the `inMemory` call.

**Listing 4.9 Create An InMemoryLRUCacheStoreSupplier**

```
Stores.create(rewardsStateStoreName).withStringKeys().withIntegerValues()
    .inMemory().maxEntries(1000).build();
```

The LRU cache is the functional equivalent of the in-memory key-value store, but it has an upper-bound on the number of key-value pairs it will store. Once reaching the max size, key-value pairs with the least amount of access are eligible for eviction from the cache.

### 4.3.7 Persistent Stores

Not only do we have the option of using in-memory stores, but there are persistent stores as well, which we'll discuss next.

Persistent `StateStore` instances provide local (off-heap) storage using RocksDB [11].

Footnote 11 github.com/facebook/rocksdb/wiki/RocksDB-Basics#3-high-level-architecture

Why choose a persistent store over an in-memory? One factor could be if you plan on saving a lot of static values. Using a persistent store would not count against Java heap space.

### 4.3.8 RocksDBKeyValueStoreSupplier

Specifying a persistent `StateStoreSupplier` over an in-memory one is a matter of switching the `inMemory()` call for `persistent()`.

**Listing 4.10 Create RocksDBKeyValueStoreSupplier**

```
Stores.create(rewardsStateStoreName).withStringKeys().withIntegerValues().persistent().build();
```

### 4.3.9 RocksDBWindowStoreSupplier

The final `StateStoreSupplier` we have is a persistent *windowed* store supplier. As the name implies, a windowed store allows for specifying a window size and retention period for retaining records. To define a persistent windowed store, we add a `windowed` call immediately after `persistent()`.

**Listing 4.11 Create RocksDBWindowStoreSupplier**

```
Stores.create(rewardsStateStoreName).withStringKeys().withIntegerValues().persistent()
    .windowed(60 * 1000, 300 * 1000, 5, false).build();
```

We pass four parameters to the `windowed` call

1. The size of the window in time.
2. Retention period (in millis) to keep records in the store.
3. Number of segments; used to "roll" the window in intervals of retention period/(number segments - 1).
4. Boolean used to determine if duplicates should be kept in the store.

Windowed `StateStoreSupplier` instances are created automatically in `KStream` and `KTable` aggregation and join operations. We cover joins in the next section. Coverage of The `KTable` API and aggregations are in the next chapter.

When creating a persistent `StateStoreSupplier` instance, we also have the option to enable caching via the `enableCaching()` method call. Caching with state stores is a broad topic and will be covered in-depth in the next chapter along with aggregations.

However, before we move on to the next section, we have a little more ground to cover regarding `StateStore` usage.

### 4.3.10 StateStore Fault Tolerance

All the `StateStoreSupplier` types have logging enabled as a default. Logging in this context means there is a Kafka topic used as a changelog to record the current values in the store and can be used to recover lost values or (re)populate a state store on startup. This logging can be disabled when using the `Stores` factory with the method `disableLogging()`. However, one should give serious consideration before disabling logging; as this makes state stores lose fault tolerance and the ability to repopulate on startup.

The changelogs for state stores are configurable via the `enableLogging(Map<String, String> config)` method. You can use any configuration parameters available for topics in the map. Again careful consideration needs to given setting these parameters.

Although we spent some considerable time discussing the types of `StateStoreSupplier` that are available, this was done purely for getting a good feel for

what options you have for constructing `StateStore` instances in Kafka Streams. In practice one would almost always refer to the parent class:

**Listing 4.12 Creating StateStoreSupplier in Practice**

```
StateStoreSupplier stateStoreSupplier = Stores.create(rewardsStateStoreName)
    .withStringKeys().withIntegerValues().persistent().build();
```

We have just learned about the in-memory and persistent state stores Kafka Streams provides and how we can include them in our streaming applications. We also learned the importance of data locality and fault tolerance when using state in a streaming application. Now that we have covered basic stateful operations and state stores let's move on to joining streams.

## 4.4 Joining Streams for Added Insight

Remember earlier in the chapter we discussed the concept of streams needing state when events in the stream do not stand alone. In this section, we are going to learn how we can take different events from two streams with the same key and combine them to form a new event. The reason we are going to do this is sometimes the state or context we need is another stream.

The best way for us to learn about joining streams is a concrete example. For this particular example, we will return to our ZMart scenario. If you recall, ZMart opened a new line of stores that carried electronics and related merchandise (CD's, DVD's and smartphones, etc.).

Trying a new approach, ZMart has partnered with a national coffee house and has embedded a cafe in each store. If you recall, in Chapter 3 you were asked to branch out the purchase transactions in those stores into two distinct streams. Here's how the topology looked for this requirement:

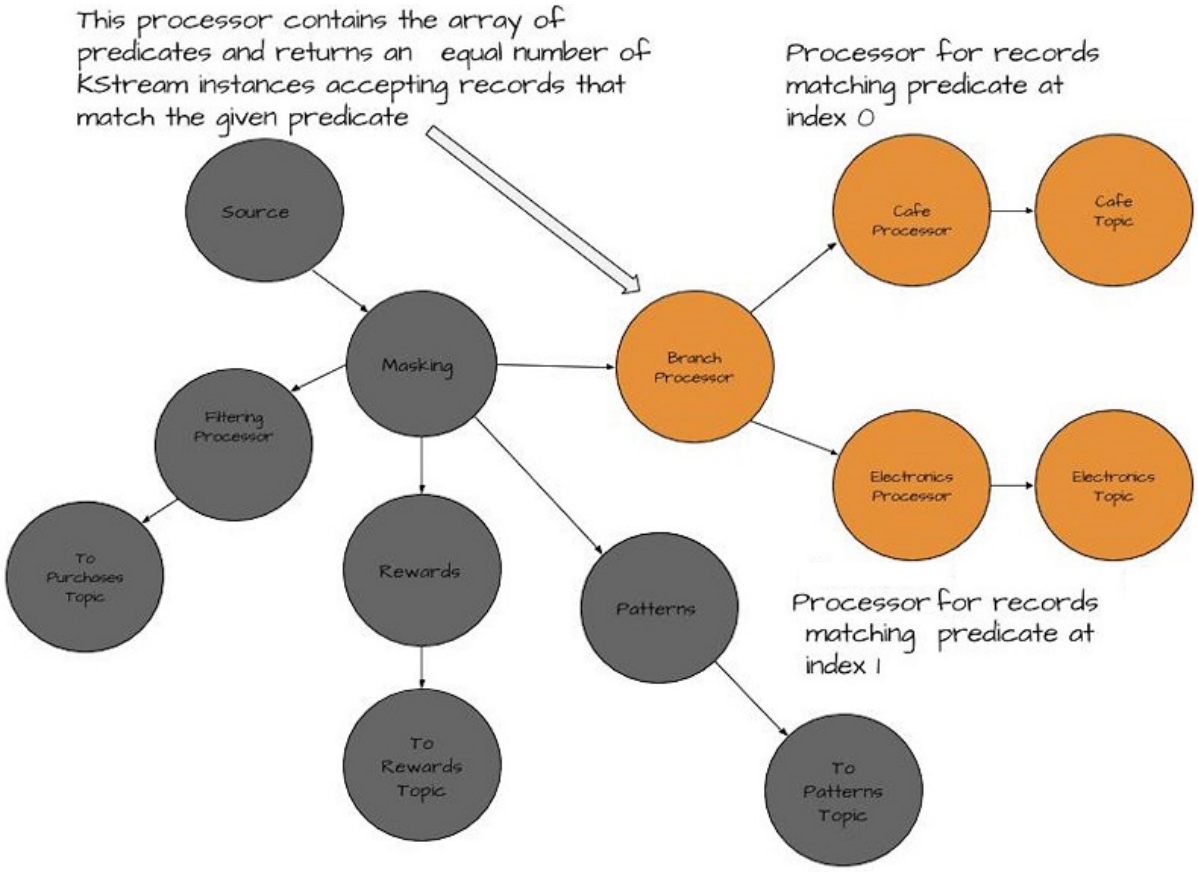



**Figure 4.11 Here's another look at the branch processor and where it stands in the overall topology.**

Initially, this approach of embedding cafe has been a big success for ZMart. Naturally, ZMart would like to see this trend continue, so they have decided to start a new program. They want to keep traffic on the electronics store high (hoping that increased traffic leads to additional purchases) by offering coupons to the cafe.

ZMart wants to give coupons to customers that have bought coffee and made a purchase in the electronics store almost immediately after the second transaction. ZMart intends to see if they can generate some sort Pavlovian response to buying in their new store.

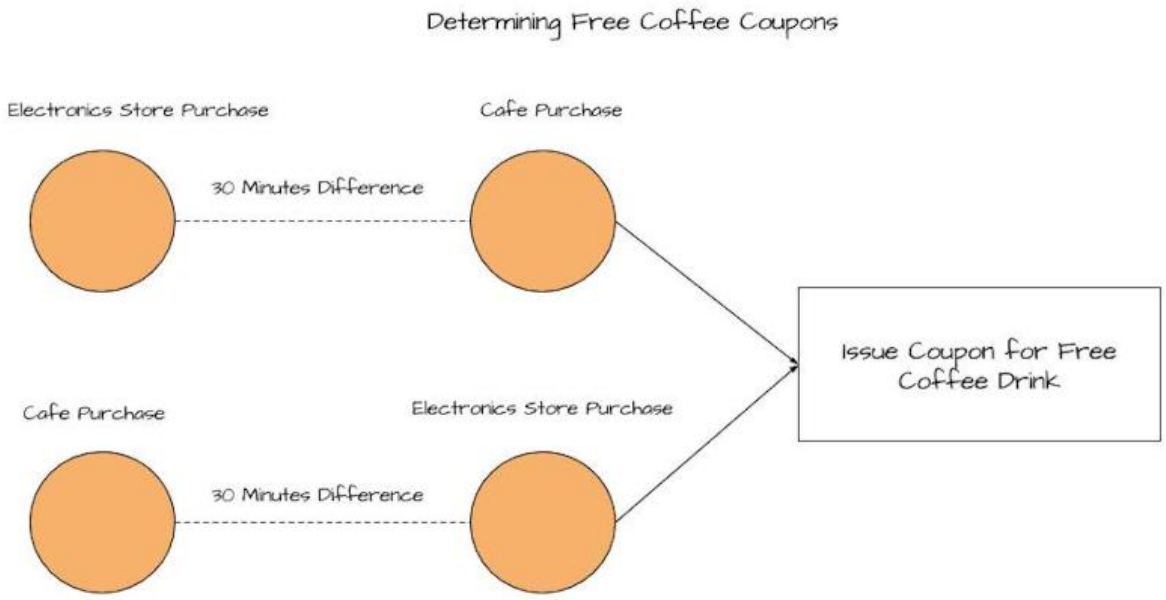



**Figure 4.12 Purchase records with timestamps within 30 minutes of each other are joined by customer id and used to issue a reward to the customer, a free coffee drink in this case.**

To make the determination to issue a coupon, we will need to join the sales from the cafe against the sales in the electronics store. Joining streams is relatively straight forward as far as the code you will need to write. Let's get starting by setting up the data we'll need to process for doing joins.

### 4.4.1 Data Setup

First off, let's take another view of the relevant portion of the topology responsible for branching the streams:

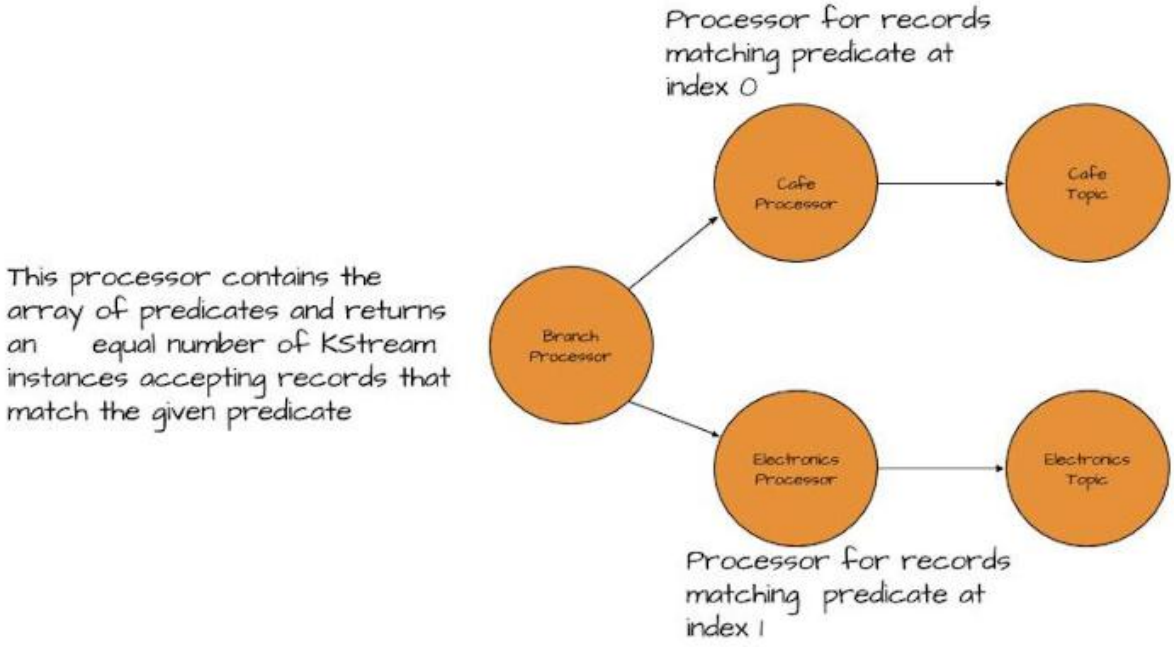



**Figure 4.13 In order to perform a join, we need more than one stream. The branch processor takes care of this for us by creating two streams; one stream contains cafe purchases and the other electronics purchases.**

In addtion, let's review the code used to implement the branching requirement:

**Listing 4.13 Branching Into Two Streams**

```
Predicate<String, Purchase> coffeePurchase = (key, purchase)
    -> purchase.getDepartment().equalsIgnoreCase("coffee");
Predicate<String, Purchase> electronicPurchase = (key, purchase) ->
    purchase.getDepartment().equalsIgnoreCase("electronics");   (1)

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

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

(1) Defining the predicates for matching records.

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

(3) Creating the branched stream.

A quick review of the code above shows to perform branching we use predicates to match incoming records into an array of `KStream` instances. The order of matching is the position in the array of `KStream` objects. The branching process drops any record not matching any predicate.

Although we have our two streams to join, we still have an additional step to perform. Remember purchase records come into our Kafka Streams application with no keys. As a result, we'll need to add another processor to generate a key containing the customer-id. We need to have populated keys because it's what we use to join records together.

### 4.4.2 Generating Keys of Customer ID To Perform Joins

To generate a key, we'll select the customer-id from the purchase data in the stream. With this in mind we'll need to update the original `KStream` instance (transactionStream) and create another processing node between it and the branch node:

**Listing 4.14 Generating New Keys**

```
KStream<String, Purchase>[] branchesStream = transactionStream.selectKey((k,v)->
   v.getCustomerId()).branch(coffeePurchase, electronicPurchase);   (1)
```

(1) Inserting the "selectKey" processing node.

As a result of the code implemented above here's an updated view of the processing topology:

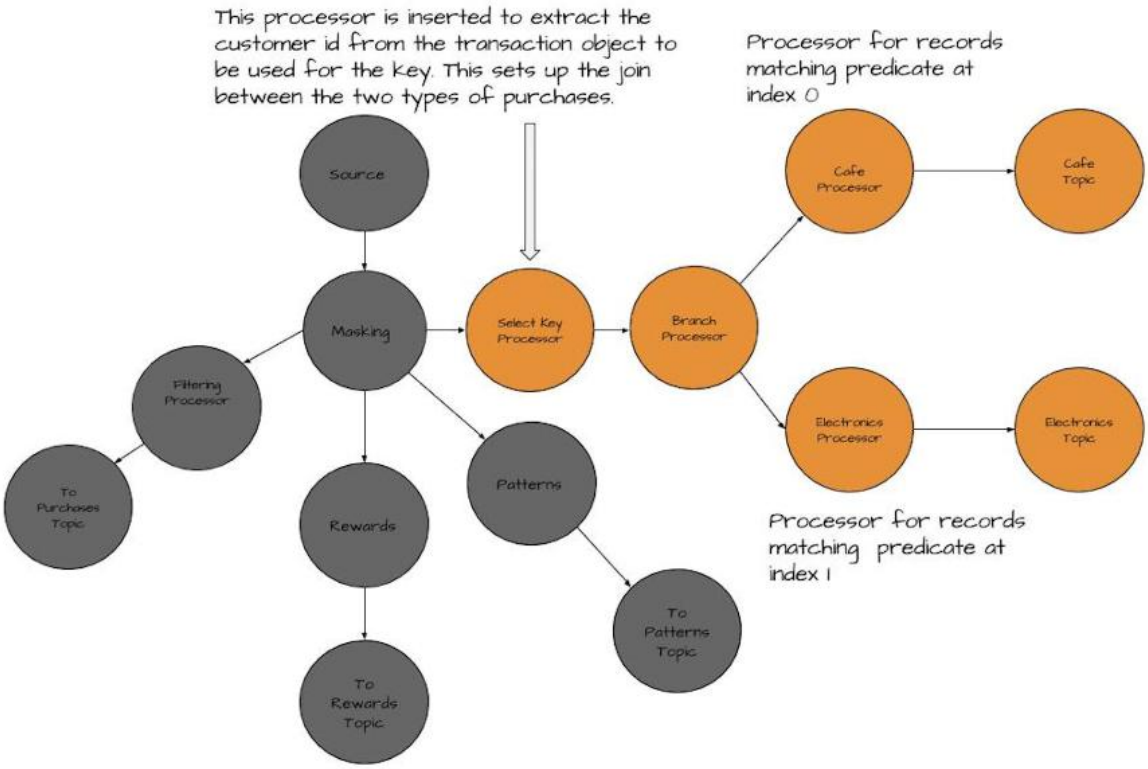



**Figure 4.14 We need to re-map the key-value purchase records into ones where the key contains the customer id. Fortunately, we can extract the customer id from the Purchase object.**

We've seen before that changing the key may require us to repartition the data. That is true in this example as well, so why didn't we see a repartitioning step?

In Kafka Streams whenever invoking a method that could result in a generating a new key (`selectKey`, `map` or `transform`) an internal boolean flag is set to "true" indicating the new `KStream` instance requires re-partitioning. With this boolean flag set, at any point, if you perform a join, reduce or aggregation operation the repartitioning is handled for you under the covers automatically.

**NOTE** **Finer Grained Control of Partition Assignment**
In the current example we wanted to repartition by the key only. But there may be instances where you either don't want to use the key or you want to use some ombination of the key and value. In these cases we can use the `StreamPartitioner<K, V>` interface as we saw in the example in section 4.2.4.

Now that we have two separate streams with populated keys we are ready for our next step: joining the streams by key.

### 4.4.3 Constructing the Join

The next step is to perform the actual join. We'll take our two branched streams and join them by using the `KStream.join` method. Our topology will look like this:

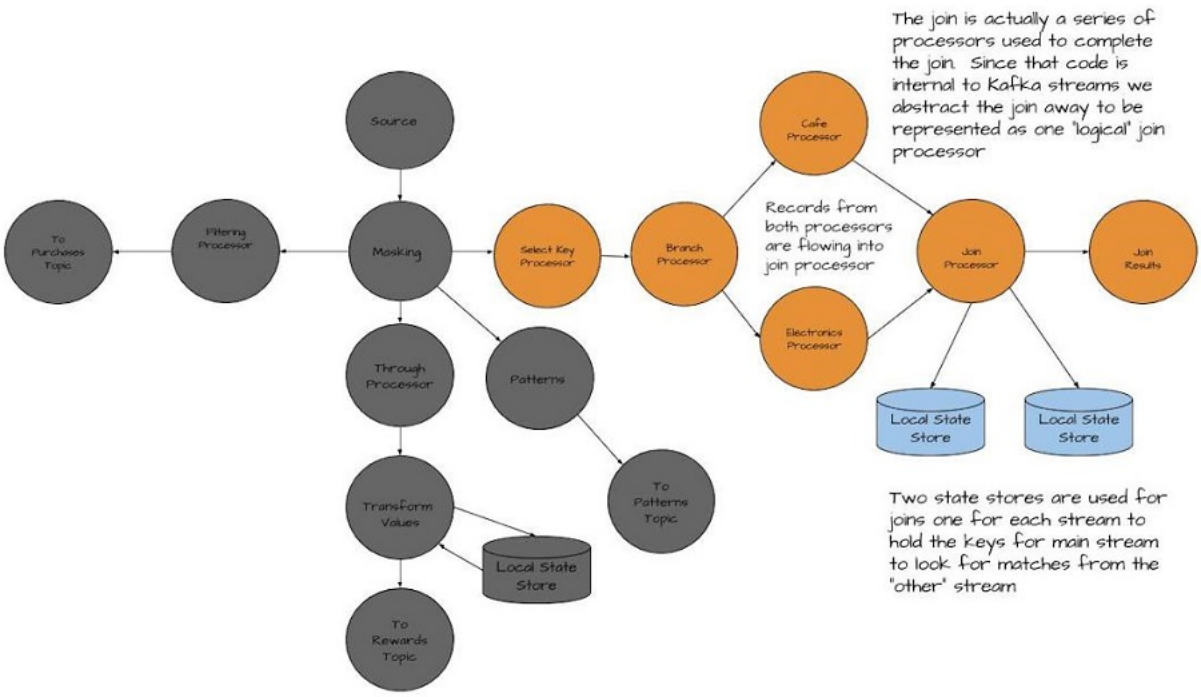



**Figure 4.15 Here's how the updated topology looks now. Both the cafe and electronics processors forward their records to the join processor. The join processor uses two state stores to search for matches on a corresponding record from the "other" stream.**

### JOINING PURCHASE RECORDS

First off, to create the joined record, we need to create an instance of a `ValueJoiner<V1, V2, R>`. The `ValueJoiner` takes two objects of the same type and returns a single object, possibly of a different type. In this case, our `ValueJoiner` takes two `Purchase` objects and returns a `CorrelatedPurchase` object. Let's take a look at the code:

**Listing 4.15 ValueJoiner Implementation**

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

    @Override
    public CorrelatedPurchase apply(Purchase purchase, Purchase purchase2) {

        CorrelatedPurchase.Builder builder = CorrelatedPurchase.newBuilder();  (1)

        List<String> purchasedItems = Arrays.asList(purchase.getItemPurchased(),
            purchase2.getItemPurchased());  (2)

        builder.withCustomerId(purchase.getCustomerId())
                .withFirstPurchaseDate(purchase.getPurchaseDate())
                .withSecondPurchaseDate(purchase2.getPurchaseDate())
                .withItemsPurchased(purchasedItems)
                .withTotalAmount(purchase.getPrice() + purchase2.getPrice());  (3)

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

(1) Instantiating the builder.

2. Adding the items purchased.
3. Calculating the total purchase amount.
4. Returning the new CorrelatedPurchase object.

To create our `CorrelatedPurchase` object, we extract some information from each purchase object. Because of the number of items we need to construct our new object, we use the builder pattern which makes the code clearer and eliminates any errors due to misplaced parameters. Next, we'll move on to implementing the join between streams.

### IMPLEMENTING THE JOIN

Now we now how we will merge records resulting from the join between the streams, let's move on to calling the actual `KStream.join` method.

**Listing 4.16 Using the Join Method**

```
KStream<String, Purchase> coffeeStream = branchesStream[COFFEE_PURCHASE];          (1)
KStream<String, Purchase> electronicsStream = branchesStream[ELECTRONICS_PURCHASE];
ValueJoiner<Purchase, Purchase, CorrelatedPurchase>
    purchaseJoiner = new PurchaseJoiner();                                         (2)
JoinWindows thirtyMinuteWindow =  JoinWindows.of(60 * 1000 * 30);

KStream<String, CorrelatedPurchase> joinedKStream = coffeeStream.join(electronicsStream,
    purchaseJoiner, thirtyMinuteWindow, stringSerde,
    purchaseSerde, purchaseSerde);                                                 (3)
joinedKStream.print("joinedStream");                                              (4)
```

1. Extracting the branched streams and repartitioning with "through" method.
2. The ValueJoiner instance used to perform the join
3. Constructing the Join.
4. Printing the join results to the console

Since we are supplying six parameters to the `KStream.join` method, let's quickly review what they are:

1. electronicsStream - the stream of electronic purchases to join with
2. purchaseJoiner - an implementation of the`ValueJoiner<V1, V2, R>` interface. The `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 our example, the `PurchaseJoiner` will add some relevant information from both `Purchase` objects and return a `CorrelatedPurchase` object.
3. thirtyMinuteWindow - A `JoinWindows` instance. The `JoinWindow.of` method specifies a maximum time difference between the two values to be included in the join. In this case, the timestamps are within 30 minutes of each other.
4. The final three parameters are the serde for the keys and the value serde for the 'calling' stream and the value serde for the secondary stream. We only have one key serde because when joining records, keys must be of the same type.

Although we specified the purchases needed to be within 30 minutes of each other, there is no order implied. As long as the timestamps are within 30 minutes of each other, the join will occur.

There are two additional methods where we can specify the order of events.

- JoinWindow.after - `stream.join(otherStream,…,JoinWindow.after(N)….)` We can also specify that one streams event needs to occur within a given time range *before* the other streams corresponding event
- JoinWindow.before - `stream.join(otherStream,…,JoinWindow.before(N),…)` the converse that one stream's event takes place within the given time range *after* the other streams event.

The time windows used for the join are an example of "sliding" windows, and we'll cover windowing operations in detail in the next chapter.

Now we have constructed our joined stream; purchases made within 30 minutes of a coffee purchase will result in a coupon for a free drink on their next visit to ZMart. Next, let's take a quick look at some of the other join options we have available to us.

### 4.4.4 Other Join Options

The example join presented here is a case of an 'inner join'. With an inner join, if either record is not present the join doesn't occur, and we don't emit a `CorrelatedPurchase` object. However, you have other options that don't require both records. These other join options are necessary if you have use cases where you need information even if the desired record for joining is not 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 is not present once the join window expires, we send the record that is available downstream. Of course of both events are present within the window then the issued record contains both events. The following image demonstrates the three possible outcomes of the outer join:

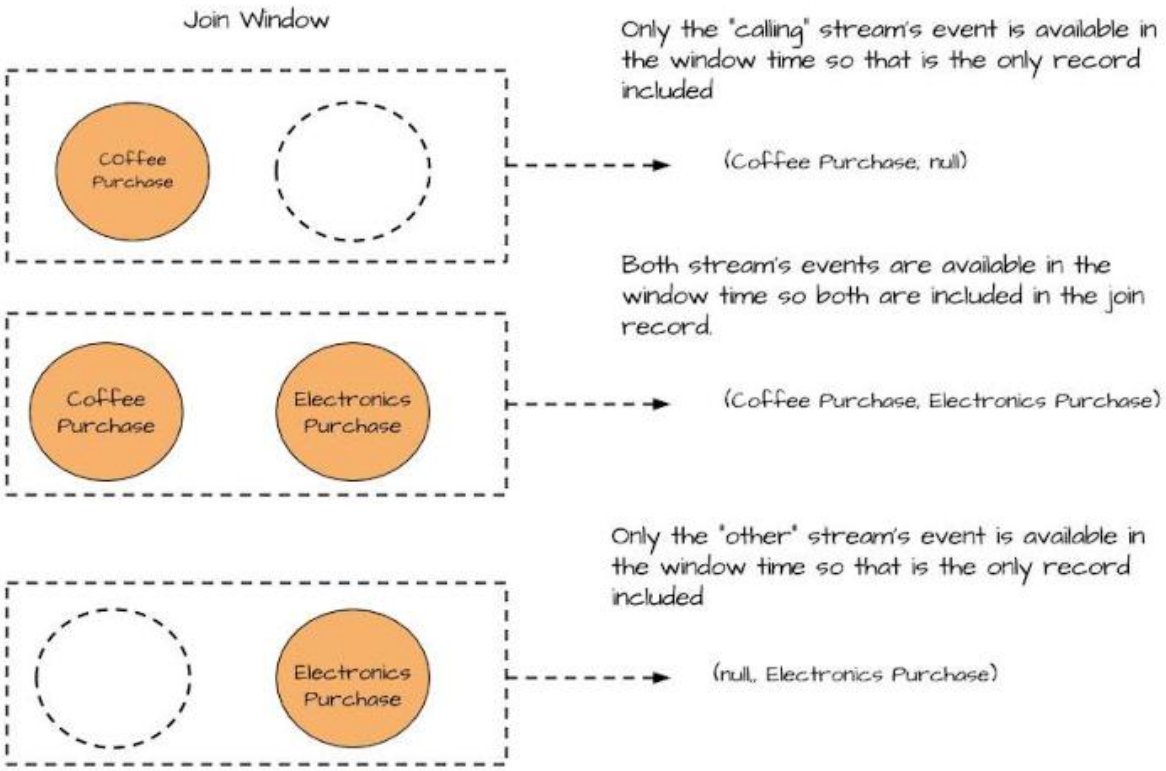



**Figure 4.16 Three outcomes are possible with outer joins, calling stream event only, both events, "other" stream event only.**

### 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 is no output at all. The graphic below shows the outcomes of the left-outer join

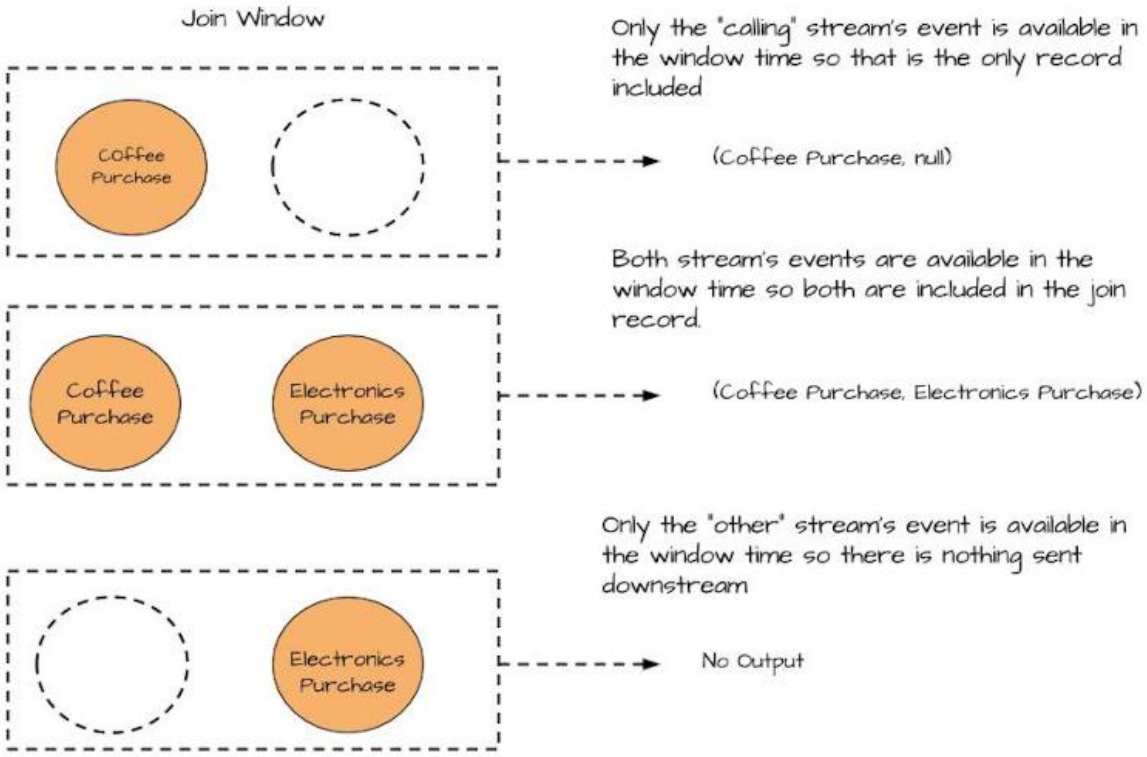



**Figure 4.17 Three outcomes are possible with the left outer join, but if only the "other"**

**stream's record is available there is no output**

In this section, we have covered joining streams, but I have casually mentioned a concept that deserves a more detailed discussion. I'm referring to timestamps and the impact they have on your Kafka Streams application.

In our join example, we specified a max time difference between events of 30 minutes. In this case, it's the difference between purchases, but if you review the code for our example, how we set or extract these timestamps is not specified. With this in mind, let's move on to our next section.

## *4.5 Timestamps In Kafka Streams*

In Chapter 2, section 5.4 we discussed timestamps in Kafka records. In this section, we'll discuss the use of timestamps specific to their role in Kafka Streams.

We need to cover timestamps because they come into play in key areas of Kafka Streams functionality such as:

1. The joining of streams.
2. Making an update to a changelog (KTable API).
3. Helps decide when the `Processor.punctuate` method is triggered (Processor API).

I know we haven't covered the KTable or Processor API at this point yet, but that's ok. We don't need them to understand this section.

In stream processing, in general, we can place timestamps into three categories summarized by this image:

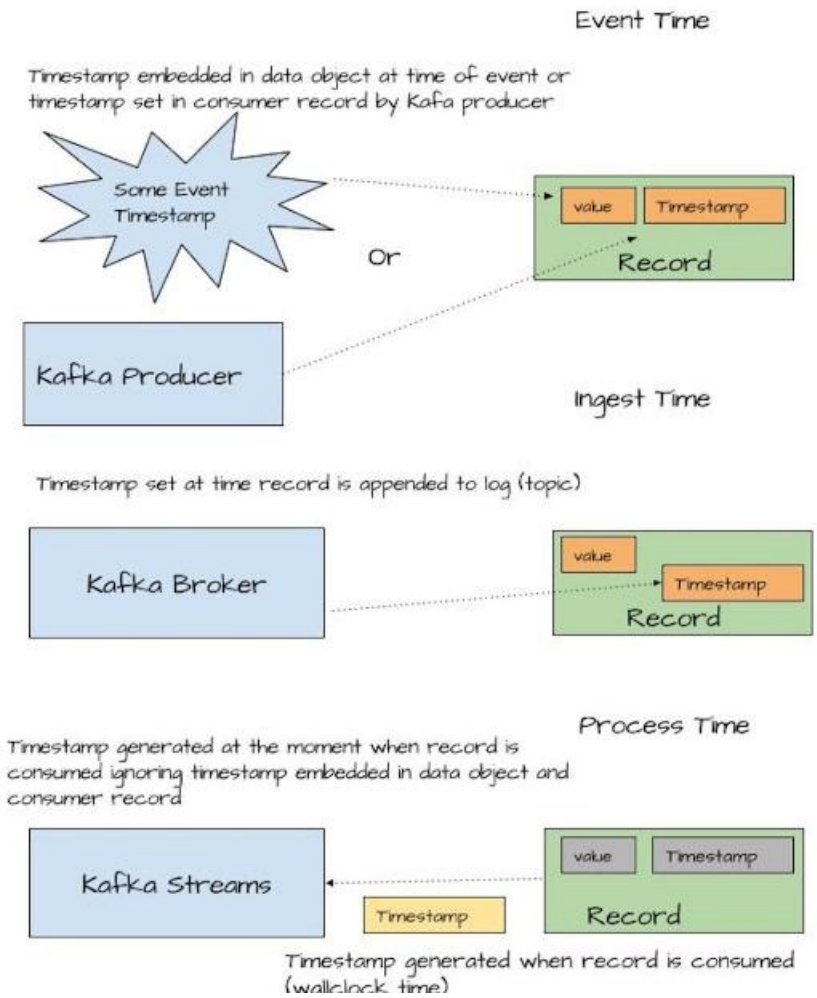



**Figure 4.18 There are three categories of timestamps in Kafka Streams. Event time is a timestamp embedded in the message itself or the timestamp set by the producer when sending the message. Ingest time is the timestamp set by the broker when appending the message to the log (topic) and process time is the timestamp when Kafka Streams processes the record.**

Let's describe each category of the timestamp in a little more detail:

1. Event Time - timestamp when the event occurred. Usually, a timestamp embedded in the object used to represent the event. For our purposes, we will consider the timestamp set when the `ProducerRecord` is created as event-time as well.
2. Ingestion Time - ingestion time is a timestamp set when the data first enters into the data processing pipeline. Here we can consider the timestamp set by the Kafka broker (assuming configuration setting of `LogAppendTime`) at ingestion time.
3. Processing Time - processing time is a timestamp set when the data or event record is first started to flow through a processing pipeline.

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

We are going to consider the three cases timestamp processing semantics mentioned above:

1. An embedded timestamp in the actual event/message object (event-time semantics).
2. Using the timestamp set when creating the `ProducerRecord` to send to Kafka (event-time semantics).

3. Using the current timestamp (current local time) when Kafka Streams ingests the record (processing-time semantics).

For event-time semantics using the timestamp placed the `ProducerRecord` is sufficient. However, there may be cases when you have different needs. For example:

- You are sending messages to Kafka with events that have their timestamps recorded in the message object. Additionally, there is some lag time in when these event objects are made available to the Kafka producer, so you want to consider only the embedded timestamp from the message object.
- You want to consider the time when your Kafka Streams application consumes records.

To enable different processing semantics, Kafka Stream provides a `TimestampExtactor` interface, with two implementations provided. Let's briefly describe the two included implementations and provide an example of implementing a custom `TimestampExtractor`.

### 4.5.1 ConsumerRecordTimestampExtractor

The `ConsumerRecorodTimestampExtractor` provides event-time processing semantics. It extracts the timestamp set by the producer or broker depending on your configuration. The figure below demonstrates the pulling the timestamp from of the `ConsumerRecord` object:

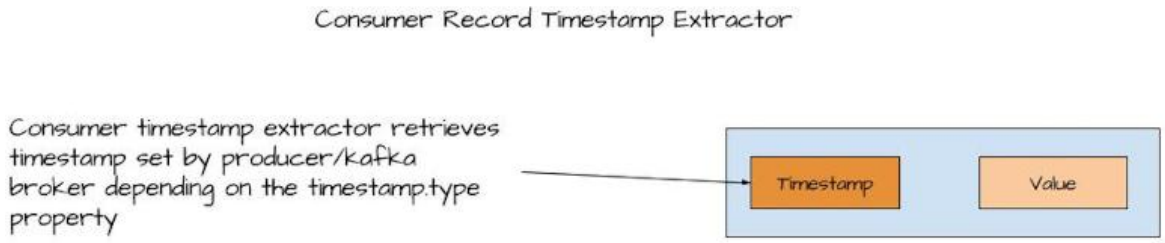



**Figure 4.19 Here we are using the timestamp found in the ConsumerRecord object. Either the producer or broker set this timestamp depending on your configuration.**

Although we are assuming the setting of `CreateTime` for the timestamp, bear in mind if you were to use the `LogAppendTime` this would return the timestamp value when the Kafka broker appended the record to the log.

### 4.5.2 WallclockTimestampExtractor

The `WallclockTimestampExtractor` provides process/ingest-time semantics. In the graphic below the timestamp and value fields are grayed out to emphasize the fact this extractor ignores any timestamps present in the `ConsumerRecord`.

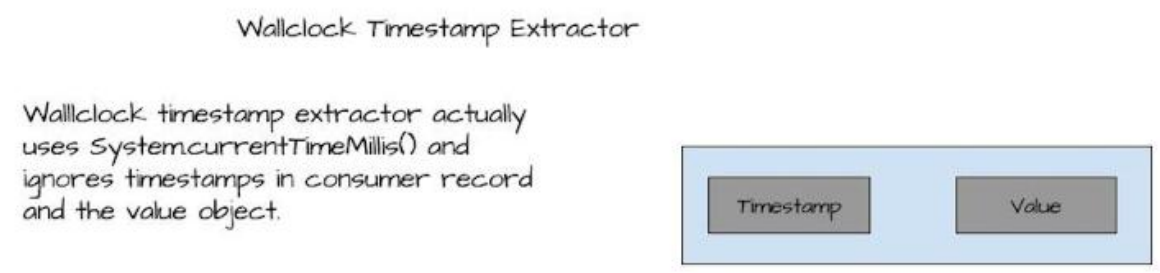



**Figure 4.20 In this case we are not using either the ConsumerRecord timestamp or any potential timestamp from the message value. This timestamp is the wallclock time when Kafka Streams starts processing the record.**

This extractor returns the current time in milliseconds via a call to the `System.currentTimeMillis()` method.

### 4.5.3 Custom TimestampExtractor

If you have a need to use an existing timestamp contained in the value object of the `ConsumerRecored` you can write a custom extractor implementing the `TimestampExtractor` interface. In the following image, we emphasize the fact our application won't use any timestamp set by Kafka (either producer or broker).

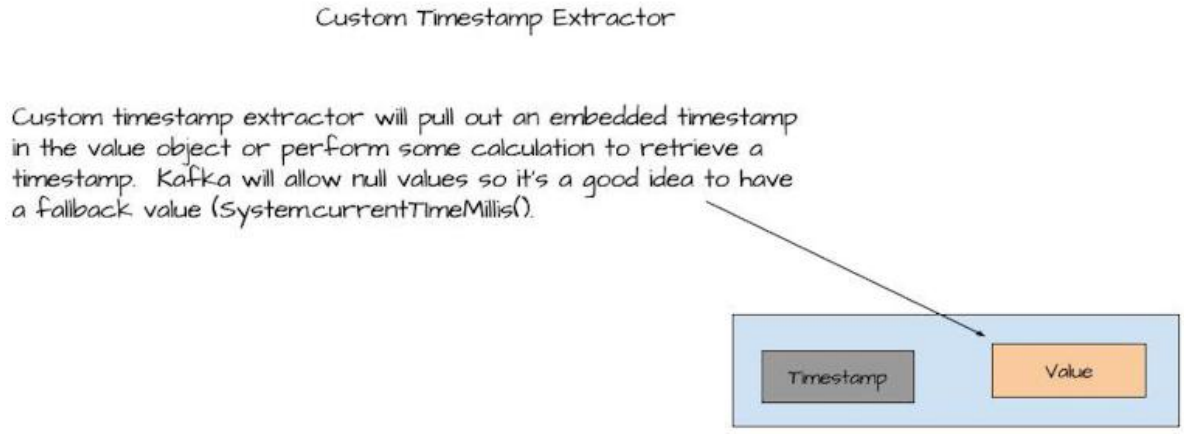



**Figure 4.21 A custom TimestampExctractor provides a timestamp from the value contained in the ConsumerRecord. This timestamp could be an existing value or one calculated from properties contained in the value object.**

While the other two concrete implementations should handle the majority of use cases, you may have a use case where you want to extract or calculate a timestamp from the actual message delivered to Kafka. Here's an example of a `TimestampExtractor` and the concrete implementation used for our join example above.

**Listing 4.17 Custom TimestampExtractor**

```
public class TransactionTimestampExtractor implements TimestampExtractor {

    @Override
    public long extract(ConsumerRecord<Object, Object> record, long previousTimestamp) {
        Purchase purchaseTransaction = (Purchase) record.value();          (1)
        return purchaseTransaction.getTransactionDate().getTime();         (2)
```

```
    }
}
```

1. Retrieving the Purchase object from the key-value pair sent to Kafka.
2. Returning the timestamp recorded at the point of sale.

In our example, we chose to use a custom `TimestampExtractor` since there could be delays when the data available to send to Kafka. In the case of delays, we want to rely on the timestamp recorded at the time of purchase. This approach will allow us to join our records even with delays in delivery or out of order arrivals.

**WARNING** **Clever TimestampExtractor Implementations**
Take care when creating a custom TimestampExtractor to not get too 'clever'. Log retention / log rolling is now 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

Finally, we will talk about how to specify which timestamp extractor to use. The timestamp extractor is defined in the `StreamsConfig` object when setting up your Kafka Streams application. The default setting is `ConsumerRecorodTimestampExtractor.class`. To use our custom implementation the configuration setting is:

**Listing 4.18 Configuring A TimestampExtractor**

```
props.put(StreamsConfig.TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
    TransactionTimestampExtractor.class);
```

We've come to the end of the discussion on timestamp usage. In the coming sections, we'll run into situations where the difference between timestamps drives some action, flushing the cache of a `KTable` for instance. While I don't expect you remember all three types of timestamp extractors, the key point here is timestamps are an important part of how Kafka/Kafka Streams function.

## 4.6 Conclusion

We've covered a lot of ground in this chapter. This section could be considered the first half of the core concepts of Kafka Streams. Although we touched on several topics here are the key takeaways:

- Stream processing needs state. Sometimes events can stand on their own, but most of the time we need additional info to make good decisions.
- Kafka Streams provides useful abstractions for stateful transformations including joins.
- StateStores 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 and use only one timestamp source with far-reaching impact. The chosen timestamp source needs careful consideration.

Next chapter we'll continue exploring state in streams with more significant operations like aggregations and grouping. We'll also explore the KTable API. Where 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. Querable state allows us to directly observe the state of our stream, without having to materialize the information by reading data from a topic in an external application.

# 5 The KTable API

In this chapter:

- Relationship Between Streams and Tables and how Streams are Tables.
- Updates to records and the KTable abstraction.
- Aggregations and Windowing Operations
- Joining KStreams and KTables
- Global KTables
- Queryable State Stores

So far we have covered the KStream API and how to add state to our Kafka Streams application. In this chapter, we're going to look deeper into adding state. Along the way, we'll get introduced to new abstraction, the KTable.

When covering the KStream API, we are talking about individual events or an event stream. If we go back to the original ZMart example, when Jane Doe made a purchase, we considered the purchase as an individual event. We didn't keep track of how often or how many purchases Jane carried out before.

If you think about the purchase event stream concerning a database, it can be considered a series of inserts. Since each record is new and unrelated to any other records, we continually insert records into a table.

Now let's add some context, by adding a key to each purchase event. Going back to Jane Doe, we now have a series of updated purchase events. Since we are using a primary key (customer-id) each purchase now is more of an update to any previous purchase Jane made. Treating an event stream with keys as inserts and events with keys as updates is what we use to define the relationship between streams and tables.

Now let's discuss what we're going to cover in this chapter. First, we'll cover more in-depth the relationship between streams and tables. This relationship is important as it helps your understanding how the KTable operates.

Second, you'll learn about 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 we introduced stateful transformations. In section 2.6 of Chapter 4, 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 within a given period. For example, how many purchases have been over the past hour, updated every ten minutes. Windowing allows us to gather data in chunks vs. having an unbounded collection.

**NOTE** **What's Windowing and Bucketing?**
Windowing and bucketing are somewhat synonymous terms. Both operate by placing information into smaller chunks or categories. While windowing implies categorizing by time, the result of either operation is the same.

Our final concept this chapter is queryable state stores. Queryable state stores are an exciting feature of Kafka Streams. I say exciting because 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.

With these points in mind let's move on to our first topic.

## 5.1 The Relationship Between Streams and Tables

If you remember in Chapter one, I defined a stream as an infinite sequence of events. But stating 'infinite' sequence of events' is generic, so let's narrow it down with a specific example.

### 5.1.1 The Record Stream

Let's say we want to view a series of stock price updates, so we'll recast our generic diagram to look like this:

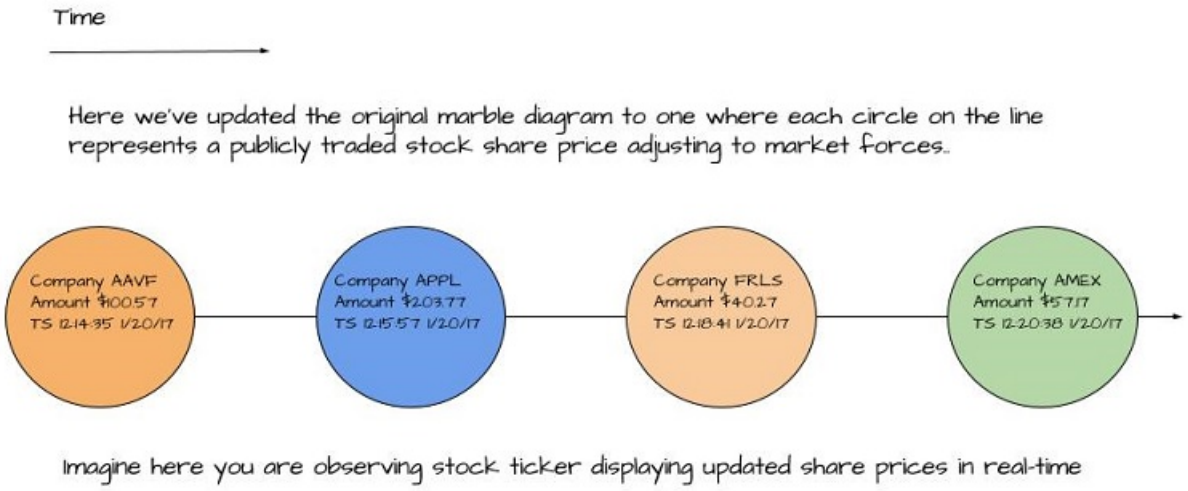



**Figure 5.1 Here we revised our simple marble diagram to be an unbounded stream of stock quotes.**

We can see from this picture that each stock price is a discrete event and aren't

related to each other. Even if the same company accounts for many price quotes, we are only looking at them one at a time.

This view of events is how the KStream works. The KStream is a stream of records. Now we can show how this concept ties into database tables. In this image we present a simple stock quote table:

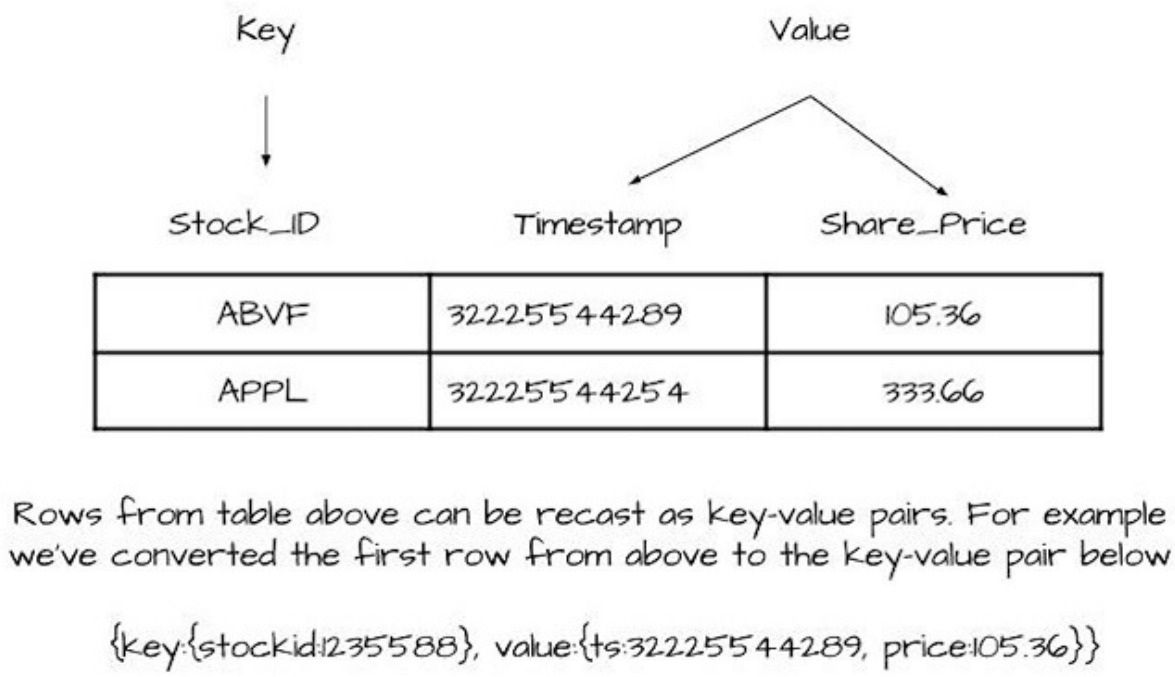



**Figure 5.2 A simple database table is representing stock prices for companies. There is a key column, and other columns are containing the value. But it is fair to say we have a key-value pair if we lump the other columns into a "value" container.**

| NOTE | **Simplifying Assumptions**<br>To keep our discussion straight forward we are going to assume a key to be a singular value. |
|---|---|

Now let's take another look at the record stream. Since each record stands on its own, the stream represents inserts into a table. In the following image we've combined the two concepts to illustrate this point:

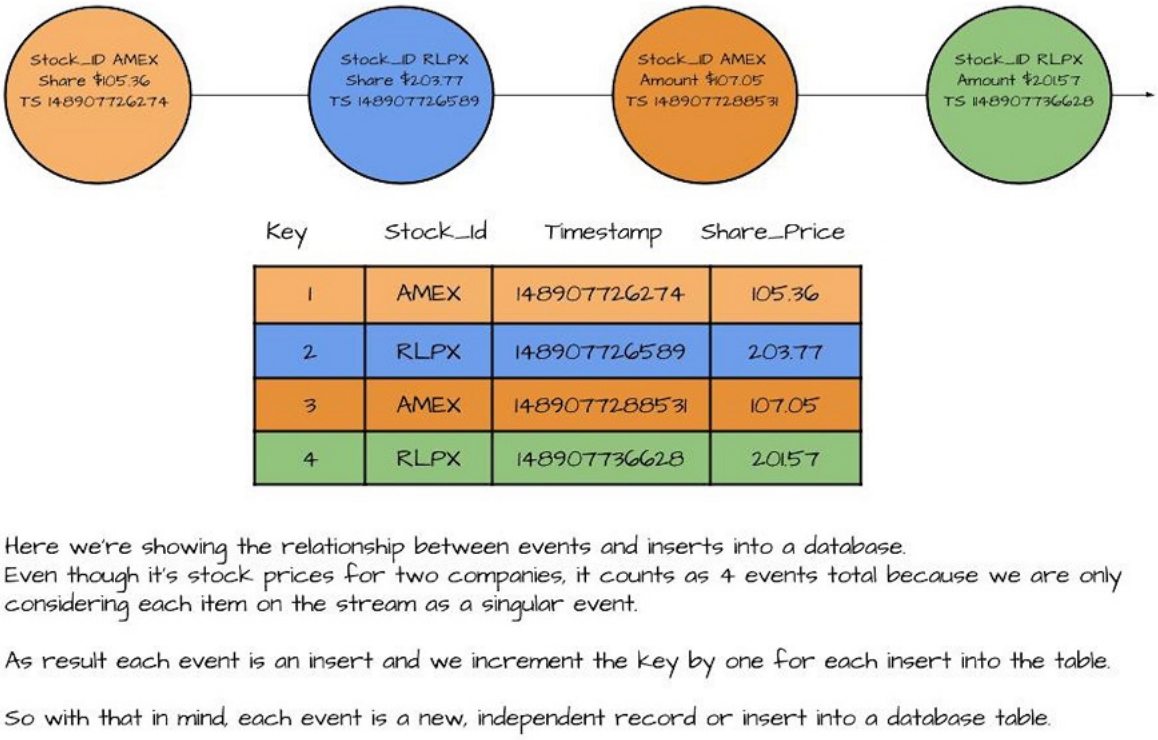



**Figure 5.3 Here we can see how a stream of individual events compares to inserts in a database table. We can flip this around and look at how we could stream each row of the table.**

What's important here is that we can view a stream of events in the same light as inserts into a table, which helps give us a deeper understanding of using streams for working with events.

The next step in our comparison of streams to tables is for us to consider the case when a flow of events does have something to do with each other.

### *5.1.2 Updates To Records or The Change Log*

Let's take the same stream of customer transactions, but now we want to track activity over time. If we alter the key to the customer id, instead of a stream of records we have a stream of updates. In this case, each record does not stand alone, so we can consider this a stream of updates or a change log. The image below demonstrates this concept:

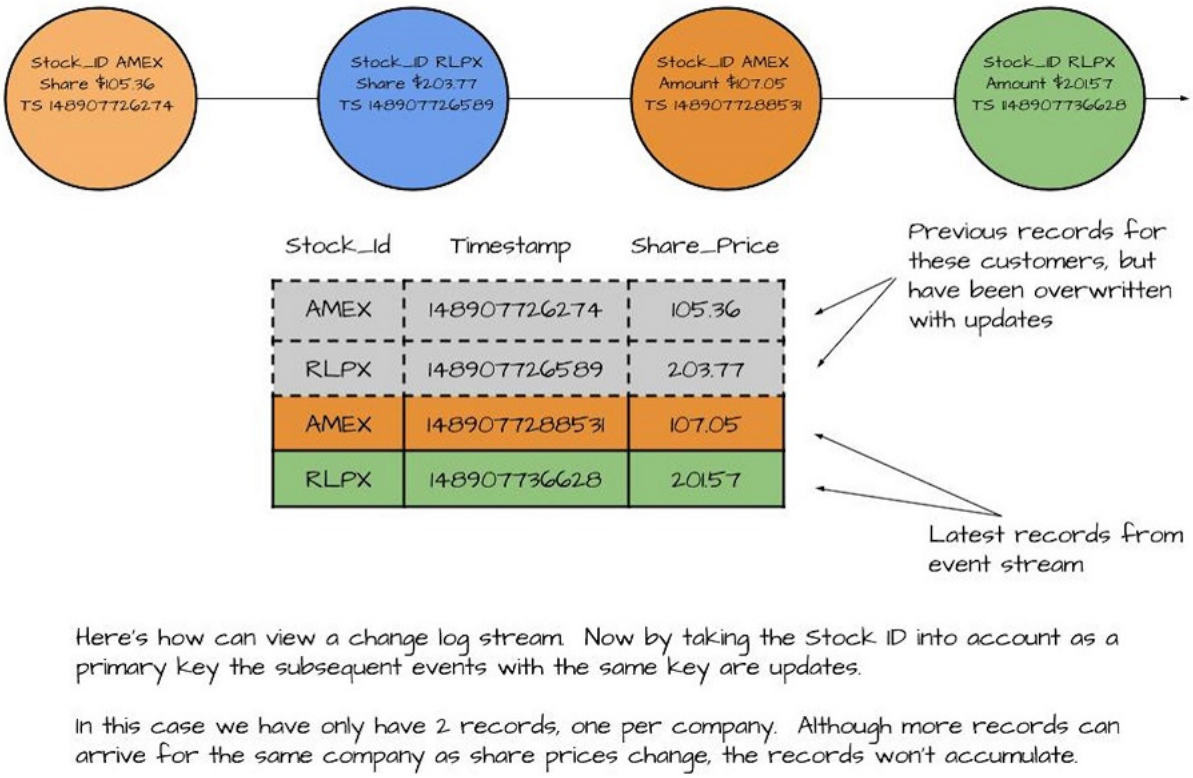



**Figure 5.4 Each record here does not stand on its own. With a record stream, we would have a total count of four events. In the case of updates or a change log, we only have one. Each incoming record overwrites the previous one with the same key.**

Here we can see the relationship between a stream of updates and a database table. Both the log and change log represent incoming records appended to the end of a file. In a log, we need to see all the records. But a change log, but we only want to keep the latest record for any given key

SIDEBAR .Log vs. Change Log

**SIDEBAR** With both the log and change log, as records come in they are always appended at the end of the file. The distinction between the two is in a log you want to read *all* records, but in change log, you only want the *latest* record for each key.

Now we need a way to trim down the log while maintaining those latest arriving records (per key). The good news is we have just such an approach. In chapter two we discussed the concept of log compaction. Let's take another look at the image for review:

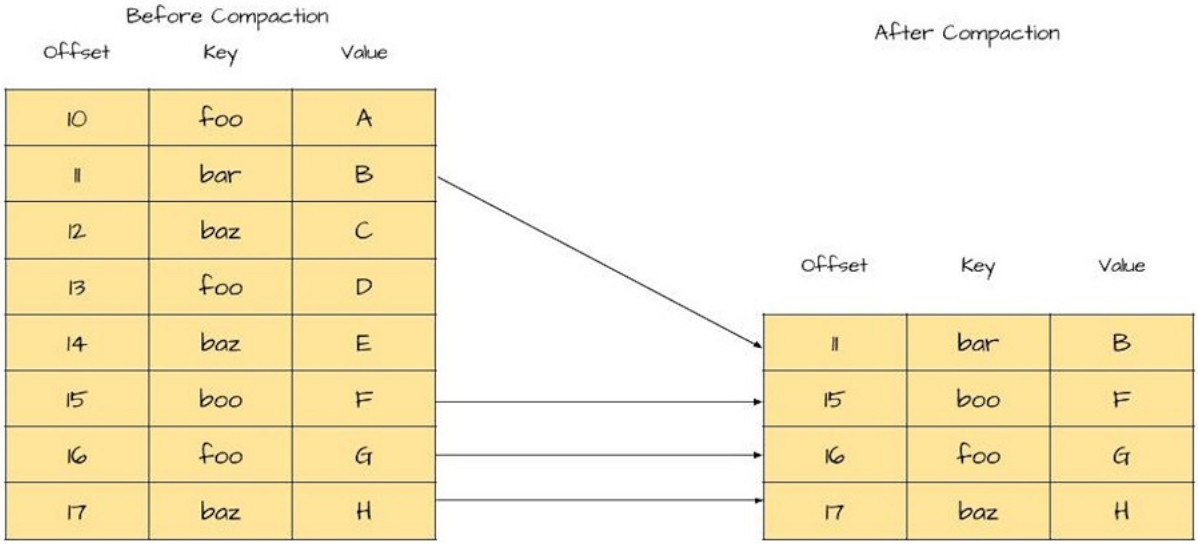



**Figure 5.5 On the left is a log before compaction. You'll notice duplicate keys with different values which are updates. On the right is the log after compaction. Notice we keep the latest value for each key, but the log is smaller in size.**

We can see the impact of the compacted log in this image. Since we only care about the latest values, we can remove older key-value pairs [12].

Footnote 12 This section derived information from Jay Kreps in www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/

We're already familiar with record or event streams from working with KStreams. For a change log or stream of updates, Kafka Streams provides the KTable. Now that we've established the relationship between streams and tables, our next step is to compare an event stream to an update stream. We'll make this comparison by watching a `KStream` and `KTable` in action side by side.

### 5.1.3 Event Streams vs Update Streams

In this section, we'll show event and update streams by comparing the KStream to the KTable. We'll do this by running a simple stock ticker application. The stock ticker application writes the current share price for three (fictitious!) companies. We'll produce three iterations of stock quotes for a total of 9 records.

A KStream and KTable instance reads the records and writes them to the console via the `print` method.

The following image shows the results of running the application:

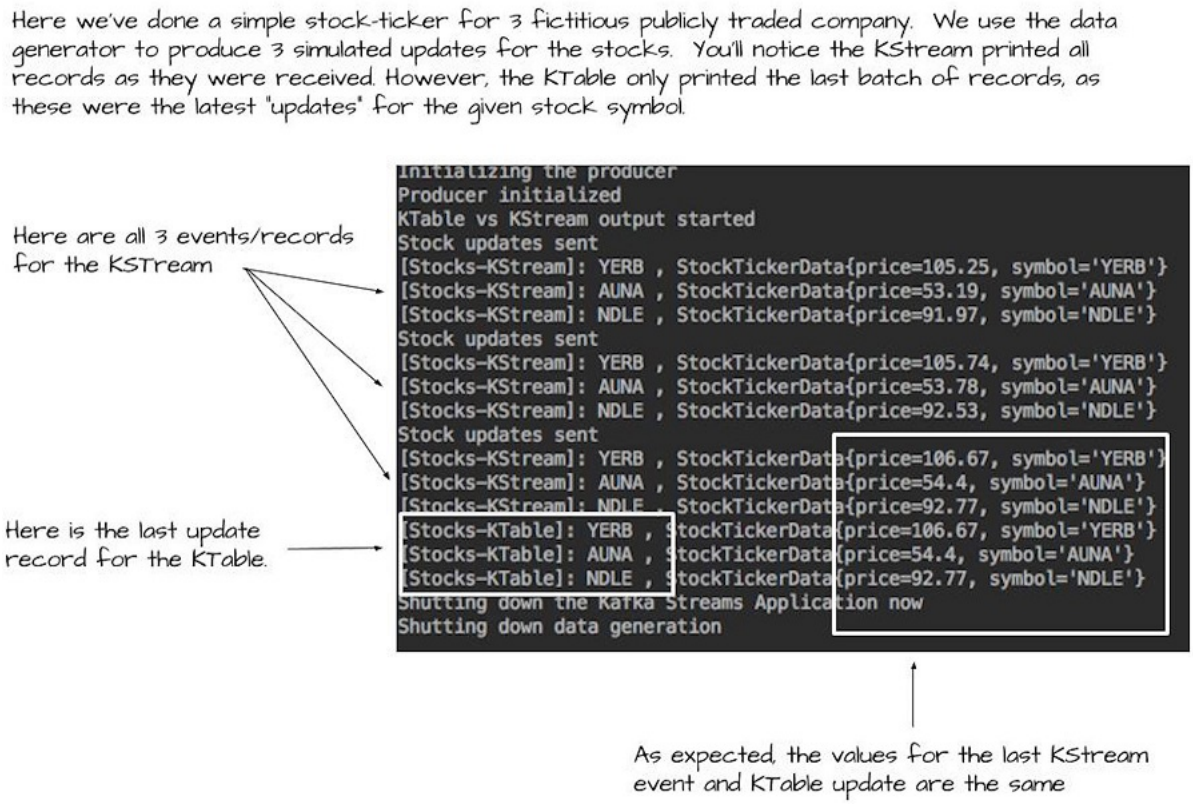



**Figure 5.6 KTable vs KStream printing results from messages with the same keys**

As you can see from the results, the KStream printed all nine records out. We expect the KStream to behave this way as it views each record as an individual event. In contrast, the KTable only printed three records. From what we learned above, though, this is how we'd expect the KTable to perform.

**NOTE** **About Primary Keys**

The example in the previous section demonstrated 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. As you can't update a record in a database table without having the key, the same rule applies here as well.

From the KTable point of view, it didn't receive nine individual records. The KTable received three original records and updates to those records twice. The KTable only printed the last round of updates. Notice the KTable records are the same as the last three records published by the KStream. We'll discuss the mechanisms of how the KTable emitted only the updates in the next section.

Here is our program prining stock ticker results to the console:

**Listing 5.1 KTable and KStreams Printing to the Console**

```
KTable<String, StockTickerData> stockTickerTable =
    kStreamBuilder.table(STOCK_TICKER_TABLE_TOPIC, "ticker-store");  (1)
```

```
KStream<String, StockTickerData> stockTickerStream =
    kStreamBuilder.stream(STOCK_TICKER_STREAM_TOPIC);  ❷

stockTickerTable.toStream().print( "Stocks-KTable");  ❸
stockTickerStream.print( "Stocks-KStream");  ❹
```

❶ Creating the KTable instance.

❷ Creating the KStream instance.

❸ KTable printing results to the console.

❹ KStream printing results to the console.

**NOTE** **Using Default Serdes**

I should point out here that in creating the KTable and KStream we did not specify any `Serdes` to use. The same is true with both calls to the `print` method. We were able to do this because we registered our default `Serdes` in the configuration like so:

```
props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
    Serdes.String().getClass().getName());
props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
    StreamsSerdes.StockTickerSerde().getClass().getName());
```

If we used different types then we'd need to provide `Serdes` in any method reading or writing records.

The takeaway here is records in a stream with the same key are updates to events, not new events by themselves. A stream of updates is the main concept behind working with the KTable.

Now we've seen the KTable in action, let's discuss the mechanisms behind the KTable functionality.

## 5.2 Record Updates and KTable Configuration

To explain how the KTable functions we should ask the following questions: how and where are records stored and how does the 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 aren't going to want an update each time the count increments by one.

To answer the first question let's take a look at the line creating the KTable:

```
kStreamBuilder.table(STOCK_TICKER_TABLE_TOPIC, "ticker-store");
```

The first parameter is the topic name and the second parameter is the name of the state store, but this case, you don't need to do anything. The `KStreamBuilder` creates the

`StateStore` when constructing the KTable instance. So we have our answer to the first question. The `KTable` uses the local state integrated with Kafka Streams for storage (We covered state stores in chapter 4, sections 3.3 through 3.10).

Now let's move on to what determines when the `KTable` emits updates to downstream processors? To answer this question, we need to consider a few factors:

1. The number of records flowing into the application. Higher data rates would tend to increase the rate of emitting updated records.
2. How many distinct keys you have in your data. Again the greater the number of distinct keys may lead to more updates sent downstream.
3. The configuration parameters "cache.max.bytes.buffering" and "commit.interval.ms"

From this list, we'll only cover what we can control, the configuration parameters. First, let's take a look at the "cache.max.bytes.buffering."

### 5.2.1 Setting Cache Buffering Size

The "cache.max.bytes.buffering" setting controls the amount of memory for the record cache. The memory gets divided across the number of stream threads. (The number of stream threads is another configuration parameter).

**WARNING** **Disabling the Cache**
To turn of caching one can set the "cache.max.bytes.buffering" setting to "0". But this setting will result in sending every KTable update downstream. This could lead to a large data flow if you have consistent updates to the same keys.

The cache deduplicates updates to records with the same key. All other factors being equal, a larger cache will reduce the amount of updates emitted. Here's an image describing the flow of the cache.

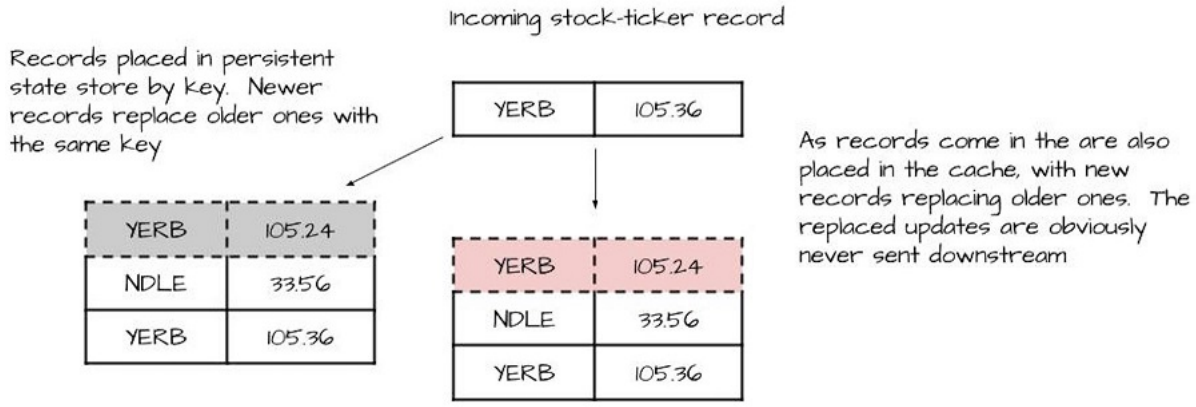



**Figure 5.7 KTable caching of records reduces duplicate date from consecutive updates from going downstream**

As we can see in the picture above, not all records get forwarded downstream. The cache automatically deduplicates records with the same key. Only the latest record for any given key is in the cache. As a result, flushing the cache forwards only the most

recent updates downstream.

### 5.2.2 Setting the Commit Interval

The other setting is the "commit.interval.ms" parameter. The commit interval determines how often (in milliseconds) to save the state of a processor. When saving the state of the processor (committing), also forces a cache flush. The key point here is that a commit forces the forwarding of the latest updated, deduplicated records downstream.

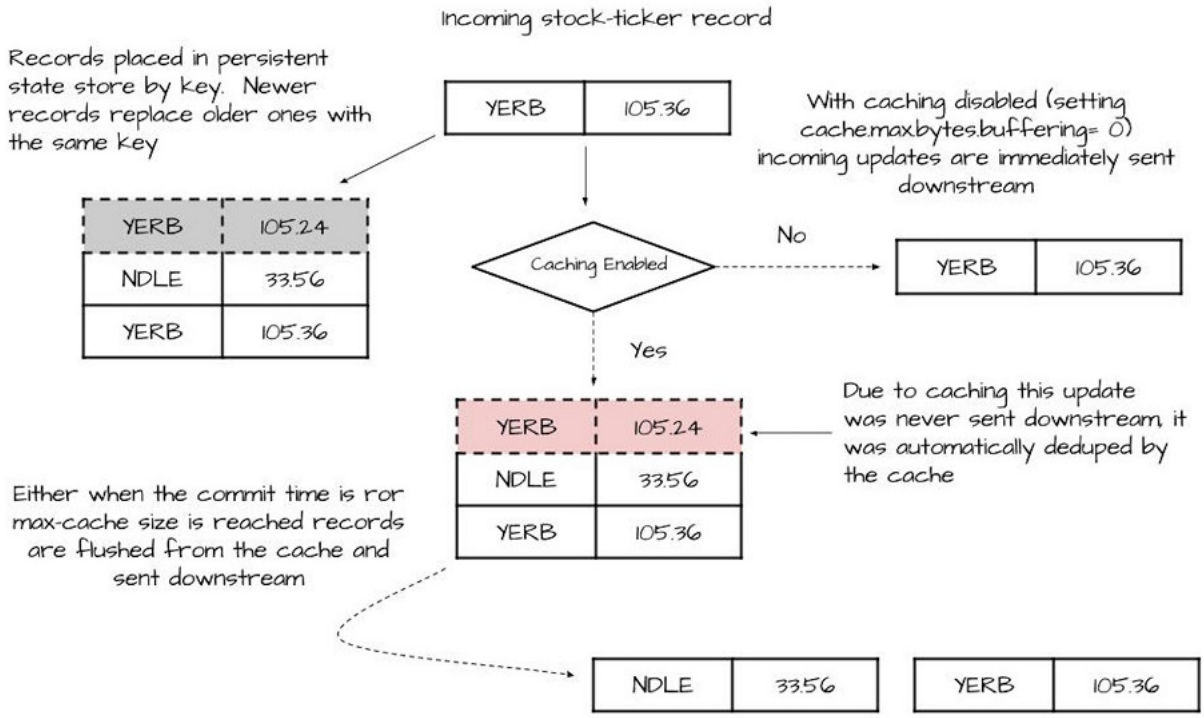



**Figure 5.8 Full caching workflow. If caching enabled, records are de-duped and sent downstream on cache-flush or commit**

In the full caching workflow above, we see two forces at play when it comes to sending records downstream. Either a commit or the cache hittinh its configured max size, will send records downstream. We should also be able to see that by disabling the cache, all records are sent downstream, including all duplicates. While there may be some use cases requiring all records, generally speaking, it's best to have caching enabled when using a KTable.

So 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. But, a longer commit interval could lead to more updates as 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 thirty seconds (commit time) and ten MB (cache size).

The key thing to remember is the rate of updated records sent from a KTable is configurable.

Next, let's take a look at how we can use the KTable in our applications.

## 5.3 Aggregations and Windowing Operations

In this section, we move on to cover some of the most potent parts of Kafka Streams. We can consider this section the "capstone" of your Kafka Streams experience. So far we've covered:

1. How to set up a processing topology
2. How to use state in our streaming application.
3. Performing joins between streams.
4. The difference between an event stream (KStream) and an update stream (KTable).

In the examples to follow we are going to tie all these elements together. Additionally, we are going to introduce Windowing another powerful tool in streaming applications. In the following section, we'll tackle our first example, a straightforward aggregation.

### 5.3.1 Aggregating Share Volume by Industry

All our examples to this point have been in the retails sales arena. We are going to shift gears and draw our examples from the financial industry in this chapter.

We are going to take on the role of being a day trader for the next set of examples. In the first one, we'll track the share volume of companies across a list of selected industries. In particular, we are only interested in the top five (again by share volume) in each industry.

Aggregation and grouping are necessary tools to use when working with streaming data. Many times reviewing single records as they arrive is not enough. To gain the information/insight you need often, we need grouping and combining of some sort.

To do this aggregation, we'll need a few steps to set the data up in the correct format. At a high level here are the steps we'll take:

1. Create a source from a topic publishing raw stock-trade information. We'll need to map a `StockTransaction` object into a `StockVolume` object. The reason for performing this mapping step is simple. The `StockTransaction` object contains a lot of information about the trade itself. But we only want the volume of shares involved in the trade.
2. Then we'll group `StockVolume` by its ticker symbol. Once grouped by symbol, we can reduce to a rolling total of share volume. We should note here calling `KStream.groupBy` returns a `KGroupedStream` instance. Then calling `KGroupedStream.reduce` is what gets us to our `KTable` instance.

Let's pause here for a minute and take a look at an image of what we're building so far:

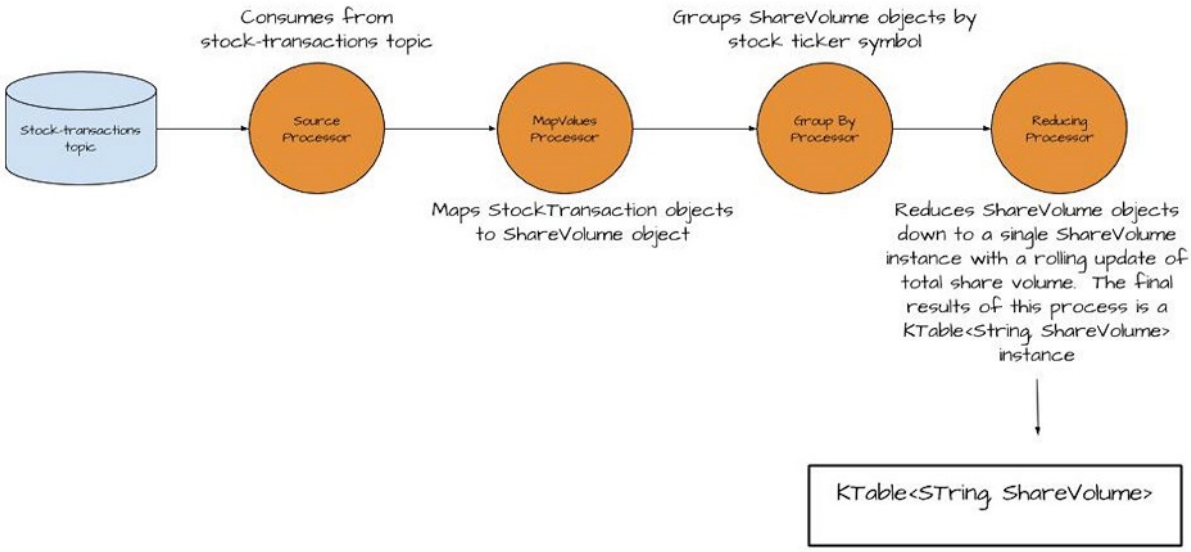



**Figure 5.9 Mapping and Reducing StockTransaction objects into StockVolume objects then reducing to rolling total.**

This topology is something we are familiar with by now. Now we'll look at the code behind the topology:

**Listing 5.2 Source for map-reduce of stock transactions**

```
KTable<String, ShareVolume> shareVolume = kStreamBuilder.stream(EARLIEST,
   stringSerde, stockTransactionSerde, STOCK_TOPIC)                    ❶
    .mapValues(st -> ShareVolume.newBuilder(st).build())               ❷
    .groupBy((k, v) -> v.getSymbol(), stringSerde, shareVolumeSerde)    ❸
    .reduce(ShareVolume::reduce, "stock-transaction-reductions");       ❹
```

❶ Source processor consuming from topic

❷ Mapping StockTransaction objects to ShareVolume objects

❸ Grouping the ShareVolume objects by their stock ticker symbol

❹ Reducing ShareVolume objects to contain a rolling aggregate of share volume

This code is concise, and we are squeezing a lot of power into a few number of lines.

If we look at the first parameter of the `kSteamBuilder.stream` method we see something new. It's the `AutoOffsetReset.EARLIEST` enum (there is also a LATEST value as well). This enum allows us to specify the offset reset strategy for each KStream/KTable. Using the enum takes precedence over the offset reset setting in the streams config.

While it's clear what `mapValues` and `groupBy` are doing, let's look into our reduce method:

**Listing 5.3 Source of the ShareVolume reduce method**

```
public static ShareVolume reduce(ShareVolume csv1, ShareVolume csv2) {
```

```
        Builder builder = newBuilder(csv1);                    (1)
        builder.shares = csv1.shares + csv2.shares;            (2)
        return builder.build();                                (3)
}
```

(1) Using a Builder for a copy constructor.

(2) Setting the number of shares to the total of both ShareVolume objects.

(3) Calling build and returning a new ShareVolume object.

**NOTE** **Builder Pattern usage**
While we've seen the Builder Pattern in use earlier in the book, we see it here in a somewhat different context. In this example, we are using the Builder to make a copy of an object and update a field without modifying the original object.

The `ShareVolume.reduce` method gives us our rolling total of share volume. The outcome of entire processing chain is a `KTable<String, StockVolume>` object. Now we can see the role of the KTable. As `ShareVolume` objects come in, the associated KTable keeps the most recent update.

**NOTE** **Aggregation vs. Reduce**
Why did we choose to do a reduce over an aggregation? While reducing is a form of aggregation, a reduce operation will yield the same type of object. An aggregation also sums results but could return a different type of object.

Next, we'll take our `KTable` and use it to perform a top-five aggregation (by share volume) summary. The steps we'll take here are similar to the steps we took for our first operation:

1. We need to perform another `groupBy` operation. We'll group the individual `ShareVolume` objects by their industry.
2. With the `ShareVolume` grouped by industry, we can start to add them together. This time our aggregation object is a fixed size priority queue. The fixed size queue only keeps the top five companies by share volume.
3. Then we'll map the queue into a string reporting the top five stocks per industry by share volume.
4. The string result is then written out to a topic.

Before we look at the code, let's take a look at a topology graph to cement our understanding of the data flow:

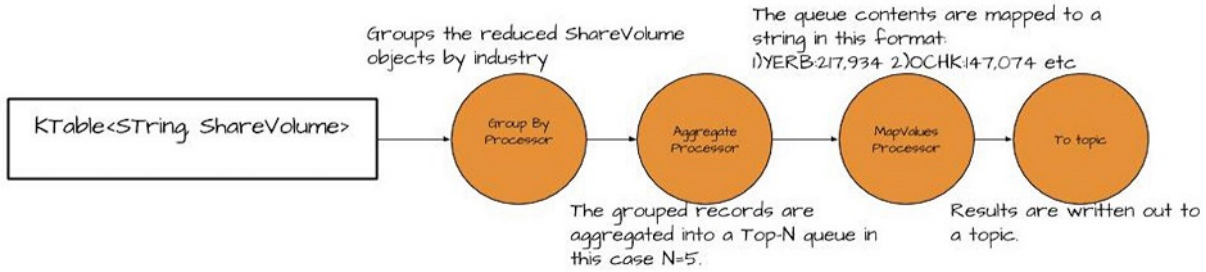



**Figure 5.10 Topology for grouping by industry and aggregating by top 5. Then we map the top 5 queue to a string and write the string out to a topic**

We can see from the image of the graph above, this second round of processing is straight forward. With a clear picture of the structure of this second round of processing, it's time to look at the source code:

**Listing 5.4 KTable GroupBy and Aggregation**

```
Comparator<ShareVolume> comparator = (st1, st2) -> st2.getShares() - st1.getShares();
FixedSizePriorityQueue<ShareVolume> fixedQueue =
    new FixedSizePriorityQueue<>(comparator, 5);

shareVolume.groupBy((k, v) -> KeyValue.pair(v.getIndustry(), v),
    stringSerde, shareVolumeSerde)                                    (1)
    .aggregate(() -> fixedQueue,                                      (2)
               (k, v, agg) -> agg.add(v),                             (3)
               (k, v, agg) -> agg.remove(v),                          (4)
               fixedSizePriorityQueueSerde,                           (5)
               "volume-shares-industry-store")                        (6)
    .mapValues(valueMapper)                                           (7)
    .print("Stock volume by Industry");                               (8)
```

1. Grouping by industry and provided required Serdes
2. The Aggregate intitializer an instance of the FixedSizePriorityQueue class (custom for demonstration purposes not for production!)
3. Aggregate adder for new updates
4. Aggregate remover for removing old updates
5. Serde for the Aggregator
6. Name of the state store for aggregator held in KTable.
7. ValueMapper instance to convert aggregator to a string used for reporting.
8. Printing results to console (in production this would be a to("some_topic") )

In our initializer there is `fixedQueue` variable. It comes from these lines earlier in the program: .FixedSizePriorityQueue creation

```
Comparator<ShareVolume> comparator =
    (st1, st2) -> st2.getShares() - st1.getShares();                  (1)
FixedSizePriorityQueue<ShareVolume> fixedQueue =
```

```
new FixedSizePriorityQueue<>(comparator, 5);  ❷
```

❶ Comparator used for determining order

❷ FixedSizePriorityQueue - a wrapper around a TreeSet.

For me, the code sample is easier to understand by viewing the queue creation alone. Although I'm sure, we could make a case for the opposite approach. With this clarification out of the way let's go over the rest of the code sample.

In this code example, we've seen the `groupBy`, `mapValues` and `print` calls before so we won't go over them again. But we haven't seen the KTable version of aggregate before, so let's take a minute to discuss it.

To review what makes a KTable unique is that it considers new records with the same key are updates. The KTable replaces the old record with the new one. Aggregation works in the same manner. It's an aggregation of the most recent records with the same key. So as a record arrives we add it to the `FixedSizePriorotyQueue`, but if an another record exists with the same key, we remove the old record.

What this means is our aggregator, `FixedSizePriorityQueue`, does not aggregate *all* values with the same key. Instead, the aggregator keeps a running tally of the top N stocks which have the largest volume. Each record coming has the total volume of shares traded so far. Our `KTable` will show us which companies have the top number of shares traded at the moment; we don't want a running aggregation of each update.

We should pause for a minute and review some key points I want you to remember:

1. We've learned how to group values in a KTable by a common key.
2. Additionally, we've learned how to perform useful operations like reducing and aggregation with those grouped values.

The ability to perform these operations is important when you are trying to make sense of or determine what your data is telling you as it flows through your KafkaStreams application.

Also, we've brought together some of the key concepts discussed earlier in the book. In chapter 4 we learned about the importance of having fault-tolerant, local state for streaming applications. The first example of this chapter shows why local state is so important. To briefly restate those reasons, state allows you to keep track of what you have seen. Having *local* access does not introduce any network latency and is isolated from other processes.

Anytime we execute a reducing or aggregation operation we provide the name of state store. Reductions and aggregation operations return a KTable instance. The KTable uses the state store to replace older results with the newer ones.

Not every update gets forwarded downstream, and that is important because we perform aggregation operations to gather *summary* information. If we didn't use local state, the KTable would forward *every* aggregation or reduction result. Sending every

incremental update downstream is essentially an event-stream and renders the aggregation operations less useful.

Next, we'll show how to perform aggregation like operations over distinct periods of time called windowing.

### 5.3.2 Windowing Operations

In the first example in this chapter, we provided a "rolling" reduction and aggregation. The application performed a continuous reduction of share volume, then a top 5 aggregation of shares traded in the stock market. Sometimes we'll want an ongoing aggregation/reduction of results. Other times we want only to do operations over a given time range.

How many stock transactions have involved a given company over the last ten minutes? How many users have clicked to view a new advertisement in the last 15 minutes? While the applications perform these operations many times, the results are only for the defined period or "window" of time.

#### COUNTING STOCK TRANSACTIONS BY CUSTOMER

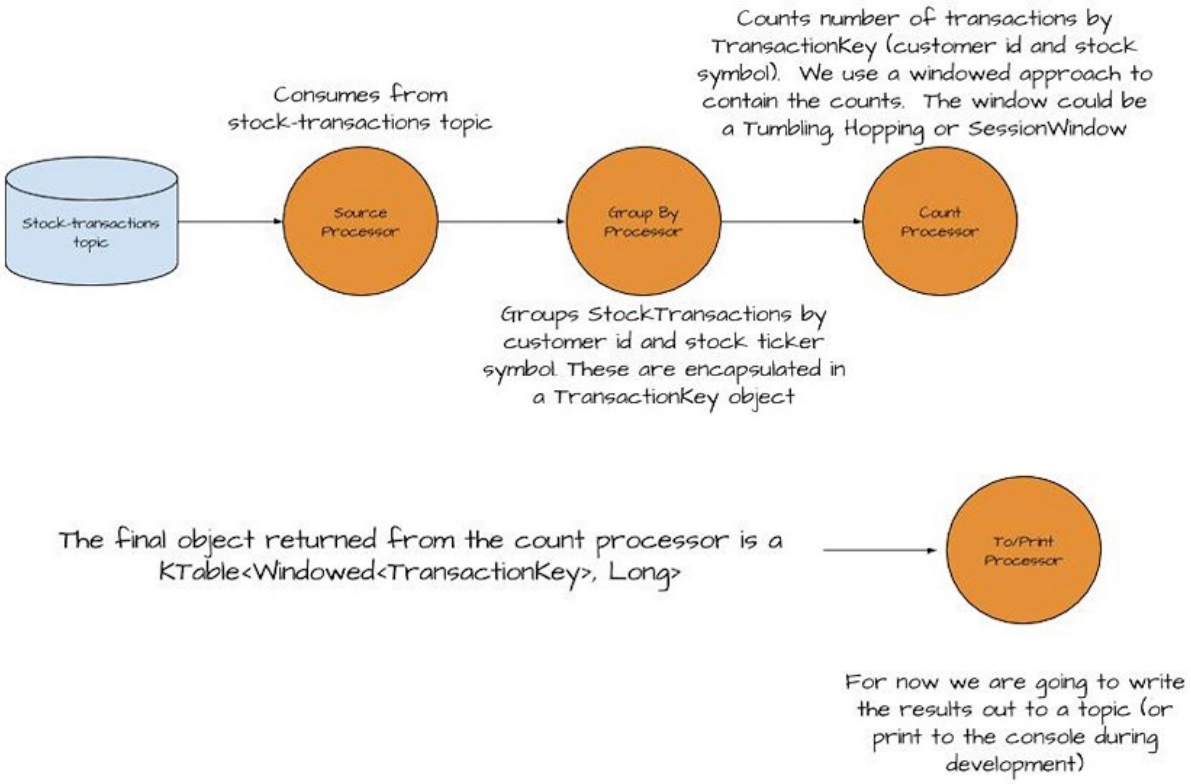
For our next example, we'll have the following scenario. We'll want to track stock transactions for a handful of traders. These could be large institutional traders to financially savvy individuals. We have two potential reasons for doing this tracking.

One case is we want to see where the market leaders are buying and selling. If these big hitters/savvy investors see an opportunity in the market, we'll to follow the same strategy. The other case is we want to track any potential for indications of insider trading. We'll want to look into the timing of large spikes in trading and correlate them with significant news releases.

Here are the steps we'll take 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. After we have grouped the records, we'll use the `KGroupedStream.count` method. This call returns the `KTable` that will maintain the state of the latest updates.
4. As part of the `count` method, we'll provide either a `Windows` or `SessionWindows` instance. The provided window determines if the record gets included into our count.
5. Write the results to a topic or print results to console during development.

The topology for this application is straightforward, but it's helpful to have a mental picture of the structure. Here's an image of the topology:




We have laid out our topology and have introduced the concept of windows. Our next step is to describe the various windowing functionality and the corresponding code.

### WINDOW TYPES

In Kafka Stream we have three types of windows available to us:

1. Session windows
2. Tumbling windows
3. Sliding/Hopping windows

Which one we choose depends on what the business requirements demand. The tumbling and hopping windows time bound. While session windows are more about user activity the length of the session(s) is driven by the time.

A key point to keep in mind for all windows with respect to time is we are referring to the timestamps in the record and not wallclock time. Even if you use the `WallclockTimestampExtractor`, the timestamp will be when Kafka Streams processes the record.

Next, we'll implement our topology from above with each of the window types. We'll only show the full code in the first example. Other than changing the type of window operation to use, everything else will remain the same. Next, we'll start with our first windowing example, session windows.

### SESSION WINDOWS

Session windows are very different from other windowing operations. Session windows aren't bound strictly by time as much as user activity (or the activity of anything you want to track). We delineate session windows by a given period of inactivity. The image below shows how we could view session windows:

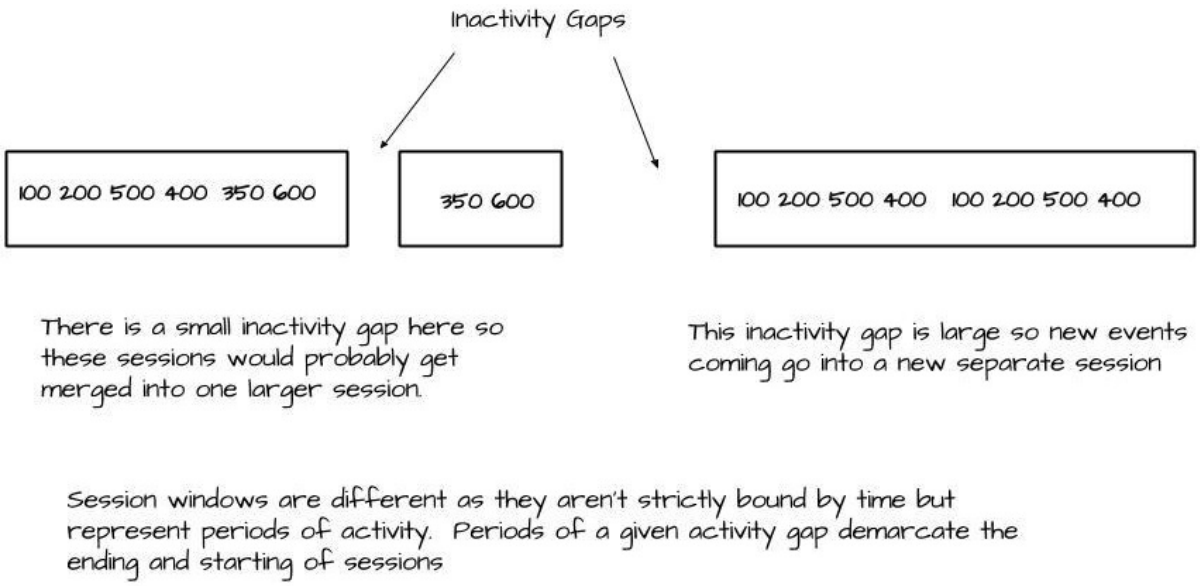



**Figure 5.11 Session windows example. We combine new sessions with a small inactivity gap to an adjacent session to form a new larger session.**

In the image above, you can see how the smaller session gets merged with the session on the left. But the session on the right will mark the start of a new session since the inactivity gap is large.

SessionWindows work on user activity, but use timestamps in the records to determine which "session" the record belongs.

### USING SESSION WINDOWS TO TRACK STOCK TRANSACTIONS

In this section, we'll show how we'll use `SessionWindows` to capture our stock transactions. The following code example is how we'll implement the session windows image above.

**Listing 5.5 Tracking Stock Transactions with Session Windows**

```
WindowedSerializer<TransactionSummary> windowedSerializer =
    new WindowedSerializer<>(transactionSummarySerde.serializer());
WindowedDeserializer<TransactionSummary> windowedDeserializer =
    new WindowedDeserializer<>(transactionSummarySerde.deserializer());
Serde<Windowed<TransactionSummary>> windowedSerde =
    Serdes.serdeFrom(windowedSerializer, windowedDeserializer);   (1)


long thirtySeconds = 1000 * 30;
long fifteenMinutes = 1000 * 60 * 15;

KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =   (2)
                kStreamBuilder.stream(LATEST, stringSerde, transactionSerde, STOCK_TOPIC)   (3)
               .groupBy((noKey, transaction) -> TransactionSummary.from(transaction), customerKeySer
```

```
4
            .count(SessionWindows.with(thirtySeconds).until(fifteenMinutes), "windowed-customer-t
5
    customerTransactionCounts.toStream().print(windowedSerde, Serdes.Long(),"Customer Transactior
6
```

1. Creating the WindowedSerde which wraps the serde for the key.
2. The KTable resulting from the groupBy and count calls.
3. Creating the stream from the STOCK_TRANSACTION (a String constant) Here we are using the offset reset strategy enum of LATEST for this stream.
4. Grouping records by customer id and stock symbol; contained in the TransactionSummary object.
5. Executing the count operation with a SessionWindow and the state store to use. The session window has an inactivity time of thirty seconds and a retention time of fifteen minutes.
6. Converting our KTable output to a KStream and printing result to StdOut.

We'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 we haven't seen before, and we'll discuss those now.

Any time we use an aggregation operation (aggregate, reduce or count), we have the option to provide a `Window` or `SessionWindow` instance. If we use a window instance the state store wraps the current key with a "windowed" key that includes a timestamp. The timestamp keeps track of data that may or may not belong in a particular window.

Because of this interaction, we need to provide a `WindowSerde` that handles the serialization of the window but uses a serde defined for the key present in the record. We can see this in listing one above, as we create a `Serde` from a `WindowedSerializer` and `WindowedDeserializer` instances that in turn wrap the respective `TransactionSummary` serializers/deserializers.

The code on listing number 5 is the crux of this example. The following image breaks it down for us:

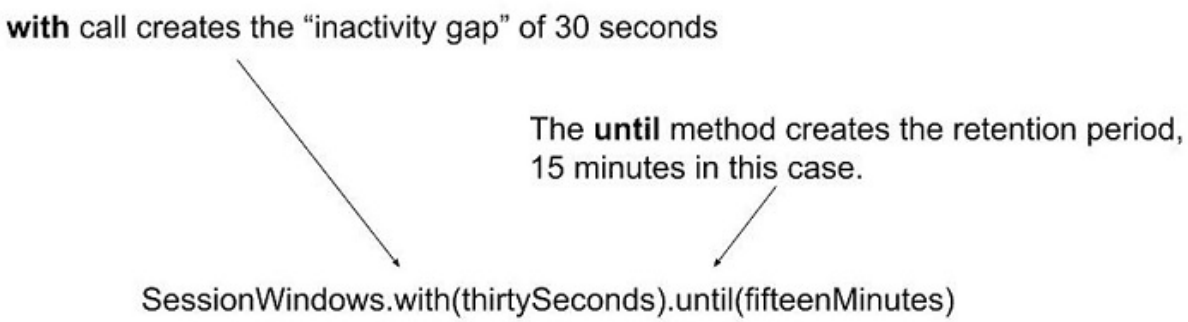



**Figure 5.12 Creating Session Windows with inactivity periods and retention**

With the call to

`SessionWindows.with(thirtySeconds).until(fifteenMinutes)` we are creating a `SessionWindow` with an inactivity gap of thirty seconds and retention period of fifteen minutes. An inactivity time of thirty seconds means the application includes any record arriving within thirty seconds of the current sessions ending or start time in the current (active) session.

If the 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. The retained session time is for late-arriving data that is outside the inactivity period of a session but can still be merged (we covered window stores and retention in Chapter 3, section 3.8). Additionally, as sessions are combined the newly created session has 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.

**Table 5.1 Sessioning Table with Thiry Second Activity 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} | 00:00:22 |

As records come in we look for existing sessions with the same key and ending times less than (current timestamp - inactivity gap) and starting times greater than (current timestamp + inactivity gap). With that in mind here's how the four records in the table end up getting merged into a single session:

1. Record 1 is of course first, so start and end are 00:00:00
2. Record 2 arrives we look for sessions with the earliest ending of 23:59:45 and the latest start of 00:00:45, we find record 1. We merge sessions 1 and 2. We keep session one start time (earliest) and session two end time (latest). So now we have one session starting 00:00:00 and ending 00:00:15
3. Record 3 arrives, and we look for sessions between 00:00:20 and 00:01:20 and find none. So we add a second session for key 123-345-654, FFBE starting and ending at 00:00:50.
4. Record 4 arrives the search is for sessions between 00:00:12 and 00:00:52. This time we find sessions 1 and 2. Now all three are merged into one session with a start time of 00:00:00 and an end of 00:00:50.

There are a couple of key points to remember from this section. First, sessions are not a fixed size window. Rather the size of a session is driven by the amount of activity within a given timeframe. Secondly, timestamps in the data determine how an event either fits into an existing session or falls into an inactivity gap.

Next, we'll move on to our 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 one minute. You would collect every event for 1 minute. After that 1 minute period is over, your window would "tumble" to a new 1 minute observation period. Here's an image to show this situation:

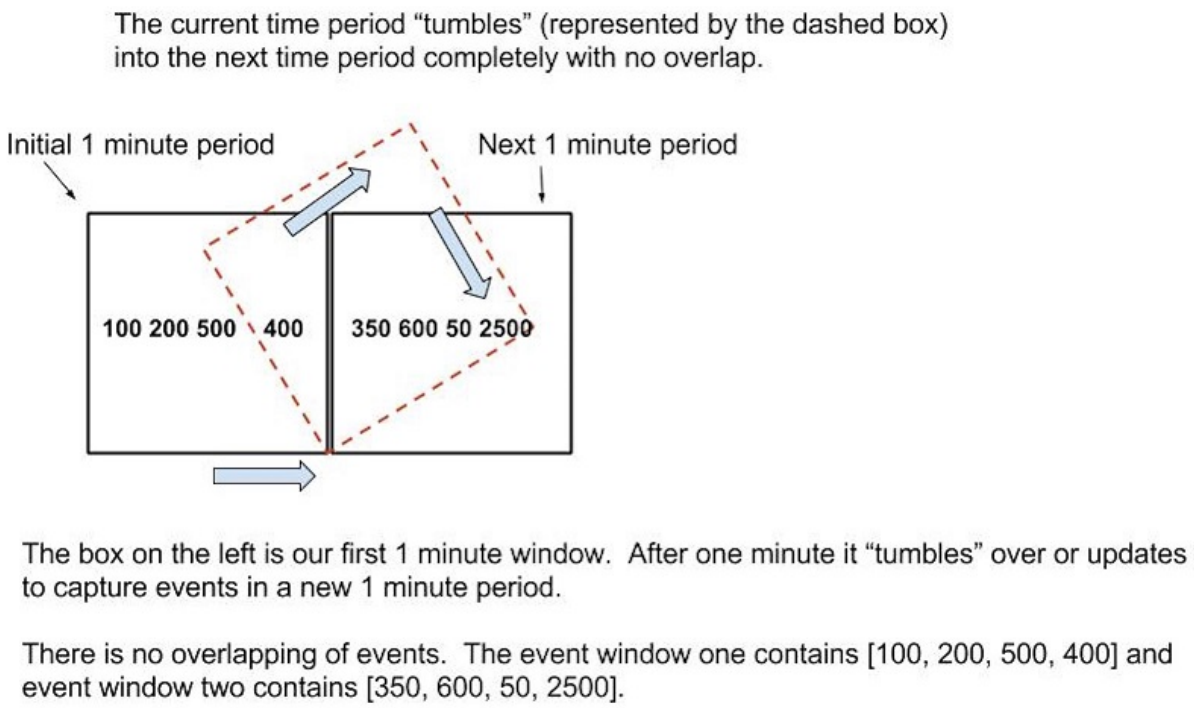



**Figure 5.13 Tumbling windows reset after a given fixed period**

As you can see, we include each event that has come in for the last minute. However, we wait the length of the specified time a new window.

Here's how we use tumbling windows to capture stock transactions every minute as defined in our scenario above:

**Listing 5.6 Using Tumbling Windows to Count User Transactions**

```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =
                kStreamBuilder.stream(LATEST, stringSerde, transactionSerde, STOCK_TOPIC)
               .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),
                   transactionSummarySerde, transactionSerde)
               .count(TimeWindows.of(twentySeconds),"windowed-customer-transaction-counts");
 (1)
```

(1) Specifying a tumbling window of twenty seconds.

So with a minor change, we can use a tumbling window with the `TimeWindows.of` call. In this example, we did not add the `until` method. By not specifying the duration of the window we'll get the default retention of twenty-four hours.

Next, we'll move on to the last of our windowing options, hopping windows.

### SLIDING OR HOPPING WINDOWS

Sliding or "Hopping" windows are like fixed windows but with a small difference. A sliding window won't doesn't wait the entire time before launching another window to process events that have occurred during the last interval. Sliding windows perform a new calculation after waiting for an interval smaller than the duration of the entire window.

To relate how hopping windows differ from tumbling windows, let's recast the stock transaction count example. We still want to count the number of transactions, but we don't wait the entire period before we update the count.

Instead, we update the count at *smaller* intervals. For example, we'll still count the number of transactions every minute, but we update the count every 30 seconds. Consider the following example in this graphic:

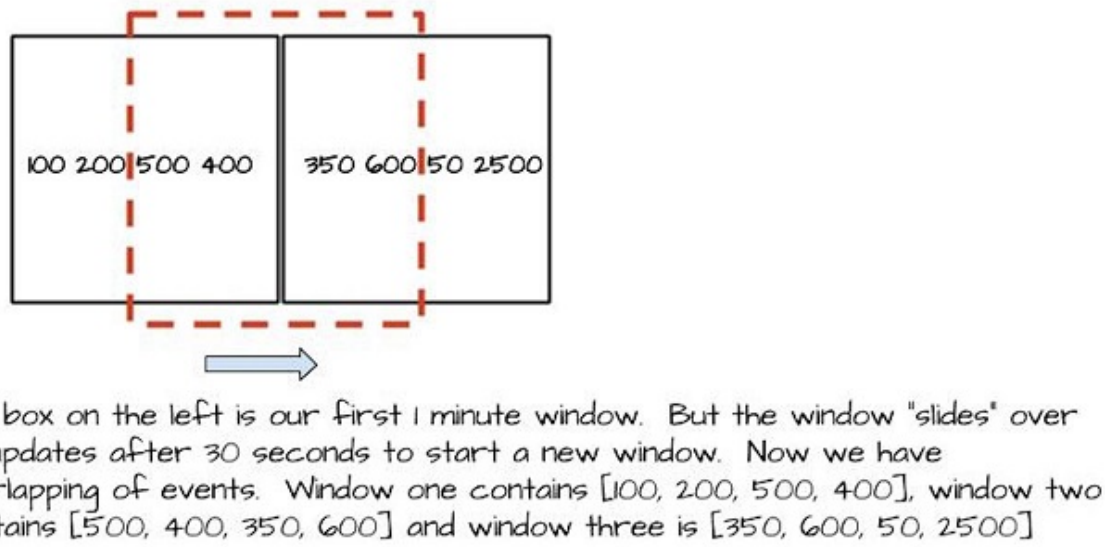



**Figure 5.14 Sliding windows update frequently and may contain overlapping data**

As you can see here, we still have one-minute windows. But we recalculate every thirty seconds. We now have three result windows with overlapping data.

And here's how to use hopping windows in the code example

**Listing 5.7 Specifying Hopping Windows to Count Transactions**

```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =
                 kStreamBuilder.stream(LATEST, stringSerde, transactionSerde, STOCK_TOPIC)
                .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),
                    transactionSummarySerde, transactionSerde)
                .count(TimeWindows.of(twentySeconds).advanceBy(fiveSeconds)
                .until(fifteenMinutes),"windowed-customer-transaction-counts");   (1)
```

(1) Using a hopping window of twenty seconds and advancing every five seconds

With the addition of the `advanceBy` method, we've converted our tumbling window

to a hopping window. This time we've specified a retention time of fifteen minutes.

To sum up, we've covered how to put our aggregation results in time windows. In particular, I want you to remember three things from this section:

1. Session windows aren't fixed by time but are driven by user activity.
2. Tumbling windows give us a *distinct* a set picture of events within the specified time frame.
3. Hopping windows are a window of size N, but is frequently updated and can contain overlapping records in each window.

Next, we'll demonstrate how we can convert a `KTable` back into a `KStream` to perform a join.

### 5.3.3 Joining KStreams and KTables

Remember from Chapter 4 we discussed joining two `KStreams`. Now we are going to show how we can join a `KTable` and a `KStream`. The reason why we'll do this is simple. KStreams are record streams, and the KTable is a stream of record updates. But there may be cases where we need to add some additional context to our record stream with the updates in a KTable.

What if we wanted to take our customer stock transaction counts and join them with financial news from relevant industry sectors? It's easy to see stock transactions as a stream of events. While economic news can be frequent, it's more static where today's news can be considered an update or replacement for old news.

To do this with our existing code, we'll need to take a few steps to make this happen:

1. Convert the KTable of stock transaction counts into KStream where we change the key to be the industry of the count by ticker symbol.
2. Create a KSTable that reads from a topic of financial news. Luckily the new is categorized by industry.
3. Join the news updates with our stock transaction counts by industry.

With these steps laid out for us, let's walk through how we can accomplish these tasks.

#### CONVERTING THE KTABLE TO KSTREAM

To do the KTable to KStream conversion, we'll take the following steps:

1. Call the `KTable.toStream()` method.
2. Then 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:

**Listing 5.8 Converting A KTable to a KStream**

```
KStream<String, TransactionSummary> countStream =
    customerTransactionCounts.toStream().map((window, count) -> {    (1)
    TransactionSummary transactionSummary = window.key();            (2)
```

```
    String newKey = transactionSummary.getIndustry();                    (3)
    transactionSummary.setSummaryCount(count);                           (4)
    return KeyValue.pair(newKey, transactionSummary);                    (5)
        });
```

(1) Calling toStream immediately followed by the map call.

(2) Extracting the TransactionSummary object from the Windowed instance.

(3) Setting the key to the industry segment of the stock purchase.

(4) Taking the count value from the aggregation and placing it in the TransactionSummary object.

(5) Returning the new KeyValue pair for the KStream.

We should note here that by performing a `KStream.map` operation, repartitioning for the returned KStream instance is automatically done when used in a join.

Now we have the conversion process completed our next step is to create the KTable to read the financial news.

### CREATING THE FINANCIAL NEWS KTABLE

Fortunately, to create the KTable is one line of code:

**Listing 5.9 KTable for Financial News**

```
KTable<String, String> financialNews = kStreamBuilder.table(EARLIEST,
    "financial-news", "financial-news-store");                           (1)
```

(1) Creating the KTable with EARLIEST enum, topic of "financial-news" and store name "financial-news-store".

It's worth noting here we didn't provide any serdes as the configuration is using string serdes. Also by using the enum `EARLIEST`, we're populating the table with records on startup.

Now we'll move on to our last step, setting up the join.

### JOINING NEWS UPDATES WITH TRANSACTION COUNTS

Setting up the join is very simple. In this case, we'll use a left join in case there is no financial news for the industry involved in the transaction.

**Listing 5.10 Setting up the Join Between the KStream and KTable**

```
ValueJoiner<TransactionSummary, String, String> valueJoiner = (txnct, news) ->
                String.format("%d shares purchased %s related news [%s]",
                    txnct.getSummaryCount(), txnct.getStockTicker(), news);   (1)

KStream<String,String> joined = countStream.leftJoin(financialNews,
    valueJoiner, stringSerde, transactionKeySerde);                      (2)
joined.print("Transactions and News");                                    (3)
```

1. ValueJoiner used for combining the values from the join result.
2. The left-join statement for the countStream KStream and the financial news KTable.
3. Printing results to console (in production we'd write this to a topic with a "to('topic-name') call)

The left-join statement straight forward. You'll notice that unlike the joins in Chapter 4, we did not provide a `JoinWindow`. That's because when performing a KStream-KTable join, there is only one record per key in the KTable. So performing the join is not a matter of time, the record is present in the KTable or not.

The key point here is we can use KTables to provide less frequently updated lookup data to enrich our KStream counterparts.

Next, we'll take a look at a more efficient way to enhance our KStream events.

### 5.3.4 GlobalKTables

At this point, we've established the need to enrich or add context to our event streams. We've seen joins between two KStreams in Chapter 4, and the previous example demonstrated a join between a KStream and a KTable. In all of these cases, if we mapped the keys to a new type or value, a repartitioning of the stream needs to happen. Sometimes we need to do the repartitioning explicitly ourselves, and other times Kafka Streams does it. The need for repartitioning makes sense. We have to make sure all keys are on the same partition, or the join doesn't happen.

#### REPARTITIONING HAS A COST

But the repartitioning is not free. There is additional overhead in this process with the creation of intermediate topics, storage of duplicate data in another topic, and the increased latency of writing to and reading from another topic. Additionally, if we want to join on more than one facet or dimension, we'll need to chain joins and map the records with a new key and repeat the repartitioning process.

#### JOINING WITH SMALLER DATASETS

In some cases, the lookup data you want to join against is relatively small and entire copies of the lookup data could fit locally on each node. In situations where the lookup data is reasonable small enough, Kafka Streams provides the `GlobalKTable`. GlobalKtables are unique because the application replicates all of the data to each node. Since the entirety of the data is on each node, we no longer have the requirement that our event stream is partitioned by the key of the lookup data, making it available to all partitions. Another important part is we can now do non-key joins. Let's revisit one of our previous examples to demonstrate this capability.

#### JOINING KSTREAMS WITH GLOBALKTABLES

Earlier in the chapter, we performed a windowed aggregation of stock transactions per customer. The output of the aggregation looked like this:

**Listing 5.11 Windowed Stock Transactions Aggregation Output**

```
{customerId='074-09-3705', stockTicker='GUTM'}, 17
{customerId='037-34-5184', stockTicker='CORK'}, 16
```

While this output accomplished our goal, it would be more impactful if we could see the client's name and the full company name. While we could perform regular joins to fill out the customer and company name, we would need to do two key mappings and repartitioning. But with GlobalKTables we can avoid the expense of those operations.

To accomplish this, we'll take our first countingStream from above and join it against two GlobalKtables. For review let's take our code for calculating the number of stock transactions using session windows:

**Listing 5.12 Aggregation Stock Transactions using Session Windows**

```
KStream<String, TransactionSummary> countStream =
    kStreamBuilder.stream(LATEST, stringSerde, transactionSerde, STOCK_TOPIC)
    .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),
        transactionSummarySerde, transactionSerde)
    .count(SessionWindows.with(twentySeconds), "session-windowed-customer-transaction-counts")
    .toStream().map(transactionMapper);
```

Since we've covered this before, we won't review it again here. But I would like to point out we've abstracted the code in the `toStream().map` function into function object vs. having an in-line lambda for readability purposes.

Our next step is to define two `GlobalKTable` instances:

**Listing 5.13 Defining the GlobalKTables for Lookup Data**

```
GlobalKTable<String, String> publicCcompanies =
    kStreamBuilder.globalTable("companies", "global-company-store");  (1)
GlobalKTable<String, String> clients = kStreamBuilder.globalTable("clients",
    "global-clients-store");  (2)
```

(1) The publicCompanies lookup is for finding companies by their stock ticker symbol

(2) The clients lookup us for getting customer names by customer id.

Now we have our components in place, we need to construct the join:

**Listing 5.14 Joining a KStream with Two GlobalKTables**

```
countStream.leftJoin(publicCompanies, (key, txn) -> txn.getStockTicker(),
    TransactionSummary::withCompmanyName)  (1)
      .leftJoin(clients, (key, txn)
        -> txn.getCustomerId(),TransactionSummary::withCustomerName)  (2)
      .print(stringSerde, transactionSummarySerde,"Resolved Transaction Summaries");  (3)
```

(1)

1. Setting up the leftJoin with the publicCompanies table, keying by stock ticker symbol and returning the transactionSummary with company name added.
2. Setting up the leftJoin with the clients table, keying by customer id and returning the transactionSummary with the customer named added.
3. Printing results out to the console.

Although there are two joins, we've chained them together as we don't use any of the results alone. We print the results at the end of the entire operation. Once we run the join operation we now get results like this:

**Listing 5.15 Resolved Windowed Stock Transaction Aggregates**

```
{customer='Barney, Smith' company="Exxon", transactions= 17}
```

While the facts haven't changed, these results are more clear to read. Including Chapter 4, we've seen several types of joins in action. At this point it will be useful to summarize the join types we have available in a table:

**Table 5.2 Kafka Streams Join Table**

| Left Join | Full 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 |

This table represents the state of join options as of Kafka Streams 0.10.2, but this may change in future releases.

In conclusion, the key thing to remember is we can combine our event streams (KStream) and update streams (KTable) leveraging local state. Additionally, for those times where the lookup data is of a manageable size, we 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 demonstrate a capability of Kafka Streams where we can observe changes to our state, without having to consume data from a Kafka topic.

### 5.3.5 Queryable State

At this point in the book, we've performed several operations involving state. In all cases, we've printed results to the console (development) or written them to a topic (production). The implications of writing the results to a topic are we need to use a Kafka Consumer to view the results. Reading the data from these topic(s) could be considered a form of "materialized views."

We'll define materialized views as "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" [13].

Footnote 13 en.wikipedia.org/wiki/Materialized_view

But Kafka Streams offers "interactive queries" from state stores, giving us the ability to read these materialized views directly. It's important to note querying state stores is a read-only operation. By making the queries read only, we don't have to worry about creating an inconsistent state while our application continues to process data.

The impact of making the state stores directly queriable is significant. This direct querying ability means we can create dashboard applications without having to consume the data from a Kafka Consumer first. There are also some gains in efficiency by not writing the data out again:

1. Since the data is local, you can access it very fast
2. We avoid duplication of data by not copying data to an external store [14].

Footnote 14 Adapted from www.confluent.io/blog/unifying-stream-processing-and-interactive-queries-in-apache-kafka/

The main thing I want to remember from this section is the ability to query state from the application directly. I can't underestimate 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 database table to store results).

We've covered a lot in this chapter, so we'll stop at this point in our discussion of interactive queries on state stores.

But fear not, in Chapter Nine we'll build a simple dashboard application with interactive queries. This dashboard application will use some our current examples to demonstrate interactive queries and how you can add it into your Kafka Streams applications.

## 5.4 Summary

Again, we have another chapter where we've covered a lot of ground. This chapter builds on the concepts of the previous one. We can consider this chapter as the capstone chapter of the KStreams API. Although we touched on several topics here are the key takeaways:

- KStreams represent event streams of comparable to inserts into a database. KTables are update streams and are more akin to updates into a database. The size of KTable doesn't continue to grow; it replaces older records with the newer records.
- KTables are essential for performing aggregation operations.
- We can place our aggregated data into "time buckets" with windowing operations.
- GlobalKTables give us lookup data across the entire application, regardless of the partition.
- We can perform joins with KStreams, KTables, and GlobalKtables.

So far we've focused on the "high-level" KStreams DSL to build our Kafka Streams applications. While the high-level approach gives us nice concise programs, everything is

a trade-off. By working with the KStreams DSL, we trade a level of control for more concise code. In the next chapter, we'll cover the "low-level" or Processor API. We'll make the different trade-offs using the Processor API; we won't have the conciseness of our applications we've built so far. But we'll gain the ability to perform or create virtually any kind of processor we need.

# 6 The Processor API

In this chapter:

- The Tradeoffs of Higher Abstractions vs. 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 CoGrouping Processor.
- Integrating the Processor API and the KStream API

Up to this point in the book, we have been working with the "High Level" or KStream API. The KStream API is a DSL allowing developers to create robust applications with minimal code. The ability to quickly put together processing topologies is an important feature of the KStreams DSL. The DSL allows you to iterate quickly to flesh out ideas for working on your data without getting bogged down in details like how to set up local state or wiring up processing nodes together. Also, you can get up and running without getting bogged down with intricate setup details some other frameworks may need.

But at some point, even when working with the best of tools, you always get to *those* situations that are just a little bit off. You have a problem to solve requiring you to deviate from the traditional path. Whatever the case may be, we need a way to dig down and write some code that just isn't possible with a higher level abstraction.

## 6.1 The Tradeoffs of Higher Abstractions vs. More Control

To me, the classic example of this trade-off is using ORM (Object Relational Mapping) frameworks. A good ORM framework maps your domain objects to database tables and will create 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 it seems inevitable that there are those few queries (very complex joins, select statements with nested sub-select statements) that just aren't working the way you want. To get the job done you need to write raw SQL to get the information from the database in the format you need. Many times you'll be able to mix in raw SQL with the mappings provided with the framework.

So we can see the tradeoff of a higher level abstraction vs. more programmatic control now. You can use a higher level abstraction that does much of the low-level work for you. In the case of the previous example the ORM framework, once configured, allows you work almost unaware of the fact you are materializing objects from a database.

To be clear, I'm not trying to draw a direct comparison between the KStreams API and using an ORM framework. I chose the example because I feel it represents the trade-off of higher abstraction vs. more programmatic control many developers experience.

This chapter is about those times you want to do stream processing in a way that the KStreams DSL doesn't seem to handle the way you want. For example, we've seen from working with the `KTable` API the framework controls the timing of forwarding records downstream.

But you may find yourself in a situation where you want explicit control when a record gets sent. For example, you are tracking trades on Wall Street, and you only want to forward records when a stock crosses a price threshold. To gain this type of control, I am talking about using the Processor API. What the Processor API lacks in ease of development, it makes up for in power. You can write your custom processors to do almost anything you want.

**NOTE** **The Processor API is Still Easy to Work With**
While the Processor API requires you to write more code vs. the KStreams API, it's still straightforward and intuitive. As a matter of fact, the KStreams API uses the Processor API under the covers.

In the following sections we'll learn how we can use the Processor API to handle situations like these:

1. Schedule actions to occur at regular intervals (as determined by timestamps in the records).
2. Full control over when records sent downstream.

3. Forwarding records to specific child nodes.
4. The ability to create functionality that doesn't exist in the KStream API. We'll see an example of this later when we build a CoGrouping processor.

First, let's understand 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 orders from distributors online, including international sales to Europe. You want to route orders within the company based on whether the order is domestic or international. Also, you'll need to convert any European sales from either British Pounds or Euros to U.S Dollars.

If you were to sketch out the flow of operation it would look something like this:

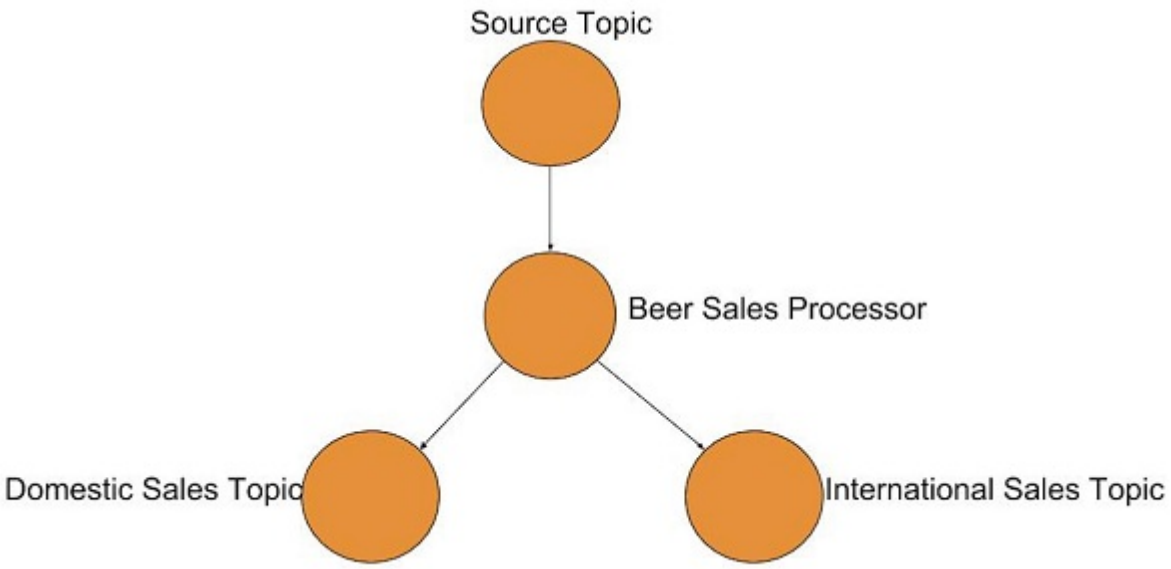



**Figure 6.1 Beer Sales Distribution Pipeline**

The best way to learn about the Processor API is to build a simple application so we can talk about each step.

Next, we'll do just that when we build topology to handle the requirement listed here. First, we'll start by creating a source node.

### 6.2.1 Adding a Source Node

The first step we take when constructing a topology is establishing the source node(s). Here's the line of code where we set the source of data for our new topology:

**Listing 6.1 Creating the Beer Application source node found in src/main/java/bbejeck.chapter_6/PopsHopsApplication.java**

```
builder.addSource(LATEST,                                    (1)
                  purchaseSourceNode,                        (2)
```

```
                stringDeserializer,                    3
                beerPurchaseDeserializer,              4
                "pops-hops-purchases");                5
```

1. Specifying the offset reset to use.
2. The name of this node.
3. A deserializer for the keys.
4. The deserializer for the values.
5. The name of the topic to consume data.

I've listed the parameters here vertically so we can discuss a few things we haven't seen until now. At listing number two, we are naming the source node. When using the KStreams API, we didn't need to pass in a name, because KStreams generates a name for the node. But when using the Processor API, we need to provide the names of the nodes in our topology.

The node name is used to wire up a child node to a parent node. And we've seen that's how data flows in a Kafka Streams application, from parent to child node. Listings three and four should seem familiar. With the Processor API instead of using a `Serde` for the keys and values, we provide a serializer/deserializer directly for them.

Next, let's look at how we'll work with purchase records coming into our application from the "pops-hops-purchases" topic.

### 6.2.2 Adding a Processor Node

Now we'll add a processor to work with the records coming in from the source node:

**Listing 6.2 Adding a processor node found in src/main/java/bbejeck.chapter_6/PopsHopsApplication.java**

```
BeerPurchaseProcessor beerProcessor = new BeerPurchaseProcessor(domesticSalesSink,
    internationalSalesSink);

builder.addSource(LATEST, purchaseSourceNode, stringDeserializer, beerPurchaseDeserializer,
    "pops-hops-purchases")
       .addProcessor(purchaseProcessor,        1
                     () -> beerProcessor,      2
                     purchaseSourceNode);      3
```

1. Naming the processor node.
2. Adding the processor defined in line one above, using a Java 8 lambda instead of a ProcessorSupplier. A ProcessorSupllier is a single method interface used to provide a Processor instance. Since it's a single method interface, we can substitute a Java 8 lambda.
3. Specifying the name of the parent node(s);

Right away we can see we still use the fluent pattern for constructing the topology. The difference from the KStream API lies in the return type. With the KStreams API, every call on a `KStream` operator results in returning a new KStream/KTable instance. In the Processor API, each call to the `TopologyBuiler` returns the same `TopologyBuilder` instance each time. The difference in return types makes sense if you think about it some. In the KStream API, you are *building* the topology as you go. On the other hand with the Processor API, you are *adding* nodes and building at the end.

In listing number two, we pass in the processor instantiated on the first line of the code example. The `TopologyBuiler.addProcessor` method takes an instance of a `ProcessorSupplier` interface for the second parameter. Since the `ProcessorSupplier` is a single method interface, we can substitute it with a lambda expression, which we've done here.

The key point in this section is the third parameter is the same value as the second parameter of the `addSource` method. This process establishes the parent-child relationship between nodes. The image below demonstrates this process.

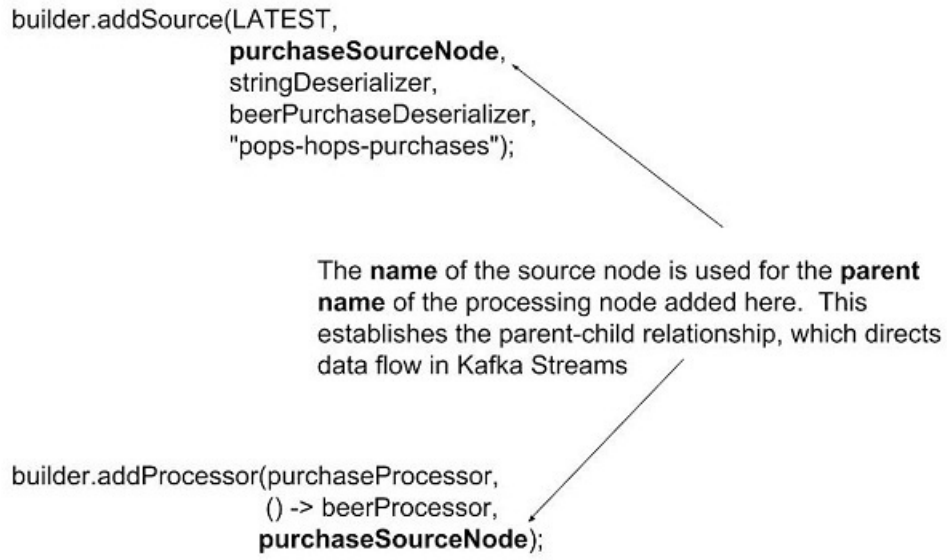



**Figure 6.2 Wiring up nodes in Processor API using a node name for a parent node name.**

By providing the name of the source node as the parent name, we've established the parent-child relationship between the source and processing node. The parent-child relationship is important because that determines the flow data in a Kafka Streams application.

We should take this opportunity to review what we've built so far:

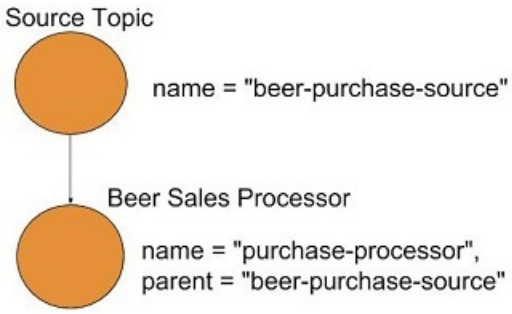



**Figure 6.3 Processor API topology built so far including node names and parent names.**

Now let's take a second to discuss the `BeerPurchaseProcessor` functions. The processor has two responsibilities: . Convert international sales amounts (in Euros) to US Dollars. . Based on the origin of sale, (domestic or international) route the record to the appropriate sink node.

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

1. Check the currency type. If it's not in dollars, then convert it to dollars.
2. If you have a non-domestic sale, after converting the sales amount, forward the updated record to the international sales topic.
3. Otherwise, we can forward the record directly to the domestic sales topic.

The code for this processor is below:

**Listing 6.3 BeerPurchaseProcessor**

```
public class BeerPurchaseProcessor extends 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.Builder builder = BeerPurchase.newBuilder(beerPurchase);
            double internationalSaleAmount = beerPurchase.getTotalSale();
            builder.totalSale(transactionCurrency.convertToDollars(internationalSaleAmount));
(3)
            beerPurchase = builder.build();
            context().forward(key, beerPurchase, internationalSalesNode);        (4)
        } else {
            context().forward(key, beerPurchase, domesticSalesNode);             (5)
        }

    }
}
```

1. Setting the names for different nodes to forward records.
2. The process method, where the action takes place.
3. Converting international sales to U.S Dollars.
4. Using the ProcessorContext, we forward records to a topic for international transactions.
5. Sending records for domestic sales to the right child node.

While there is a `Processor` interface, we've chosen to use the extend the `AbstractProcessor` class which gives us no-op overrides for all methods except for the `process()` method. The `Processor.process()` method is where we perform actions for the records flowing through the topology. Right away we see the flexibility offered by the Procesor API (listings three and four). The ability to select which child nodes to forward records.

The `context()` method shown here (listings four and five) retrieves the 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 StreamTask when initializing the topology. In the next example, we'll discuss `init()` and the other methods on the `Processor` interface in depth.

Now that we've covered how we process our records, the next step is to demonstrate how we connect a sink node (topic) for us to write records.

### 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 source we used `addSource` for adding a processor we have an `addProcessor` so as you can imagine we use the `addSink` method to wire up a sink node (topic) to a processor node. Our updated topology will look like this:

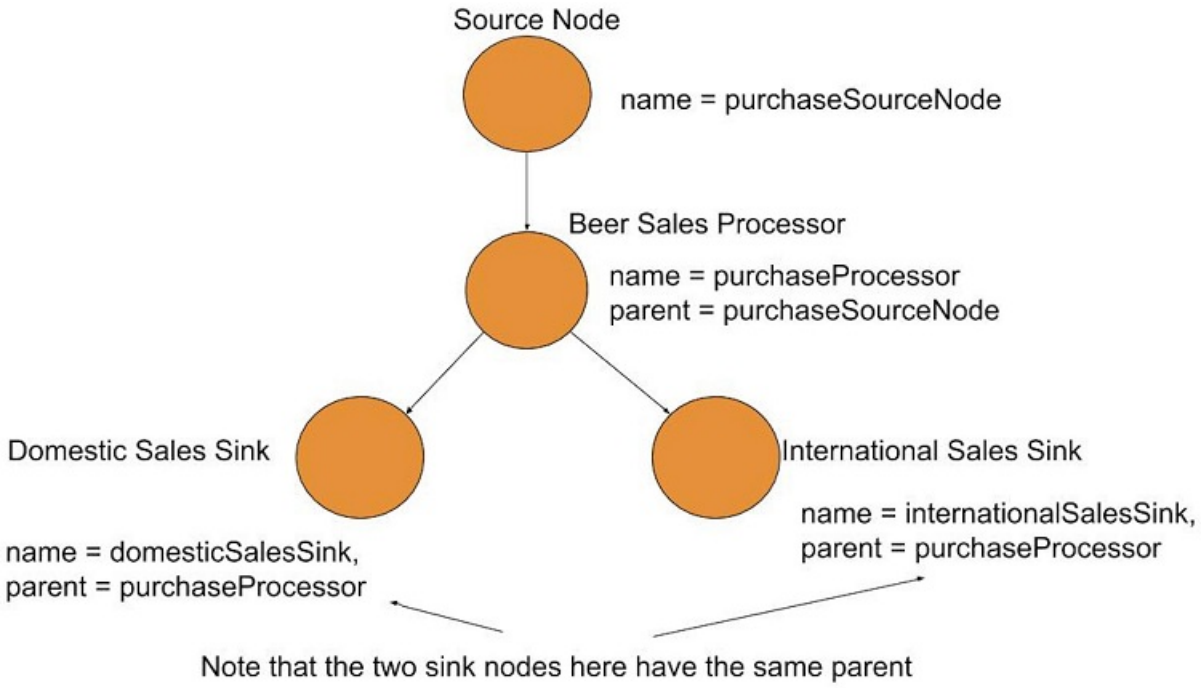



**Figure 6.4 Completing the topology by adding sink nodes.**

Let's update the topology we're building by adding a sink nodes now in the code now:

**Listing 6.4 Adding a sink node found in src/main/java/bbejeck.chapter_6/PopsHopsApplication.java**

```
builder.addSource(LATEST, purchaseSourceNode, stringDeserializer,
    beerPurchaseDeserializer, "pops-hops-purchases")
       .addProcessor(purchaseProcessor, () -> beerProcessor, purchaseSourceNode)

       .addSink(internationalSalesSink,                    (1)
                "international-sales",                     (2)
                stringSerializer,                          (3)
                beerPurchaseSerializer,                    (4)
                purchaseProcessor)                         (5)

       .addSink(domesticSalesSink,                         (1)
                "domestic-sales",                          (2)
                stringSerializer,                          (3)
                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.

Okay, I'm a liar as we added two sink nodes, one for Dollars and another for Euros. But if you recall from the source code of our processor, this makes sense. As depending on the currency of origin of the transaction we write our records out to two distinct topics. The key to adding two sink nodes is both have the same parent name.

By supplying the same parent name to both sink nodes, we have wired both of them to our processor (as shown in the image above). While the Processor API is more verbose than the KStream API, we can see it's still easy to construct topologies.

The key point I want you to remember from our first example is how we wired topologies together. In the Processor API, we explicitly wire up the nodes in the topology by using the names of other nodes for parent nodes. Since we are using names of previously defined nodes, the order in which we put together the topology doesn't matter.

Conversely, in the KStream API, parent nodes are specified by the order in which we call methods to build the topology. This explicit naming the parent nodes allows us to get creative with how we build our KStream applications.

Next, we'll take a deeper look into the Processor API by developing a processor where the requirements direct us to use the Processor API.

## 6.3 Digging Deeper into the Processor API with a Stock Analysis Processor

Now we are going to return to the world of finance. I want you to put on your day trading hat. As day traders, we want to analyze how stocks are changing in price in the hopes of picking the best time to buy and sell. Our goal is to take advantage of market fluctuations and make a quick profit. We have a few key indicators to consider in the hopes they will tip us off when we want to make a move. Our entire list of requirements are as follows:

1. Show the current value of the stock.
2. Indicate if the price per share is trending up or down.
3. Include the total share volume so far, and if the volume is trending up or down.
4. Restrict sending records downstream to those displaying %2 trending (up or down).

Now let's take a walk through how we might handle this analysis manually trying to determine when we see a deal to make. In the figure below (Figure 6.2) we show the decision tree of sorts that we'll want to create to help us make decisions.


**Figure 6.5 Stock Trends Updates**

We a handful calculations we need to perform to complete our analysis. Additionally, we'll use our calculation results to determine if and when we should forward records downstream. This restriction of sending records means we can't rely on the standard mechanisms of commits or cache flushes to handle the flow for us, which rules out using the KStream API. It goes without saying that we'll also require state so we can keep track of changes over time.

While we may be able to cobble together something already existing in the KStream API, the requirement calls for control over sending records downstream (as outlined above). What we need here is the ability to write a custom processor.

**IMPORTANT** **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 from this example. This model bears no similarity to a real-life approach and is presented only as a way of demonstrating more complex processing situations. Remember you're not a day trader, and neither am I!

In the next section, we are going to present the solution to our problem, creating a new class implementing the `Processor` interface to satisfy all of our requirements.

### 6.3.1 Building A Custom Processor

In this section, we're going to build our custom `Processor`, and along the way, we'll describe the finer details to consider when creating custom processors. We've mentioned the `Processor` interface several times, so now is an excellent opportunity to review it.

**Listing 6.5 The Processor Interface**

```
public interface Processor<K, V> {

    void init(ProcessorContext context);    (1)

    void process(K key, V value);           (2)

    void punctuate(long timestamp);         (3)

    void close();                           (4)
}
```

(1) The init method used for setting up state stores and scheduling punctuate calls.

(2) The process method, where we perform any actions on the key-value pair.

(3) Punctuate method. Called to perform operations at an interval specified in init.

(4) The close() method is used to clean up any resources on shutdown.

With four methods, the `Processor` interface is straight forward. Over the next four sections, we'll build our processor one step at a time. At each stage, we'll discuss the method's role in the processor and how we've decided to implement it.

#### THE INIT METHOD

The `Processor.init()` method where we do all of our initialization tasks and where the framework passes in the `ProcessorContext` reference. The `ProcessorContext` exposes critical information and functionality necessary to work with records in the processor. Instead of listing everything here, we'll discover what we use the `ProcessorContext` for as we go along in the rest of the chapter. Here's what we are going to do in the `init()` method:

1. Initialize our instance variable of the `ProcessorContext` with reference passed as a parameter.
2. Grab a reference to state store and set it to a local variable.
3. Schedule to potentially flush records every ten seconds.

**Listing 6.6 Init Method tasks**

```
@Override
public void init(ProcessorContext processorContext) {
  this.processorContext = processorContext;                                    (1)
  this.keyValueStore = (KeyValueStore) this.processorContext.getStateStore(stateStoreName);   (2)
```

```
  this.processorContext.schedule(10000);    ❸
}
```

❶ Intializing ProcessorContext instance variable.

❷ Retrieving state store created when building topology.

❸ Scheduling to have the Processor.punctate called every ten seconds.

As we can see, what we do in the `init()` method is straight forward, as we'd expect in any initialization setup. Having said that, we'll have some interesting scenarios to discuss when we cover the `Processor.punctuate()` method. Next, let's take a look at the `Processor.process()` method.

### THE PROCESS METHOD

In the `process()` method is where we perform all of our calculations to evaluate stock performance. We have several steps to take when we receive a record.

1. First, we check the state store to see if we have a corresponding `StockPerformance` object for the given stock ticker symbol.
2. If the store does not contain the object, we'll record the current share price and volume.
3. Otherwise, we'll add the current share price and share volume to the object and update our calculations.
4. We only start performing calculations once we hit twenty transactions for any given stock.

While financial analysis is beyond the scope of this book, we should take a minute to explain our calculations. For both the share price and volume we are going to perform a simple moving average. In the financial trading world, a simple moving average or SMA is used to calculate an average data points of size N.

For our purposes here we are going to set our size N to twenty. Setting a max size means as new trades come in we'll collect the share price and amount of shares traded up to twenty. Once we hit that threshold, we'll remove the oldest value and add the latest one. Using the SMA, we'll get a "rolling average" of stock price and volume over the last 20 trades. It's important to note we won't have to recalculate the entire amount as new values come in.

Now let's take a look at the following image for a high-level walk through or our process method, with the mindset of what we would do if we were to perform these steps manually:

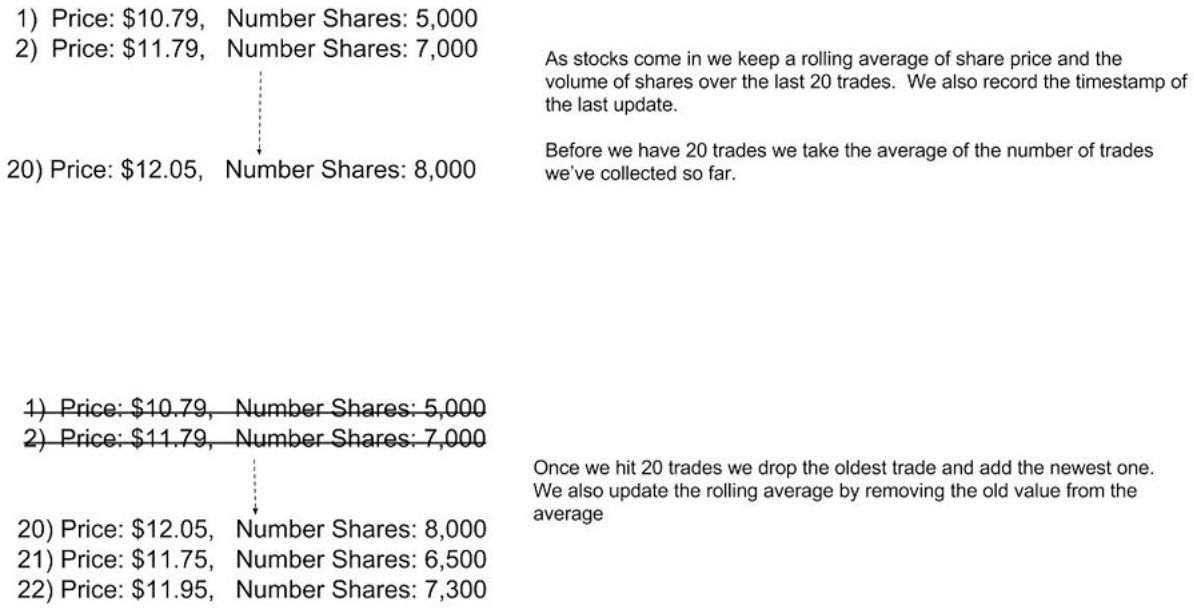



**Figure 6.6 Stock Analysis Process**

As we can see from our walk through the process method is where we perform all of the calculations. Now let's take a look at the code that makes up our process method:

**Listing 6.7 Process Implementation found in bbejeck.chapter_6.processor.StockPerformanceProcessor**

```
@Override
public void process(String symbol, StockTransaction transaction) {
 StockPerformance stockPerformance = keyValueStore.get(symbol);   (1)

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

 stockPerformance.updatePriceStats(transaction.getSharePrice());   (3)
 stockPerformance.updateVolumeStats(transaction.getShares());   (4)
 stockPerformance.setLastUpdateSent(Instant.now());   (5)

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

(1) Retrieving previous performance stats, possibly null.

(2) Creating a new StockPerformance object when not in the state store.

(3) Updating the price statistics for this stock.

(4) Updating the volume statistics for this stock.

(5) Setting the timestamp of the last update.

(6) Placing the updated StockPerformance object into the state store.

From viewing the code, we can see the process method is not very complicated. We take the latest share price and the number of shares involved in the transaction and add

them to the `StockPerformance` object. Notice that we've abstracted all details of how we perform the updates inside the `StockPerformance` object. Keeping most of the business logic out of the processor is a good idea, and we'll come back to that point when we cover testing in Chapter seven.

While we won't show the entirety of the code for the `StockPerformance` object, let's talk about the two key calculations we do in the class, determining the moving average and the differential of stock price/volume from the current respective average.

First, let's discuss how we've implemented our simple moving average. We don't want to calculate an average until we've collected data from twenty transactions. So we defer doing anything until the processor receives twenty trades. But as soon as we get data from twenty trades, we calculate our first average. The first average is the only time we need to sum all values. After the initial calculation, we use the update approach outlined before.

**Listing 6.8 Implementation of Simple Moving Average found in bbejeck.model.StockPerformance**

```
private double calculateNewAverage(double newValue, double currentAverage, ArrayDeque<Double> deque)
  if (deque.size() < MAX_LOOK_BACK) {                                    (1)
      deque.add(newValue);

    if (deque.size() == MAX_LOOK_BACK) {

        currentAverage = deque.stream().reduce(0.0, Double::sum) / MAX_LOOK_BACK;   (2)
    }

  } else {

     double oldestValue = deque.poll();                                  (3)

     deque.add(newValue);                                                (4)
     currentAverage = (currentAverage + (newValue / MAX_LOOK_BACK))

       - oldestValue / MAX_LOOK_BACK;                                    (5)
  }
 return currentAverage;
}
```

1. Checking if we have accumulated 20 data points if so we add it to our ArrayDeque.
2. If the added value hits 20 calculate a new average. The first time reaching 20 transactions is the only time we need to sum all values.
3. We have 20 values, so remove the oldest from the head.
4. Add the new value to the end.
5. This is our SMA implementation, which does not require us to re-sum all values again to obtain a new average.

Next, let us cover how we determine the percentage difference between the current price or number of shares from their respective average. We're using this calculation to try and identify a situation where we should either sell or buy a particular stock. An

actual percentage increase over the average could indicate a good time to sell, and a percentage decrease could mean a purchase is possible. Here is the simple calculation so we can jump right to the code:

**Listing 6.9 Calcuate percentage difference between current price and average over twenty trades**

```
private double calculateDifferentialFromAverage(double value, double average) {
  return average != Double.Min_Value ? ((value / average) - 1) * 100.0 : 0.0;  (1)
}
```

(1) If we've defined the average return the percentage difference from the current value, otherwise zero

So now we've seen how the process method works and the calculations we perform. But if you've been paying attention we make no mention of forwarding the records. All of the processing is not worth much to us unless we send records downstream to other nodes, precisely the sink nodes where the information is written out.

So if the `process` method does not handle sending records downstream, where do we take care of sending records onwards in the topology? It's in the `punctuate` method where we make our determination of sending records downstream.

So the next step for us is to discuss the `Processor.punctuate()` method.

### 6.3.2 The Punctuate Method

Once the `punctuate` method is scheduled, the `StreamTask` will execute the method at the specified intervals. We set the schedule when we use the `ProcessorContext.schedule` call in the `init` method as we described before. The `punctuate` method gives us the ability to perform any task at regular intervals. There are some caveats to using the `punctate` method, but we'll discuss some of the 'fine print' with the `punctate` method later in this section).

For our purposes, we are going to check if we have any records meeting our criteria during `punctuate` execution.

To perform the test, we'll iterate over all the records in the state store. For each record, we'll check if the absolute value of either the price or volume differential from the average meets or exceeds our established threshold. If the record does, we'll forward it downstream.

Here's the code for our puncutiation call:

**Listing 6.10 Iterating over stock records in punctate call found in bbejeck.model.StockPerformance**

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

```
 while (performanceIterator.hasNext()) {
   KeyValue<String, StockPerformance> keyValue = performanceIterator.next();
   StockPerformance stockPerformance = keyValue.value;
   String key = keyValue.key;
   if (stockPerformance.priceDifferential() >= differentialThreshold ||
                    stockPerformance.volumeDifferential() >= differentialThreshold) {
       this.processorContext.forward(key, stockPerformance);
   }
 }
}
```

❶ Retrieving the iterator to go over all key values in the state store.

❷ Checking the threshold for the current stock.

❸ If we've met or exceeded the threshold, forward the record.

**NOTE** **Kafka Streams does not share data**

While we're demonstrating access to a state store, it's a good time to reiterate what we said about Kafka Streams architecture. We need to remember a few key things. First, each task created has its *own* copy of a *local* state, and there is one task per thread, so there is no sharing of data across threads or multi-threaded access. Secondly, as records make their way through the topology, each node is visited depth-first meaning there is never concurrent access to state stores from any given processor.

The `ProcessorContext` forwards the record to all child nodes of our `StockPerformanceProcessor`, in this case, it's a single sink node. What if we wanted to forward records to only select child node selectively? Well, the `ProcessorContext` also offers two overloaded versions of the `forward` method.

The overloaded methods look like this:

**Listing 6.11 Overloaded ProcessorContext.forward methods**

```
ProcessorContext.forward(K key, V value, int childIndex)
ProcessorContext.forward(K key, V value, String childName)
```

❶ Forwarding records to a particular child node by index.

❷ Forwarding records to a particular child node by name.

The first overloaded method allows us to target a particular child node by index. The index would be the position in which you added the child node when building the topology (Note: the `KStreamBranch` method uses this approach under the covers). The second overloaded method allows us to specify the name of the child node to forward. In

either case, it's a powerful tool to have at your disposal. By specifying specific child processing nodes, we can use conditional logic to cherry-pick which nodes we forward records.

A littler earlier in this section, I mentioned some "fine print" regarding the scheduling of the `punctuate` method. A key point to consider with scheduling is that the timing used to determine to call punctuate based on timestamps of the data and not wallclock time. Let's take a look at an image describing this process:

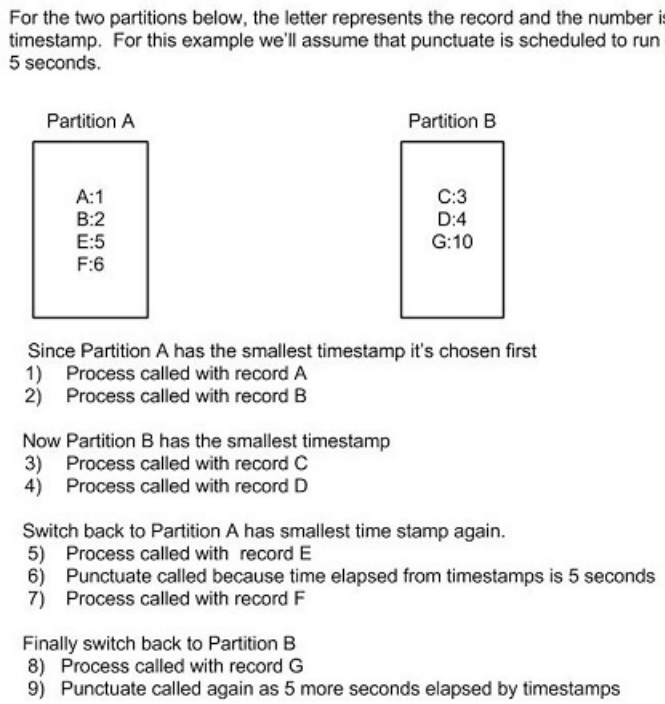



**Figure 6.7 Punctuation Scheduling**

Using the image above to guide us let's walk through how the schedule is determined:

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

The key point here is the execution of the punctuate function is entirely dependent on the rate of data coming into the topology. Even if you are using the `WallclockTimestampExtractor`, timestamps only trigger punctuate calls from data arrival.

The key takeaway from this discussion on`Processor.punctuate` is in the current implementation of Kafka Streams the `Processor.punctuate()` method is driven by time elapsing via timestamps in the data, and not wallclock time. This is important to

remember as you can't be guaranteed your data arrives in order. So if you design your logic to expect a uniform punctuate call every N number of seconds or milliseconds, you may have unexpected results.

**NOTE** **Changes to punctuate semantics**
There is talk in the Kafka Streams community now about changing the semantics of the punctuate method. In a nutshell, there are proposed changes in KIP-138 that would give developers the opportunity to specify whether punctuate execution should be data driven or time driven. For more details, readers are encouraged to read cwiki.apache.org/confluence/display/KAFKA/KIP-138:+Change+punctuate+ for more information.

Next, we'll cover how we can perform any necessary cleanup of resources before shutting down. To do any clean-up, we'll use the last method of the `Processor` interface, the `close` method.

### 6.3.3 The Close Method

While we didn' have any use to use the `Processor.close` method in our example, we'll still discuss it here. When shutting down a Kafka Streams application, each `StreamTask` will as part of its shutdown process, iterate over all of its processors and call the `Processor.close` on each one. The close method on the processor is your chance to do any final tasks and clean up state that is specific to that processor. For example closing a file or any connections to a database etc.

As a general rule, you don't want to attempt to clean up any of the state stores you may have used, as the application handles the clean up of state stores. Additionally, calling the close method is part of the internal clean up during the shutdown process, so you can't do any final writes to Kafka as the application has already closed the underlying clients.

Now that we've covered how to write a custom processor let's take a look at how we integrate the processor into an application.

### 6.3.4 Putting it all together, the completed Stock Analysis Application

Here's how our final application looks. (Note that we've left out serde and configuration code for clarity):

**Listing 6.12 Final stock performance application with custom processor**

```
TopologyBuilder builder = new TopologyBuilder();
String stocksStateStore = "stock-performance-store";

double differentialThreshold = 0.02;  (1)

StockPerformanceProcessor stockPerformanceProcessor =

   new StockPerformanceProcessor(stocksStateStore, differentialThreshold);  (2)
```

```
builder.addSource(LATEST, "stocks-source", stringDeserializer, stockTransactionDeserializer,
   "stock-transactions")

      .addProcessor("stocks-processor", () -> stockPerformanceProcessor, "stocks-source")  ❸
      .addStateStore(Stores.create("stock-transactions").withStringKeys()
                     .withValues(stockPerformanceSerde).inMemory().maxEntries(100).build(),
                     "stocks-processor")  ❹
      .addSink("stocks-sink", "stock-performance", stringSerializer,
          stockPerformanceSerializer, "stocks-processor");
```

❶ Setting the percentage differential for forwarding stock information.

❷ Instantiation our custom processor.

❸ Adding the processor to the topology.

❹ Creating the state store.

Even though this is the low-level API, our code here is still concise. All of the complexity is in our `StockPerformanceProcessor` (we will see the benefits of having all the business logic contained in the processor when we get to testing in Chapter 8). While this example worked to give us an excellent introduction to writing a custom processor, we can take writing custom processors a bit farther. With this in mind, we'll move on to adding a Cogroup processor.

## 6.4 The CoGroup Processor

Back in Chapter four, we discussed joins between two streams; specifically, we joined purchases from different departments within a given timeframe to promote business.

We use joins to bring together records with the same key and arriving in the same time window. With joins there's an implied one-to-one mapping of records from stream A to stream B. The following image depicts this relationship clearly:

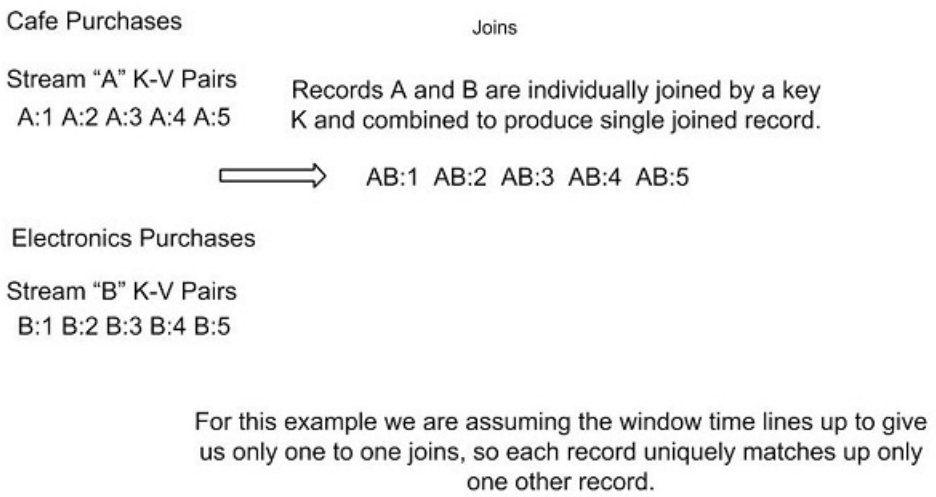



**Figure 6.8 Records A and B joined by a common key**

Now let's imagine you want to do a similar type of analysis, but instead of a one-to-one join by key, you want two collections of data by a common key, or a *co-grouping* of data.

You are the manager of a popular online day-trading application. Day traders are on 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 have defined an event as when a user clicks on the ticker symbol to read more about a company and its financial outlook. You want to do some deeper analysis between user clicks on the application and when users purchase stocks. But you want more course grained results, by comparing multiple clicks and purchases to determine some overall pattern. What you want is a tuple with two collections of each event type by the company trading symbol as pictured here:

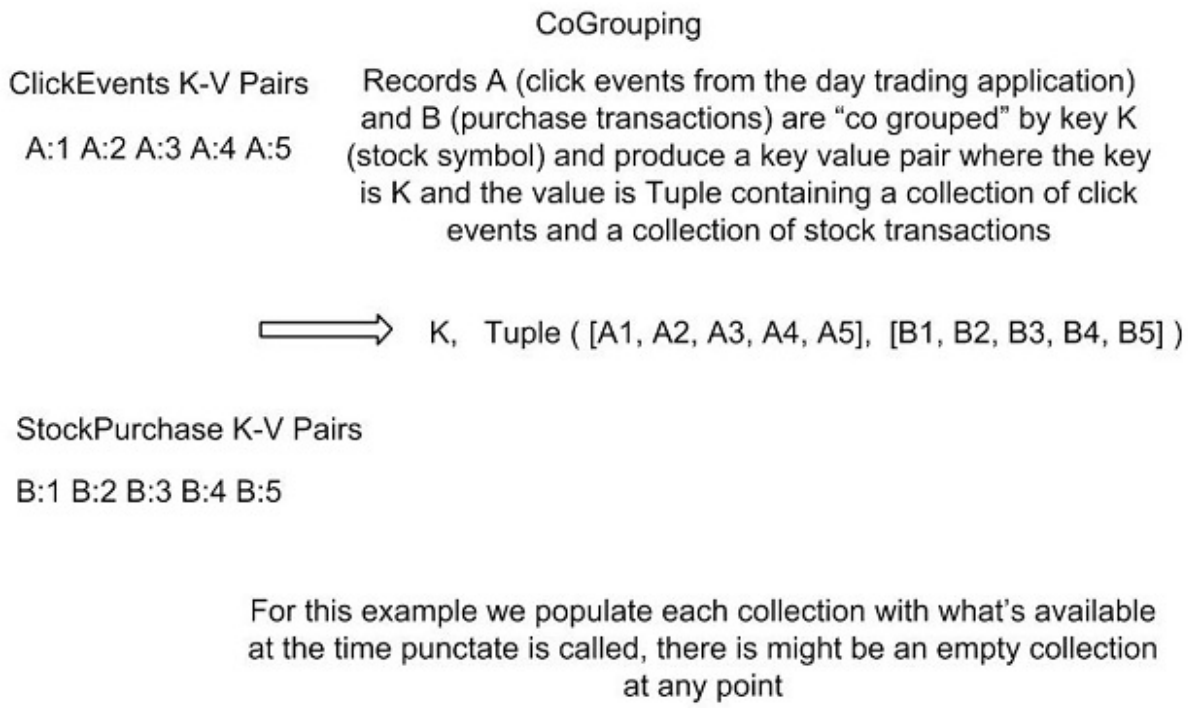



**Figure 6.9 Output of a Key with a Tuple containing two collections of data, a co-grouped result.**

Your desire is to combine "snapshots" of click events and transactions for a given company every N seconds. But you are not 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 transactions by company ticker symbol. If either type of event is not present when the timer elapses, then one of the collections in the tuple is empty. For those readers familiar with Apache Spark or Apache Flink this functionality is similar to those found in the CoGroupDataSet respectively.

Next, let’s start walking through the steps we’ll take in constructing this processor.

### 6.4.1 Building the CoGroup Processor

While creating the co-grouping processor will be straight forward, we have a few pieces we need to tie together:

1. Define sources for two topics (stock-transactions, click-events)
2. Add two processors to consume records from a topic

3. We'll also need two state stores, one for each processor.
4. A sink node to write our results (or a printing processor to print results to console).

Now we'll now walk through the steps to put this processor together.

### DEFINING THE SOURCE NODES

By this point we are familiar with the first step we take when building a topology, creating the source nodes. This time we'll need to create two source nodes to support reading both the click-event stream and the stock transaction stream. To help us keep track of where we are in the topology, we'll build on the following diagram for a graphical display of the topology:

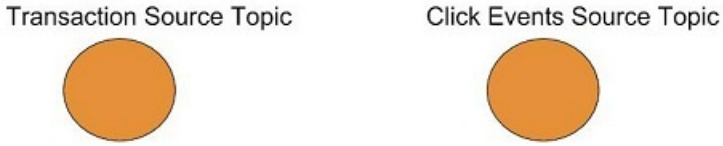



**Figure 6.10 Cogrouping Source Nodes**

The accompanying code for creating our source nodes looks like this:

**Listing 6.13 Source nodes for CoGrouping Processor found in**

```
//Note that we've left out configuration and Serde creation for clarity.

builder.addSource(LATEST, "txn-source", stringDeserializer, stockTransactionDeserializer,
    "stock-transactions")                                                          (1)
       .addSource(LATEST, "events-source", stringDeserializer, clickEventDeserializer,
        "click-events")                                                            (2)
```

(1) Source node for the stock transactions topic.

(2) Source node for the click-events topic.

A subtle point to notice here is we only provide deserializers in the Processor API vs. a `Serde` in the KStream DSL. Next, we'll move on to adding the processor nodes.

**NOTE** **Serde vs. Deserializers**
Another difference between the KStream API and the Processor API is the serialization/deserialization of objects. In the KStream API, we use a `Serde` which contains a serializer and deserializer for the object in question. In the Processor API, we directly pass in either a serializer or deserializer as required.

### ADDING THE PROCESSOR NODES

Now we'll add the workhorses of the toplogy, the processors. The toplogy graph is updated accordingly:

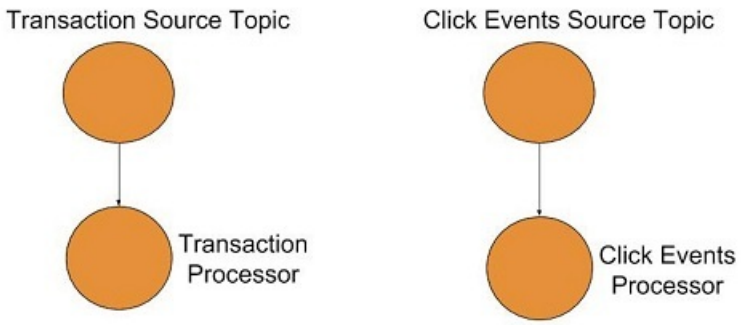



**Figure 6.11 Adding Processor Nodes**

At this point, we've added the processors. The only detail here to emphasize is that we'll need to make sure we get the names of the parent nodes correct, as using names is how we wire the topology together. (We've seen this before with the earlier examples in this chapter, but it's worth emphasizing again as this is a detail you don't have to take care of in the KStream API).

Here's the code for adding these new processors:

**Listing 6.14 The Processor Nodes**

```
.addProcessor("txn-processor", () -> transactionProcessor, "txn-source")  (1)
.addProcessor("evnts-processor", () -> eventProcessor, "events-source")   (2)
```

(1) Adding the StockTransactionProcessor.

(2) Adding the DayTradingClickEventProcessor.

While these are simple two lines of code, there are two things to note here. First, the names of the parent nodes match the names of the source nodes in the listing above. Secondly, we've again used a lambda expression instead of a `ProcessorSupplier` instance. At this point we should discuss how each processor functions, but let's hold off on that discussion until after we take the next step, adding the state stores.

### ADDING THE STATE STORES

What makes this application unique is for the first time we are going to demonstrate giving a processor access to more than one state store. The ability of a processor to access more than one state store is what makes this particular application tick. Let's look at a visual depiction of how using state stores work in this application:

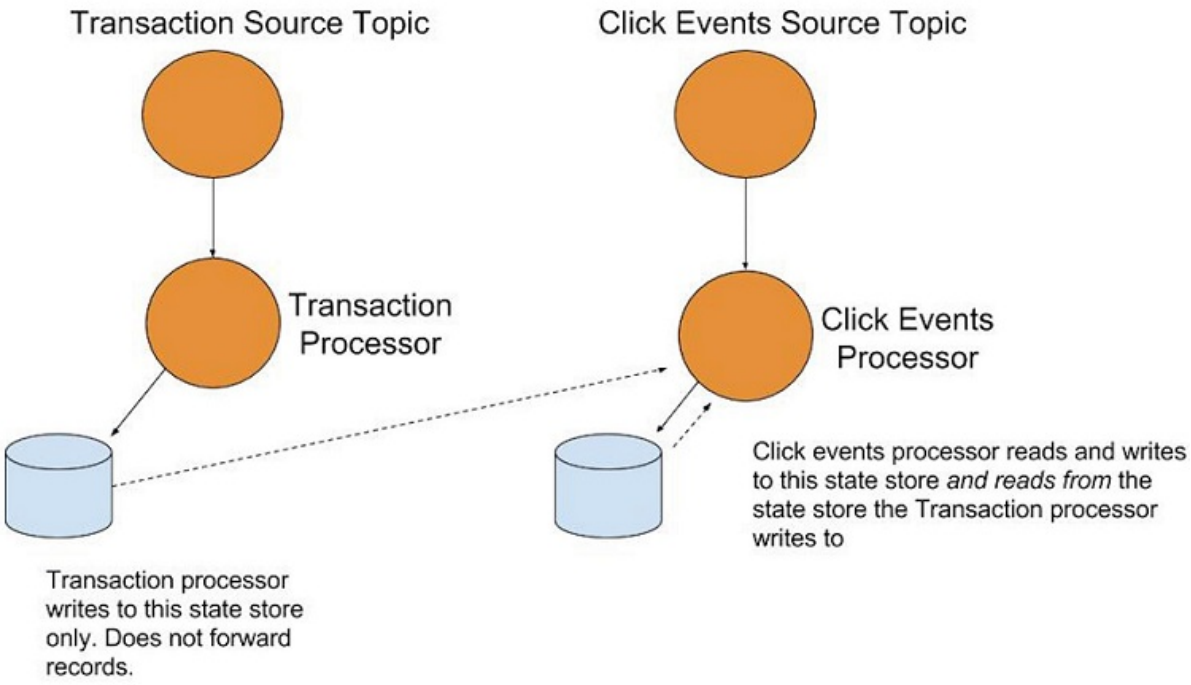



**Figure 6.12 Attaching state stores to processors in the topology**

**TIP** **Non Concurrent Access**

It's important to remember that we traverse processors depth-first and there is no multi-threaded access per stream task. So there are no concurrency concerns with sharing a state store between two processors.

From the image above, we can see the `StockTransactionCogroupingProcessor` writes (and never reads) stock transaction records to the attached state store. To perform the co-grouping, we need one of the processors to have access to click events and stock transaction records from both source nodes, in this case, we chose the `ClickEventCogroupingProcessor`. While we can define a source node to consume from more than one topic, processors in KafkaStreams expect records of a particular type.

We are going to use the state store as a proxy for the other processor. By taking this approach the `ClickEventCogroupingProcessor` processes the expected record types, but still, has access to records output by the `StockTransactionCogroupingProcessor`. Now let's take a look at the code for adding the state stores:

**Listing 6.15 Adding State Store nodes**

```
.addStateStore(Stores.create(stocksStateStore).withStringKeys()
   .withValues(txnListSerde).inMemory().maxEntries(100).build(),  "txn-processor",
    "evnts-processor")  (1)
.addStateStore(Stores.create(dayTradingEventClicksStore).withStringKeys()
   .withValues(eventListSerde).inMemory().maxEntries(100).build(),
    "evnts-processor")  (2)
```

1. Adding a state store to both processors, notice two names at the end. The stock transaction processor only writes to this store while the events processor will only read from this store.
2. Adding a state store to the events processor. The events processor will read and write to this store.

From the code listing here we can see we are using in-memory key-value stores. We opted for in-memory stores in this case because the data we are analyzing is transient and we won't be using it for any deeper analysis.

Now that we have the state store use laid out our next step is to demonstrate how the processors work.

### PROCESSOR FUNCTIONALITY

First, we'll cover the `StockTransactionCogroupingProcessor` as it has the simplest functionality. The role of this first processor is to take `StockTransaction` records and place them in the state store by its trading symbol. But we aren't storing a single transaction; we are storing a `List<StockTransaction>` so we can hold more than one instance.

**Listing 6.16 The StockTransactionCogroupProcessor found in src/main/java/bbejeck/chapter_6/processor**

```
public class StockTransactionCogroupingProcessor
    extends AbstractProcessor<String, StockTransaction> {

    private String storeName;
    private KeyValueStore<String, List<StockTransaction>> keyListStore;

    public StockTransactionCogroupingProcessor(String storeName) {
        this.storeName = storeName;
    }

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        super.init(context);

        keyListStore = (KeyValueStore) context().getStateStore(storeName);   (1)
    }

    @Override
    public void process(String key, StockTransaction value) {

        if (key != null) {                                                    (2)
            List<StockTransaction> transactions = keyListStore.get(key);
            if (transactions == null) {

                transactions = new ArrayList<>();                             (3)
            }

            transactions.add(value);

            keyListStore.put(key, transactions);                              (4)
        }
    }

}
```

1. Retrieving the state store.
2.

2 Records with null keys are ignored.

3 Initializing the list if null.

4 Adding the transaction to the list and placing the list in the store.

While the first processor is simple in its functionality, remember we are using the state store as a proxy to forward records to the next processor. Next, we'll look at the code for the `ClickEventCogroupingProcessor`. I should mention here that I'll be leaving out some repetitive details here for clarity, the following example is not meant to stand alone as is.

**Listing 6.17 The ClickEventCogroupingProcessor code found in src/main/java/bbejeck/chapter_6/processor**

```
    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        super.init(context);
        transactionsStore = (KeyValueStore)context().getStateStore(transactionStoreName);
        clickEventStore = (KeyValueStore)context().getStateStore(clickEventStoreName);   1
        context().schedule(15000);                                                        2
    }
   //I'm leaving out the process method as it does the same thing as the other processor

    @Override
    public void punctuate(long timestamp) {
      KeyValueIterator<String, List<DayTradingAppClickEvent>> clickEvents =
        clickEventStore.all();                                                            3
      while(clickEvents.hasNext()){

          KeyValue<String,List<DayTradingAppClickEvent>> keyValue = clickEvents.next();
          String key = keyValue.key;

          List<DayTradingAppClickEvent> eventsSoFar = keyValue.value;
          List<StockTransaction> txnsSoFar = transactionsStore.delete(key);               4

          if (txnsSoFar == null){
              txnsSoFar = new ArrayList<>();
          }

          if (!eventsSoFar.isEmpty() || !txnsSoFar.isEmpty()) {
              Tuple<List<DayTradingAppClickEvent>,
                List<StockTransaction>> tuple = Tuple.of(eventsSoFar, txnsSoFar);       5
              context().forward(key, tuple);
          }

          clickEventStore.delete(key);                                                    6
      }

    }
}
```

1 Retrieving both state stores.

2 Scheduling to call punctuate every 15 seconds.

3 Iterating over everything in the store could be costly.

4 Deleting the list of stock transactions from the store.

5 If either list isn't empty, create a tuple and forward the key-value pair.

6 Delete the list of events from the store.

There are a few key points we need to discuss from the code listing. First is we are retrieving both state stores, as this will enable us to group both collections into a tuple. In the `punctuate` method we take advantage of the fact the `KeyValueStore.delete` the method returns the item removed from the store.

We don't want records to collect indefinitely so as soon as we can we remove them from the store. Since we are using the trading symbol as the key for both source topics, we can retrieve the relevant objects from the `transactionsStore` populated by the other processor in the topology.

At the bottom of the loop iterating over the records in the `clickEventStore`, we potentially forward the tuple containing a list of transactions and click events "co-grouped" by the stock trading symbol.

Next, we'll take a look at the last piece of this topology, creating the sink node.

### ADDING THE SINK NODE

The final part of our topology is adding the sink node to write our results out to a topic, as shown in the following image:

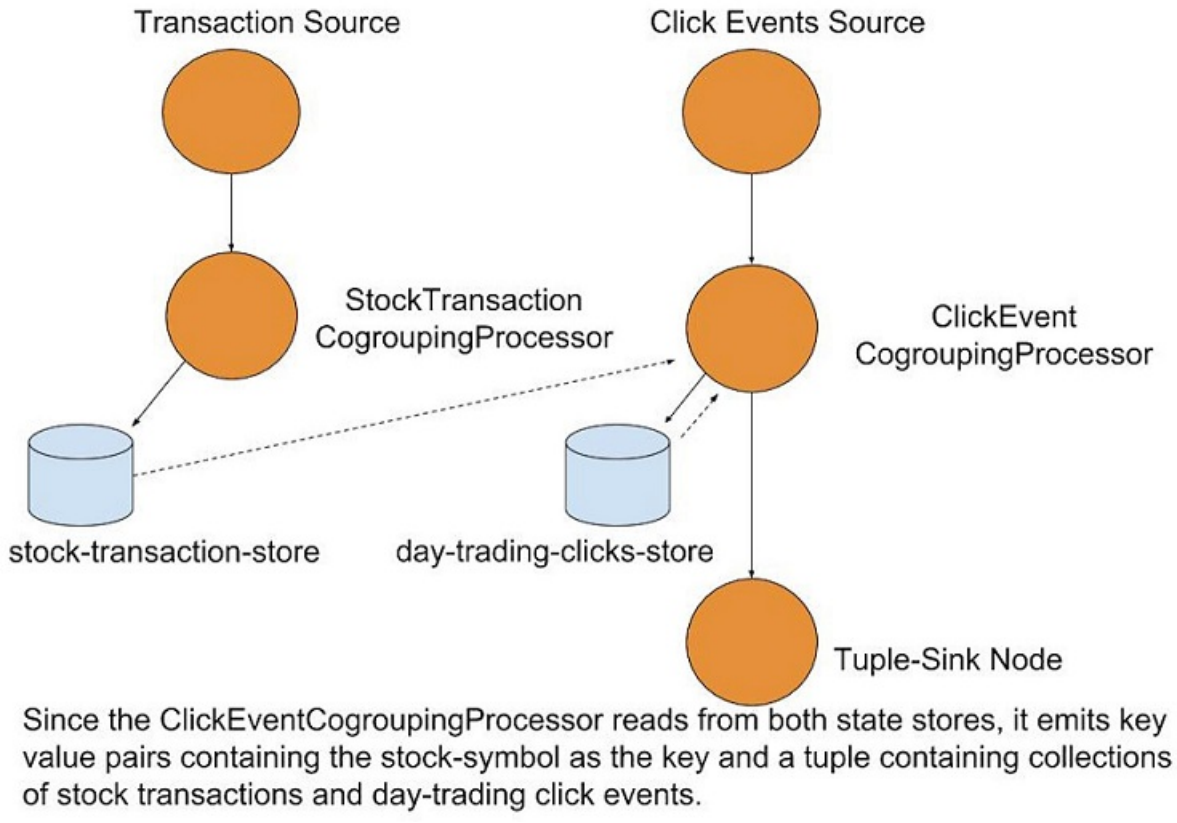



**Figure 6.13 Adding a sink node completes the co-grouping topology.**

As you can see, since the two processors share a state store, the click-events processor can read from both to emit records to the sink node.

**Listing 6.18 The sink node and a printing processor**

```
.addSink("tuple-sink", "cogrouped-results", stringSerializer, tupleSerializer, "evnts-processor");
 (1)

builder.addProcessor("print", new KStreamPrinter("Co-Grouping"), "evnts-processor");  (2)
```

(1) The sink node writing our co-grouped tuples out to a topic.

(2) A processor that prints results to stdout for use during development.

We've reached the end of building our co-grouping processor. The key point I want you to remember from this section is that while you have more code to write the Processor API gives you the flexibility to create virtually any kind of streaming topology you need.

Our next step is to demonstrate how we can integrate some Processor API functionality into a KStreams application.

## 6.5 Integrating the Processor API and the KStream API

Imagine you've started building a KafkaStreams application using the KStreams API. This application analyzes trends in stock prices over the last several trades (if this sounds familiar we're re-using our example from section). But our issue is we want control over when we emit records downstream depending on the performance of an individual stock.

We know we can write our processor to handle this requirement, but how can we integrate a custom processor into the KStream API? In the next section, we'll discuss options on how we can integrate custom processor functionality into the KStream API.

### 6.5.1 Integrating Processor API into KStream API

The KStream API offers three methods providing integration with the Processor API `KStream.process`, `KStream.transform` and `KStream.transformValues`. The following table describes how each one operates:

**Table 6.1 Transform Process Method Comparison**

| Supplier | Interface Type | Return Type | Terminal Node |
|---|---|---|---|
| ProcessorSupplier | Processor | Void | Yes |
| TransformerSupplier | Transformer | Generic Type <R> | No |

We can see there is a method taking a `ProcessorSupplier` instance, but calling `KStream.process` creates a terminal node, indicated by the return type of `void`. A terminal node is the end of the line for that *individual* branch of the topology. `KStream.transform` and `KStream.transformValues` are essentially the same methods, except for the fact that `transformValues` is expected to work with values only, thus gives you no access to the key. For our purposes the `KStream.transform` method fits the bill, so we'll discuss how we can use transform in the next section.

### USING THE TRANSFORM METHOD

The `KStream.transform` method takes a `TransformerSupplier` instance (or lambda expression substitute) which provides an instance of a `Transformer`. You can think of the `TransformerSupplier` and `Transformer` interfaces as mirror images of the `ProcessorSupplier` and `Processor` interfaces from the Processor API. For the most part, the functionality is same with two small differences:

1. The `Transformer` uses `transform` to do its work vs. the `process` method in the `Processor` interface.
2. The `punctuate` and `transform` methods return a Java generic type of `R` where the `process` and `punctuate` methods on the `Processor` interface don't return anything.

Despite these minor differences, we can use a `Transformer` as a drop-in replacement for a `Processor`, as we'll demonstrate next.

### REPLACING A PROCESSOR WITH A TRANSFORMER

Let's say you've used both the KStream and Processor API 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 it provides some of the lower lever control you need. What you'll want to do in this case is replace the `Processor` with a `Transformer`.

To use a `Transformer` in the same way, we'd use a `Processor` from the Processor API we have two simple guidelines:

1. Whenever you emit a *single* record in the `Processor` via a `ProcessorContext.forward` call you now *return* the value either in the `Transformer.transform` or `Transform.punctuate` methods.
2. If you need to emit *multiple* records, you take advantage of the fact that the `Transformer` also has a reference to the `ProcessorContext`. So for the case of sending multiple values you call `ProcessorContext.forward` for each record to send downstream, just as you do in a `Processor` and after forwarding you return null.

Let's take a look at the code we'll use in the `Transformer`. We'll only show the `transform` and `punctuate` methods

**Listing 6.19 Transformer replacement for Processor found in src/main/java/bbejeck/chapter_6/transformer/StockPerformanceTransformerSuppl**

```
@Override
public KeyValue<String, StockPerformance> transform(String symbol, StockTransaction transaction) {
       if (symbol != null) {
            StockPerformance stockPerformance = keyValueStore.get(symbol);
         if (stockPerformance == null) {
             stockPerformance = new StockPerformance();
         }
           stockPerformance.updatePriceStats(transaction.getSharePrice());
           stockPerformance.updateVolumeStats(transaction.getShares());
           stockPerformance.setLastUpdateSent(Instant.now());
           keyValueStore.put(symbol, stockPerformance);
       }
```

```
   return null;
}

@Override
public KeyValue<String, StockPerformance> punctuate(long timestamp) {
     KeyValueIterator<String, StockPerformance> performanceIterator = keyValueStore.all();
           while (performanceIterator.hasNext()) {
               KeyValue<String, StockPerformance> keyValue = performanceIterator.next();
               StockPerformance stockPerformance = keyValue.value;
            if (stockPerformance != null) {
               if (stockPerformance.priceDifferential() >= differentialThreshold ||
                   stockPerformance.volumeDifferential() >= differentialThreshold) {
                  this.processorContext.forward(keyValue.key, stockPerformance);
                }
              }
           }
        }
   return null;
}
```

(1) (2) (3)

(1) Returning null as we just store records in the transform method.

(2) Using the ProcessorContext to forward records as we iterate over the store.

(3) Returning null again as we exit the punctuate method.

Except for the return statements, and the transform method name this code matches exactly to what we have in the `StockPerformanceProcessor`. To cap this example off let's look at the application code we use

**Listing 6.20 Application code using Low level and High level APIs found in src/main/java/bbejeck/chapter_6/StockPerformanceStreamsAndProcessorApplicat**

```
StockPerformanceTransformerSupplier performanceTransformer = new StockPerformanceTransformerSupplier(
  (1)

builder.addStateStore(Stores.create(stocksStateStore).withStringKeys()
       .withValues(stockPerformanceSerde).inMemory().maxEntries(100).build());  (2)

builder.stream(LATEST, stringSerde, stockTransactionSerde, "stock-transactions")
       .transform(performanceTransformer, stocksStateStore)  (3)
       .print(stringSerde, stockPerformanceSerde, "StockPerformance");  (4)
```

(1) Creating the Transformer wrapped in a TransformerSupplier.

(2) Building the state store.

(3) Adding the transformer to the topology via KStream API.

(4) Printing results to console for development, in production write to a topic.

**TIP** In this example, we returned null from the `Transfomer.transform` and `Transformer.punctuate` methods because we want to forward multiple records at one time via the `ProcessorContext.forward` method. In the case where we want to forward a single record, we would return the record from the either transform/punctate and not use the `ProcessorContext.forward` method.

Now you've successfully combined the low-level Processor API with the KStreams DSL! The key point I want you to remember from this section is we can use a `Transformer` to give us the low-level control while staying in the KStreams API.

## 6.6 Conclusion

We've reached the end of our coverage of the low-level Processor API. While we covered a lot of ground here are the key points I want you to take away from this chapter:

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

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