SAMPLE CHAPTER

Kafka Streams IN ACTION

Real-time apps and microservices with the Kafka Streams API

William P. Bejeck Jr.

Foreword by Neha Narkhede

MANNING

Kafka Streams in Action

by William P. Bejeck Jr.

Chapter 1

brief contents

PART 1 GETTING STARTED WITH KAFKA STREAMS 1

1 ■ Welcome to Kafka Streams 3
2 ■ Kafka quickly 22

PART 2 KAFKA STREAMS DEVELOPMENT 55

3 ■ Developing Kafka Streams 57
4 ■ Streams and state 84
5 ■ The KTable API 117
6 ■ The Processor API 145

PART 3 ADMINISTERING KAFKA STREAMS 173

7 ■ Monitoring and performance 175
8 ■ Testing a Kafka Streams application 199

PART 4 ADVANCED CONCEPTS WITH KAFKA STREAMS 215

9 ■ Advanced applications with Kafka Streams 217

1 Welcome to Kafka Streams

This chapter covers

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

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

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

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

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

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

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

1.1.1 The genesis of big data

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

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

Figure 1.1 offers a graphical representation of the PageRank algorithm:

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

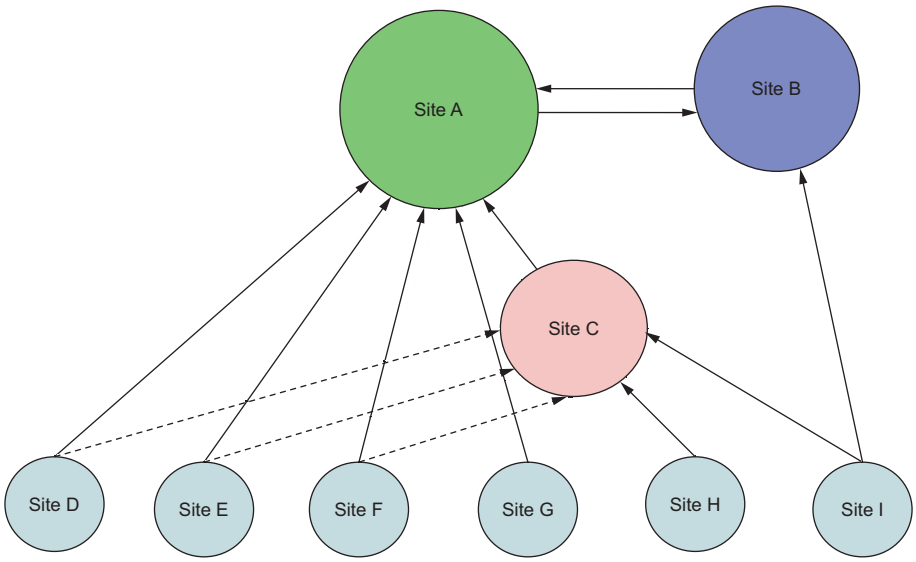


Figure 1.1 The PageRank algorithm in action. The circles represent websites, and the larger ones represent sites with more links pointing to them from other sites.

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

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

Google developed another revolutionary approach for processing all that data: the MapReduce paradigm. Not only did MapReduce enable Google to do the work it needed to as a company, it inadvertently spawned an entire new industry in computing.

1.1.2 Important concepts from MapReduce

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

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

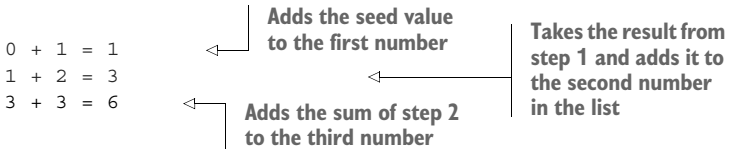
```
Function<LocalDate, String> addDate =
       (date) -> "The Day of the week is " + date.getDayOfWeek();
```

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

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

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

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



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

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

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

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

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

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

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

DISTRIBUTING DATA ACROSS A CLUSTER TO ACHIEVE SCALE IN PROCESSING

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

As you can see from the table, you may start out with an unwieldy amount of data to process, but by spreading the load across more servers, you eliminate the difficulty

Table 1.1 How splitting up 5 TB improves processing throughput

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

of processing the data. The 1 GB of data in the last line of the table is something a laptop could easily handle.

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

USING KEY/VALUE PAIRS AND PARTITIONS TO GROUP DISTRIBUTED DATA

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

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

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

Figure 1.2 shows how you could apply a hashing function to take results from Olympic events stored on separate servers and group them on partitions for different events.

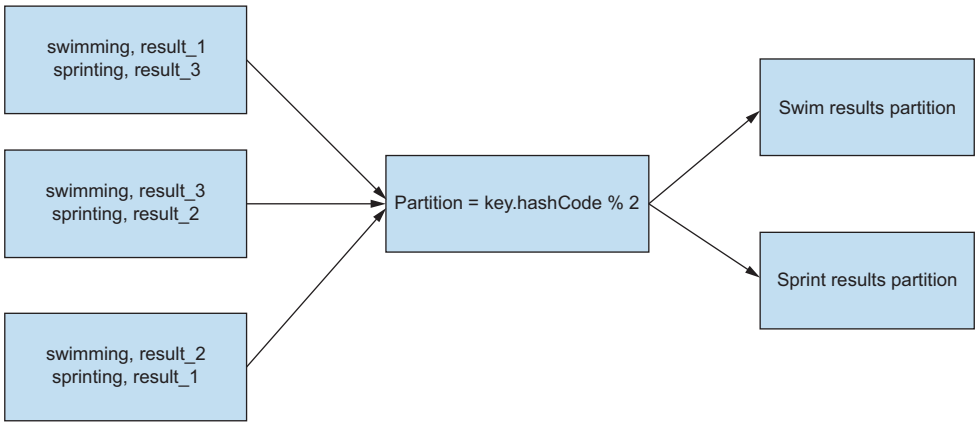


Figure 1.2 Grouping records by key on partitions. Even though the records start out on separate servers, they end up in the appropriate partitions.

All the data is stored as key/value pairs. In the image below the key is the name of the event, and the value is a result for an individual athlete.

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

EMBRACING FAILURE BY USING REPLICATION

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

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

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

1.1.3 Batch processing is not enough

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

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

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

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

1.2 Introducing stream processing

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

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

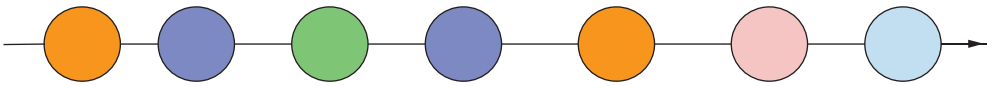


Figure 1.3 This marble diagram is a simple representation of stream processing. Each circle represents some information or an event occurring at a particular point in time. The number of events is unbounded and moves continually from left to right.

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

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

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

- *Credit card fraud*—A credit card owner may not notice a card has been stolen, but by reviewing purchases as they happen against established patterns (location, general spending habits), you may be able to detect a stolen credit card and alert the owner.
- *Intrusion detection*—Analyzing application log files after a breach has occurred may be helpful to prevent future attacks or to improve security, but the ability to monitor aberrant behavior in real time is critical.
- *A large race, such as the New York City Marathon*—Almost all runners will have a chip on their shoe, and when runners pass sensors along the course, you can use that information to track the runners' positions. By using the sensor data, you can determine the leaders, spot potential cheating, and detect whether a runner is potentially having problems.
- *The financial industry*—The ability to track market prices and direction in real time is essential for brokers and consumers to make effective decisions about when to sell or buy.

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

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

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

1.3 Handling a purchase transaction

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

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

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

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

1.3.1 Weighing the stream-processing option

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

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

- *Privacy*—First and foremost, ZMart values its relationship with its customers. With all of today's privacy concerns, your first goal is to protect customers' privacy, and protecting their credit card numbers is the highest priority. However you use the transaction information, customer credit card information should never be at risk of exposure.
- *Customer rewards*—A new customer-rewards program is in place, with customers earning bonus points based on the amount of money they spend on certain

items. The goal is to notify customers quickly, once they've received a reward—you want them back in the store! Again, appropriate monitoring of activity is required here. Remember how Jane received an email immediately after leaving the store? That's the kind of exposure you want for the company.

- *Sales data*—ZMart would like to refine its advertising and sales strategy. The company wants to track purchases by region to figure out which items are more popular in certain parts of the country. The goal is to target sales and specials for best-selling items in a given area of the country.
- *Storage*—All purchase records need to be saved in an off-site storage center for historical and ad hoc analysis.

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

1.3.2 Deconstructing the requirements into a graph

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

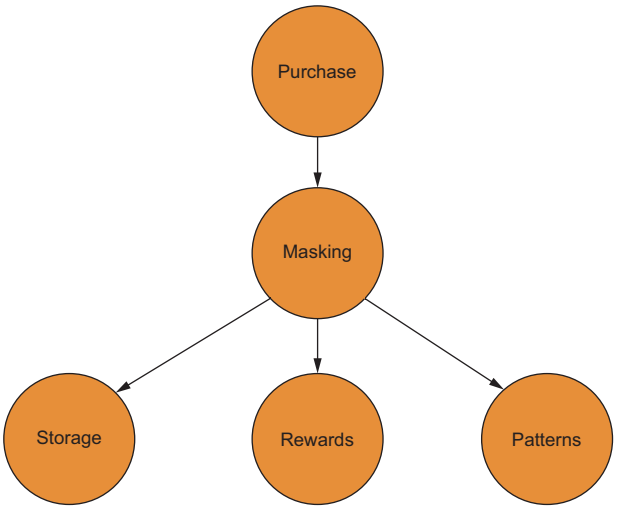


Figure 1.4 The business requirements for the streaming application presented as a directed acyclic graph. Each vertex represents a requirement, and the edges show the flow of data through the graph.

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

1.4 Changing perspective on a purchase transaction

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

1.4.1 Source node

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

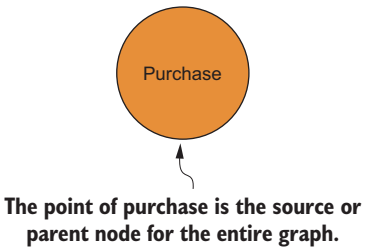


Figure 1.5 The simple start for the sales transaction graph. This node is the source of raw sales transaction information that will flow through the graph.

1.4.2 Credit card masking node

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

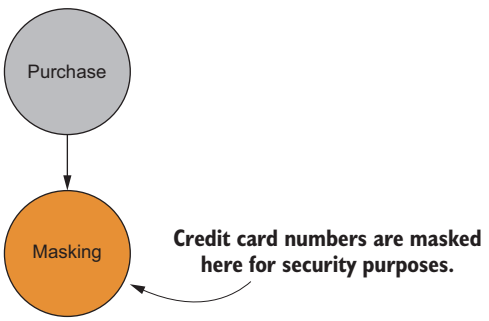


Figure 1.6 The first node in the graph that represents the business requirements. This node is responsible for masking credit card numbers and is the only node that receives the raw sales data from the source node, effectively making it the source for all other nodes connected to it.

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

1.4.3 Patterns node

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

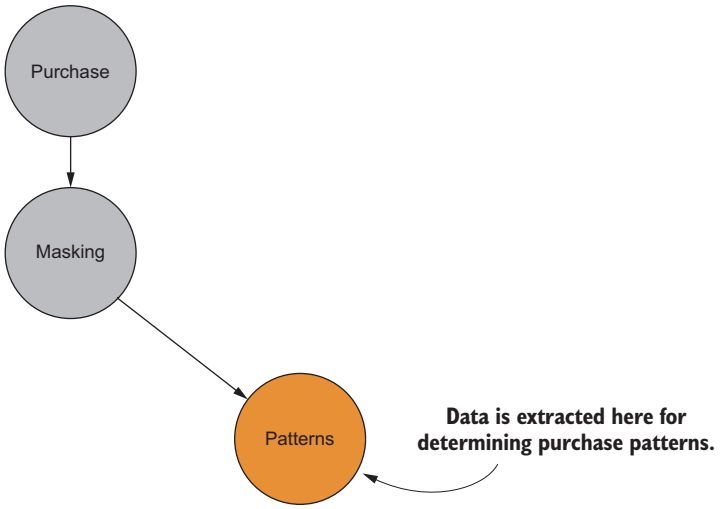


Figure 1.7 The patterns node consumes purchase information from the masking node and converts it into a record showing when a customer purchased an item and the ZIP code where the customer completed the transaction.

1.4.4 Rewards node

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

1.4.5 Storage node

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

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

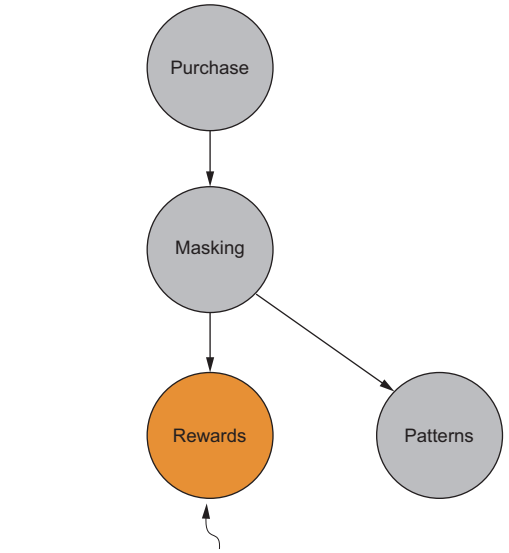


Figure 1.8 The rewards node is responsible for consuming sales records from the masking node and converting them into records containing the total of the purchase and the customer ID.

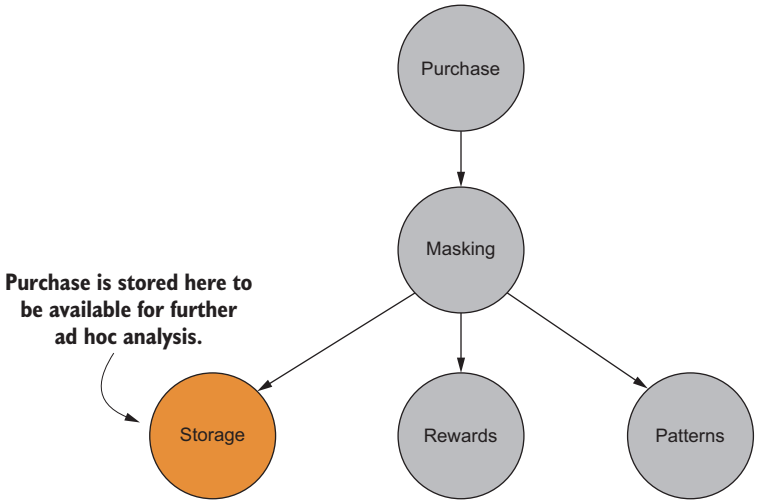


Figure 1.9 The storage node consumes records from the masking node as well. These records aren't converted into any other format but are stored in a NoSQL data store for ad hoc analysis later.

1.5 Kafka Streams as a graph of processing nodes

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

Most of ZMart's goals are time sensitive, in that you want to take action as soon as possible. Preferably, you'll be able to collect information as events occur. Additionally, there are several ZMart locations across the country, so you'll need all the transaction records to funnel into a single flow or *stream* of data for analysis. For these reasons, Kafka Streams is a perfect fit. Kafka Streams allows you to process records as they arrive and gives you the low-latency processing you require.

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

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

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

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

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

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

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

> **Kafka Streams and Kafka**
>
> As you might have guessed from the name, Kafka Streams runs on top of Kafka. In this introductory chapter, you don't need to know about Kafka, because we're focusing more how Kafka Streams works conceptually. A few Kafka-specific terms may be mentioned, but for the most part, we'll be concentrating on the stream-processing aspects of Kafka Streams.
>
> If you're new to Kafka or are unfamiliar with it, you'll learn what you need to know about Kafka in chapter 2. Knowledge of Kafka is essential for working effectively with Kafka Streams.

1.6 Applying Kafka Streams to the purchase transaction flow

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

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

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

- ZMart customer ID (scanned from the member card)
- Total dollar amount spent
- Item(s) purchased
- ZIP code of the store where the purchase took place
- Date and time of the transaction
- Debit or credit card number

1.6.1 Defining the source

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

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

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

It's important to note that to Kafka, the Kafka Streams program looks like any other combination of consumers and producers. Any number of applications could

be reading from the same topic in conjunction with your streaming program. Figure 1.10 represents the source node in the topology.

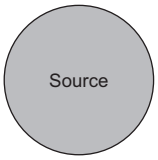


Figure 1.10 The source node: a Kafka topic

1.6.2 The first processor: masking credit card numbers

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

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

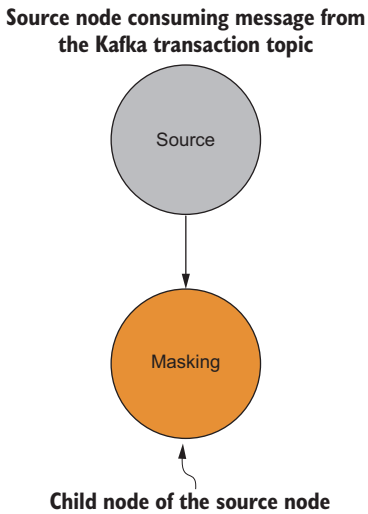


Figure 1.11 The masking processor is a child of the main source node. It receives all the raw sales transactions and emits new records with the credit card number masked.

CREATING PROCESSOR TOPOLOGIES

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

It's important to note that calling a method that returns a new `KStream` instance doesn't cause the original instance to stop consuming messages. A transforming method

creates a new processor and adds it to the existing processor topology. The updated topology is then used as a parameter to create the next `KStream` instance, which starts receiving messages from the point of its creation.

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

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

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

1.6.3 The second processor: purchase patterns

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

The purchase-patterns processor receives a `Purchase` object from its parent node and maps the object to a new `PurchasePattern` object. The mapping process extracts

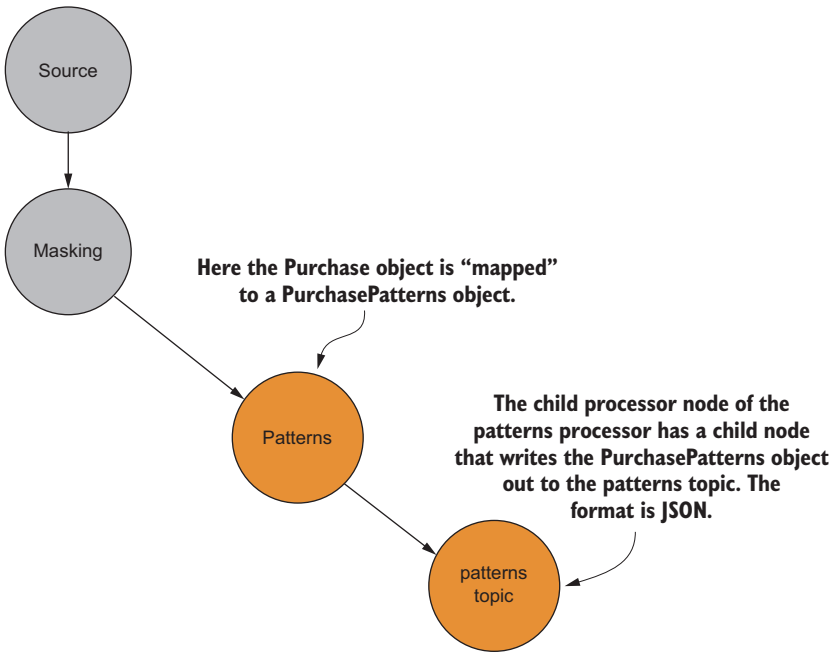


Figure 1.12 The purchase-pattern processor takes `Purchase` objects and converts them into `PurchasePattern` objects containing the items purchased and the ZIP code where the transaction took place. A new processor takes records from the patterns processor and writes them out to a Kafka topic.

the item purchased (toothpaste, for example) and the ZIP code it was bought in and uses that information to create the `PurchasePattern` object. We'll go over exactly how this mapping process occurs in chapter 3.

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

1.6.4 The third processor: customer rewards

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

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

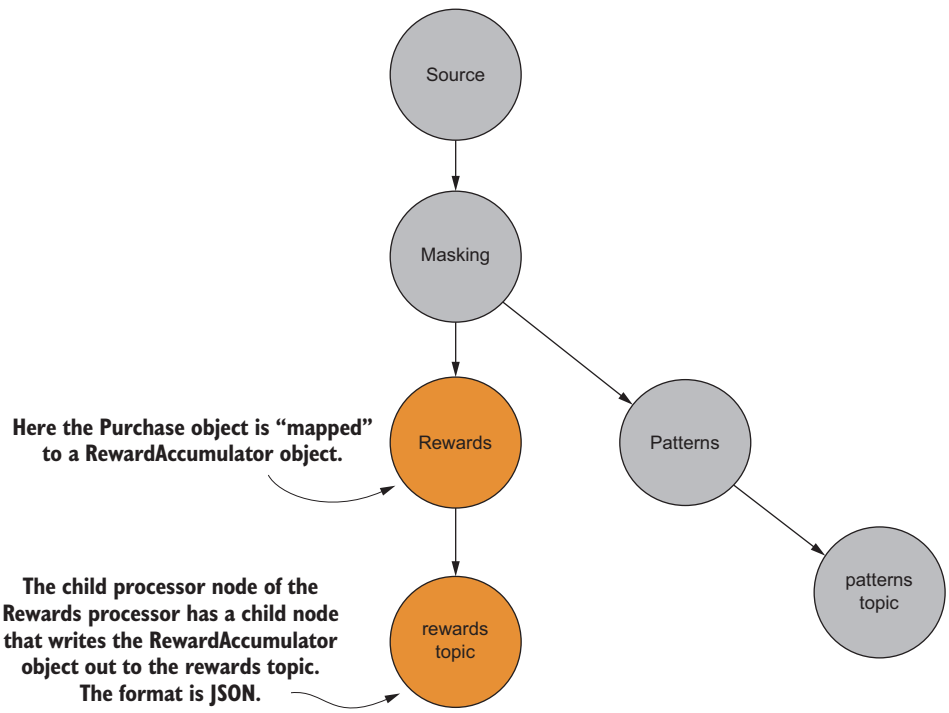


Figure 1.13 The customer rewards processor is responsible for transforming `Purchase` objects into a `RewardAccumulator` object containing the customer ID, date, and dollar amount of the transaction. A child processor writes the Rewards objects to another Kafka topic.

1.6.5 The fourth processor—writing purchase records

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

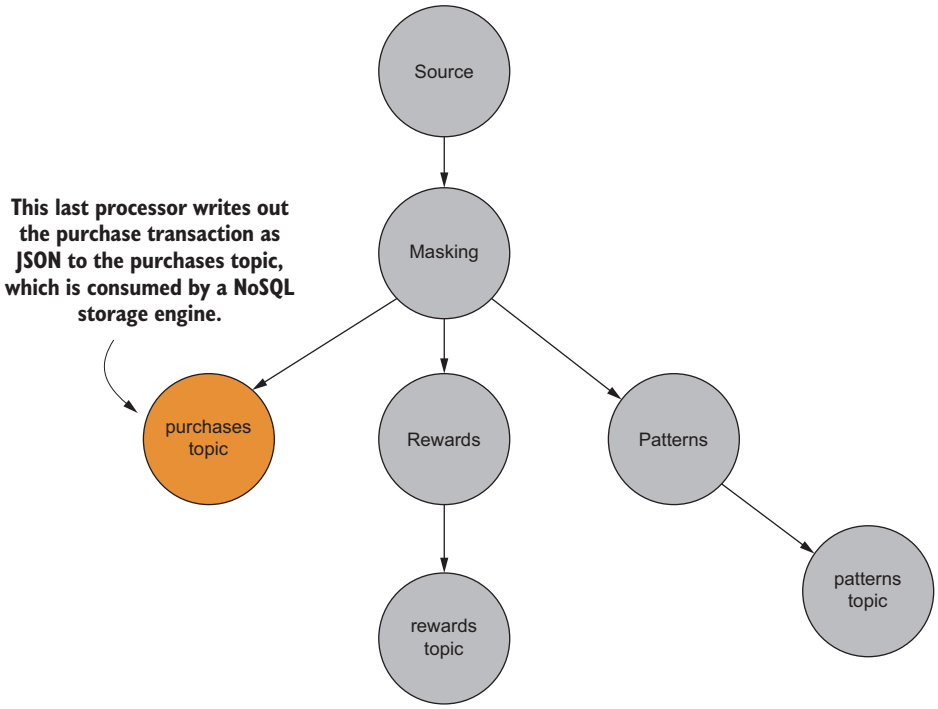


Figure 1.14 The final processor is responsible for writing out the entire `Purchase` object to another Kafka topic. The consumer for this topic will store the results in a NoSQL store such as MongoDB.

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

Summary

- Kafka Streams is a graph of processing nodes that combine to provide powerful and complex stream processing.
- Batch processing is powerful, but it's not enough to satisfy real-time needs for working with data.

- Distributing data, key/value pairs, partitioning, and data replication are critical for distributed applications.

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

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

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

SOFTWARE DEVELOPMENT

Kafka Streams IN ACTION

William P. Bejeck Jr.

Not all stream-based applications require a dedicated processing cluster. The lightweight Kafka Streams library provides exactly the power and simplicity you need for message handling in microservices and real-time event processing. With the Kafka Streams API, you filter and transform data streams with just Kafka and your application.

Kafka Streams in Action teaches you to implement stream processing within the Kafka platform. In this easy-to-follow book, you'll explore real-world examples to collect, transform, and aggregate data, work with multiple processors, and handle real-time events. You'll even dive into streaming SQL with KSQL! Practical to the very end, it finishes with testing and operational aspects, such as monitoring and debugging.

What's Inside

- Using the KStream API
- Filtering, transforming, and splitting data
- Working with the Processor API
- Integrating with external systems

Assumes some experience with distributed systems. No knowledge of Kafka or streaming applications required.

Bill Bejeck is a Kafka Streams contributor and Confluent engineer with over 15 years of software development experience.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/kafka-streams-in-action

Free eBook
See first page

"A great way to learn about Kafka Streams and how it is a key enabler of event-driven applications."

—From the Foreword by Neha Narkhede
Cocreator of Apache Kafka

"A comprehensive guide to Kafka Streams—from introduction to production!"

—Bojan Djurkovic, Cvent

"Bridges the gap between message brokering and real-time streaming analytics."

—Jim Mantheiy Jr.
Next Century

"Valuable both as an introduction to streams as well as an ongoing reference."

—Robin Coe, TD Bank

ISBN-13: 978-1-61729-447-1
ISBN-10: 1-61729-447-0
54499
9 781617 294471

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]