Mastering Kafka Streams and ksqlDB

Building Real-Time Data Systems by Example

Mitch Seymour

Mastering Kafka Streams and ksqlDB

by Mitch Seymour

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman

Development Editor: Jeff Bleiel

Production Editor: Daniel Elfanbaum

Copyeditor: Kim Cofer

Proofreader: JM Olejarz

Indexer: Ellen Troutman-Zaig

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

February 2021: First Edition

Revision History for the First Edition

- 2021-02-04: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492062493 for release details.

978-1-492-06249-3

[LSI]

Foreword

Businesses are increasingly built around events—the real-time activity data of what is happening in a company—but what is the right infrastructure for harnessing the power of events? This is a question I have been thinking about since 2009, when I started the Apache Kafka project at LinkedIn. In 2014, I cofounded Confluent to definitively answer it. Beyond providing a way to store and access discrete events, an event streaming platform needs a mechanism to connect with a myriad of external systems. It also requires global schema management, metrics, and monitoring. But perhaps most important of all is stream processing—continuous computation over never-ending streams of data—without which an event streaming platform is simply incomplete.

Now more than ever, stream processing plays a key role in how businesses interact with the world. In 2011, Marc Andreessen wrote an article titled "Why Software Is Eating the World." The core idea is that any process that can be moved into software eventually will be. Marc turned out to be prescient. The most obvious outcome is that software has permeated every industry imaginable.

But a lesser understood and more important outcome is that businesses are increasingly defined in software. Put differently, the core processes a business executes—from how it creates a product, to how it interacts with customers, to how it delivers services—are increasingly specified, monitored, and executed in software. What has changed because of that dynamic? Software, in this new world, is far less likely to be directly interacting with a human. Instead, it is more likely that its purpose is to programmatically trigger actions or react to other pieces of software that carry out business directly.

It begs the question: are our traditional application architectures, centered around existing databases, sufficient for this emerging world? Virtually all databases, from the most established relational databases to the newest key-value stores, follow a paradigm in which data is passively stored and the database waits for commands to retrieve or modify it. This paradigm was driven by human-facing applications in which a user looks at an interface and initiates actions that are translated into database queries. We think that is only half the problem, and the problem of storing data needs to be complemented with the ability to react to and process events.

Events and stream processing are the keys to succeeding in this new world. Events support the continuous flow of data throughout a business, and stream processing automatically executes code in response to change at any level of detail—doing it in concert with knowledge of all changes that came before it. Modern stream processing systems like Kafka Streams and ksqlDB make it easy to build applications for a world that speaks software.

In this book, Mitch Seymour lucidly describes these state-of-the-art systems from first principles. *Mastering Kafka Streams and ksqlDB* surveys core concepts, details the nuances of how each system works, and provides hands-on examples for using them for business in the real world. Stream processing has never been a more essential programming paradigm—and *Mastering Kafka Streams and ksqlDB* illuminates the path to succeeding at it.

Jay Kreps
Cocreator of Apache Kafka,
Cofounder and CEO of Confluent

Preface

For data engineers and data scientists, there’s never a shortage of technologies that are competing for our attention. Whether we’re perusing our favorite subreddits, scanning Hacker News, reading tech blogs, or weaving through hundreds of tables at a tech conference, there are so many things to look at that it can start to feel overwhelming.

But if we can find a quiet corner to just think for a minute, and let all of the buzz fade into the background, we can start to distinguish patterns from the noise. You see, we live in the age of explosive data growth, and many of these technologies were created to help us store and process data at scale. We’re told that these are modern solutions for modern problems, and we sit around discussing “big data” as if the idea is avant-garde, when really the focus on data volume is only half the story.

Technologies that only solve for the data volume problem tend to have batch-oriented techniques for processing data. This involves running a job on some pile of data that has accumulated for a period of time. In some ways, this is like trying to drink the ocean all at once. With modern computing power and paradigms, some technologies actually manage to achieve this, though usually at the expense of high latency.

Instead, there’s another property of modern data that we focus on in this book: data moves over networks in steady and never-ending streams. The technologies we cover in this book, Kafka Streams and ksqlDB, are specifically designed to process these continuous data streams in real time, and provide huge competitive advantages over the ocean-drinking variety. After all, many business problems are time-sensitive, and if you need to enrich, transform, or react to data as soon as it comes in, then Kafka Streams and ksqlDB will help get you there with ease and efficiency.

Learning Kafka Streams and ksqlDB is also a great way to familiarize yourself with the larger concepts involved in stream processing. This includes modeling data in different ways (streams and tables), applying stateless transformations of data, using local state for more advanced operations (joins, aggregations), understanding the different time semantics and methods for grouping data into time buckets/windows, and more. In other words, your knowledge of Kafka Streams and ksqlDB will help you distinguish and evaluate different stream processing solutions that currently exist and may come into existence sometime in the future.

I’m excited to share these technologies with you because they have both made an impact on my own career and helped me accomplish technological feats that I thought were beyond my own capabilities. In fact, by the time you finish reading this sentence, one of my Kafka Streams applications will have processed nine million events. The feeling you’ll get by providing real business value without having to invest exorbitant amounts of time on the solution will keep you working with these technologies for years to come, and the succinct and expressive language constructs make the process feel more like an art form than a labor. And just like any other art form, whether it be a life-changing song or a beautiful painting, it's human nature to want to share it. So consider this book a mixtape from me to you, with my favorite compilations from the stream processing space available for your enjoyment: Kafka Streams and ksqlDB, Volume 1.

Who Should Read This Book

This book is for data engineers who want to learn how to build highly scalable stream processing applications for moving, enriching, and transforming large amounts of data in real time. These skills are often needed to support business intelligence initiatives, analytic pipelines, threat detection, event processing, and more. Data scientists and analysts who want to upgrade their skills by analyzing real-time data streams will also find value in this book, which is an exciting departure from the batch processing space that has typically dominated these fields. Prior experience with Apache Kafka is not required, though some familiarity with the Java programming language will make the Kafka Streams tutorials easier to follow.

Navigating This Book

This book is organized roughly as follows:

- Chapter 1 provides an introduction to Kafka and a tutorial for running a single-node Kafka cluster.
- Chapter 2 provides an introduction to Kafka Streams, starting with a background and architectural review, and ending with a tutorial for running a simple Kafka Streams application.
- Chapters 3 and 4 discuss the stateless and stateful operators in the Kafka Streams high-level DSL (domain-specific language). Each chapter includes a tutorial that will demonstrate how to use these operators to solve an interesting business problem.
- Chapter 5 discusses the role that time plays in our stream processing applications, and demonstrates how to use windows to perform more advanced stateful operations, including windowed joins and aggregations. A tutorial inspired by predictive healthcare will demonstrate the key concepts.
- Chapter 6 describes how stateful processing works under the hood, and provides some operational tips for stateful Kafka Streams applications.
- Chapter 7 dives into Kafka Streams' lower-level Processor API, which can be

used for scheduling periodic functions, and provides more granular access to application state and record metadata. The tutorial in this chapter is inspired by IoT (Internet of Things) use cases.

- Chapter 8 provides an introduction to ksqlDB, and discusses the history and architecture of this technology. The tutorial in this chapter will show you how to install and run a ksqlDB server instance, and work with the ksqlDB CLI.
- Chapter 9 discusses ksqlDB's data integration features, which are powered by Kafka Connect.
- Chapters 10 and 11 discuss the ksqlDB SQL dialect in detail, demonstrating how to work with different collection types, perform push queries and pull queries, and more. The concepts will be introduced using a tutorial based on a Netflix use case: tracking changes to various shows/films, and making these changes available to other applications.
- Chapter 12 provides the information you need to deploy your Kafka Streams and ksqlDB applications to production. This includes information on monitoring, testing, and containerizing your applications.

Source Code

The source code for this book can be found on GitHub at *https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb*.

Instructions for building and running each tutorial will be included in the repository.

Kafka Streams Version

At the time of this writing, the latest version of Kafka Streams was version 2.7.0. This is the version we use in this book, though in many cases, the code will also work with older or newer versions of the Kafka Streams library. We will make efforts to update the source code when newer versions introduce breaking changes, and will stage these updates in a dedicated branch (e.g., kafka-streams-2.8).

ksqlDB Version

At the time of this writing, the latest version of ksqlDB was version 0.14.0. Compatibility with older and newer versions of ksqlDB is less guaranteed due to the ongoing and rapid development of this technology, and the lack of a major version (e.g., 1.0) at the time of this book's publication. We will make efforts to update the source code when newer versions introduce breaking changes, and will stage these updates in a dedicated branch (e.g., ksqldb-0.15). However, it is recommended to avoid versions older than 0.14.0 when running the examples in this book.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

Tip

This element signifies a tip or suggestion.
Note

This element signifies a general note.
Warning

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) can be found on the book's GitHub page, *https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb*.

If you have a technical question or a problem using the code examples, please email *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Mastering Kafka Streams and ksqlDB* by Mitch Seymour (O'Reilly). Copyright 2021 Mitch Seymour, 978-1-492-06249-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

Note

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/mastering-kafka-streams*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*.

Follow us on Twitter: *http://twitter.com/oreillymedia*.

Watch us on YouTube: *http://www.youtube.com/oreillymedia*.

Acknowledgments

First and foremost, I want to thank my wife, Elyse, and my daughter, Isabelle. Writing a book is a huge time investment, and your patience and support through the entire process helped me immensely. As much as I enjoyed writing this book, I missed you both greatly, and I look forward to having more date nights and daddy-daughter time again.

I also want to thank my parents, Angie and Guy, for teaching me the value of hard work and for being a never-ending source of encouragement. Your support has helped me overcome many challenges over the years, and I am eternally grateful for you both.

This book would not be possible without the following people, who dedicated a lot of their time to reviewing its content and providing great feedback and advice along the way: Matthias J. Sax, Robert Yokota, Nitin Sharma, Rohan Desai, Jeff Bleiel, and Danny Elfanbaum. Thank you all for helping me create this book, it's just as much yours as it is mine.

Many of the tutorials were informed by actual business use cases, and I owe a debt of gratitude to everyone in the community who openly shared their experiences with Kafka Streams and ksqlDB, whether it be through conferences, podcasts, blogs, or even in-person interviews. Your experiences helped shape this book, which puts a special emphasis on practical stream processing applications. Nitin Sharma also provided ideas for the Netflix-inspired ksqlDB tutorials, and Ramesh Sringeri shared his stream processing experiences at Children's Healthcare of Atlanta, which inspired the predictive healthcare tutorial. Thank you both.

Special thanks to Michael Drogalis for being a huge supporter of this book, even when it was just an outline of ideas. Also, thank you for putting me in touch with many of this book's reviewers, and also Jay Kreps, who graciously wrote the foreword. The technical writings of Yeva Byzek and Bill Bejeck have also set a high bar for what this book should be. Thank you both for your contributions in this space.

There have been many people in my career that helped get me to this point. Mark Conde and Tom Stanley, thank you for opening the doors to my career as a software engineer. Barry Bowden, for helping me become a better engineer, and for being a great mentor. Erin Fusaro, for knowing exactly what to say whenever I felt overwhelmed, and for just being a rock in general. Justin Isasi, for your continuous encouragement, and making sure my efforts don't go unrecognized. Sean Sawyer, for a suggestion you made several years ago, that I try a new thing called "Kafka Streams," which has clearly spiraled out of control. Thomas Holmes and Matt Farmer, for sharing your technical expertise with me on many occasions, and helping me become a better engineer. And to the Data Services team at Mailchimp, thanks for helping me solve some really cool problems, and for inspiring me with your own work.

Finally, to my friends and family, who continue to stick by me even when I disappear for months at a time to work on a new project. Thanks for sticking around, this was a long one.

Part I. Kafka

Chapter 1. A Rapid Introduction to Kafka

The amount of data in the world is growing exponentially and, according to the World Economic Forum, the number of bytes being stored in the world already far exceeds the number of stars in the observable universe.

When you think of this data, you might think of piles of bytes sitting in data warehouses, in relational databases, or on distributed filesystems. Systems like these have trained us to think of data in its *resting state*. In other words, data is sitting somewhere, resting, and when you need to process it, you run some query or job against the pile of bytes.

This view of the world is the more traditional way of thinking about data. However, while data can certainly pile up in places, more often than not, it's moving. You see, many systems generate continuous streams of data, including IoT sensors, medical sensors, financial systems, user and customer analytics software, application and server logs, and more. Even data that eventually finds a nice place to rest likely travels across the network at some point before it finds its forever home.

If we want to process data in real time, while it moves, we can't simply wait for it to pile up somewhere and then run a query or job at some interval of our choosing. That approach can handle some business use cases, but many important use cases require us to process, enrich, transform, and respond to data incrementally as it becomes available. Therefore, we need something that has a very different worldview of data: a technology that gives us access to data in its *flowing state*, and which allows us to work with these continuous and unbounded data streams quickly and efficiently. This is where Apache Kafka comes in.

Apache Kafka (or simply, Kafka) is a streaming platform for ingesting, storing, accessing, and processing streams of data. While the entire platform is very interesting, this book focuses on what I find to be the most compelling part of Kafka: the stream processing layer. However, to understand Kafka Streams and ksqlDB (both of which operate at this layer, and the latter of which also operates at the stream ingestion layer), it is necessary to have a working knowledge of how Kafka, as a platform, works.

Therefore, this chapter will introduce you to some important concepts and terminology that you will need for the rest of the book. If you already have a working knowledge of Kafka, feel free to skip this chapter. Otherwise, keep reading.

Some of the questions we will answer in this chapter include:

- How does Kafka simplify communication between systems?

- What are the main components in Kafka's architecture?
- Which storage abstraction most closely models streams?
- How does Kafka store data in a fault-tolerant and durable manner?
- How is high availability and fault tolerance achieved at the data processing layers?

We will conclude this chapter with a tutorial showing how to install and run Kafka. But first, let's start by looking at Kafka's communication model.

Communication Model

Perhaps the most common communication pattern between systems is the synchronous, client-server model. When we talk about systems in this context, we mean applications, microservices, databases, and anything else that reads and writes data over a network. The client-server model is simple at first, and involves direct communication between systems, as shown in Figure 1-1.

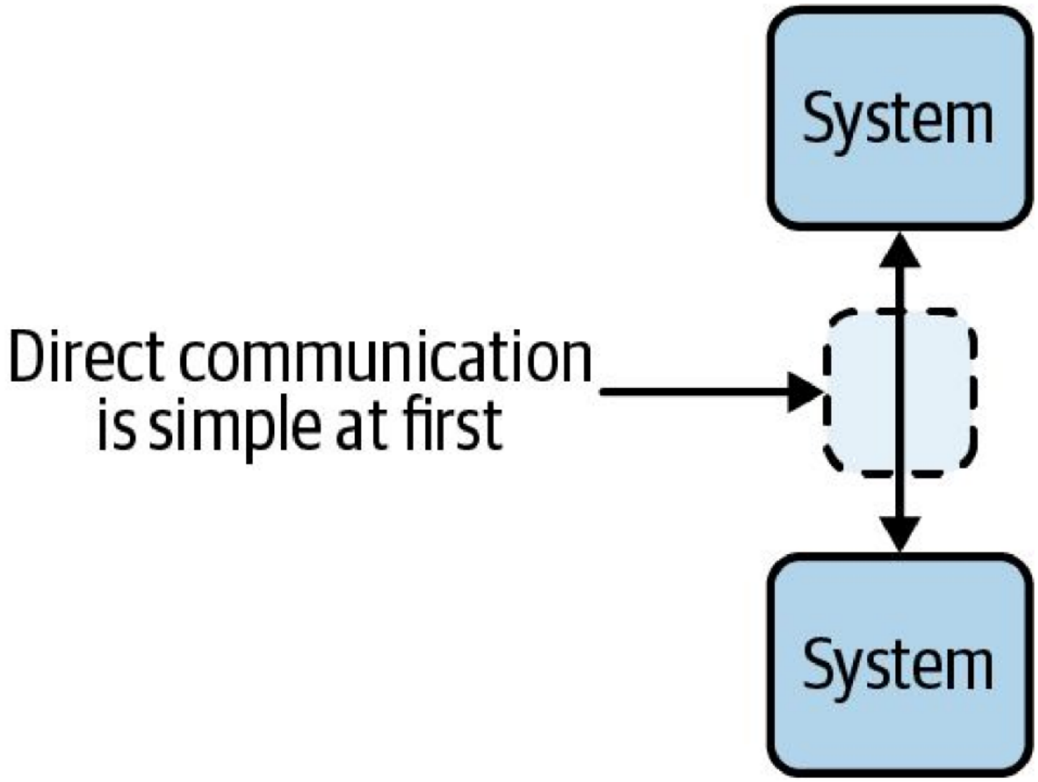


Figure 1-1. Point-to-point communication is simple to maintain and reason about when you have a small number of systems

For example, you may have an application that synchronously queries a database for some data, or a collection of microservices that talk to each other directly.

However, when more systems need to communicate, point-to-point communication becomes difficult to scale. The result is a complex web of communication pathways that can be difficult to reason about and maintain. Figure 1-2 shows just how confusing it can get, even with a relatively small number of systems.

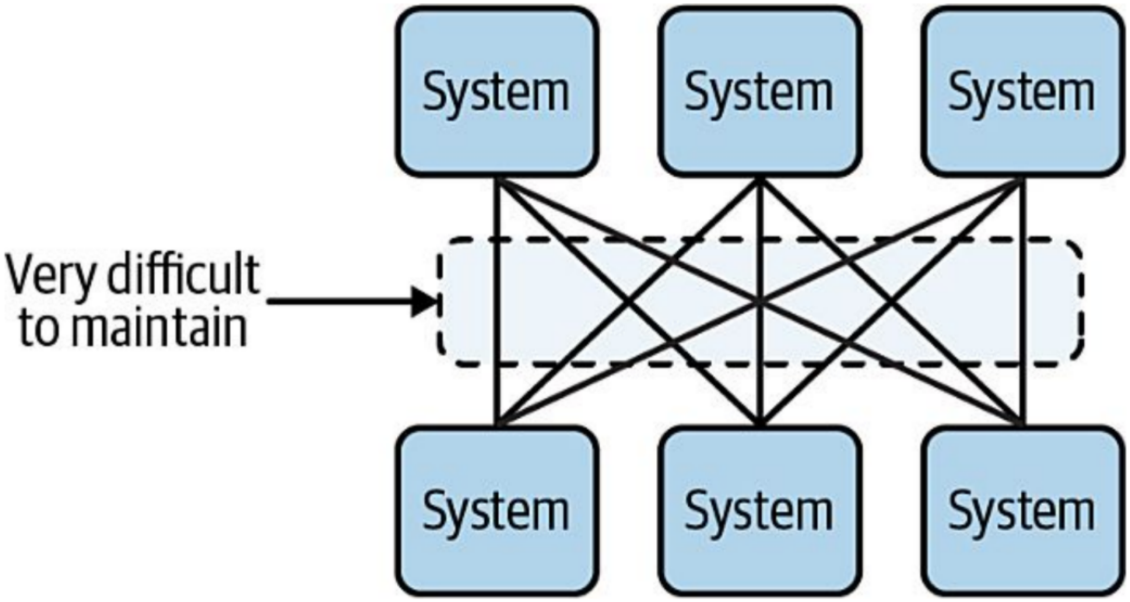


Figure 1-2. The result of adding more systems is a complex web of communication channels, which is difficult to maintain

Some of the drawbacks of the client-server model include:

- Systems become *tightly coupled* because their communication depends on knowledge of each other. This makes maintaining and updating these systems more difficult than it needs to be.
- Synchronous communication leaves little room for error since there are *no delivery guarantees* if one of the systems goes offline.
- Systems may use different communication protocols, scaling strategies to deal with increased load, failure-handling strategies, etc. As a result, you may end up with multiple species of systems to maintain (*software speciation*), which hurts maintainability and defies the common wisdom that we should treat applications like cattle instead of pets.
- Receiving systems can easily be overwhelmed, since they don't control the pace at which new requests or data comes in. *Without a request buffer*, they

operate at the whims of the applications that are making requests.

- There isn't a strong notion for what is being communicated between these systems. The nomenclature of the client-server model has put too much emphasis on *requests* and *responses*, and not enough emphasis on the data itself. Data should be the focal point of data-driven systems.
- Communication is *not replayable*. This makes it difficult to reconstruct the state of a system.

Kafka simplifies communication between systems by acting as a centralized communication hub (often likened to a central nervous system), in which systems can send and receive data without knowledge of each other. The communication pattern it implements is called the *publish-subscribe pattern* (or simply, pub/sub), and the result is a drastically simpler communication model, as shown in Figure 1-3.

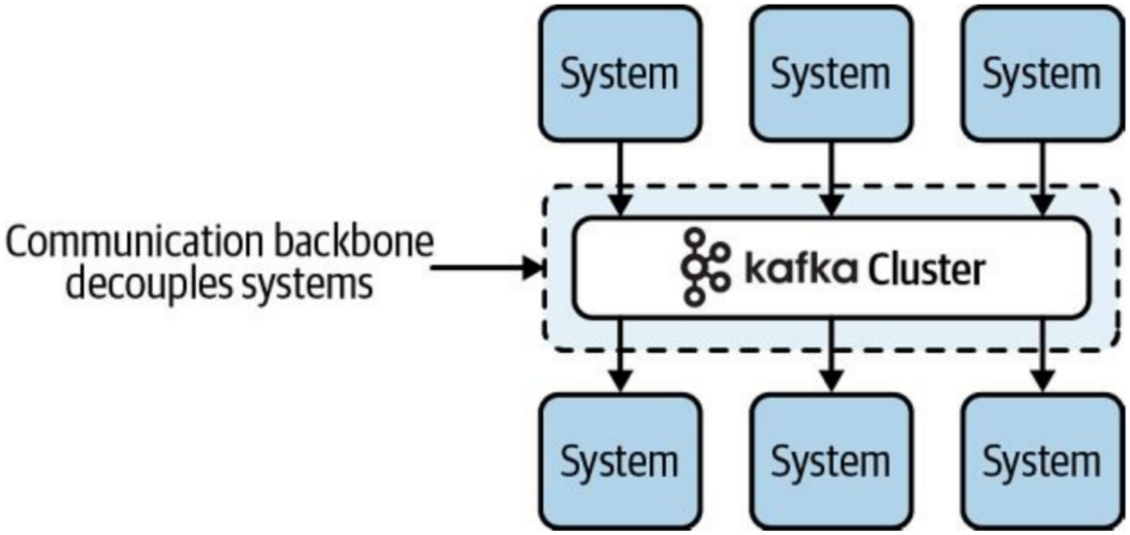


Figure 1-3. Kafka removes the complexity of point-to-point communication by acting as a communication hub between systems

If we add more detail to the preceding diagram, we can begin to identify the main components involved in Kafka's communication model, as shown in Figure 1-4.

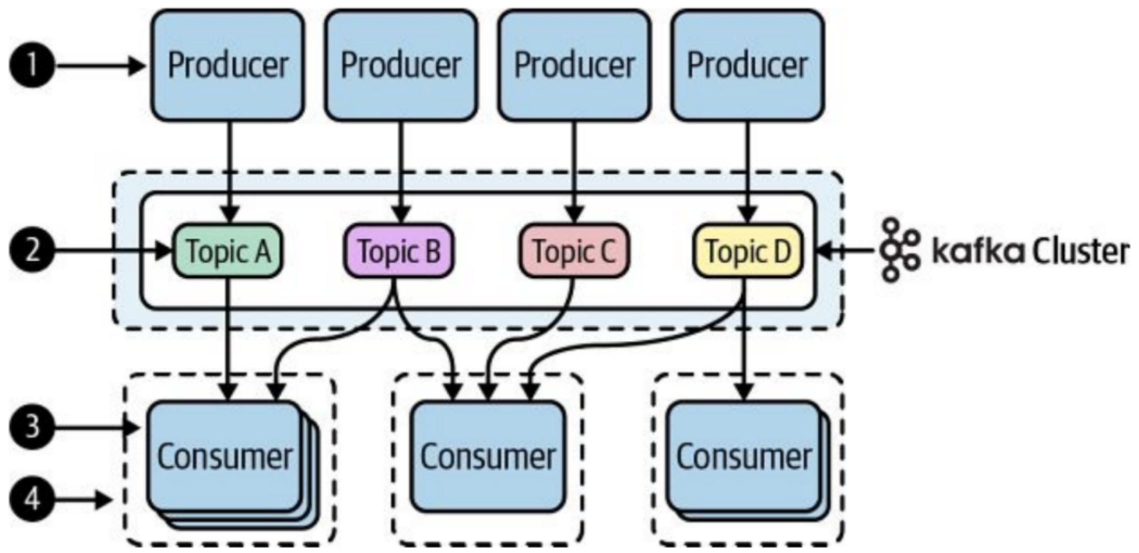


Figure 1-4. The Kafka communication model, redrawn with more detail to show the main components of the Kafka platform

❶ Instead of having multiple systems communicate directly with each other, *producers* simply publish their data to one or more topics, without caring who comes along to read the data.

❷ *Topics* are named streams (or channels) of related data that are stored in a Kafka cluster. They serve a similar purpose as tables in a database (i.e., to group related data). However, they do not impose a particular schema, but rather store the raw bytes of data, which makes them very flexible to work with.[1]

❸ *Consumers* are processes that read (or subscribe) to data in one or more topics. They do not communicate directly with the producers, but rather listen to data on any stream they happen to be interested in.

❹ Consumers can work together as a group (called a *consumer group*) in order to distribute work across multiple processes.

Kafka's communication model, which puts more emphasis on flowing streams of data that can easily be read from and written to by multiple processes, comes with several advantages, including:

- Systems become *decoupled* and easier to maintain because they can produce and consume data without knowledge of other systems.

- Asynchronous communication comes with *stronger delivery guarantees*. If a consumer goes down, it will simply pick up from where it left off when it comes back online again (or, when running with multiple consumers in a consumer group, the work will be redistributed to one of the other members).
- Systems can standardize on the communication protocol (a high-performance binary TCP protocol is used when talking to Kafka clusters), as well as scaling strategies and fault-tolerance mechanisms (which are driven by consumer groups). This allows us to write software that is broadly consistent, and which fits in our head.
- Consumers can process data at a rate they can handle. Unprocessed data is stored in Kafka, in a durable and fault-tolerant manner, until the consumer is ready to process it. In other words, if the stream your consumer is reading from suddenly turns into a firehose, the Kafka cluster will act as a buffer, preventing your consumers from being overwhelmed.
- A stronger notion of what data is being communicated, in the form of *events*. An event is a piece of data with a certain structure, which we will discuss in "Events". The main point, for now, is that we can focus on the data flowing through our streams, instead of spending so much time disentangling the communication layer like we would in the client-server model.
- Systems can rebuild their state anytime by replaying the events in a topic.

One important difference between the pub/sub model and the client-server model is that communication is not bidirectional in Kafka's pub/sub model. In other words, streams flow one way. If a system produces some data to a Kafka topic, and relies on another system to do something with the data (i.e., enrich or transform it), the enriched data will need to be written to another topic and subsequently consumed by the original process. This is simple to coordinate, but it changes the way we think about communication.

As long as you remember the communication channels (topics) are stream-like in nature (i.e., flowing unidirectionally, and may have multiple sources and multiple downstream consumers), it's easy to design systems that simply listen to whatever stream of flowing bytes they are interested in, and produce data to topics (named streams) whenever they want to share data with one or more systems. We will be working a lot with Kafka topics in the following chapters (each Kafka Streams and ksqlDB application we build will read, and usually write to, one or more Kafka topics), so by the time you reach the end of this book, this will be second nature for you.

Now that we've seen how Kafka's communication model simplifies the way systems communicate with each other, and that named streams called *topics* act as the communication medium between systems, let's gain a deeper understanding of how streams come into play in Kafka's storage layer.

How Are Streams Stored?

When a team of LinkedIn engineers[2] saw the potential in a stream-driven data platform, they had to answer an important question: how should unbounded and continuous data streams be modeled at the storage layer?

Ultimately, the storage abstraction they identified was already present in many types of data systems, including traditional databases, key-value stores, version control systems, and more. The abstraction is the simple, yet powerful *commit log* (or simply, *log*).

Note

When we talk about *logs* in this book, we're not referring to *application logs*, which emit information about a running process (e.g., HTTP server logs). Instead, we are referring to a specific data structure that is described in the following paragraphs.

Logs are *append-only* data structures that capture an *ordered sequence* of events. Let's examine the italicized attributes in more detail, and build some intuition around logs, by creating a simple log from the command line. For example, let's create a log called `user_purchases`, and populate it with some dummy data using the following command:

```
# create the logfile
touch users.log

# generate four dummy records in our log
echo "timestamp=1597373669,user_id=1,purchases=1" >> users.log
echo "timestamp=1597373669,user_id=2,purchases=1" >> users.log
echo "timestamp=1597373669,user_id=3,purchases=1" >> users.log
echo "timestamp=1597373669,user_id=4,purchases=1" >> users.log
```

Now if we look at the log we created, it contains four users that have made a single purchase:

```
# print the contents of the log
cat users.log

# output
timestamp=1597373669,user_id=1,purchases=1
timestamp=1597373669,user_id=2,purchases=1
timestamp=1597373669,user_id=3,purchases=1
```

```
timestamp=1597373669,user_id=4,purchases=1
```

The first attribute of logs is that they are written to in an append-only manner. This means that if `user_id=1` comes along and makes a second purchase, we do not update the first record, since each record is *immutable* in a log. Instead, we just append the new record to the end:

```
# append a new record to the log
echo "timestamp=1597374265,user_id=1,purchases=2" >> users.log

# print the contents of the log
cat users.log

# output
timestamp=1597373669,user_id=1,purchases=1 ❶
timestamp=1597373669,user_id=2,purchases=1
timestamp=1597373669,user_id=3,purchases=1
timestamp=1597373669,user_id=4,purchases=1
timestamp=1597374265,user_id=1,purchases=2 ❷
```

❶ Once a record is written to the log, it is considered immutable. Therefore, if we need to perform an update (e.g., to change the purchase count for a user), then the original record is left untouched.

❷ In order to model the update, we simply append the new value to the end of the log. The log will contain both the old record and the new record, both of which are immutable.

Any system that wants to examine the purchase counts for each user can simply read each record in the log, in order, and the last record they will see for `user_id=1` will contain the updated purchase amount. This brings us to the second attribute of logs: they are ordered.

The preceding log happens to be in timestamp order (see the first column), but that's not what we mean by *ordered*. In fact, Kafka does store a timestamp for each record in the log, but the records do not have to be in timestamp order. When we say a log is ordered, what we mean is that a record's *position* in the log is fixed, and never changes. If we reprint the log again, this time with line numbers, you can see the position in the first column:

```
# print the contents of the log, with line numbers
cat -n users.log
```

```
# output
1       timestamp=1597373669,user_id=1,purchases=1
2       timestamp=1597373669,user_id=2,purchases=1
3       timestamp=1597373669,user_id=3,purchases=1
4       timestamp=1597373669,user_id=4,purchases=1
5       timestamp=1597374265,user_id=1,purchases=2
```

Now, imagine a scenario where ordering couldn't be guaranteed. Multiple processes could read the `user_id=1` updates in a different order, creating disagreement about the actual purchase count for this user. By ensuring the logs are ordered, the data can be processed deterministically[3] by multiple processes.[4]

Furthermore, while the position of each log entry in the preceding example uses line numbers, Kafka refers to the position of each entry in its distributed log as an *offset*. Offsets start at 0 and they enable an important behavior: they allow multiple consumer groups to each read from the same log, and maintain their own positions in the log/stream they are reading from. This is shown in Figure 1-5.

Now that we've gained some intuition around Kafka's log-based storage layer by creating our own log from the command line, let's tie these ideas back to the higher-level constructs we identified in Kafka's communication model. We'll start by continuing our discussion of topics, and learning about something called *partitions*.

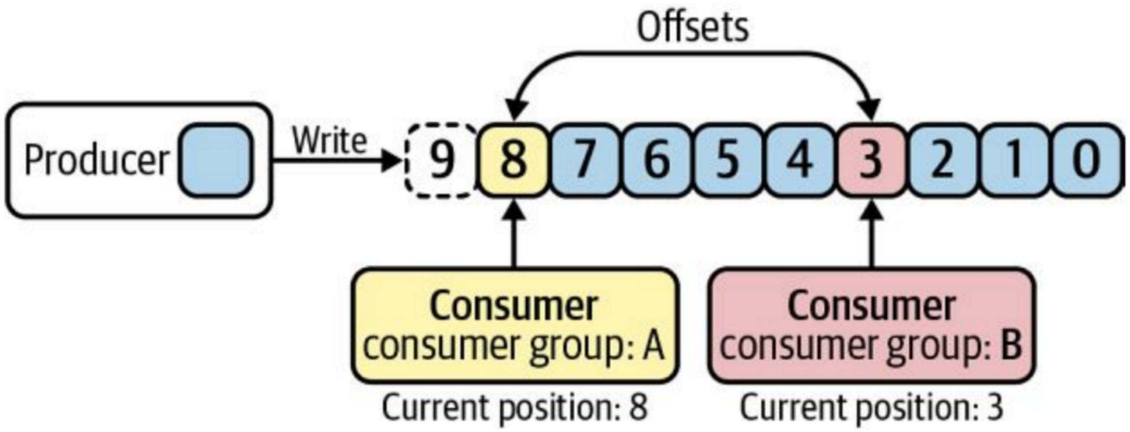


Figure 1-5. Multiple consumer groups can read from the same log, each maintaining their position based on the offset they have read/processed

Topics and Partitions

In our discussion of Kafka's communication model, we learned that Kafka has the concept of named streams called topics. Furthermore, Kafka topics are extremely flexible with what you store in them. For example, you can have *homogeneous topics* that contain only one type of data, or *heterogeneous topics* that contain multiple types of data.[5] A depiction of these different strategies is shown in Figure 1-6.

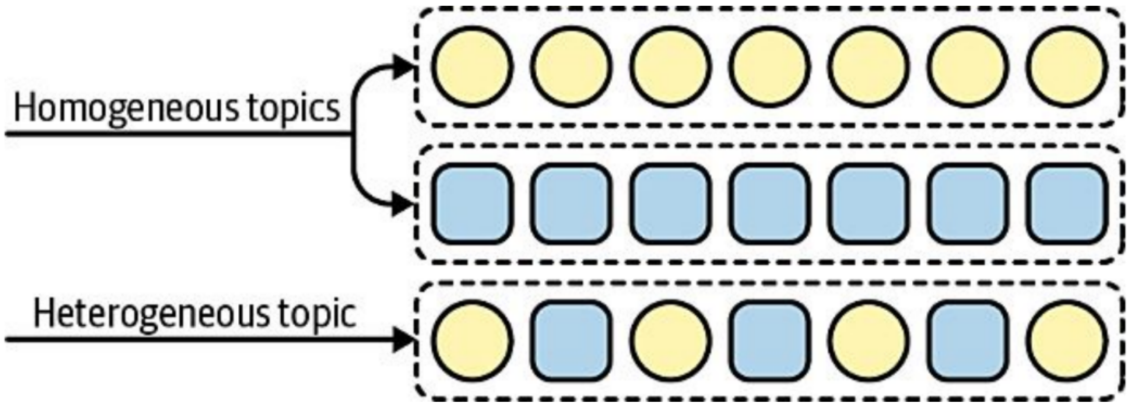


Figure 1-6. Different strategies exist for storing events in topics; homogeneous topics generally contain one event type (e.g., `clicks`*) while heterogeneous topics contain multiple event types (e.g.,* `clicks` *and* `page_views`*)*

We have also learned that append-only commit logs are used to model streams in Kafka's storage layer. So, does this mean that each topic correlates with a log file? Not exactly. You see, Kafka is a distributed log, and it's hard to distribute just one of something. So if we want to achieve some level of parallelism with the way we distribute and process logs, we need to create lots of them. This is why Kafka topics are broken into smaller units called *partitions*.

Partitions are individual logs (i.e., the data structures we discussed in the previous section) where data is produced and consumed from. Since the commit log abstraction is implemented at the partition level, this is the level at which ordering is guaranteed, with each partition having its own set of offsets. Global ordering is not supported at the topic level, which is why producers often route related records to the same partition.[6]

Ideally, data will be distributed relatively evenly across all partitions in a topic. But you could also end up with partitions of different sizes. Figure 1-7 shows an example of a topic with three different partitions.

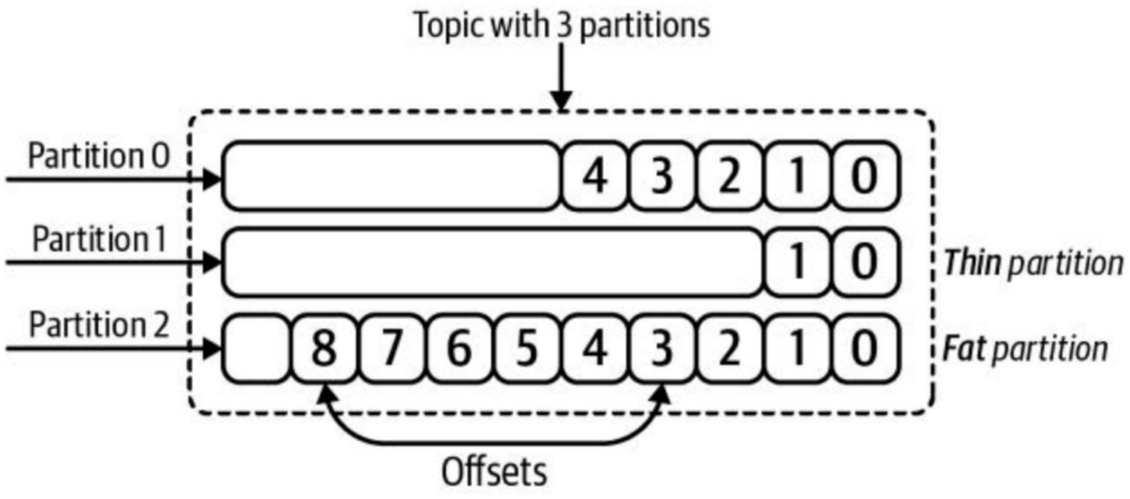


Figure 1-7. A Kafka topic configured with three partitions

The number of partitions for a given topic is configurable, and having more partitions in a topic generally translates to more parallelism and throughput, though there are some trade-offs of having too many partitions.[7] We'll talk about this more throughout the book, but the important takeaway is that only one consumer per consumer group can consume from a partition (individual members across different consumer groups can consume from the same partition, however, as shown in Figure 1-5).

Therefore, if you want to spread the processing load across *N* consumers in a single consumer group, you need *N* partitions. If you have fewer members in a consumer group than there are partitions on the source topic (i.e., the topic that is being read from), that's OK: each consumer can process multiple partitions. If you have more members in a consumer group than there are partitions on the source topic, then some consumers will be idle.

With this in mind, we can improve our definition of what a topic is. A topic is a named stream, composed of multiple partitions. And each partition is modeled as a commit log that stores data in a totally ordered and append-only sequence. So what exactly is stored in a topic partition? We'll explore this in the next section.

Events

In this book, we spend a lot of time talking about processing data in topics. However, we still haven't developed a full understanding of what kind of data is stored in a Kafka topic (and, more specifically, in a topic's partitions).

A lot of the existing literature on Kafka, including the official documentation, uses a variety of terms to describe the data in a topic, including messages, records, and events. These terms are often used interchangeably, but the one we have favored in this book (though we still use the other terms occasionally) is *event*. An event is a timestamped key-value pair that records *something that happened*. The basic anatomy of each event captured in a topic partition is shown in Figure 1-8.

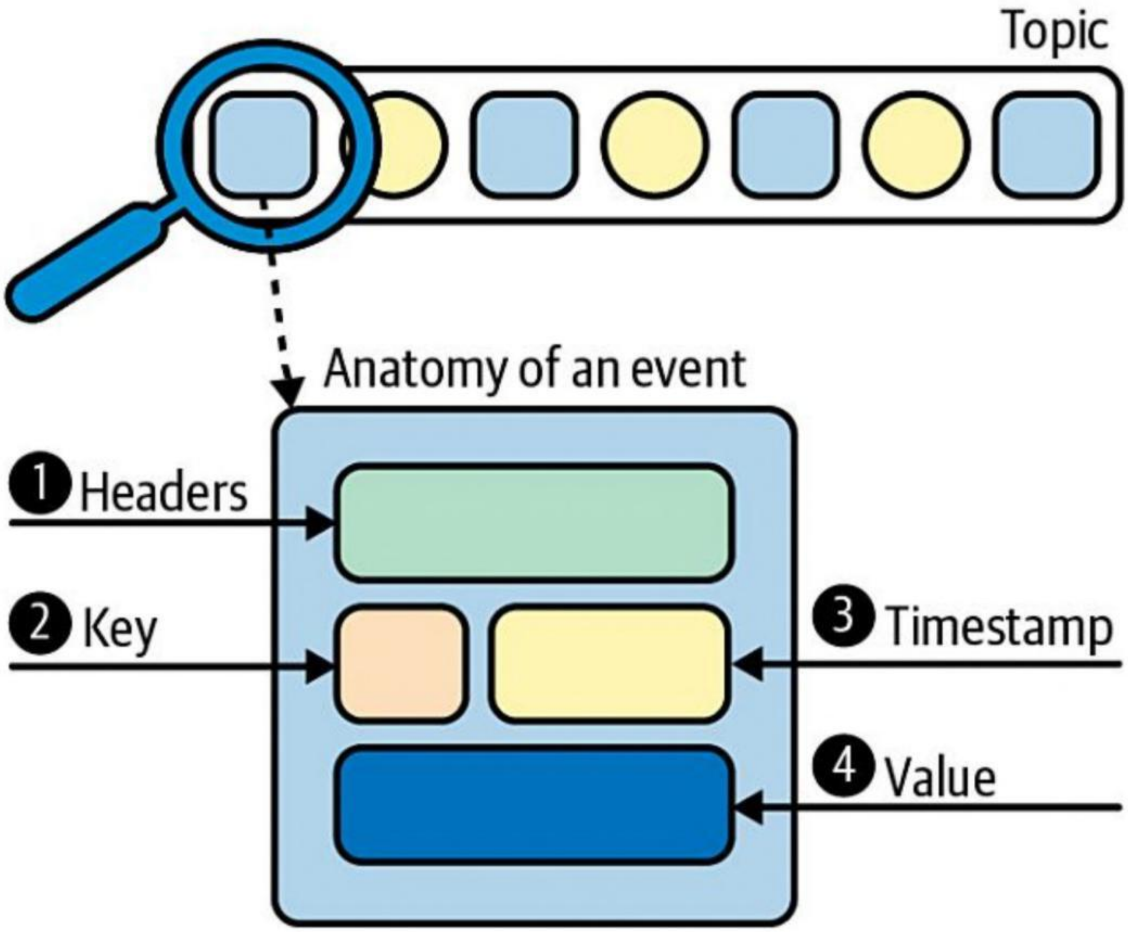


Figure 1-8. Anatomy of an event, which is what is stored in topic partitions

❶ Application-level headers contain optional metadata about an event. We don't work with these very often in this book.

❷ Keys are also optional, but play an important role in how data is distributed across partitions. We will see this over the next few chapters, but generally speaking, they are used to identify related records.

❸ Each event is associated with a timestamp. We'll learn more about timestamps in Chapter 5.

❹ The value contains the actual message contents, encoded as a byte array. It’s up to clients to deserialize the raw bytes into a more meaningful structure (e.g., a JSON object or Avro record). We will talk about byte array deserialization in detail in “Serialization/Deserialization”.

Now that we have a good understanding of what data is stored in a topic, let’s get a deeper look at Kafka’s clustered deployment model. This will provide more information about *how* data is physically stored in Kafka.

Kafka Cluster and Brokers

Having a centralized communication point means reliability and fault tolerance are extremely important. It also means that the communication backbone needs to be scalable, i.e., able to handle increased amounts of load. This is why Kafka operates as a cluster, and multiple machines, called *brokers*, are involved in the storage and retrieval of data.

Kafka clusters can be quite large, and can even span multiple data centers and geographic regions. However, in this book, we will usually work with a single-node Kafka cluster since that is all we need to start working with Kafka Streams and ksqlDB. In production, you’ll likely want at least three brokers, and you will want to set the replication of your Kafka topic so that your data is replicated across multiple brokers (we’ll see this later in this chapter’s tutorial). This allows us to achieve high availability and to avoid data loss in case one machine goes down.

Now, when we talk about data being stored and replicated across brokers, we’re really talking about individual partitions in a topic. For example, a topic may have three partitions that are spread across three brokers, as shown in Figure 1-9.

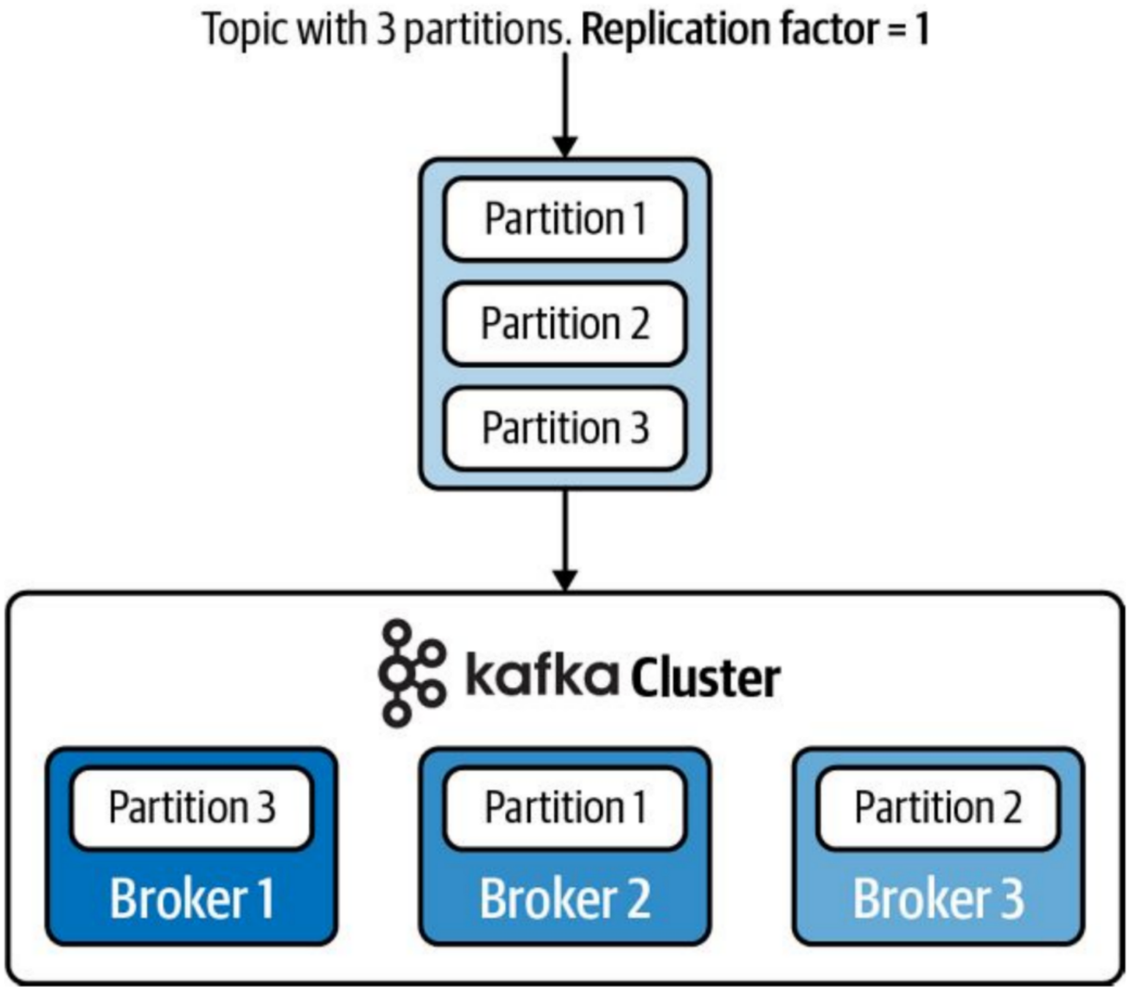


Figure 1-9. Partitions are spread across the available brokers, meaning that a topic can span multiple machines in the Kafka cluster

As you can see, this allows topics to be quite large, since they can grow beyond the capacity of a single machine. To achieve fault tolerance and high availability, you can set a replication factor when configuring the topic. For example, a replication factor of 2 will allow the partition to be stored on two different brokers. This is shown in Figure 1-10.

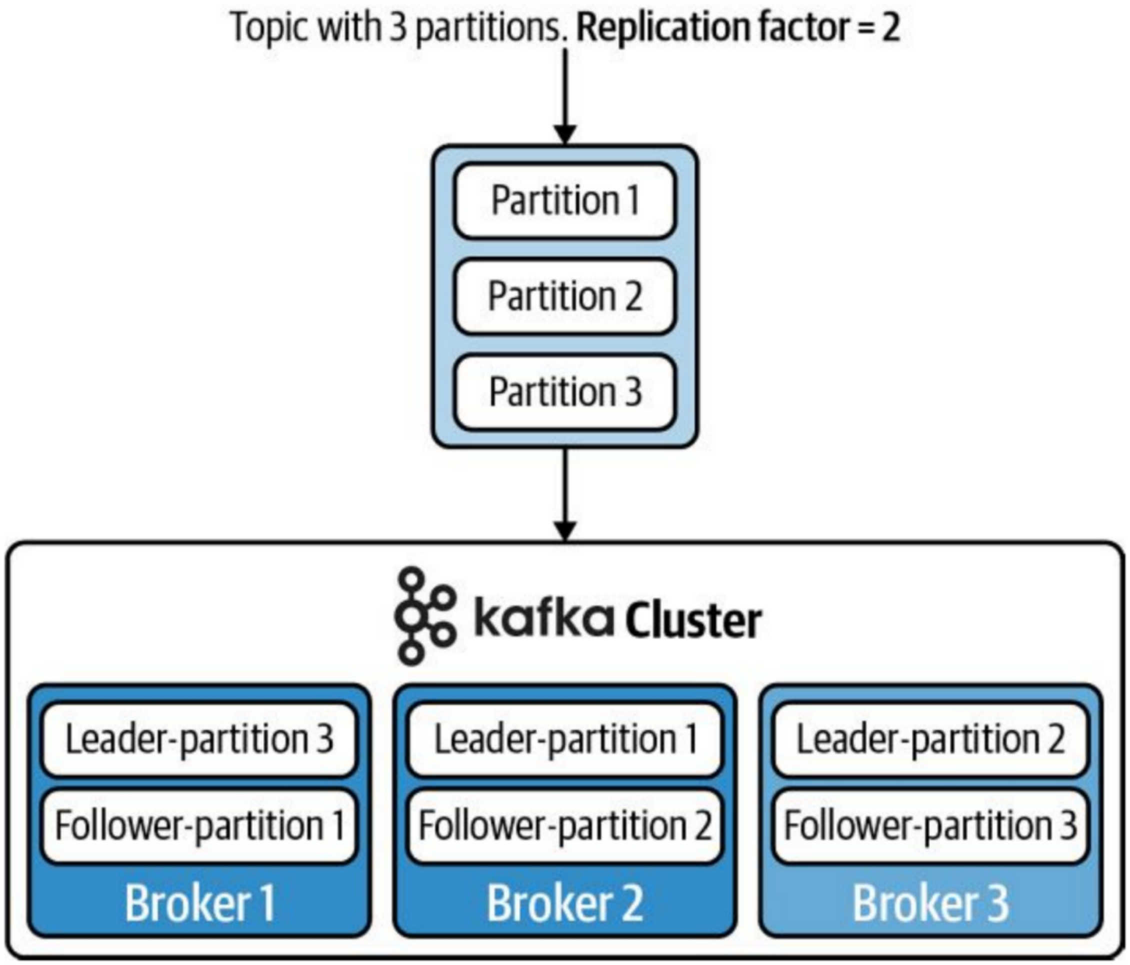


Figure 1-10. Increasing the replication factor to 2 will cause the partitions to be stored on two different brokers

Whenever a partition is replicated across multiple brokers, one broker will be designated as the *leader*, which means it will process all read/write requests from producers/consumers for the given partition. The other brokers that contain the replicated partitions are called *followers*, and they simply copy the data from the leader. If the leader fails, then one of the followers will be promoted as the new leader.

Furthermore, as the load on your cluster increases over time, you can expand your cluster by adding even more brokers, and triggering a partition reassignment. This will allow you to migrate data from the old machines to a fresh, new machine.

Finally, brokers also play an important role with maintaining the membership of consumer groups. We'll explore this in the next section.

Consumer Groups

Kafka is optimized for high throughput and low latency. To take advantage of this on the consumer side, we need to be able to parallelize work across multiple processes. This is accomplished with consumer groups.

Consumer groups are made up of multiple cooperating consumers, and the membership of these groups can change over time. For example, new consumers can come online to scale the processing load, and consumers can also go offline either for planned maintenance or due to unexpected failure. Therefore, Kafka needs some way of maintaining the membership of each group, and redistributing work when necessary.

To facilitate this, every consumer group is assigned to a special broker called the *group coordinator*, which is responsible for receiving heartbeats from the consumers, and triggering a *rebalance* of work whenever a consumer is marked as dead. A depiction of consumers heartbeating back to a group coordinator is shown in Figure 1-11.

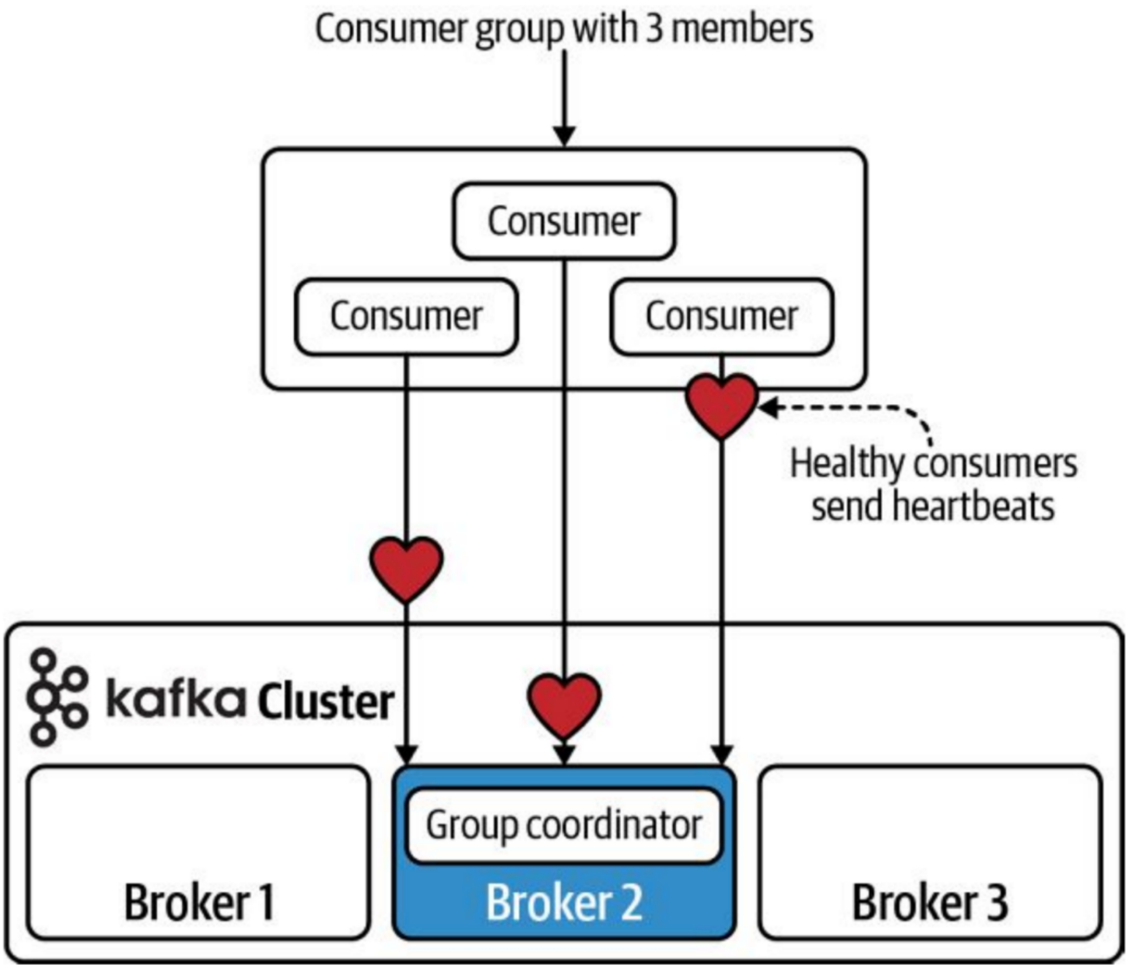


Figure 1-11. Three consumers in a group, heartbeating back to group coordinator

Every active member of the consumer group is eligible to receive a partition assignment. For example, the work distribution across three healthy consumers may look like the diagram in Figure 1-12.

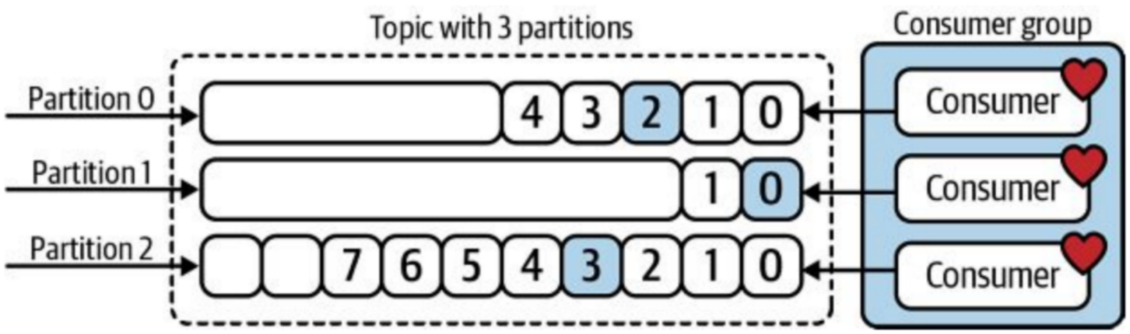


Figure 1-12. Three healthy consumers splitting the read/processing workload of a three-partition Kafka topic

However, if a consumer instance becomes unhealthy and cannot heartbeat back to the cluster, then work will automatically be reassigned to the healthy consumers. For example, in Figure 1-13, the middle consumer has been assigned the partition that was previously being handled by the unhealthy consumer.

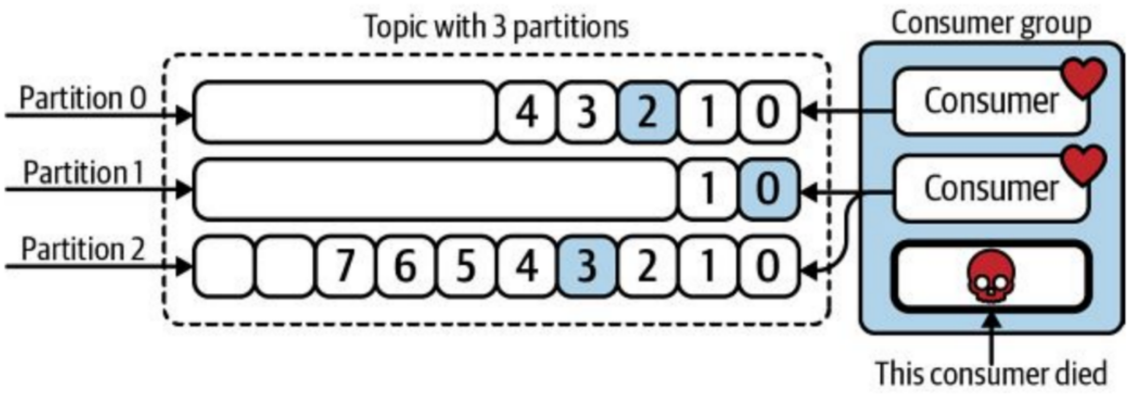


Figure 1-13. Work is redistributed when consumer processes fail

As you can see, consumer groups are extremely important in achieving high availability and fault tolerance at the data processing layer. With this, let's now commence our tutorial by learning how to install Kafka.

Installing Kafka

There are detailed instructions for installing Kafka manually in the official documentation. However, to keep things as simple as possible, most of the tutorials in this book utilize Docker, which allows us to deploy Kafka and our stream processing applications inside a containerized environment.

Therefore, we will be installing Kafka using Docker Compose, and we'll be using Docker images that are published by Confluent.[8] The first step is to download and install Docker from the Docker install page.

Next, save the following configuration to a file called *docker-compose.yml*:

```yaml
---
version: '2'

services:
  zookeeper: ❶
    image: confluentinc/cp-zookeeper:6.0.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  kafka: ❷
    image: confluentinc/cp-enterprise-kafka:6.0.0
    hostname: kafka
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "29092:29092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: |
         PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: |
         PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
```

❶ The first container, which we've named `zookeeper`, will contain the ZooKeeper installation. We haven't talked about ZooKeeper in this introduction since, at the time of this writing, it is being actively removed from Kafka. However, it is a centralized service for storing metadata such as topic configuration. Soon, it will no longer be included in Kafka, but we are including it here since this book was published before ZooKeeper was fully removed.

❷ The second container, called `kafka`, will contain the Kafka installation. This is where our broker (which comprises our single-node cluster) will run and where we will execute some of Kafka's console scripts for interacting with the cluster.

Finally, run the following command to start a local Kafka cluster:

```
docker-compose up
```

With our Kafka cluster running, we are now ready to proceed with our tutorial.

Hello, Kafka

In this simple tutorial, we will demonstrate how to create a Kafka *topic*, write data to a topic using a *producer*, and finally, read data from a topic using a *consumer*. The first thing we need to do is log in to the container that has Kafka installed. We can do this by running the following command:

```
docker-compose exec kafka bash
```

Now, let's create a topic, called `users`. We'll use one of the console scripts (`kafka-topics`) that is included with Kafka. The following command shows how to do this:

```
kafka-topics \ ❶
    --bootstrap-server localhost:9092 \ ❷
    --create \ ❸
    --topic users \ ❹
    --partitions 4 \ ❺
    --replication-factor 1 ❻

# output
Created topic users.
```

❶ `kafka-topics` is a console script that is included with Kafka.

❷ A bootstrap server is the host/IP pair for one or more brokers.

❸ There are many flags for interacting with Kafka topics, including `--list`, `--describe`, and `--delete`. Here, we use the `--create` flag since we are creating a new topic.

❹ The topic name is `users`.

❺ Split our topic into four partitions.

❻ Since we're running a single-node cluster, we will set the replication factor to 1. In production, you will want to set this to a higher value (such as 3) to ensure high-availability.

Note

The console scripts we use in this section are included in the Kafka source distribution. In a vanilla Kafka installation, these scripts include the *.sh* file extension (e.g., *kafka-topics.sh*, *kafka-console-producer.sh*, etc.). However, the file extension is dropped in Confluent Platform (which is why we ran *kafka-topics* instead of *kafka-topics.sh* in the previous code snippet).

Once the topic has been created, you can print a description of the topic, including its configuration, using the following command:

```
kafka-topics \
    --bootstrap-server localhost:9092 \
    --describe \ ❶
    --topic users

# output
Topic: users    PartitionCount: 4       ReplicationFactor: 1    Configs:
        Topic: users    Partition: 0    Leader: 1       Replicas: 1     Isr: 1
        Topic: users    Partition: 1    Leader: 1       Replicas: 1     Isr: 1
        Topic: users    Partition: 2    Leader: 1       Replicas: 1     Isr: 1
        Topic: users    Partition: 3    Leader: 1       Replicas: 1     Isr: 1
```

❶ The `--describe` flag allows us to view configuration information for a given topic.

Now, let's produce some data using the built-in `kafka-console-producer` script:

```
kafka-console-producer \ ❶
    --bootstrap-server localhost:9092 \
    --property key.separator=, \ ❷
    --property parse.key=true \
    --topic users
```

❶ The `kafka-console-producer` script, which is included with Kafka, can be used to produce data to a topic. However, once we start working with Kafka Streams and ksqlDB, the producer processes will be embedded in the underlying Java library, so we won't need to use this script outside of testing and development purposes.

❷ We will be producing a set of key-value pairs to our `users` topic. This property states that our key and values will be separated using the `,` character.

The previous command will drop you in an interactive prompt. From here, we can input several key-value pairs to produce to the `users` topic. When you are finished, press Control-C on your keyboard to exit the prompt:

```
>1,mitch
>2,elyse
>3,isabelle
>4,sammy
```

After producing the data to our topic, we can use the `kafka-console-consumer` script to read the data. The following command shows how:

```
kafka-console-consumer \ ❶
    --bootstrap-server localhost:9092 \
    --topic users \
    --from-beginning ❷

# output
mitch
elyse
isabelle
sammy
```

❶ The `kafka-console-consumer` script is also included in the Kafka distribution. Similar to what we mentioned for the `kafka-console-producer` script, most of the tutorials in this book will leverage consumer processes that are built into Kafka Streams and ksqlDB, instead of using this standalone console script (which is useful for testing purposes).

❷ The `--from-beginning` flag indicates that we should start consuming from the beginning of the Kafka topic.

By default, the `kafka-console-consumer` will only print the message value. But as we learned earlier, events actually contain more information, including a key, a timestamp, and headers. Let's pass in some additional properties to the console consumer so that we can see the timestamp and key values as well:[9]

```
kafka-console-consumer \
    --bootstrap-server localhost:9092 \
    --topic users \
    --property print.timestamp=true \
    --property print.key=true \
    --property print.value=true \
    --from-beginning

# output
CreateTime:1598226962606        1       mitch
CreateTime:1598226964342        2       elyse
CreateTime:1598226966732        3       isabelle
CreateTime:1598226968731        4       sammy
```

That's it! You have now learned how to perform some very basic interactions with a Kafka cluster. The final step is to tear down our local cluster using the following command:

```
docker-compose down
```

Summary

Kafka's communication model makes it easy for multiple systems to communicate, and its fast, durable, and append-only storage layer makes it possible to work with fast-moving streams of data with ease. By using a clustered deployment, Kafka can achieve high availability and fault tolerance at the storage layer by replicating data across multiple machines, called brokers. Furthermore, the cluster's ability to receive heartbeats from consumer processes, and update the membership of consumer groups, allows for high availability, fault tolerance, and workload scalability at the stream processing and consumption layer. All of these features have made Kafka one of the most popular stream processing platforms in existence.

You now have enough background on Kafka to get started with Kafka Streams and ksqlDB. In the next section, we will begin our journey with Kafka Streams by seeing how it fits in the wider Kafka ecosystem, and by learning how we can use this library to work with data at the stream processing layer.

[1] We talk about the raw byte arrays that are stored in topics, as well as the process of deserializing the bytes into higher-level structures like JSON objects/Avro records, in Chapter 3.

[2] Jay Kreps, Neha Narkhede, and Jun Rao initially led the development of Kafka.

[3] Deterministic means the same inputs will produce the same outputs.

[4] This is why traditional databases use logs for replication. Logs are used to capture each write operation on the leader database, and process the same writes, *in order*, on a replica database in order to deterministically re-create the same dataset on another machine.

[5] Martin Kleppmann has an interesting article on this topic, which can be found at *https://oreil.ly/tDZMm*. He talks about the various trade-offs and the reasons why one might choose one strategy over another. Also, Robert Yokota's follow-up article goes into more depth about how to support multiple event types when using Confluent Schema Registry for schema management/evolution.

[6] The partitioning strategy is configurable, but a popular strategy, including the one that is implemented in Kafka Streams and ksqlDB, involves setting the partition based on the record key (which can be extracted from the payload of the record or set explicitly). We'll discuss this in more detail over the next few chapters.

[7] The trade-offs include longer recovery periods after certain failure scenarios, increased resource utilization (file descriptors, memory), and increased end-to-end latency.

[8] There are many Docker images to choose from for running Kafka. However, the Confluent images are a convenient choice since Confluent also provides Docker images for some of the other technologies we will use in this book, including ksqlDB and Confluent Schema Registry.

[9] As of version 2.7, you can also use the `--property print.headers=true` flag to print the message headers.

Part II. Kafka Streams

Chapter 2. Getting Started with Kafka Streams

Kafka Streams is a lightweight, yet powerful Java library for enriching, transforming, and processing real-time streams of data. In this chapter, you will be introduced to Kafka Streams at a high level. Think of it as a first date, where you will learn a little about Kafka Streams' background and get an initial glance at its features.

By the end of this date, er...I mean *chapter*, you will understand the following:

- Where Kafka Streams fits in the Kafka ecosystem
- Why Kafka Streams was built in the first place
- What kinds of features and operational characteristics are present in this library
- Who Kafka Streams is appropriate for
- How Kafka Streams compares to other stream processing solutions
- How to create and run a basic Kafka Streams application

So without further ado, let's get our metaphorical date started with a simple question for Kafka Streams: *where do you live* (...in the Kafka ecosystem)?

The Kafka Ecosystem

Kafka Streams lives among a group of technologies that are collectively referred to as the *Kafka ecosystem*. In Chapter 1, we learned that at the heart of Apache Kafka is a distributed, append-only log that we can produce messages to and read messages from. Furthermore, the core Kafka code base includes some important APIs for interacting with this log (which is separated into categories of messages called *topics*). Three APIs in the Kafka ecosystem, which are summarized in Table 2-1, are concerned with the *movement* of data to and from Kafka.

Table 2-1. APIs for moving data to and from Kafka

API	Topic interaction	Examples
Producer API	*Writing* messages to Kafka topics.	• Filebeat • rsyslog • Custom producers
Consumer API	*Reading* messages from Kafka topics.	• Logstash • kafkacat • Custom consumers
Connect API	*Connecting* external data stores, APIs, and filesystems to Kafka topics. Involves both *reading* from topics (sink connectors) and *writing* to topics (source connectors).	• JDBC source connector • Elasticsearch sink connector • Custom connectors

However, while moving data through Kafka is certainly important for creating data pipelines, some business problems require us to also *process* and *react* to data as it becomes available in Kafka. This is referred to as *stream processing*, and there are multiple ways of building stream processing applications with Kafka. Therefore, let's take a look at how stream processing applications were implemented before Kafka Streams was introduced, and how a dedicated stream processing library came to exist alongside the other APIs in the Kafka ecosystem.

Before Kafka Streams

Before Kafka Streams existed, there was a void in the Kafka ecosystem.[1] Not the kind of void you might encounter during your morning meditation that makes you feel refreshed and enlightened, but the kind of void that made building stream processing applications more difficult than it needed to be. I'm talking about the lack of library support for processing data in Kafka topics.

During these early days of the Kafka ecosystem, there were two main options for building Kafka-based stream processing applications:

- Use the Consumer and Producer APIs directly
- Use another stream processing framework (e.g., Apache Spark Streaming, Apache Flink)

With the Consumer and Producer APIs, you can read from and write to the event stream directly using a number of programming languages (Python, Java, Go, C/C++, Node.js, etc.) and perform any kind of data processing logic you'd like, as long as you're willing to write a lot of code from scratch. These APIs are very basic and lack many of the primitives that would qualify them as a stream processing API, including:

- Local and fault-tolerant state[2]
- A rich set of operators for transforming streams of data
- More advanced representations of streams[3]
- Sophisticated handling of time[4]

Therefore, if you want to do anything nontrivial, like aggregate records, join separate streams of data, group events into windowed time buckets, or run ad hoc queries against your stream, you will hit a wall of complexity pretty quickly. The Producer and Consumer APIs do not contain any abstractions to help you with these kinds of use cases, so you will be left to your own devices as soon as it's time to do something more advanced with your event stream.

The second option, which involves adopting a full-blown streaming platform like Apache Spark or Apache Flink, introduces a lot of unneeded complexity. We talk about the downsides of this approach in "Comparison to Other Systems", but the short version is that if we're optimizing for simplicity *and* power, then we need a solution that gives us the stream processing primitives without the overhead of a processing cluster. We also need better integration with Kafka, especially when it comes to working with intermediate representations of data outside of the source and sink topics.

Fortunately for us, the Kafka community recognized the need for a stream processing API in the Kafka ecosystem and decided to build it.[5]

Enter Kafka Streams

In 2016, the Kafka ecosystem was forever changed when the first version of Kafka Streams (also called the *Streams API*) was released. With its inception, the landscape of stream processing applications that relied so heavily on hand-rolled features gave way to more advanced applications, which leveraged community-developed patterns and abstractions for transforming and processing real-time event streams.

Unlike the Producer, Consumer, and Connect APIs, Kafka Streams is dedicated to helping you *process* real-time data streams, not just *move* data to and from Kafka.[6] It makes it easy to consume real-time streams of events as they move through our data pipeline, apply data transformation logic using a rich set of stream processing operators and primitives, and optionally write new representations of the data back to Kafka (i.e., if we want to make the transformed or enriched events available to downstream systems).

Figure 2-1 depicts the previously discussed APIs in the Kafka ecosystem, with Kafka Streams operating at the stream processing layer.

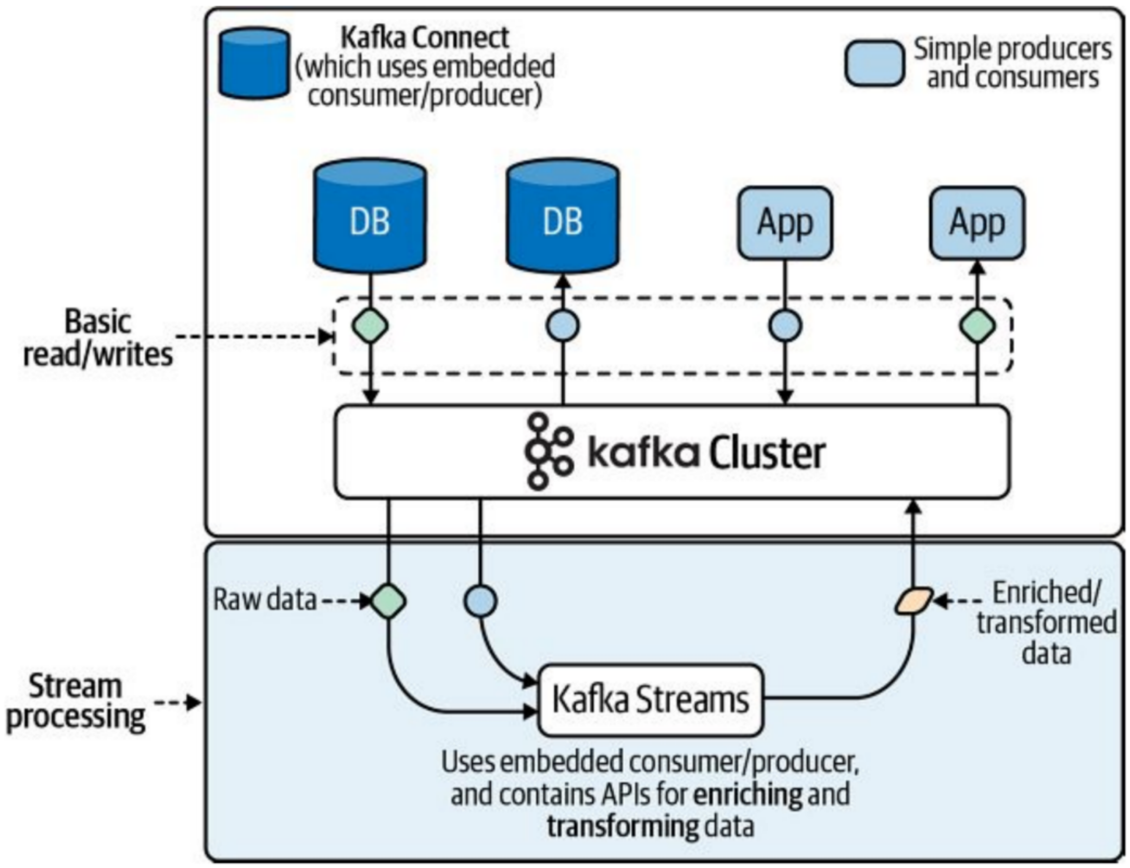

Kafka Connect (which uses embedded consumer/producer)
Simple producers and consumers
DB
DB
App
App
Basic read/writes
kafka Cluster
Raw data
Enriched/ transformed data
Stream processing
Kafka Streams
Uses embedded consumer/producer, and contains APIs for enriching and transforming data

Figure 2-1. Kafka Streams is the "brain" of the Kafka ecosystem, consuming records from the event stream, processing the data, and optionally writing enriched or transformed records back to Kafka

As you can see from the diagram in Figure 2-1, Kafka Streams operates at an exciting layer of the Kafka ecosystem: the place where data from many sources converges. This is the layer where sophisticated data *enrichment*, *transformation*, and *processing* can happen. It’s the same place where, in a pre–Kafka Streams world, we would have tediously written our own stream processing abstractions (using the Consumer/Producer API approach) or absorbed a complexity hit by using another framework. Now, let’s get a first look at the features of Kafka Streams that allow us to operate at this layer in a fun and efficient way.

Features at a Glance

Kafka Streams offers many features that make it an excellent choice for modern stream processing applications. We’ll be going over these in detail in the following chapters, but here are just a few of the features you can look forward to:

- A high-level DSL that looks and feels like Java’s streaming API. The DSL provides a fluent and functional approach to processing data streams that is easy to learn and use.
- A low-level Processor API that gives developers fine-grained control when they need it.
- Convenient abstractions for modeling data as either streams or tables.
- The ability to join streams and tables, which is useful for data transformation and enrichment.
- Operators and utilities for building both stateless and stateful stream processing applications.
- Support for time-based operations, including windowing and periodic functions.
- Easy installation. It’s just a library, so you can add Kafka Streams to any Java application.[7]
- Scalability, reliability, maintainability.

As you begin exploring these features in this book, you will quickly see why this library is so widely used and loved. Both the high-level DSL and low-level Processor API are not only easy to learn, but are also extremely powerful. Advanced stream processing tasks (such as joining live, moving streams of data) can be accomplished with very little code, which makes the development experience truly enjoyable.

Now, the last bullet point pertains to the long-term stability of our stream processing applications. After all, many technologies are exciting to learn in the beginning, but what really counts is whether or not Kafka Streams will continue to be a good choice as our relationship gets more complicated through real-world, long-term usage of this library. Therefore, it makes sense to evaluate the long-term viability of Kafka Streams before getting too far down the rabbit hole. So how should we go about doing this? Let's start by looking at Kafka Streams' operational characteristics.

Operational Characteristics

In Martin Kleppmann's excellent book, *Designing Data-Intensive Applications* (O'Reilly), the author highlights three important goals for data systems:

- Scalability
- Reliability
- Maintainability

These goals provide a useful framework for evaluating Kafka Streams, so in this section, we will define these terms and discover how Kafka Streams achieves each of them.

Scalability

Systems are considered *scalable* when they can cope and remain performant as load increases. In Chapter 1, we learned that scaling Kafka topics involves adding more partitions and, when needed, more Kafka brokers (the latter is only needed if the topic grows beyond the existing capacity of your Kafka cluster).

Similarly, in Kafka Streams, the unit of work is a single topic-partition, and Kafka automatically distributes work to groups of cooperating consumers called consumer groups.[8] This has two important implications:

- Since the unit of work in Kafka Streams is a single topic-partition, and since topics can be expanded by adding more partitions, the amount of work a Kafka Streams application can undertake can be scaled by increasing the number of partitions on the source topics.[9]
- By leveraging consumer groups, the total amount of work being handled by a Kafka Streams application can be distributed across multiple, cooperating instances of your application.

A quick note about the second point. When you deploy a Kafka Streams application, you will almost always deploy multiple application instances, each handling a subset of the work (e.g., if your source topic has 32 partitions, then you have 32 units of work that can be distributed across all cooperating consumers). For example, you could deploy four application instances, each handling eight partitions (`4 * 8 = 32`), or you could just as easily deploy sixteen instances of your application, each handling two partitions (`16 * 2 = 32`).

However, regardless of how many application instances you end up deploying, Kafka Streams' ability to cope with increased load by adding more partitions (units of work) and application instances (workers) makes Kafka Streams *scalable*.

On a similar note, Kafka Streams is also *elastic*, allowing you to seamlessly (albeit manually) scale the number of application instances in or out, with a limit on the scale-out path being the number of tasks that are created for your topology. We'll discuss tasks in more detail in "Tasks and Stream Threads".

Reliability

Reliability is an important feature of data systems, not only from an engineering perspective (we don't want to be woken up at 3 a.m. due to some fault in the system), but also from our customers' perspective (we don't want the system to go offline in any noticeable way, and we certainly can't tolerate data loss or corruption). Kafka Streams comes with a few fault-tolerant features,[10] but the most obvious one is something we've already touched on in "Consumer Groups": automatic failovers and partition rebalancing via consumer groups.

If you deploy multiple instances of your Kafka Streams application and one goes offline due to some fault in the system (e.g., a hardware failure), then Kafka will automatically redistribute the load to the other healthy instances. When the failure is resolved (or, in more modern architectures that leverage an orchestration system like Kubernetes, when the application is moved to a healthy node), then Kafka will rebalance the work again. This ability to gracefully handle faults makes Kafka Streams *reliable*.

Maintainability

> *It is well known that the majority of the cost of software is not in its initial development, but in its ongoing maintenance—fixing bugs, keeping its systems operational, investigating failures...*
>
> *Martin Kleppmann*

Since Kafka Streams is a Java library, troubleshooting and fixing bugs is relatively straightforward since we're working with standalone applications, and patterns for both troubleshooting and monitoring Java applications are well established and may already be in use at your organization (collecting and analyzing application logs, capturing application and JVM metrics, profiling and tracing, etc.).

Furthermore, since the Kafka Streams API is succinct and intuitive, code-level maintenance is less time-consuming than one would expect with more complicated libraries, and is very easy to understand for beginners and experts alike. If you build a Kafka Streams application and then don't touch it for months, you likely won't suffer the usual project amnesia and require a lot of ramp-up time to understand the previous code you've written. For the same reasons, new project maintainers can typically get up to speed pretty quickly with a Kafka Streams application, which improves maintainability even further.

Comparison to Other Systems

By this point, you should be starting to feel comfortable about starting a long-term relationship with Kafka Streams. But before things get too serious, isn't it natural to see if there are other fish in the sea?

Actually, it is sometimes difficult to evaluate how good a technology is without knowing how it stacks up against its competitors. So let's take a look at how Kafka Streams compares to some other popular technologies in the stream processing space.[11] We'll start by comparing Kafka Streams' deployment model with other popular systems.

Deployment Model

Kafka Streams takes a different approach to stream processing than technologies like Apache Flink and Apache Spark Streaming. The latter systems require you to set up a dedicated processing cluster for submitting and running your stream processing program. This can introduce a lot of complexity and overhead. Even experienced engineers at well-established companies have conceded that the overhead of a processing cluster is nontrivial. In an interview with Nitin Sharma at Netflix, I learned that it took around six months to adapt to the nuances of Apache Flink and build a highly reliable production Apache Flink application and cluster.

On the other hand, Kafka Streams is implemented as a Java *library*, so getting started is much easier since you don't need to worry about a cluster manager; you simply need to add the Kafka Streams dependency to your Java application. Being able to build a stream processing application as a standalone program also means you have a lot of freedom in terms of how you monitor, package, and deploy your code. For example, at Mailchimp, our Kafka Streams applications are deployed using the same patterns and tooling we use for other internal Java applications. This ability to immediately integrate into your company's systems is a huge advantage for Kafka Streams.

Next, let's explore how Kafka Streams' processing model compares to other systems in this space.

Processing Model

Another key differentiator between Kafka Streams and systems like Apache Spark Streaming or Trident is that Kafka Streams implements *event-at-a-time processing*, so events are processed immediately, one at a time, as they come in. This is considered true streaming and provides lower latency than the alternative approach, which is called *micro-batching*. Micro-batching involves grouping records into small groups (or *batches*), which are buffered in memory and later emitted at some interval (e.g., every 500 milliseconds). Figure 2-2 depicts the difference between event-at-a-time and micro-batch processing.

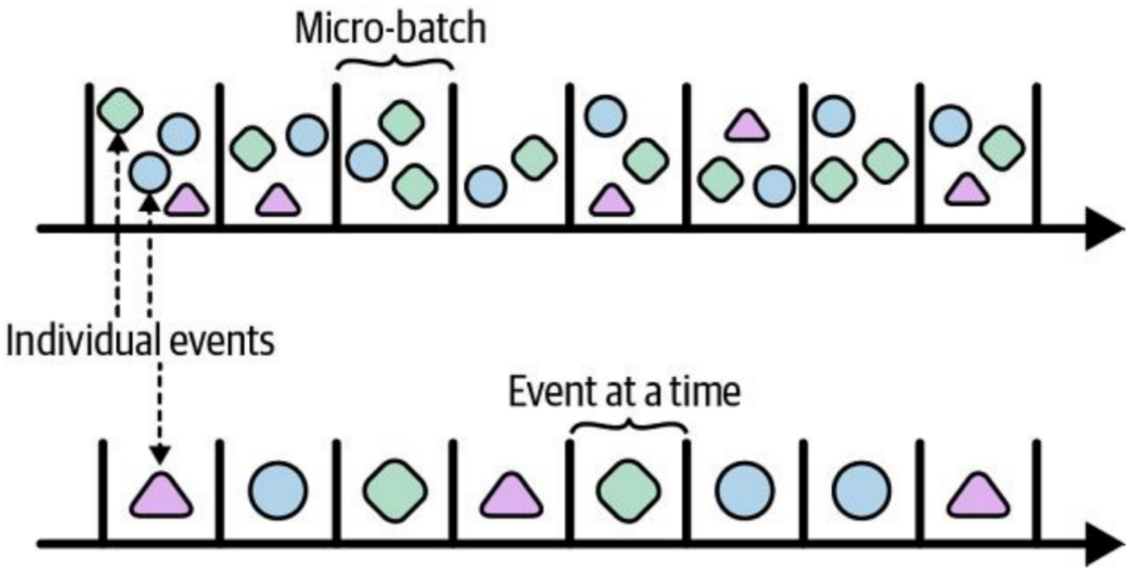


Figure 2-2. Micro-batching involves grouping records into small batches and emitting them to downstream processors at a fixed interval; event-at-a-time processing allows each event to be processed at soon as it comes in, instead of waiting for a batch to materialize

Note

Frameworks that use micro-batching are often optimized for greater *throughput* at the cost of higher *latency*. In Kafka Streams, you can achieve extremely low latency while also maintaining high throughput by splitting data across many partitions.

Finally, let's take a look at Kafka Streams' data processing architecture, and see how its focus on streaming differs from other systems.

Kappa Architecture

Another important consideration when comparing Kafka Streams to other solutions is whether or not your use case requires support for both batch and stream processing. At the time of this writing, Kafka Streams focuses solely on streaming use cases[12] (this is called a *Kappa architecture*), while frameworks like Apache Flink and Apache Spark support both batch and stream processing (this is called a *Lambda architecture*). However, architectures that support both batch and streaming use cases aren't without their drawbacks. Jay Kreps discussed some of the disadvantages of a hybrid system nearly two years before Kafka Streams was introduced into the Kafka ecosystem:

The operational burden of running and debugging two systems is going to be very high. And any new abstraction can only provide the features supported by the intersection of the two systems.

These challenges didn't stop projects like Apache Beam, which defines a unified programming model for batch and stream processing, from gaining popularity in recent years. But Apache Beam isn't comparable to Kafka Streams in the same way that Apache Flink is. Instead, Apache Beam is an API layer that relies on an execution engine to do most of the work. For example, both Apache Flink and Apache Spark can be used as execution engines (often referred to as *runners*) in Apache Beam. So when you compare Kafka Streams to Apache Beam, you must also consider the execution engine that you plan on using in addition to the Beam API itself.

Furthermore, Apache Beam–driven pipelines lack some important features that are offered in Kafka Streams. Robert Yokota, who created an experimental Kafka Streams Beam Runner and who maintains several innovative projects in the Kafka ecosystem,[13] puts it this way in his comparison of different streaming frameworks:

One way to state the differences between the two systems is as follows:

- *Kafka Streams is a stream-relational processing platform.*
- *Apache Beam is a stream-only processing platform.*

A stream-relational processing platform has the following capabilities which are typically missing in a stream-only processing platform:

- *Relations (or tables) are first-class citizens, i.e., each has an independent identity.*
- *Relations can be transformed into other relations.*
- *Relations can be queried in an ad-hoc manner.*

We will demonstrate each of the bulleted features over the next several chapters, but for now, suffice it to say that many of Kafka Streams' most powerful features (including the ability to query the state of a stream) are not available in Apache Beam or other more generalized frameworks.[14] Furthermore, the Kappa architecture offers a simpler and more specialized approach for working with streams of data, which can improve the development experience and simplify the operation and maintenance of your software. So if your use case doesn't require batch processing, then hybrid systems will introduce unnecessary complexity.

Now that we've looked at the competition, let's look at some Kafka Streams use cases.

Use Cases

Kafka Streams is optimized for processing unbounded datasets quickly and efficiently, and is therefore a great solution for problems in low-latency, time-critical domains. A few example use cases include:

- Financial data processing (Flipkart), purchase monitoring, fraud detection
- Algorithmic trading
- Stock market/crypto exchange monitoring
- Real-time inventory tracking and replenishment (Walmart)
- Event booking, seat selection (Ticketmaster)
- Email delivery tracking and monitoring (Mailchimp)
- Video game telemetry processing (Activision, the publisher of *Call of Duty*)
- Search indexing (Yelp)
- Geospatial tracking/calculations (e.g., distance comparison, arrival projections)
- Smart Home/IoT sensor processing (sometimes called AIOT, or the Artificial Intelligence of Things)
- Change data capture (Redhat)
- Sports broadcasting/real-time widgets (Gracenote)
- Real-time ad platforms (Pinterest)
- Predictive healthcare, vitals monitoring (Children's Healthcare of Atlanta)
- Chat infrastructure (Slack), chat bots, virtual assistants
- Machine learning pipelines (Twitter) and platforms (Kafka Graphs)

The list goes on and on, but the common characteristic across all of these examples is that they require (or at least benefit from) *real-time decision making* or data processing. The spectrum of these use cases, and others you will encounter in the wild, is really quite fascinating. On one end of the spectrum, you may be processing streams at the hobbyist level by analyzing sensor output from a Smart Home device. However, you could also use Kafka Streams in a healthcare setting to monitor and react to changes in a trauma victim's condition, as Children's Healthcare of Atlanta has done.

Kafka Streams is also a great choice for building microservices on top of real-time event streams. It not only simplifies typical stream processing operations (filtering, joining, windowing, and transforming data), but as you will see in "Interactive Queries", it is also capable of exposing the state of a stream using a feature called *interactive queries*. The state of a stream could be an aggregation of some kind (e.g., the total number of views for each video in a streaming platform) or even the latest representation for a rapidly changing entity in your event stream (e.g., the latest stock price for a given stock symbol).

Now that you have some idea of who is using Kafka Streams and what kinds of use cases it is well suited for, let's take a quick look at Kafka Streams' architecture before we start writing any code.

Processor Topologies

Kafka Streams leverages a programming paradigm called *dataflow programming* (DFP), which is a data-centric method of representing programs as a series of inputs, outputs, and processing stages. This leads to a very natural and intuitive way of creating stream processing programs and is one of the many reasons I think Kafka Streams is easy to pick up for beginners.

Note

This section will dive a little deeper into Kafka Streams architecture. If you prefer to get your feet wet with Kafka Streams and revisit this section later, feel free to proceed to "Introducing Our Tutorial: Hello, Streams".

Instead of building a program as a sequence of steps, the stream processing logic in a Kafka Streams application is structured as a directed acyclic graph (DAG). Figure 2-3 shows an example DAG that depicts how data flows through a set of stream processors. The nodes (the rectangles in the diagram) represent a processing step, or processor, and the edges (the lines connecting the nodes in the diagram) represent input and output streams (where data flows from one processor to another).

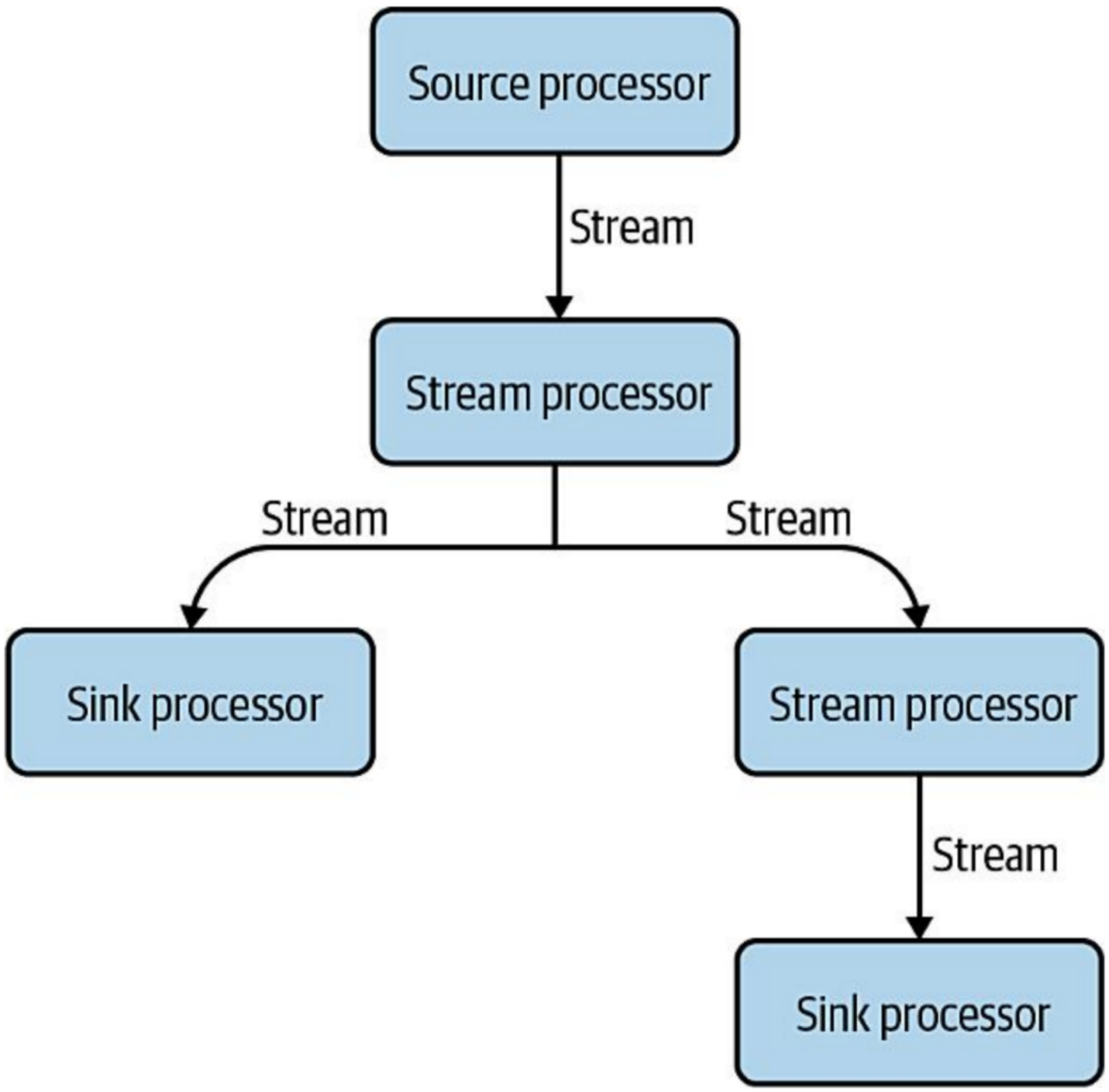


Figure 2-3. Kafka Streams borrows some of its design from dataflow programming, and structures stream processing programs as a graph of processors through which data flows

There are three basic kinds of processors in Kafka Streams:

Source processors

> Sources are where information flows into the Kafka Streams application. Data is read from a Kafka topic and sent to one or more *stream processors*.

Stream processors

> These processors are responsible for applying data processing/transformation logic on the input stream. In the high-level DSL, these processors are defined using a set

of built-in *operators* that are exposed by the Kafka Streams library, which we will be going over in detail in the following chapters. Some example operators are `filter`, `map`, `flatMap`, and `join`.

Sink processors

Sinks are where enriched, transformed, filtered, or otherwise processed records are written *back* to Kafka, either to be handled by another stream processing application or to be sent to a downstream data store via something like Kafka Connect. Like source processors, sink processors are connected to a Kafka topic.

A collection of processors forms a *processor topology*, which is often referred to as simply *the topology* in both this book and the wider Kafka Streams community. As we go through each tutorial in this part of the book, we will first design the topology by creating a DAG that connects the source, stream, and sink processors. Then, we will simply implement the topology by writing some Java code. To demonstrate this, let's go through the exercise of translating some project requirements into a processor topology (represented by a DAG). This will help you learn how to *think* like a Kafka Streams developer.

Scenario

Say we are building a chatbot with Kafka Streams, and we have a topic named `slack-mentions` that contains every Slack message that mentions our bot, `@StreamsBot`. We will design our bot so that it expects each mention to be followed by a command, like `@StreamsBot restart myservice`.

We want to implement a basic processor topology that does some preprocessing/validation of these Slack messages. First, we need to consume each message in the source topic, determine if the command is valid, and if so, write the Slack message to a topic called `valid-mentions`. If the command is not valid (e.g., someone makes a spelling error when mentioning our bot, such as `@StreamsBot restart serverrr`), then we will write to a topic named `invalid-mentions`).

In this case, we would translate these requirements to the topology shown in Figure 2-4.

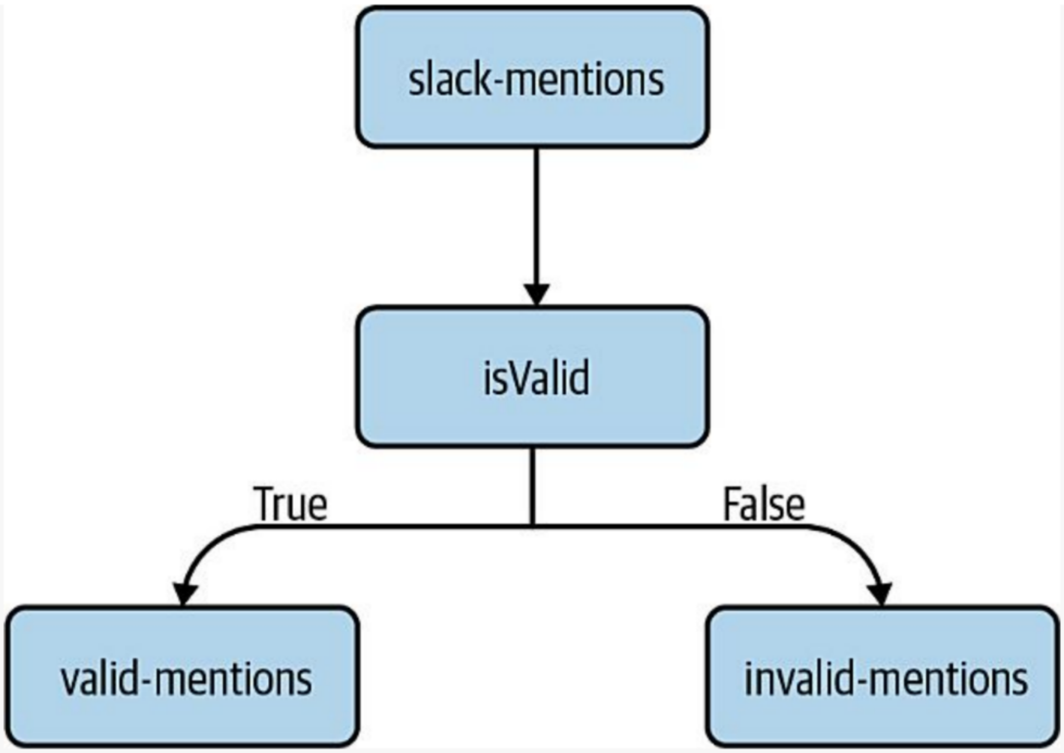


Figure 2-4. An example processor topology that contains a single source processor for reading Slack messages from Kafka (`slack-mentions`), a single stream processor that checks the validity of each message (`isValid`), and two sink processors that route the message to one of two output topics based on the previous check (`valid-mentions`, `invalid-mentions`)

Starting with the tutorial in the next chapter, we will then begin to implement any topology we design using the Kafka Streams API. But first, let's take a look at a related concept: sub-topologies.

Sub-Topologies

Kafka Streams also has the notion of sub-topologies. In the previous example, we designed a processor topology that consumes events from a single source topic (`slack-mentions`) and performs some preprocessing on a stream of raw chat messages. However, if our application needs to consume from multiple source topics, then Kafka Streams will (under most circumstances[15]) divide our topology into smaller sub-topologies to parallelize the work even further. This division of work is possible since operations on one input stream can be executed independently of operations on another input stream.

For example, let's keep building our chatbot by adding two new stream processors: one that consumes from the `valid-mentions` topic and performs whatever command was issued to StreamsBot (e.g., `restart server`), and another processor that consumes from the `invalid-mentions` topic and posts an error response back to Slack.[16]

As you can see in Figure 2-5, our topology now has three Kafka topics it reads from: `slack-mentions`, `valid-mentions`, and `invalid-mentions`. Each time we read from a new source topic, Kafka Streams divides the topology into smaller sections that it can execute independently. In this example, we end up with three sub-topologies for our chatbot application, each denoted by a star in the figure.

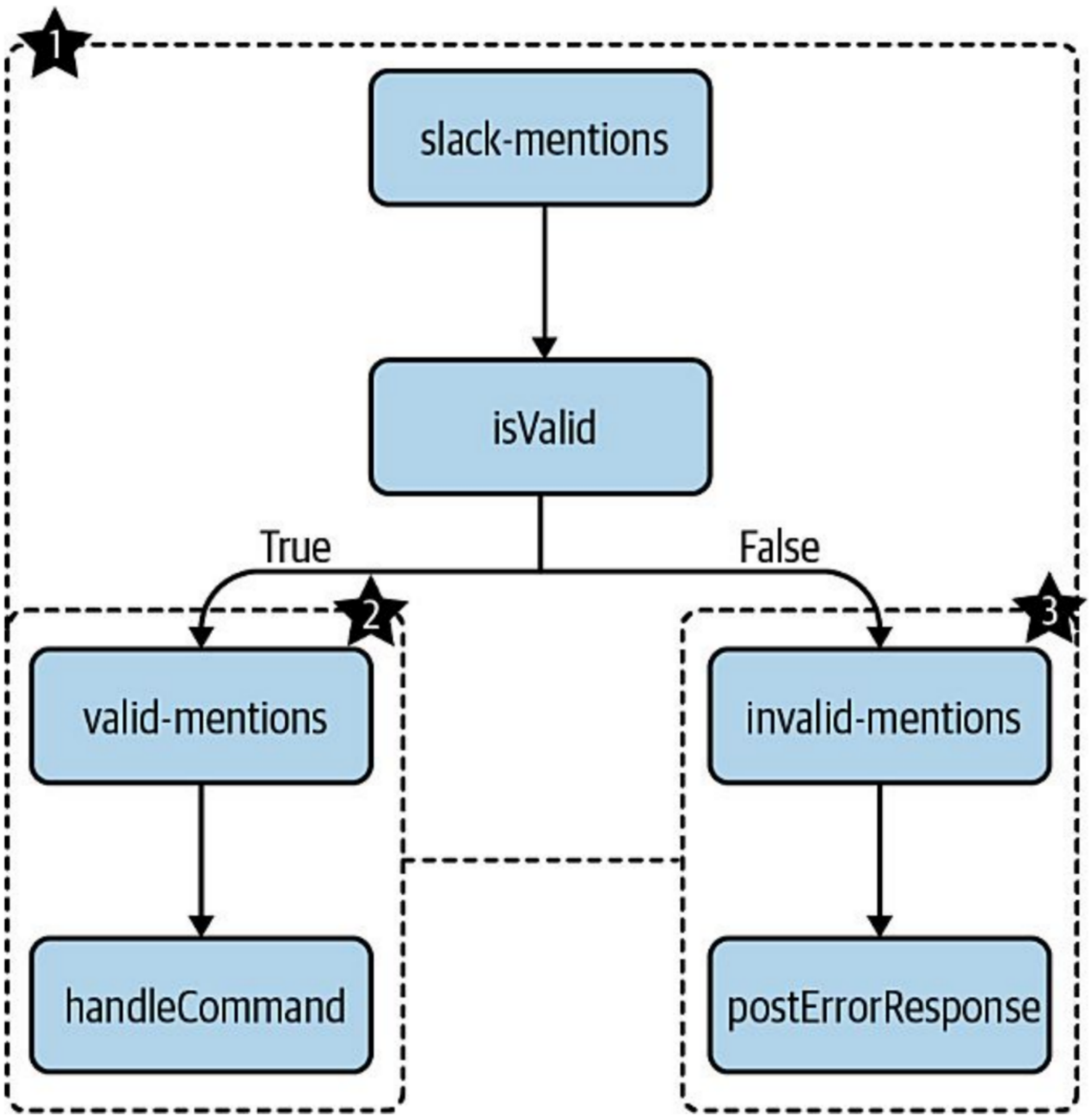


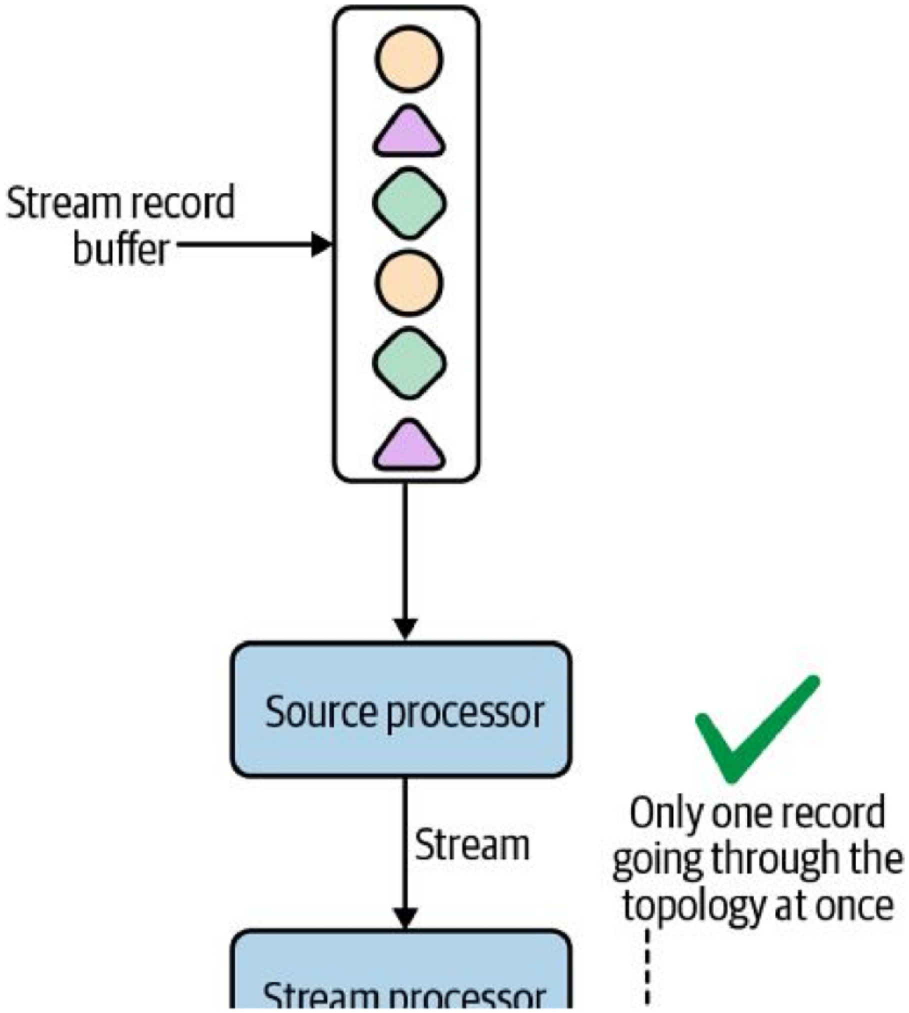
Figure 2-5. A processor topology, subdivided into sub-topologies (demarcated by dotted lines)

Notice that both the `valid-mentions` and `invalid-mentions` topics serve as a sink processor in the first sub-topology, but as a source processor in the second and third sub-topologies. When this occurs, there is no direct data exchange between sub-topologies. Records are produced to Kafka in the sink processor, and reread from Kafka by the source processors.

Now that we understand how to represent a stream processing program as a processor topology, let's take a look at how data actually flows through the interconnected processors in a Kafka Streams application.

Depth-First Processing

Kafka Streams uses a depth-first strategy when processing data. When a new record is received, it is routed through each stream processor in the topology before another record is processed. The flow of data through Kafka Streams looks something like what's shown in Figure 2-6.



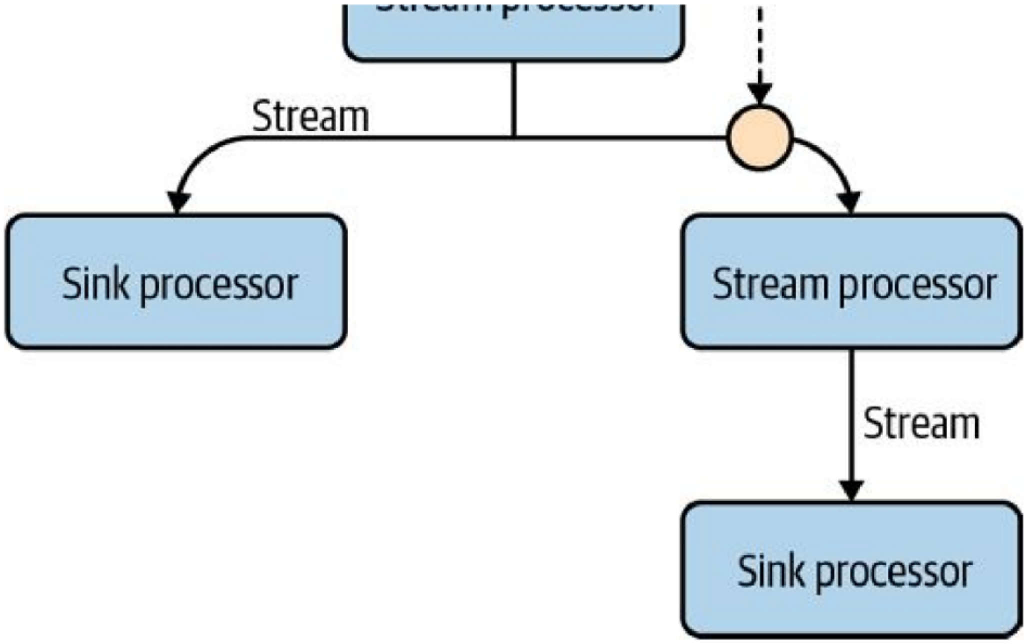


Figure 2-6. In depth-first processing, a single record moves through the entire topology before another record is processed

This depth-first strategy makes the dataflow much easier to reason about, but also means that slow stream processing operations can block other records from being processed in the same thread. Figure 2-7 demonstrates something you will never see happen in Kafka Streams: multiple records going through a topology at once.

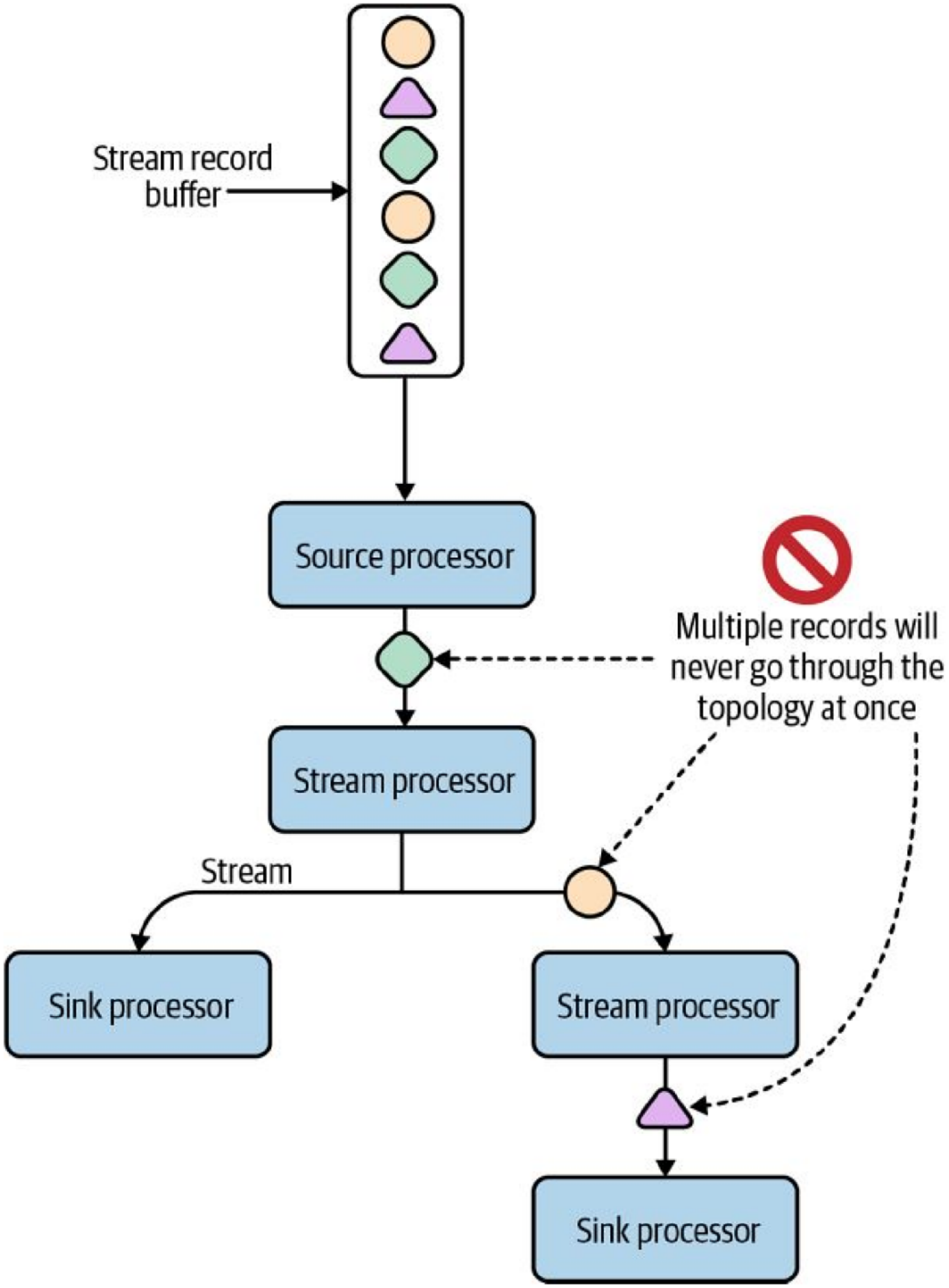


Figure 2-7. Multiple records will never go through the topology at the same time

Note

When multiple sub-topologies are in play, the single-event rule does not apply to the entire topology, but to each sub-topology.

Now that we know how to design processor topologies and how data flows through them, let's take a look at the advantages of this data-centric approach to building stream processing applications.

Benefits of Dataflow Programming

There are several advantages of using Kafka Streams and the dataflow programming model for building stream processing applications. First, representing the program as a directed graph makes it easy to reason about. You don't need to follow a bunch of conditionals and control logic to figure out how data is flowing through your application. Simply find the source and sink processors to determine where data enters and exits your program, and look at the stream processors in between to discover how the data is being processed, transformed, and enriched along the way.

Furthermore, expressing our stream processing program as a directed graph allows us to standardize the way we frame real-time data processing problems and, subsequently, the way we build our streaming solutions. A Kafka Streams application that I write will have some level of familiarity to anyone who has worked with Kafka Streams before in their own projects—not only due to the reusable abstractions available in the library itself, but also thanks to a common problem-solving approach: defining the flow of data using operators (nodes) and streams (edges). Again, this makes Kafka Streams applications easy to reason about and maintain.

Directed graphs are also an intuitive way of visualizing the flow of data for non-technical stakeholders. There is often a disconnect about how a program works between engineering teams and nonengineering teams. Sometimes, this leads to nontechnical teams treating the software as a closed box. This can have dangerous side effects, especially in the age of data privacy laws and GDPR compliance, which requires close coordination between engineers, legal teams, and other stakeholders. Thus, being able to simply communicate how data is being processed in your application allows people who are focused on another aspect of a business problem to understand or even contribute to the design of your application.

Finally, the processor topology, which contains the source, sink, and stream processors, acts as a *template* that can be instantiated and parallelized very easily across multiple threads and application instances. Therefore, defining the dataflow in this way allows us to realize performance and scalability benefits since we can easily replicate our stream processing program when data volume demands it.

Now, to understand how this process of replicating topologies works, we first need to understand the relationship between tasks, stream threads, and partitions.

Tasks and Stream Threads

When we define a topology in Kafka Streams, we are not actually executing the program. Instead, we are building a template for how data should flow through our application. This template (our topology) can be instantiated multiple times in a single application instance, and parallelized across many *tasks* and *stream threads* (which we'll refer to as simply threads going forward.[17]) There is a close relationship between the number of tasks/threads and the amount of work your stream processing application can handle, so understanding the content in this section is especially important for achieving good performance with Kafka Streams.

Let's start by looking at tasks:

> *A task is the smallest unit of work that can be performed in parallel in a Kafka Streams application...*
>
> *Slightly simplified, the maximum parallelism at which your application may run is bounded by the maximum number of stream tasks, which itself is determined by the maximum number of partitions of the input topic(s) the application is reading from.*
>
> *Andy Bryant*

Translating this quote into a formula, we can calculate the number of tasks that can be created for a given Kafka Streams sub-topology[18] with the following math:

```
max(source_topic_1_partitions, ... source_topic_n_partitions)
```

For example, if your topology reads from one source topic that contains 16 partitions, then Kafka Streams will create 16 tasks, each of which will instantiate its own copy of the underlying processor topology. Once Kafka Streams has created all of the tasks, it will assign the source partitions to be read from to each task.

As you can see, tasks are just logical units that are used to instantiate and run a processor topology. *Threads*, on the other hand, are what actually execute the task. In Kafka Streams, the stream threads are designed to be isolated and thread-safe.[19] Furthermore, unlike tasks, there isn't any formula that Kafka Streams applies to figure out how many threads your application should run. Instead, you are responsible for specifying the thread count using a configuration property named `num.stream.threads`. The upper bound for the number of threads you can utilize corresponds to the task count, and there are different strategies for deciding on the number of stream threads you should run with.[20]

Now, let's improve our understanding of these concepts by visualizing how tasks and threads are created using two separate configs, each specifying a different number of threads. In each example, our Kafka Streams application is reading from a source topic that contains four partitions (denoted by `p1` - `p4` in Figure 2-8).

First, let's configure our application to run with two threads (`num.stream.threads = 2`). Since our source topic has four partitions, four tasks will be created and distributed across each thread. We end up with the task/thread layout depicted in Figure 2-8.

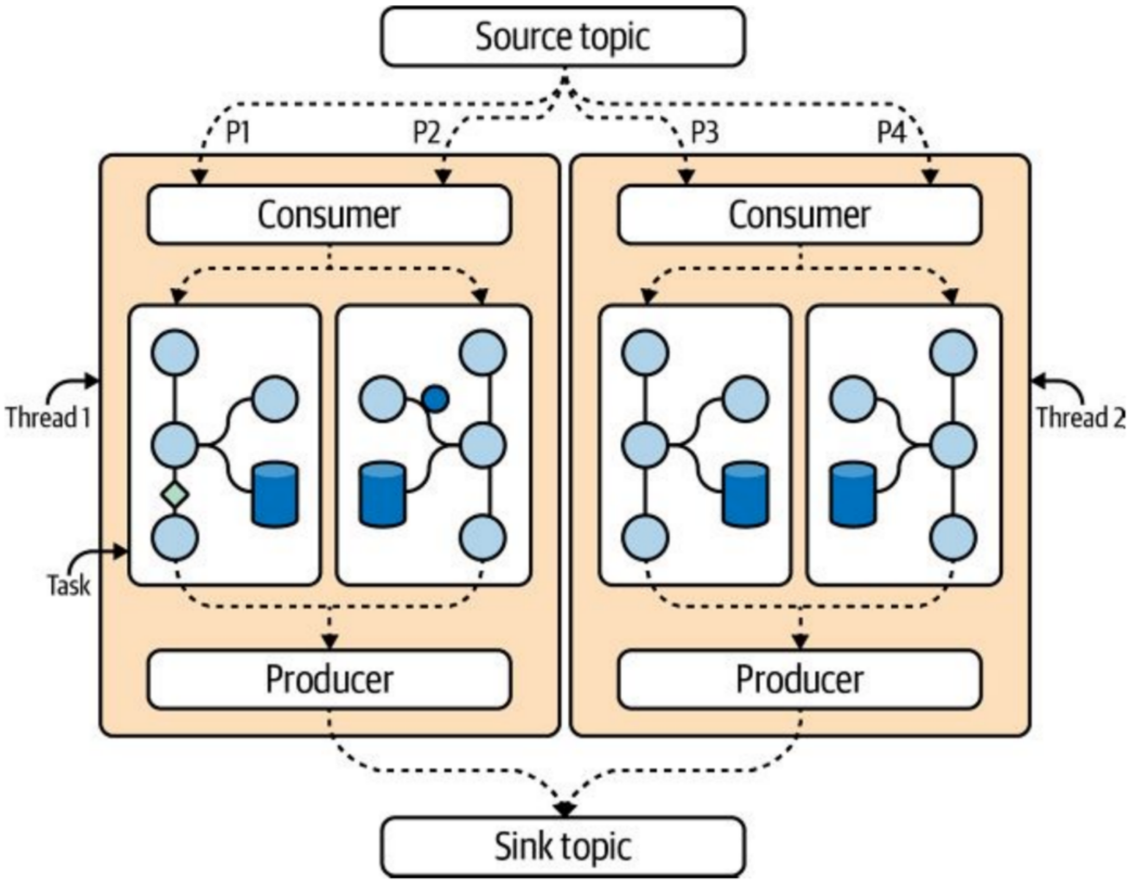


Figure 2-8. Four Kafka Streams tasks running in two threads

Running more than one task per thread is perfectly fine, but sometimes it is often desirable to run with a higher thread count to take full advantage of the available CPU resources. Increasing the number of threads doesn't change the number of tasks, but it does change the distribution of tasks among threads. For example, if we reconfigure the same Kafka Streams application to run with four threads instead of two (`num.stream.threads = 4`), we end up with the task/thread layout depicted in Figure 2-9.

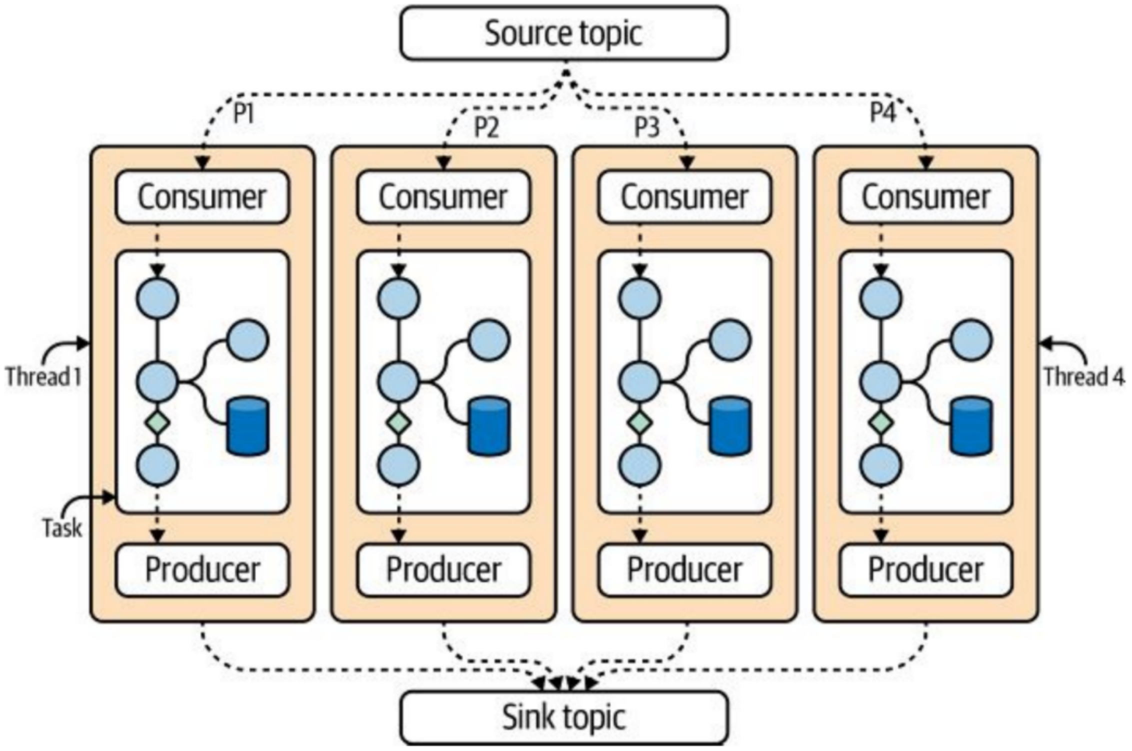


Figure 2-9. Four Kafka Streams tasks running in four threads

Now that we've learned about Kafka Streams' architecture, let's take a look at the APIs that Kafka Streams exposes for creating stream processing applications.

High-Level DSL Versus Low-Level Processor API

Different solutions present themselves at different layers of abstraction.

James Clear[21]

A common notion in the software engineering field is that abstraction usually comes at a cost: the more you abstract the details away, the more the software feels like "magic," and the more control you give up. As you get started with Kafka Streams, you may wonder what kind of control you will be giving up by choosing to implement a stream processing application using a high-level library instead of designing your solution using the lower-level Consumer/Producer APIs directly.

Luckily for us, Kafka Streams allows developers to choose the abstraction level that works best for them, depending on the project and also the experience and preference of the developer.

The two APIs you can choose from are:

- The high-level DSL
- The low-level Processor API

The relative abstraction level for both the high-level DSL and low-level Processor API is shown in Figure 2-10.

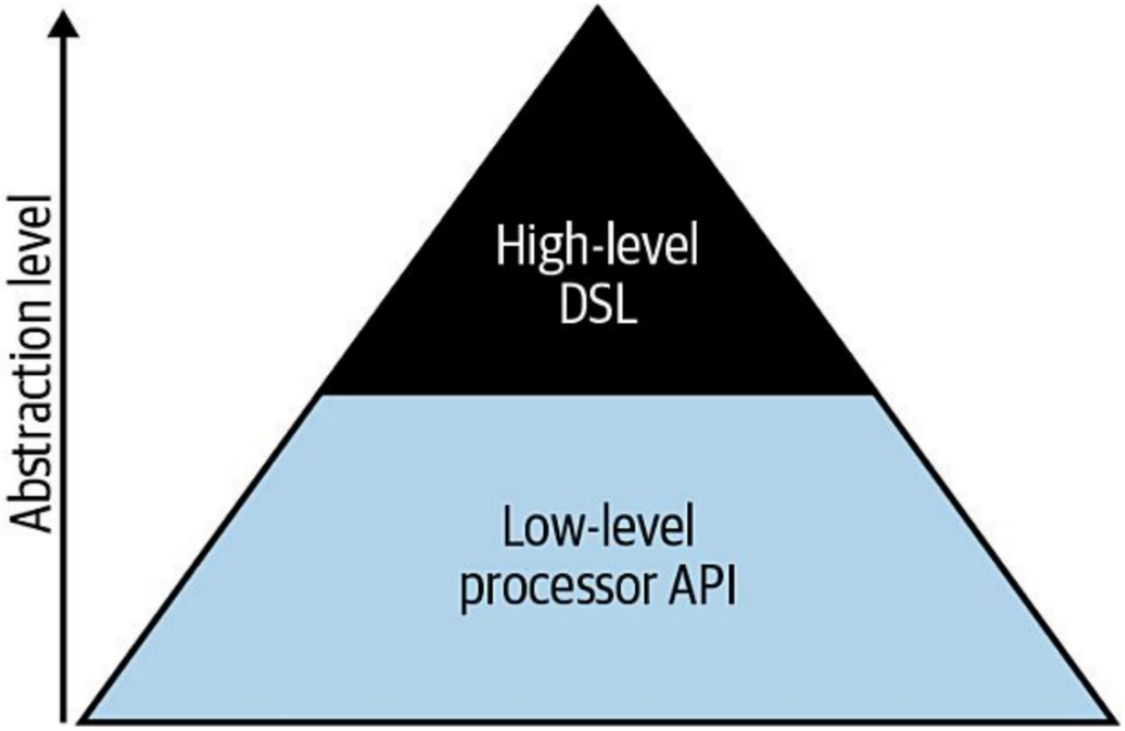


Figure 2-10. Abstraction levels of Kafka Streams APIs

The high-level DSL is built on top of the Processor API, but the interface each exposes is slightly different. If you would like to build your stream processing application using a functional style of programming, and would also like to leverage some higher-level abstractions for working with your data (streams and tables), then the DSL is for you.

On the other hand, if you need lower-level access to your data (e.g., access to record metadata), the ability to schedule periodic functions, more granular access to your application state, or more fine-grained control over the timing of certain operations, then the Processor API is a better choice. In the following tutorial, you will see examples of both the DSL and Processor API. In subsequent chapters, we will explore both the DSL and Processor API in further detail.

Now, the best way to see the difference between these two abstraction levels is with a code example. Let's move on to our first Kafka Streams tutorial: Hello Streams.

Introducing Our Tutorial: Hello, Streams

In this section, we will get our first hands-on experience with Kafka Streams. This is a variation of the "Hello, world" tutorial that has become the standard when learning new programming languages and libraries. There are two implementations of this tutorial: the first uses the high-level DSL, while the second uses the low-level Processor API. Both programs are functionally equivalent, and will print a simple greeting whenever they receive a message from the `users` topic in Kafka (e.g., upon receiving the message `Mitch`, each application will print `Hello, Mitch`).

Before we get started, let's take a look at how to set up the project.

Project Setup

All of the tutorials in this book will require a running Kafka cluster, and the source code for each chapter will include a *docker-compose.yml* file that will allow you to run a development cluster using Docker. Since Kafka Streams applications are meant to run outside of a Kafka cluster (e.g., on different machines than the brokers), it's best to view the Kafka cluster as a separate infrastructure piece that is required but distinct from your Kafka Streams application.

To start running the Kafka cluster, clone the repository and change to the directory containing this chapter's tutorial. The following commands will do the trick:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-02/hello-streams
```

Then, start the Kafka cluster by running:

```
docker-compose up
```

The broker will be listening on port 29092.[22] Furthermore, the preceding command will start a container that will precreate the `users` topic needed for this tutorial. Now, with our Kafka cluster running, we can start building our Kafka Streams application.

Creating a New Project

In this book, we will use a build tool called Gradle[23] to compile and run our Kafka Streams applications. Other build tools (e.g., Maven) are also supported, but we have chosen Gradle due to the improved readability of its build files.

In addition to being able to compile and run your code, Gradle can also be used to quickly bootstrap new Kafka Streams applications that you build outside of this book.

This can be accomplished by creating a directory for your project to live in and then by running the `gradle init` command from within that directory. An example of this workflow is as follows:

```
$ mkdir my-project && cd my-project

$ gradle init \
 --type java-application \
 --dsl groovy \
 --test-framework junit-jupiter \
 --project-name my-project \
 --package com.example
```

The source code for this book already contains the initialized project structure for each tutorial, so it's not necessary to run `gradle init` unless you are starting a new project for yourself. We simply mention it here with the assumption that you will be writing your own Kafka Streams applications at some point, and want a quick way to bootstrap your next project.

Here is the basic project structure for a Kafka Streams application:

```
.
├── build.gradle ❶
└── src
    ├── main
    │   ├── java ❷
    │   └── resources ❸
    └── test
        └── java ❹
```

❶ This is the project's build file. It will specify all of the dependencies (including the Kafka Streams library) needed by our application.

❷ We will save our source code and topology definitions in *src/main/java*.

❸ *src/main/resources* is typically used for storing configuration files.

❹ Our unit and topology tests, which we will discuss in "Testing Kafka Streams", will live in *src/test/java*.

Now that we've learned how to bootstrap new Kafka Streams projects and have had an initial look at the project structure, let's take a look at how to add Kafka Streams to our project.

Adding the Kafka Streams Dependency

To start working with Kafka Streams, we simply need to add the Kafka Streams library as a dependency in our build file. (In Gradle projects, our build file is called *build.gradle*.) An example build file is shown here:

```
plugins {
    id 'java'
    id 'application'
}

repositories {
    jcenter()
}

dependencies {
    implementation 'org.apache.kafka:kafka-streams:2.7.0' ❶
}

task runDSL(type: JavaExec) { ❷
    main = 'com.example.DslExample'
    classpath sourceSets.main.runtimeClasspath
}

task runProcessorAPI(type: JavaExec) { ❸
    main = 'com.example.ProcessorApiExample'
    classpath sourceSets.main.runtimeClasspath
}
```

❶ Add the Kafka Streams dependency to our project.

❷ This tutorial is unique among others in this book since we will be creating two different versions of our topology. This line adds a Gradle task to execute the DSL version of our application.

❸ Similarly, this line adds a Gradle task to execute the Processor API version of our application.

Now, to build our project (which will actually pull the dependency from the remote repository into our project), we can run the following command:

```
./gradlew build
```

That's it! Kafka Streams is installed and ready to use. Now, let's continue with the tutorial.

DSL

The DSL example is exceptionally simple. We first need to use a Kafka Streams class called `StreamsBuilder` to build our processor topology:

```
StreamsBuilder builder = new StreamsBuilder();
```

Next, as we learned in "Processor Topologies", we need to add a source processor in order to read data from a Kafka topic (in this case, our topic will be called `users`). There are a few different methods we could use here depending on how we decide to model our data (we will discuss different approaches in "Streams and Tables"), but for now, let's model our data as a stream. The following line adds the source processor:

```
KStream<Void, String> stream = builder.stream("users");❶
```

❶ We'll discuss this more in the next chapter, but the generics in `KStream<Void, String>` refer to the key and value types. In this case, the key is empty (`Void`) and the value is a `String` type.

Now, it's time to add a stream processor. Since we're just printing a simple greeting for each message, we can use the `foreach` operator with a simple lambda like so:

```
stream.foreach(
    (key, value) -> {
        System.out.println("(DSL) Hello, " + value);
    });
```

Finally, it's time to build our topology and start running our stream processing application:

```
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();
```

The full code, including some boilerplate needed to run the program, is shown in Example 2-1.

Example 2-1. Hello, world—DSL example

```
class DslExample {

  public static void main(String[] args) {
    StreamsBuilder builder = new StreamsBuilder(); ❶

    KStream<Void, String> stream = builder.stream("users"); ❷

    stream.foreach( ❸
        (key, value) -> {
          System.out.println("(DSL) Hello, " + value);
        });

    // omitted for brevity
    Properties config = ...; ❹

    KafkaStreams streams = new KafkaStreams(builder.build(), config); ❺
    streams.start();

    // close Kafka Streams when the JVM shuts down (e.g., SIGTERM)
    Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ❻
  }
}
```

❶ The builder is used to construct the topology.

❷ Add a source processor that reads from the `users` topic.

❸ Use the DSL's `foreach` operator to print a simple message. The DSL includes many operators that we will be exploring in upcoming chapters.

❹ We have omitted the Kafka Streams configuration for brevity, but will discuss this in upcoming chapters. Among other things, this configuration allows us to specify which Kafka cluster our application should read from and what consumer group this application belongs to.

❺ Build the topology and start streaming.

❻ Close Kafka Streams when the JVM shuts down.

To run the application, simply execute the following command:

```
./gradlew runDSL --info
```

Now your Kafka Streams application is running and listening for incoming data. As you may recall from "Hello, Kafka", we can produce some data to our Kafka cluster using the `kafka-console-producer` console script. To do this, run the following commands:

```
docker-compose exec kafka bash ❶

kafka-console-producer \ ❷
    --bootstrap-server localhost:9092 \
    --topic users
```

❶ The console scripts are available in the `kafka` container, which is running the broker in our development cluster. You can also download these scripts as part of the official Kafka distribution.

❷ Start a local producer that will write data to the `users` topic.

Once you are in the producer prompt, create one or more records by typing the name of the user, followed by the Enter key. When you are finished, press Control-C on your keyboard to exit the prompt:

```
>angie
>guy
>kate
>mark
```

Your Kafka Streams application should emit the following greetings:

```
(DSL) Hello, angie
(DSL) Hello, guy
(DSL) Hello, kate
(DSL) Hello, mark
```

We have now verified that our application is working as expected. We will explore some more interesting use cases over the next several chapters, but this process of defining a topology and running our application is a foundation we can build upon. Next, let's look at how to create the same Kafka Streams topology with the lower-level Processor API.

Processor API

The Processor API lacks some of the abstractions available in the high-level DSL, and its syntax is more of a direct reminder that we're building processor topologies, with methods like `Topology.addSource`, `Topology.addProcessor`, and `Topology.addSink` (the latter of which is not used in this example). The first step in using the processor topology is to instantiate a new `Topology` instance, like so:

```
Topology topology = new Topology();
```

Next, we will create a source processor to read data from the `users` topic, and a stream processor to print a simple greeting. The stream processor references a class called `SayHelloProcessor` that we'll implement shortly:

```
topology.addSource("UserSource", "users"); ❶
topology.addProcessor("SayHello", SayHelloProcessor::new, "UserSource"); ❷
```

❶ The first argument for the `addSource` method is an arbitrary name for this stream processor. In this case, we simply call this processor `UserSource`. We will refer to this name in the next line when we want to connect a child processor, which in turn defines how data should flow through our topology. The second argument is the topic name that this source processor should read from (in this case, `users`).

❷ This line creates a new downstream processor called `SayHello` whose processing logic is defined in the `SayHelloProcessor` class (we will create this in the next section). In the Processor API, we can connect one processor to another by specifying the name of the parent processor. In this case, we specify the `UserSource` processor as the parent of the `SayHello` processor, which means data will flow from the `UserSource` to `SayHello`.

As we saw before, in the DSL tutorial, we now need to build the topology and call `streams.start()` to run it:

```
KafkaStreams streams = new KafkaStreams(topology, config);
streams.start();
```

Before running the code, we need to implement the `SayHelloProcessor` class. Whenever you build a custom stream processor using the Processor API, you need to implement the `Processor` interface. The interface specifies methods for initializing the stream processor (`init`), applying the stream processing logic to a single record (`process`), and a cleanup function (`close`). The initialization and cleanup function aren't needed in this example.

The following is a simple implementation of `SayHelloProcessor` that we will use for this example. We will explore more complex examples, and all of the interface methods in the Processor interface (`init`, `process`, and `close`), in more detail in Chapter 7.

```
public class SayHelloProcessor implements Processor<Void, String, Void, Void> { ❶
  @Override
  public void init(ProcessorContext<Void, Void> context) {} ❷

  @Override
  public void process(Record<Void, String> record) { ❸
    System.out.println("(Processor API) Hello, " + record.value());
  }

  @Override
  public void close() {} ❹
}
```

❶ The first two generics in the `Processor` interface (in this example, `Processor<Void, String, ..., ...>`) refer to the *input* key and value types. Since our keys are null and our values are usernames (i.e., text strings), `Void` and `String` are the appropriate choices. The last two generics (`Processor<..., ..., Void, Void>`) refer to the *output* key and value types. In this example, our `SayHelloProcessor` simply prints a greeting. Since we aren't forwarding any output keys or values downstream, `Void` is the appropriate type for the final two generics.[24]

❷ No special initialization is needed in this example, so the method body is empty. The generics in the `ProcessorContext` interface (`ProcessorContext<Void, Void>`) refer to the output key and value types (again, as we're not forwarding any messages downstream in this example, both are `Void`).

❸ The processing logic lives in the aptly named `process` method in the `Processor` interface. Here, we print a simple greeting. Note that the generics in the `Record` interface refer to the key and value type of the *input* records.

❹ No special cleanup needed in this example.

We can now run the code using the same command we used in the DSL example:

```
./gradlew runProcessorAPI --info
```

You should see the following output to indicate your Kafka Streams application is working as expected:

```
(Processor API) Hello, angie
(Processor API) Hello, guy
(Processor API) Hello, kate
(Processor API) Hello, mark
```

Now, despite the Processor API's power, which we will see in Chapter 7, using the DSL is often preferable because, among other benefits, it includes two very powerful abstractions: streams and tables. We will get our first look at these abstractions in the next section.

Streams and Tables

If you look closely at Example 2-1, you will notice that we used a DSL operator called `stream` to read a Kafka topic into a *stream*. The relevant line of code is:

```
KStream<Void, String> stream = builder.stream("users");
```

However, kafka streams also supports an additional way to view our data: as a *table*. in this section, we'll take a look at both options and learn when to use *streams* and when to use *tables*.

As discussed in "Processor Topologies", designing a processor topology involves specifying a set of source and sink processors, which correspond to the topics your application will read from and write to. However, instead of working with Kafka topics *directly*, the Kafka Streams DSL allows you to work with different *representations* of a topic, each of which are suitable for different use cases. There are two ways to model the data in your Kafka topics: as a *stream* (also called a *record stream*) or a *table* (also known as a *changelog stream*). The easiest way to compare these two data models is through an example.

Say we have a topic containing ssh logs, where each record is keyed by a user ID as shown in Table 2-2.

Table 2-2. Keyed records in a single topic-partition

Key	Value	Offset
mitch	`{ "action": "login" }`	0
mitch	`{ "action": "logout" }`	1
elyse	`{ "action": "login" }`	2
isabelle	`{ "action": "login" }`	3

Before consuming this data, we need to decide which abstraction to use: a stream or a table. When making this decision, we need to consider whether or not we want to track only the latest state/representation of a given key, or the entire history of messages. Let's compare the two options side by side:

Streams

These can be thought of as inserts in database parlance. Each distinct record remains in this view of the log. The stream representation of our topic can be seen in Table 2-3.

Table 2-3. Stream view of ssh logs

Key	Value	Offset
mitch	`{ "action": "login" }`	0
mitch	`{ "action": "logout" }`	1
elyse	`{ "action": "login" }`	2
isabelle	`{ "action": "login" }`	3

Tables

Tables can be thought of as updates to a database. In this view of the logs, only the current state (either the latest record for a given key or some kind of aggregation) for each key is retained. Tables are usually built from *compacted topics* (i.e., topics that are configured with a `cleanup.policy` of `compact`, which tells Kafka that you only want to keep the latest representation of each key). The table representation of our topic can be seen in Table 2-4.

Table 2-4. Table view of ssh logs

Key	Value	Offset
mitch	`{ "action": "logout" }`	1
elyse	`{ "action": "login" }`	2
isabelle	`{ "action": "login" }`	3

Tables, by nature, are *stateful*, and are often used for performing aggregations in Kafka Streams.[25] In Table 2-4, we didn't really perform a mathematical aggregation, we just kept the latest ssh event for each user ID. However, tables also support mathematical aggregations. For example, instead of tracking the latest record for each key, we could have just as easily calculated a rolling `count`. In this case, we would have ended up with a slightly different table, where the values contain the result of our `count` aggregation. You can see a count-aggregated table in Table 2-5.

Table 2-5. Aggregated table view of ssh logs

Key	Value	Offset
mitch	`2`	1
elyse	`1`	2
isabelle	`1`	3

Careful readers may have noticed a discrepancy between the design of Kafka's storage layer (a distributed, append-only log) and a table. Records that are written to Kafka are immutable, so how is it possible to model data as updates, using a *table* representation of a Kafka topic?

The answer is simple: the table is materialized on the Kafka Streams side using a key-value store which, by default, is implemented using RocksDB.[26] By consuming an ordered stream of events and keeping only the latest record for each key in the client-side key-value store (more commonly called a *state store* in Kafka Streams terminology), we end up with a table or map-like representation of the data. In other words, the table isn't something we *consume* from Kafka, but something we *build* on the client side.

You can actually write a few lines of Java code to implement this basic idea. In the following code snippet, the `List` represents a stream since it contains an ordered collection of records,[27] and the table is constructed by iterating through the list (`stream.forEach`) and only retaining the latest record for a given key using a `Map`. The following Java code demonstrates this basic idea:

```
import java.util.Map.Entry;

var stream = List.of(
    Map.entry("a", 1),
    Map.entry("b", 1),
    Map.entry("a", 2));

var table = new HashMap<>();

stream.forEach((record) -> table.put(record.getKey(), record.getValue()));
```

If you were to print the stream and table after running this code, you would see the following output:

```
stream ==> [a=1, b=1, a=2]

table ==> {a=2, b=1}
```

Of course, the Kafka Streams implementation of this is more sophisticated, and can leverage fault-tolerant data structures as opposed to an in-memory `Map`. But this ability to construct a table representation of an unbounded stream is only one side of a more complex relationship between streams and tables, which we will explore next.

Stream/Table Duality

The *duality* of tables and streams comes from the fact that tables can be represented as streams, and streams can be used to reconstruct tables. We saw the latter transformation of a stream into a table in the previous section, when discussing the discrepancy between Kafka's append-only, immutable log and the notion of a mutable table structure that accepts updates to its data.

This ability to reconstruct tables from streams isn't unique to Kafka Streams, and is in fact pretty common in various types of storage. For example, MySQL's replication process relies on the same notion of taking a stream of events (i.e., row changes) to reconstruct a source table on a downstream replica. Similarly, Redis has the notion of an append-only file (AOF) that captures every command that is written to the in-memory key-value store. If a Redis server goes offline, then the stream of commands in the AOF can be replayed to reconstruct the dataset.

What about the other side of the coin (representing a table as a stream)? When viewing a table, you are viewing a single point-in-time representation of a stream. As we saw earlier, tables can be updated when a new record arrives. By changing our view of the table to a stream, we can simply process the update as an insert, and append the new record to the end of the log instead of updating the key. Again, the intuition behind this can be seen using a few lines of Java code:

```
var stream = table.entrySet().stream().collect(Collectors.toList());

stream.add(Map.entry("a", 3));
```

This time, if you print the contents of the stream, you'll see we're no longer using update semantics, but instead insert semantics:

```
stream ==> [a=2, b=1, a=3]
```

So far, we've been working with the standard libraries in Java to build intuition around streams and tables. However, when working with streams and tables in Kafka Streams, you'll use a set of more specialized abstractions. We'll take a look at these abstractions next.

KStream, KTable, GlobalKTable

One of the benefits of using the high-level DSL over the lower-level Processor API in Kafka Streams is that the former includes a set of abstractions that make working with streams and tables extremely easy.

The following list includes a high-level overview of each:

`KStream`

A `KStream` is an abstraction of a partitioned *record stream*, in which data is represented using insert semantics (i.e., each event is considered to be *independent* of other events).

`KTable`

A `KTable` is an abstraction of a partitioned table (i.e., *changelog stream*), in which data is represented using update semantics (the latest representation of a given key is tracked by the application). Since `KTables` are partitioned, each Kafka Streams task contains only a subset of the full table.[28]

`GlobalKTable`

This is similar to a `KTable`, except each `GlobalKTable` contains a complete (i.e., unpartitioned) copy of the underlying data. We'll learn when to use `KTables` and when to use `GlobalKTables` in Chapter 4.

Kafka Streams applications can make use of multiple stream/table abstractions, or just one. It's entirely dependent on your use case, and as we work through the next few chapters, you will learn when to use each one. This completes our initial discussion of streams and tables, so let's move on to the next chapter and explore Kafka Streams in more depth.

Summary

Congratulations, you made it through the end of your first date with Kafka Streams. Here's what you learned:

- Kafka Streams lives in the stream processing layer of the Kafka ecosystem. This is where sophisticated data processing, transformation, and enrichment happen.
- Kafka Streams was built to simplify the development of stream processing applications with a simple, functional API and a set of stream processing primitives that can be reused across projects. When more control is needed, a lower-level Processor API can also be used to define your topology.
- Kafka Streams has a friendlier learning curve and a simpler deployment model than cluster-based solutions like Apache Flink and Apache Spark Streaming. It also supports event-at-a-time processing, which is considered true streaming.
- Kafka Streams is great for solving problems that require or benefit from real-time decision making and data processing. Furthermore, it is reliable, maintainable, scalable, and elastic.
- Installing and running Kafka Streams is simple, and the code examples in this chapter can be found at *https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb*.

In the next chapter, we'll learn about stateless processing in Kafka Streams. We will also get some hands-on experience with several new DSL operators, which will help us build more advanced and powerful stream processing applications.

[1] We are referring to the official ecosystem here, which includes all of the components that are maintained under the Apache Kafka project.

[2] Jay Kreps, one of the original authors of Apache Kafka, discussed this in detail in an O'Reilly blog post back in 2014.

[3] This includes aggregated streams/tables, which we'll discuss later in this chapter.

[4] We have an entire chapter dedicated to time, but also see Matthias J. Sax's great presentation on the subject from Kafka Summit 2019.

[5] Guozhang Wang, who has played a key role in the development of Kafka Streams, deserves much of the recognition for submitting the original KIP for what would later become Kafka Streams. See *https://oreil.ly/l2wbc*.

[6] Kafka Connect veered a little into event processing territory by adding support for something called *single message transforms*, but this is extremely limited compared to what Kafka Streams can do.

[7] Kafka Streams will work with other JVM-based languages as well, including Scala and Kotlin. However, we exclusively use Java in this book.

[8] Multiple consumer groups can consume from a single topic, and each consumer group processes messages independently of other consumer groups.

[9] While partitions can be added to an existing topic, the recommended pattern is to create a new source topic with the desired number of partitions, and to migrate all of the existing workloads to the new topic.

[10] Including some features that are specific to *stateful applications*, which we will discuss in Chapter 4.

[11] The ever-growing nature of the stream processing space makes it difficult to compare every solution to Kafka Streams, so we have decided to focus on the most popular and mature stream processing solutions available at the time of this writing.

[12] Although there is an open, albeit dated, proposal to support batch processing in Kafka Streams.

[13] Including a Kafka-backed relational database called KarelDB, a graph analytics library built on Kafka Streams, and more. See *https://yokota.blog*.

[14] At the time of this writing, Apache Flink had recently released a beta version of queryable state, though the API itself was less mature and came with the following warning in the official Flink documentation: "The client APIs for queryable state are currently in an evolving state and there are no guarantees made about stability of the provided interfaces. It is likely that there will be breaking API changes on the client side in the upcoming Flink versions." Therefore, while the Apache Flink team is working to close this gap, Kafka Streams still has the more mature and production-ready API for querying state.

[15] The exception to this is when topics are joined. In this case, a single topology will read from each source topic involved in the join without further dividing the step into sub-topologies. This is required for the join to work. See "Co-Partitioning" for more information.

[16] In this example, we write to two intermediate topics (`valid-mentions` and `invalid-mentions`) and then immediately consume data from each. Using intermediate topics like this is usually only required for certain operations (for example, repartitioning data). We do it here for discussion purposes only.

[17] A Java application may execute many different types of threads. Our discussion will simply focus on the stream threads that are created and managed by the Kafka Streams library for running a processor topology.

[18] Remember, a Kafka Streams topology can be composed of multiple sub-topologies, so to get the number of tasks for the entire program, you should sum the task count across all sub-topologies.

[19] This doesn't mean a poorly implemented stream processor is immune from concurrency issues. However, by default, the stream threads do not share any state.

[20] For example, the number of cores that your application has access to could inform the number of threads you decide to run with. If your application instance is running on a 4-core machine and your topology supports 16 tasks, you may want to configure the thread count to 4, which will give you a thread for each core. On the other hand, if your 16-task application was running on a 48-core machine, you may want to run with 16 threads (you wouldn't run with 48 since the upper bound is the task count, or in this case: 16).

[21] From *First Principles: Elon Musk on the Power of Thinking for Yourself*.

[22] If you want to verify and have telnet installed, you can run `echo 'exit' | telnet localhost 29092`. If the port is open, you should see "Connected to localhost" in the output.

[23] Instructions for installing Gradle can be found at *https://gradle.org*. We used version 6.6.1 for the tutorials in this book.

[24] This version of the `Processor` interface was introduced In Kafka Streams version 2.7 and deprecates an earlier version of the interface that was available in Kafka Streams 2.6 and earlier. In the earlier version of the `Processor` interface, only input types are specified. This presented some issues with type-safety checks, so the newer form of the `Processor` interface is recommended.

[25] In fact, tables are sometimes referred to as *aggregated streams*. See "Of Streams and Tables in Kafka and Stream Processing, Part 1" by Michael Noll, which explores this topic further.

[26] RocksDB is a fast, embedded key-value store that was originally developed at Facebook. We will talk more about RocksDB and key-value stores in Chapters 4–6.

[27] To go even deeper with the analogy, the index position for each item in the list would represent the offset of the record in the underlying Kafka topic.

[28] Assuming your source topic contains more than one partition.

Chapter 3. Stateless Processing

The simplest form of stream processing requires no memory of previously seen events. Each event is consumed, processed,[1] and subsequently forgotten. This paradigm is called *stateless processing*, and Kafka Streams includes a rich set of operators for working with data in a stateless way.

In this chapter, we will explore the *stateless operators* that are included in Kafka Streams, and in doing so, we'll see how some of the most common stream processing tasks can be tackled with ease. The topics we will explore include:

- Filtering records
- Adding and removing fields
- Rekeying records
- Branching streams
- Merging streams
- Transforming records into one or more outputs
- Enriching records, one at a time

We'll take a tutorial-based approach for introducing these concepts. Specifically, we'll be streaming data about cryptocurrencies from Twitter and applying some stateless operators to convert the raw data into something more meaningful: investment signals. By the end of this chapter, you will understand how to use stateless operators in Kafka Streams to enrich and transform raw data, which will prepare you for the more advanced concepts that we will explore in later chapters.

Before we jump into the tutorial, let's get a better frame of reference for what stateless processing is by comparing it to the other form of stream processing: *stateful processing*.

Stateless Versus Stateful Processing

One of the most important things you should consider when building a Kafka Streams application is whether or not your application requires stateful processing. The following describes the distinction between stateless and stateful stream processing:

- In *stateless applications*, each event handled by your Kafka Streams application is processed independently of other events, and only *stream* views are needed by your application (see "Streams and Tables"). In other words, your application treats each event as a self-contained insert and requires no memory of previously seen events.

- *Stateful applications*, on the other hand, need to remember information about previously seen events *in one or more steps* of your processor topology, usually for the purpose of aggregating, windowing, or joining event streams. These applications are more complex under the hood since they need to track additional data, or *state*.

In the high-level DSL, the type of stream processing application you ultimately build boils down to the individual *operators* that are used in your topology.[2] Operators are stream processing functions (e.g., `filter`, `map`, `flatMap`, `join`, etc.) that are applied to events as they flow through your topology. Some operators, like `filter`, are considered *stateless* because they only need to look at the current record to perform an action (in this case, `filter` looks at each record individually to determine whether or not the record should be forwarded to downstream processors). Other operators, like `count`, are *stateful* since they require knowledge of previous events (`count` needs to know how many events it has seen so far in order to track the number of messages).

If your Kafka Streams application requires *only* stateless operators (and therefore does not need to maintain any memory of previously seen events), then your application is considered *stateless*. However, if you introduce one or more stateful operators (which we will learn about in the next chapter), regardless of whether or not your application also uses stateless operators, then your application is considered *stateful*. The added complexity of stateful applications warrants additional considerations with regards to maintenance, scalability, and fault tolerance, so we will cover this form of stream processing separately in the next chapter.

If all of this sounds a little abstract, don't worry. We'll demonstrate these concepts by building a stateless Kafka Streams application in the following sections, and getting some first-hand experience with stateless operators. So without further ado, let's introduce this chapter's tutorial.

Introducing Our Tutorial: Processing a Twitter Stream

In this tutorial, we will explore the use case of algorithmic trading. Sometimes called *high-frequency trading* (HFT), this lucrative practice involves building software to evaluate and purchase securities automatically, by processing and responding to many types of market signals with minimal latency.

To assist our fictional trading software, we will build a stream processing application that will help us gauge market sentiment around different types of cryptocurrencies (Bitcoin, Ethereum, Ripple, etc.), and use these sentiment scores as investment/divestment signals in a custom trading algorithm.[3] Since millions of people use Twitter to share their thoughts on cryptocurrencies and other topics, we will use Twitter as the data source for our application.

Before we get started, let's look at the steps required to build our stream processing application. We will then use these requirements to design a processor topology, which will be a helpful guide as we build our stateless Kafka Streams application. The key concepts in each step are italicized:

1. Tweets that mention certain digital currencies (#bitcoin, #ethereum) should be consumed from a source topic called `tweets`:
    - Since each record is JSON-encoded, we need to figure out how to properly *deserialize* these records into higher-level data classes.
    - Unneeded fields should be removed during the deserialization process to simplify our code. Selecting only a subset of fields to work with is referred to as *projection*, and is one of the most common tasks in stream processing.
2. Retweets should be excluded from processing. This will involve some form of data *filtering*.
3. Tweets that aren't written in English should be *branched* into a separate stream for translating.
4. Non-English tweets need to be translated to English. This involves *mapping* one input value (the non-English tweet) to a new output value (an English-translated tweet).
5. The newly translated tweets should be *merged* with the English tweets stream to create one unified stream.
6. Each tweet should be enriched with a sentiment score, which indicates whether Twitter users are conveying positive or negative emotion when discussing certain digital currencies. Since a single tweet could mention multiple cryptocurrencies, we will demonstrate how to convert each input (tweet) into a variable number of outputs using a `flatMap` operator.
7. The enriched tweets should be serialized using Avro, and written to an output topic called `crypto-sentiment`. Our fictional trading algorithm will read from this topic and make investment decisions based on the signals it sees.

Now that the requirements have been captured, we can design our processor topology. Figure 3-1 shows what we'll be building in this chapter and how data will flow through our Kafka Streams application.

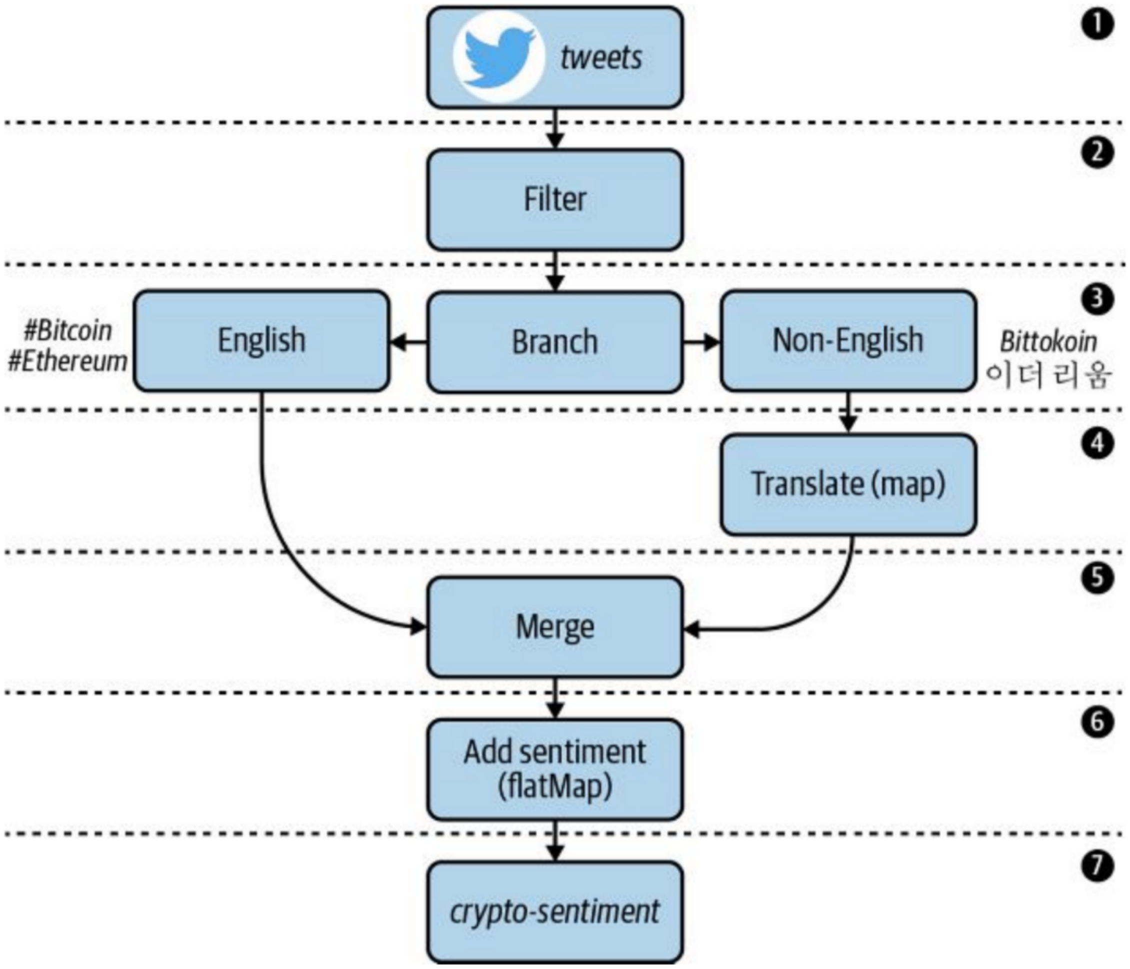


Figure 3-1. The topology that we will be implementing for our tweet enrichment application

With our topology design in hand, we can now start implementing our Kafka Streams application by working our way through each of the processing steps (labeled 1–7) in Figure 3-1. We will start by setting up our project, and then move on to the first step in our topology: streaming tweets from the source topic.

Project Setup

The code for this chapter is located at *https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git*.

If you would like to reference the code as we work our way through each topology step, clone the repository and change to the directory containing this chapter's tutorial. The following command will do the trick:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-03/crypto-sentiment
```

You can build the project anytime by running the following command:

```
$ ./gradlew build --info
```

Note

We have omitted the implementation details of tweet translation and sentiment analysis (steps 4 and 6 in Figure 3-1) since they aren't necessary to demonstrate the stateless operators in Kafka Streams. However, the source code in GitHub does include a full working example, so please consult the project's *README.md* file if you are interested in these implementation details.

Now that our project is set up, let's start creating our Kafka Streams application.

Adding a KStream Source Processor

All Kafka Streams applications have one thing in common: they consume data from one or more source topics. In this tutorial, we only have one source topic: `tweets`. This topic is populated with tweets from the Twitter source connector, which streams tweets from Twitter's streaming API and writes JSON-encoded tweet records to Kafka. An example tweet value[4] is shown in Example 3-1.

Example 3-1. Example record value in the tweets source topic

```
{
    "CreatedAt": 1602545767000,
    "Id": 1206079394583924736,
    "Text": "Anyone else buying the Bitcoin dip?",
    "Source": "",
    "User": {
        "Id": "123",
        "Name": "Mitch",
        "Description": "",
        "ScreenName": "timeflown",
        "URL": "https://twitter.com/timeflown",
        "FollowersCount": "1128",
        "FriendsCount": "1128"
    }
}
```

Now that we know what the data looks like, the first step we need to tackle is getting the data from our source topic into our Kafka Streams application. In the previous chapter, we learned that we can use the `KStream` abstraction to represent a stateless record stream. As you can see in the following code block, adding a `KStream` source processor in Kafka Streams is simple and requires just a couple of lines of code:

```
StreamsBuilder builder = new StreamsBuilder(); ❶

KStream<byte[], byte[]> stream = builder.stream("tweets"); ❷
```

❶ When using the high-level DSL, processor topologies are built using a `StreamsBuilder` instance.

❷ `KStream` instances are created by passing the topic name to the `StreamsBuilder.stream` method. The `stream` method can optionally accept additional parameters, which we will explore in later sections.

One thing you may notice is that the `KStream` we just created is parameterized with `byte[]` types:

```
KStream<byte[], byte[]>
```

We briefly touched on this in the previous chapter, but the `KStream` interface leverages two generics: one for specifying the type of keys (`K`) in our Kafka topic and the other for specifying the type of values (`V`). If we were to peel back the floorboards in the Kafka Streams library, we would see an interface that looks like this:

```
public interface KStream<K, V> {
  // omitted for brevity
}
```

Therefore, our `KStream` instance, which is parameterized as `KStream<byte[], byte[]>`, indicates that the record keys and values coming out of the `tweets` topic are being encoded as byte arrays. However, we just mentioned that the tweet records are actually encoded as JSON objects by the source connector (see Example 3-1), so what gives?

Kafka Streams, *by default*, represents data flowing through our application as byte arrays. This is due to the fact that Kafka itself stores and transmits data as raw byte sequences, so representing the data as a byte array will always work (and is therefore a sensible default). Storing and transmitting raw bytes makes Kafka flexible because it doesn't impose any particular data format on its clients, and also fast, since it requires less memory and CPU cycles on the brokers to transfer a raw byte stream over the network.[5] However, this means that Kafka clients, including Kafka Streams applications, are responsible for serializing and deserializing these byte streams in order to work with higher-level objects and formats, including strings (delimited or non-delimited), JSON, Avro, Protobuf, etc.[6]

Before we address the issue of deserializing our tweet records into higher-level objects, let's add the additional boilerplate code needed to run our Kafka Streams application. For testability purposes, it's often beneficial to separate the logic for building a Kafka Streams topology from the code that actually runs the application. So the boilerplate code will include two classes. First, we'll define a class for building our Kafka Streams topology, as shown in Example 3-2.

Example 3-2. A Java class that defines our Kafka Streams topology

```
class CryptoTopology {

  public static Topology build() {
    StreamsBuilder builder = new StreamsBuilder();

    KStream<byte[], byte[]> stream = builder.stream("tweets");
    stream.print(Printed.<byte[], byte[]>toSysOut().withLabel("tweets-stream")); ❶

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

❶ The `print` operator allows us to easily view data as it flows through our application. It is generally recommended for development use only.

The second class, which we'll call `App`, will simply instantiate and run the topology, as shown in Example 3-3.

Example 3-3. A separate Java class used to run our Kafka Streams application

```
class App {
  public static void main(String[] args) {
    Topology topology = CryptoTopology.build();

    Properties config = new Properties(); ❶
    config.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev");
    config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:29092");

    KafkaStreams streams = new KafkaStreams(topology, config); ❷

    Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ❸
```

```
      System.out.println("Starting Twitter streams");
      streams.start(); ❹
    }
  }
```

❶ Kafka Streams requires us to set some basic configuration, including an application ID (which corresponds to a consumer group) and the Kafka bootstrap servers. We set these configs using a `Properties` object.

❷ Instantiate a `KafkaStreams` object with the processor topology and streams config.

❸ Add a shutdown hook to gracefully stop the Kafka Streams application when a global shutdown signal is received.

❹ Start the Kafka Streams application. Note that `streams.start()` does not block, and the topology is executed via background processing threads. This is the reason why a shutdown hook is required.

Our application is now ready to run. If we were to start our Kafka Streams application and then produce some data to our `tweets` topic, we would see the raw byte arrays (the cryptic values that appear after the comma in each output row) being printed to the screen:

```
[tweets-stream]: null, [B@c52d992
[tweets-stream]: null, [B@a4ec036
[tweets-stream]: null, [B@3812c614
```

As you might expect, the low-level nature of byte arrays makes them a little difficult to work with. In fact, additional stream processing steps will be much easier to implement if we find a different method of representing the data in our source topic. This is where the concepts of data serialization and deserialization come into play.

Serialization/Deserialization

Kafka is a bytes-in, bytes-out stream processing platform. This means that clients, like Kafka Streams, are responsible for converting the byte streams they consume into higher-level objects. This process is called *deserialization*. Similarly, clients must also convert any data they want to write back to Kafka back into byte arrays. This process is called *serialization*. These processes are depicted in Figure 3-2.

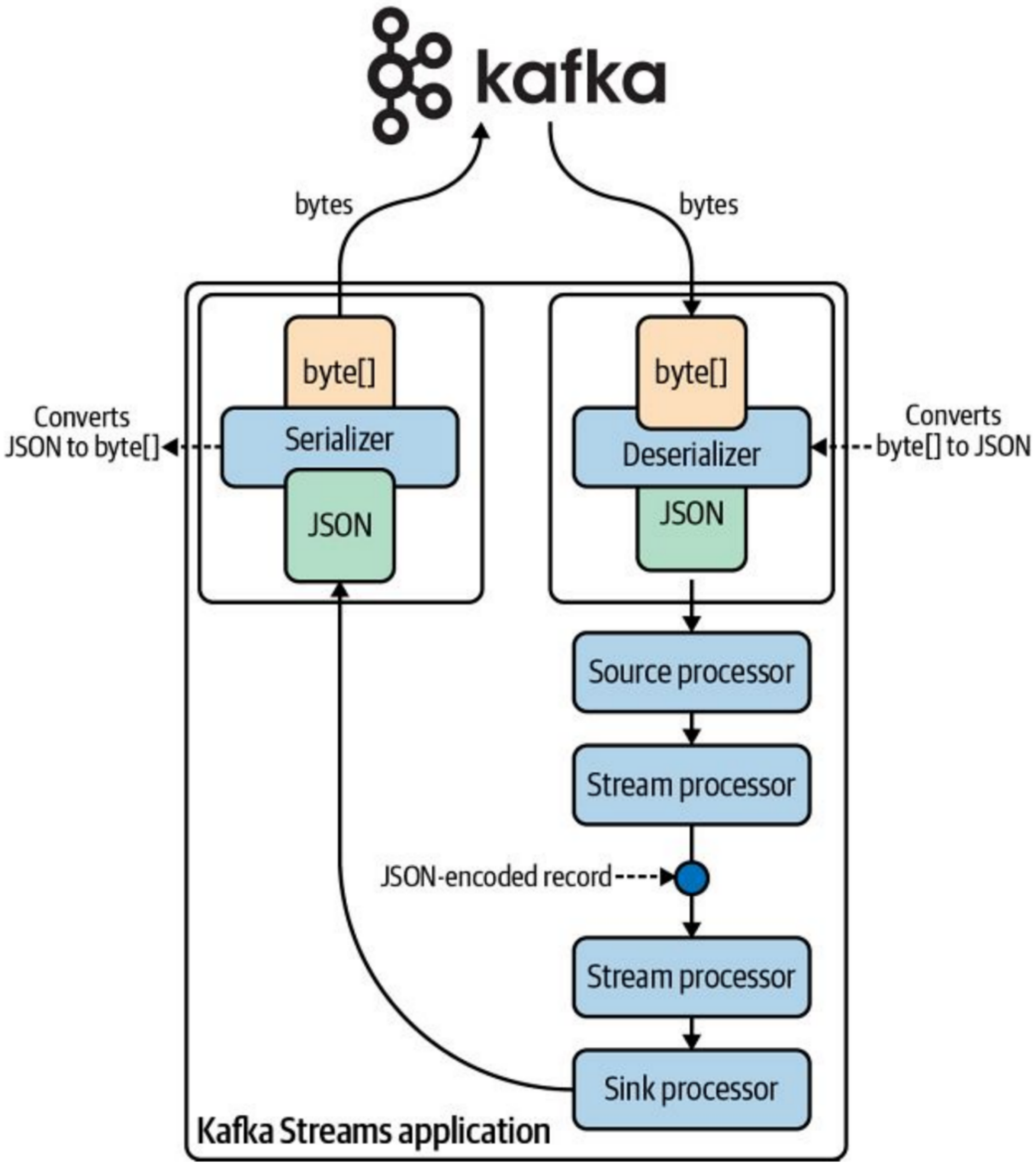


Figure 3-2. An architectural view of where the deserialization and serialization processes occur in a Kafka Streams application

In Kafka Streams, serializer and deserializer classes are often combined into a single class called a *Serdes*, and the library ships with several implementations, shown in Table 3-1.[7] For example, the String Serdes (accessible via the `Serdes.String()` method) includes both the String serializer *and* deserializer class.

image not available

project setup, Project Setup

putting it all together, Putting It All Together-Putting It All Together

registering streams and tables, Registering Streams and Tables

stateful processors, creating, Creating Stateful Processors-Creating Stateful Processors

stateless processing, Stateless Processing-Summary

processing a Twitter stream tutorial, Introducing Our Tutorial: Processing a Twitter Stream

adding KStream source processor, Adding a KStream Source Processor

branching data, Branching Data

empirically verifying the application, Empirical Verification-Empirical Verification

enriching tweets and performing sentiment analysis, Enriching Tweets-Sentiment Analysis

filtering data, Filtering Data-Filtering Data

merging streams, Merging Streams

project setup, Project Setup

running the code, Running the Code

serialization/deserialization of byte streams, Serialization/Deserialization-Building the Tweet Serdes

serializing Avro data, Serializing Avro Data-Schema Registry–Aware Avro Serdes

sink processor, Adding a Sink Processor

translating tweets, Translating Tweets-Translating Tweets

versus stateful processing, Stateless Versus Stateful Processing

stateless processors

adding to IoT digital twin service, Adding Stateless Stream Processors

creating, Creating Stateless Processors-Creating Stateless Processors statement-based replication in ksqlDB, Similarities

StateStore interface, implementing, Custom State Stores

static membership, Static Membership

sticky partition assignment, Sticky Assignment

stop word removal UDF, creating in ksqlDB, Stop-word removal UDF-Additional Resources for Custom ksqlDB Functions

storage, State Stores

- (see also state stores)
- local and remote, in ksqlDB vs. traditional SQL databases, Differences

store builders, Creating Stateful Processors

Stores factory class, Creating Stateful Processors

StoreSupplier interface, Custom State Stores, Creating Stateful Processors

stream operator, Streams and Tables

stream processing, The Kafka Ecosystem

- basics with ksqlDB, Stream Processing Basics with ksqlDB-Summary
  - basic queries, Basic Queries-Flattening/Unnesting Complex Structures
  - collections, Collections
  - conditional expressions, Conditional Expressions-Case Statements
  - data types in ksqlDB, Data Types
  - putting it all together, Putting It All Together
  - source topics, Source Topics
  - tutorial, monitoring changes at Netflix, Tutorial: Monitoring Changes at Netflix-Project Setup
  - working with streams and tables, Working with Streams and Tables-Dropping Streams and Tables
  - writing results back to Kafka, Writing Results Back to Kafka (Persistent Queries)-Terminating queries

before Kafka Streams, Before Kafka Streams

intermediate and advanced with ksqlDB, Intermediate and Advanced Stream Processing with ksqlDB-Summary

aggregations, Aggregations-Window retention

client queries of data, Clients

data enrichment, Data Enrichment-Windowed Joins

materialized views, Materialized Views-Materialized Views

pull queries, Pull Queries-Curl

push queries, Push Queries

tutorial, monitoring changes at Netflix, Intermediate and Advanced Stream Processing with ksqlDB-Bootstrapping an Environment from a SQL File

stateless versus stateful, Stateless Versus Stateful Processing

using another stream processing framework, Before Kafka Streams

using Kafka Streams, Enter Kafka Streams

stream processors, Processor Topologies

adding to chatbox example, Sub-Topologies

adding to Hello Streams tutorial, DSL

for IoT digital twin service, Putting It All Together

stream threads, Tasks and Stream Threads, Co-Partitioning, Configuration Properties

stream time, Time-Driven Dataflow, Delayed data

and retention period for windows, Window retention

showing effects of delayed data on, Delayed data

stream-relational processing, Benefits of Stateful Processing

streaming data from PostgreSQL to Kafka tutorial, Tutorial-Verifying the Source Connector

creating connectors with ksqlDB, Creating Connectors with ksqlDB

installing connectors, Installing Connectors streaming ETL, What Is ksqlDB?, Data Integration with ksqlDB

StreamPartitioner class, Adding Sink Processors

streams, Getting Started with Kafka Streams

- (see also Kafka Streams)
- aggregating, Aggregating Streams
- collections in ksqlDB, Collections
  - creating streams, Creating Source Collections
- creating derived streams, Creating Derived Collections
- extension of SQL to support, Kafka Streams Integration
- grouping, Grouping Streams
- join types in ksqlDB, Joins
- modeling users topic as stream in ksqlDB, Using the ksqlDB CLI
- nonexistent in Processor API, Adding Source Processors
- registering, Registering Streams and Tables
- registering with timestamp extractor, Registering Streams with a Timestamp Extractor
- relationships between, combining data based on, Joins
- storage of, How Are Streams Stored?
- streams and tables in Kafka Streams applications, Streams and Tables-KStream, KTable, GlobalKTable
  - KStream, KTable, and GlobalKTable, KStream, KTable, GlobalKTable
  - stream view of ssh logs, Streams and Tables
  - stream/table duality, Stream/Table Duality
- topics in Kafka, Communication Model
- windowing, Windowing Streams-Windowed Aggregation
- working with in ksqlDB, Working with Streams and Tables-Dropping Streams and Tables