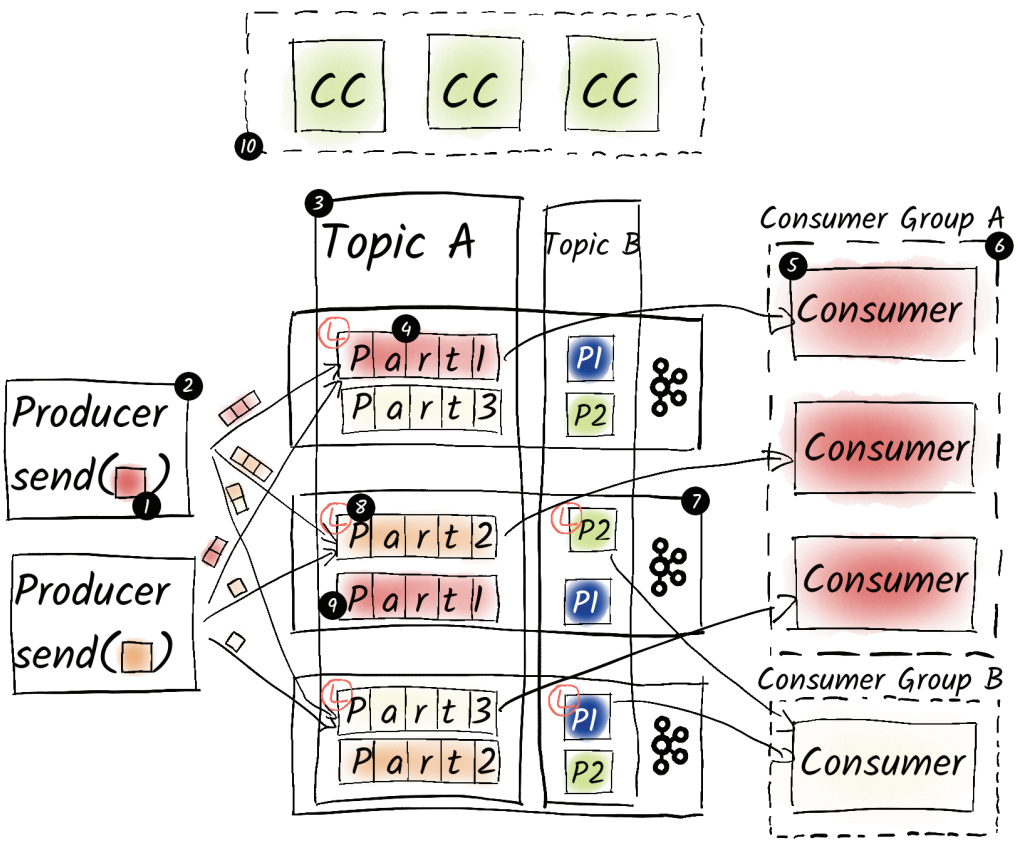
From basics to production

Apache Kafka IN ACTION

Anatoly Zelenin
Alexander Kropp
Foreword by Adam Bellemare

MANNING

Apache Kafka: At a Glance



TERMS

1. **Messages**, also called records, are the payload. They are sent as byte arrays and, under the hood, are typically grouped into batches before being sent.
2. **Producers** send messages to the leader of a partition and select the partition themselves with the help of a partitioner.
3. **Topics** are used to bundle messages of a business topic. They are comparable to tables in a database.
4. **Partitions** are the backbone of Kafka's performance. Topics are divided into partitions in order to parallelize and scale processes. To ensure fault tolerance and high availability, partitions are replicated across brokers.
5. **Consumers** receive and process messages from Kafka and can read from multiple partitions, as well as from multiple topics.
6. **Consumer Groups** allow parallel processing and scalable message consumption by dividing partitions and messages among consumers. If one consumer fails, the others in the group take over its tasks, ensuring fault tolerance.
7. **Brokers** are Kafka servers. They share replicas and tasks evenly among themselves, which improves performance. If one broker fails, another takes over, increasing reliability.
8. **Leader** is the broker that is responsible for read and write operations of a partition. Leaders are distributed as evenly as possible among all brokers.
9. **Followers** are the brokers to which the partitions are copied from the leader to increase resilience.
10. **Coordination Cluster (KRaft, previously ZooKeeper ensemble)** is used by Kafka to coordinate itself.

Apache Kafka in Action

Apache Kafka in Action

From basics to production

Anatoly Zelenin
Alexander Kropp

Foreword by Adam Bellemare

MANNING
Shelter Island

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

ISBN 9781633437593
Printed in the United States of America

Development editor:	Connor O'Brien
Technical editor:	Purushotham Chikkanayakanhalli Krishnegowda
Review editor:	Kishor Rit
Production editor:	Kathy Rossland
Copy editor:	Julie McNamee
Proofreader:	Keri Hales
Technical proofreader:	Purushotham Chikkanayakanhalli Krishnegowda
Typesetter:	Tamara Švelić Sabljić
Cover designer:	Marija Tudor

brief contents

contents

foreword

I started with Apache Kafka back in early 2015 with version 0.8.2, and as a longtime user, occasional contributor, and active observer, I've been fortunate enough to see countless organizations build, iterate, and succeed with Apache Kafka. Now, a decade later, Kafka is markedly different from how it began. There's a whole range of different deployment options, designed and implemented by a community that has lived and toiled in the problem space. Real-world experience has led to real-world capabilities, baked into the core Kafka project.

In this book, *Apache Kafka in Action*, authors Anatoly Zelenin and Alexander Kropp share their many years of real-world Kafka experience. Anatoly is an instructor and founder of the DataFlow Academy, teaching people worldwide the best ways to not only use Kafka, but how to integrate all the various supporting components: Kafka Connect, MirrorMaker, schema registries, monitoring solutions, and more.

Alex is a seasoned hands-on practitioner. A regular consultant, he helps businesses set up cloud-based data streaming platforms with modern data toolsets. He is an expert in Kubernetes and infrastructure as code deployments.

Together, Anatoly and Alex have written an excellent work on Apache Kafka. Their first book, written in 2021, may have slipped under your radar as it did mine—unless you're a German-speaker. But Anatoly and Alex have returned with this latest edition, updated and edited to take into account the wealth of changes that have visited Apache Kafka in the past few years.

One of the great things about this book is that it starts out at the very beginning of your Kafka journey. The first few sections will introduce you to event streaming and why it matters in today's world. There are a lot of benefits to reacting to changes in your business just moments after they occur, and it'll unlock a lot of powerful new architectural

patterns and capabilities for you to explore. If you've never used Apache Kafka before, then this book is for you.

If you're a seasoned expert, then good news—this book is also for you. Anatoly and Alex cover many of the more complex aspects of managing and running not only one cluster, but multiples, spread all over the world. Data replication, disaster recovery, and preventative monitoring are just some of the subjects they cover. And finally, they'll also provide you with a host of useful tips and tricks to optimize your clusters and applications, reduce costs, and keep your event streams healthy. I'm sure you'll find it an essential manual for your Apache Kafka journey.

—Adam Bellemare
Author of *Building Event-Driven Microservices*

preface

Our journey with Apache Kafka began years ago—Anatoly as a trainer and practitioner, Alex as a Kafka consultant—when the German publisher Hanser approached us about writing a book on the subject. "Sure, why not?" we thought. "How hard could it be?" Little did we know that it would take almost two years of intensive research into architectural patterns, technical details, and crafting a compelling narrative before we'd complete the first edition—in German.

The readers' responses exceeded our expectations. As first-time authors, we were overwhelmed by the fantastic feedback and the meaningful connections we made with our readers. However, these questions kept recurring: "Why did you write this book in German?" "When will there be an English edition?" Moreover, we felt the book wasn't quite complete. We hadn't had enough time to thoroughly address a crucial aspect: how to successfully integrate Kafka into an organization. Questions about the bigger picture of Kafka in organizations, architectural adaptations, and implementation strategies remained unexplored.

When Manning reached out to us about collaborating on *Apache Kafka in Action*, we saw a perfect opportunity. We proposed translating and enhancing our German edition rather than working with the existing book Manning had published previously. This way, we saw the opportunity to not only update the book to the current state of Kafka but also make the book available worldwide and add the missing pieces. Once again, we underestimated the scope of the project!

Now, after considerable effort, you're holding the fully revised and enhanced version of *Apache Kafka in Action*. We hope you find it valuable for your real-time data projects. If you encounter us at conferences or online, please say hello and connect with us. We always enjoy exchanging ideas with the Kafka community.

acknowledgments

First and foremost, we thank our families and friends for their unwavering support throughout this project. Your encouragement and assistance over the years have been invaluable.

We're deeply grateful to the Kafka community for making this book possible. Your collective efforts have shaped Kafka into the remarkable project it is today.

Our training participants and customers deserve special recognition. Your support, both through shared experiences and, of course, your financial support, has been crucial to this book's development.

We extend our gratitude to everyone who supported the first German edition with Hanser, particularly Sylvia Hasselbach, whose patience and fantastic cooperation over the years were instrumental to our success.

The Manning team has been exceptional, especially our development editor Connor O'Brien, whose discussions, comments, and suggestions significantly improved the book. We also thank Purushotham Chikkanayakanhalli Krishnegowda, seasoned IT professional with more than 21 years of experience with technologies such as Apache Kafka, Java, Spring, and microservices, who served as our technical editor and technical proofreader on this book. His thorough reviews and attention to detail has enhanced the book's technical precision and clarity.

We would like to express our sincere thanks to Adam Bellemare for writing the fantastic foreword to this book.

We must also thank the editorial and production teams at Manning, who helped make this book possible. Their attention to detail and patience in answering our questions helped focus our attention on many improvements in this edition.

Finally, we're indebted to all reviewers of both the German first edition and this book. Your hard work and invaluable feedback have enhanced the quality of this work

for all readers: Inga Blundell, Walter Forkel, Daniela Griesinger, Tobias Heller, Vincent Latzko, Andrej Olunczek, Elin Rixmann, Thomas Trepper, and David Weber. To all Manning reviewers, your suggestions helped make this a better book: Afshin Paydar, AJ Bhandal, Al Pezewski, Alexey Artemov, Alireza Aghamohammadi, Anandaganesh Balakrishnan, André Schäfer, Andres Sacco, Anthony Nandaa, Asif Iqbal, Bassam Ismail, Christian Thoudahl, David Gloyn-Cox, Gabor Laszlo Hajba, Ganesh Swaminathan, Gatikrushna Sahu, Gilberto Taccari, Glumov Konstantin, Jeremy Chen, Jim Whitfield, Jorge Bo, Joseph Pachod, Justin Reiser, Kristina Kasanicova, Lakshminarayanan A.S, Maddula Arathi, Manuel Rebello de Andrade, Mark Dechamps, Matthias J. Sax, Maxim Volgin, Michael Heil, Mikhail Malev, Milorad Imbra, Onofrei George, Peter Szabo, Rajdeep Gurmeet, Rambabu Posa, Richard Meinsen, Ronald Haring, Sharath Chandra Parashara, Simon Verhoeven, Simone Sguazza, Srihari Sridharan, Steve Goodman, Sumit Pal, Toby Lazar, Venkata Yanamadala, Viktoria Dolzhenko, Vinicios Wentz, Vivek Lakhanpal, William Jamir Silva, William Walsh, Yogesh Shetty, and Zorodzayi Mukuya.

about this book

We wrote *Apache Kafka in Action* to share our hands-on experience and make your journey with Kafka both effective and enjoyable. Through our years of training professionals and implementing Kafka in organizations, we've learned what works and what doesn't. While we focus on building practical knowledge—from your first steps with Kafka through to running production systems—we also dive deep into the theoretical foundations that are crucial for success with Kafka. Think of this book as the guide we wish we had when we started our own Kafka journey, combining essential theory with real-world expertise. Throughout the book, you'll find illustrative diagrams, practical tips, and easy-to-follow code examples that you can quickly implement yourself.

Who should read this book

We wrote this book for IT professionals who want to grow their Kafka expertise, whether they're just starting out or already working with Kafka systems. While a general understanding of modern IT architectures and distributed systems is helpful, you don't need any prior Kafka experience. Our goal is to meet you where you are and build your knowledge step by step. Whether you're a developer, system administrator, architect, or technical team lead looking to enhance your data infrastructure, you'll find practical guidance throughout the book. Even if you're a seasoned Kafka expert, you'll likely discover new insights and "Aha!" moments as we explore advanced patterns, organizational challenges, and lesser-known features.

How this book is organized: A road map

Apache Kafka in Action is organized into four parts comprising 18 chapters, each building upon the previous to create a comprehensive understanding of Kafka from basics to enterprise implementation:

- Part 1: *Getting started* (chapters 1–2) introduces you to the world of Apache Kafka. We begin by explaining what Kafka is, its architecture, and its role in modern data architectures. Through practical examples using a running Kafka cluster, you'll learn how to work with basic Kafka operations, gaining hands-on experience with topics, producers, and consumers.
- Part 2: *Concepts* (chapters 3–6) delves into the essential building blocks of Kafka. We explore the details of topics and messages, examine Kafka's role as a distributed log, and investigate how Kafka achieves reliability through replication and transactions. The section concludes with a deep dive into performance optimization, covering crucial aspects such as partitioning strategies and configuration tuning.
- Part 3: *Kafka deep dive* (chapters 7–10) takes you behind the scenes of Kafka's operations. You'll learn the intricacies of cluster management, discover how messages are produced and persisted, understand the mechanics of message consumption, and master message cleanup strategies. This section provides the technical depth needed to troubleshoot problems and optimize your Kafka deployment.
- Part 4: *Kafka in enterprise use* (chapters 11–18) bridges the gap between theory and real-world implementation. We cover essential enterprise topics such as system integration with Kafka Connect, stream processing, governance, and reference architectures. You'll learn practical skills for monitoring, performing disaster recovery, and choosing between different technologies. The section concludes with guidance on avoiding common pitfalls and implementing Kafka successfully in modern architectures.

While the book is designed to be read sequentially, experienced practitioners may choose different paths. If you're already familiar with Kafka basics, you might skim part 1, but we recommend reviewing part 2 as it contains insights that even experienced users often find valuable. Technical leaders focusing on architectural decisions might prefer to concentrate on parts 1, 2, and 4, using part 3's technical deep-dive as a reference when needed. Throughout the book, you'll find numerous practical examples that demonstrate real-world applications of the concepts being discussed.

For those new to Kafka or needing guidance on setup, appendix A provides detailed instructions for creating a test environment, while appendix B covers monitoring setup procedures. These appendixes serve as practical references you can return to whenever needed.

About the code

This book contains many examples of source code listings in line with normal text. Source code is formatted in a `fixed-width font like this` to separate it from ordinary text.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this wasn't enough, and listings include line-continuation markers

(➥). Code annotations accompany many of the listings, highlighting important concepts. You can get executable snippets of code from the liveBook (online) version of this book at https://livebook.manning.com/book/apache-kafka-in-action.

liveBook discussion forum

Purchase of *Apache Kafka in Action* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to https://livebook.manning.com/book/apache-kafka-in-action/discussion. You can also learn more about Manning's forums and the rules of conduct at https://livebook.manning.com/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

Alexander Kropp has dedicated his focus to Apache Kafka and Kubernetes, actively shaping cloud platform designs and ensuring efficient monitoring systems. With extensive experience as an architect and trainer in communication networks and cloud computing, his creative approach to solution design underscores his excellence in the field.

Anatoly Zelenin is a renowned expert and trainer in Apache Kafka. He is known for his interactive and engaging workshops, which attract customers from various industries across Europe, particularly banking and manufacturing. In addition to his role as an IT consultant and trainer, Anatoly is an avid adventurer, exploring different parts of our planet.

Together, they are part of the DataFlow Academy, a leading European training and consulting company. The academy specializes in real-time data technologies, focusing on hands-on, interactive learning experiences.

about the cover illustration

The figure on the cover of *Apache Kafka in Action* is "La Nocellara," or "The Nocellara," taken from *Uses and costumes of Naples and surroundings described and painted*, a literary work by Francesco de Bourcard published in 1853 and related to the customs and traditions of the Kingdom of Naples in the 19th century.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

Part 1

Getting started

In the ever-evolving world of data processing and system integration, Apache Kafka stands as a foundational technology. But how do we get started with Kafka? What does it take for us to unlock Kafka's full potential in our applications and workflows?

In this first part of the book, we'll guide you through the essential building blocks to get up and running with Kafka. Starting with an introduction to Kafka, we'll explore its core concepts and components, as well as its role within modern enterprise ecosystems. We'll discover what Kafka is, where it fits, and the power it brings to data-driven architectures.

In chapter 1, we lay the groundwork by explaining Kafka's architecture, including how its components—producers, consumers, topics, and brokers—work together to facilitate scalable, fault-tolerant messaging. In chapter 2, we dive into Kafka's core operations by creating topics, producing and consuming messages, and using command-line tools. By the end, we'll have hands-on experience with a basic Kafka setup, preparing us for more advanced topics ahead.

1 Introduction to Apache Kafka

This chapter covers

- What Apache Kafka is and its use cases
- How Kafka fits into enterprise ecosystems
- Architectural overview of Kafka
- Running and using Kafka

Modern enterprise applications are often built from independent components and services that communicate by exchanging messages. Managing this flow of messages becomes increasingly complex as systems scale across multiple servers, data centers, cloud platforms, and geographic regions. The challenge is further heightened by the need for reliability, fault tolerance, and near-real-time performance. Apache Kafka was designed specifically to address these demands, providing a high-throughput, distributed messaging platform capable of handling massive streams of data efficiently.

In this first chapter, we'll learn what Apache Kafka is, dive into its basic components and their purpose, and discuss where it can be used. More importantly, we'll look at how Kafka fits into broader enterprise ecosystems. Finally, we'll cover what is required to use Kafka in terms of hardware, tools, and programming languages.

1.1 What is Apache Kafka, and how does it solve our problems?

Our world runs in real time. When we make a purchase, we expect immediate confirmation. When we order parcels, we expect real-time tracking. When our credit card is used suspiciously, we want to know now, not tomorrow. As customers, we have come to expect organizations to process and react to data as fast as life happens.

This real-time expectation, combined with an explosion in data volume—from Internet of Things (IoT) sensors streaming data to customers interacting across many digital touchpoints—creates many challenges for IT teams. Traditional batch-oriented architectures, designed for end-of-day processing, struggle to adapt to this new reality in which every second counts.

Especially for established organizations with complex legacy systems, it has become a critical challenge to transform their architecture to handle real-time data flows while maintaining system reliability. As they break down monolithic applications into distributed services, the complexity grows drastically. Teams need to evolve their services independently while ensuring reliable data flow across the organization. How do you maintain system stability when dozens of services need to communicate? How do you ensure teams can deploy changes without breaking other services? And, how do you handle these massive streams of data flowing between systems without creating a tangled web of point-to-point integrations?

Kafka provides an architectural pattern that addresses these challenges by serving as the core of a central nervous system for data. It's no coincidence that, according to the Kafka community, over 80% of Fortune 100 companies rely on Kafka to solve these exact problems. From enabling independent service evolution to preventing system-wide failures through asynchronous communication, and transforming batch-oriented systems into real-time streams, Kafka has become the backbone of modern distributed architectures.

Kafka is a robust, open source, distributed streaming platform that fundamentally changes how organizations can handle their data flows. At its core, Kafka acts as a persistent, distributed log of all your organization's data events. Think of it as a central backbone where every significant piece of information—from customer interactions to system state changes—can be reliably stored and processed in real time. This architectural approach provides key advantages:

- By persisting data streams, Kafka enables both real-time processing and reliable replay of data. When a consuming system fails, it can simply pick up where it left off. When you develop a new service that needs historical data, it can process past events just as easily as new ones.
- Kafka's distributed nature means it scales horizontally to handle massive data volumes while maintaining fault tolerance. A single Kafka cluster can handle millions of events per second while ensuring no data is lost, even if parts of the system fail.

These capabilities have made Kafka essential across industries. Financial institutions use it to process millions of transactions in real time while ensuring every system—from fraud detection to customer notifications—stays in sync. Manufacturers stream sensor data from thousands of IoT devices through Kafka to enable predictive maintenance and real-time monitoring. Retailers use its capabilities for inventory management and order processing.

Kafka achieves this through a publish-subscribe model, or as the Kafka community phrases it, a producer-consumer model. Producers send messages to specific topics, and consumers process these messages as needed. But what sets Kafka apart from traditional messaging systems is its persistence layer—data written to Kafka can be stored and read multiple times, for hours, days, or even months.

This persistence, combined with Kafka's distributed nature and rich ecosystem of tools such as Kafka Connect and Kafka Streams, has transformed Kafka from its origins as LinkedIn's high-throughput messaging system into something much more powerful: the central nervous system for enterprise data.

This book is designed for a broad range of IT professionals who want to gain a solid understanding of Kafka and its integration into existing IT infrastructures. Whether you're an architect, a system administrator, a developer, or a data engineer, this book provides a comprehensive starting point with Kafka.

You shouldn't expect a developer handbook filled with dozens of Java examples. Instead, we aim to offer clear, practical insights and guidance that will enable you to effectively use Kafka in your projects. Our goal is to equip you with the knowledge and skills needed to use Kafka for data streaming and real-time processing in a variety of IT environments.

1.2 Kafka in enterprise ecosystems

What do we mean by saying that Kafka can be the central nervous system for data? The vision is that every *event* (see figure 1.1), a significant occurrence or incident within an enterprise that generates data and carries important information, is stored in Kafka. In the context of enterprise systems and Kafka, an event typically represents various activities, changes, or transactions happening in different parts of the organization, such as user interactions, system updates, financial transactions, or any other relevant business operation.

In the context of event-driven architectures and Kafka, an event is often a piece of data that encapsulates the details of a specific occurrence, providing a structured representation of the incident. These events are typically produced by various components or applications within the enterprise and are then published to a Kafka *topic*, which is a structural unit to organize data streams. Subsequently, other systems or applications can subscribe to these topics to consume and process the events, enabling real-time data flow and communication between different parts of the enterprise ecosystem.

In many companies, there's a growing separation between *legacy systems*, which are essential for current business processes and models, and newer systems built using

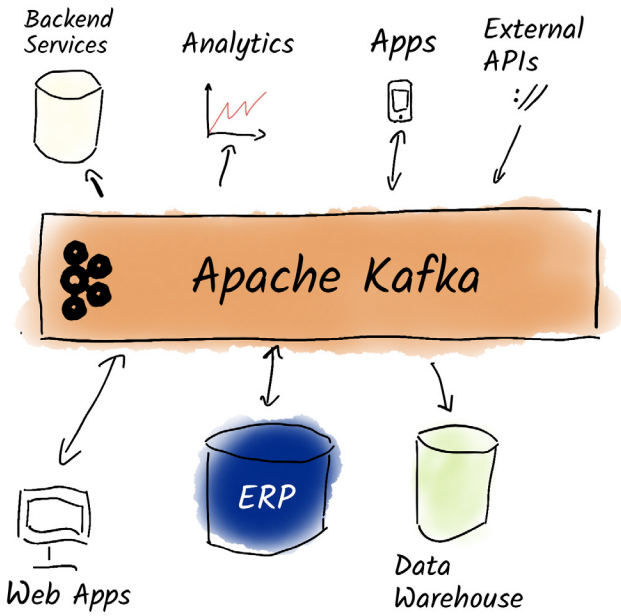


Figure 1.1 Kafka as the central nervous system for data in a company. Every event that takes place in the enterprise is stored in Kafka. Other services can react to these events asynchronously and process them further.

modern development practices. The *new world* refers to the approach of developing innovative services and solutions using contemporary methodologies. While Kafka isn't strictly necessary to serve as an interface between old and new systems (this can often be handled by adapters or anti-corruption patterns), it plays a crucial role in enabling communication between these systems. Kafka facilitates the seamless exchange of messages, connecting established legacy systems with the dynamic, evolving world of modern software services. By doing so, it ensures smooth integration without requiring significant changes to the existing systems.

Legacy systems often fall short of meeting the modern demands of both internal and external customers. These systems often rely on batch processing, where data is handled in large chunks at set intervals. However, in today's fast-paced world, immediate access to information is essential. For instance, no one wants to wait a week or even a day for their account balance to update after a credit card transaction. We now expect real-time parcel tracking and instant data updates. Modern cars, for example, generate huge amounts of data that need to be sent to corporate headquarters for analysis, especially when preparing for autonomous driving. Kafka can help businesses transition from batch-oriented processing to real-time data handling. It acts as a powerful bridge between older systems and the real-time data processing model, helping companies respond to the growing need for instant information and adapt to changing customer expectations and industry trends. While Kafka is designed for real-time processing, it can still be used in situations where messages are processed less frequently, such as once a day, depending on the use case.

The way we write software is also changing. Instead of putting more and more functionality into monolithic services and then connecting these few monoliths to each other via integration, we're breaking our services into *microservices*. In the microservices software architectural style, applications are composed of small, independently deployable services that communicate over well-defined APIs. This approach aims to reduce dependency between teams, enhance scalability, and enable more flexibility in software development and deployment. To access the benefits of microservices, asynchronous data exchange must occur. Even if one microservice is undergoing maintenance, others can continue to function independently. Additionally, microservices require communication methods that allow data formats in one service to evolve independently of other services. Kafka provides valuable support in this context, offering a robust platform for asynchronous communication and data streaming, allowing microservices to operate independently and seamlessly exchange information in a decoupled way.

Another trend, largely driven by virtualization and the growing adoption of cloud architectures, is the declining need for specialized hardware. Unlike some other messaging systems, Kafka doesn't require dedicated appliances. It runs on standard, off-the-shelf hardware and doesn't rely on fail-safe systems. Kafka is designed to handle subsystem failures gracefully. As a result, even in the event of problems within the data center, message delivery remains reliable.

So, how does Kafka achieve this reliability and performance? How can we use Kafka for our use cases, and what needs to be considered when operating Kafka? We'll give you the answers to these questions and much more on our journey through this book.

1.3 Architectural overview of Kafka

Kafka's architecture isn't just a basic framework; it's a well-thought-out system that makes the smooth transmission, storage, and processing of data possible. Let's take a closer look at Kafka's architecture, shown in figure 1.2, to understand the key components that make it a fundamental tool for distributed streaming applications.

Whether you're a developer looking for a comprehensive understanding or you're simply curious about how it all works, this exploration of the following components will shed light on how Kafka's architecture is organized and its crucial role in shaping the future of data streaming:

1. *Messages* make up the payload, also called records. They are sent as byte arrays and, under the hood, are typically grouped into batches before being sent.
2. *Producers* send messages to the leader of a partition and select the partition themselves with the help of a partitioner.
3. *Topics* are used to bundle messages of a business topic. They are comparable to tables in a database.
4. *Partitions* are the backbone of Kafka's performance. Topics are divided into partitions to parallelize and scale processes. To ensure fault tolerance and high availability, partitions are replicated across brokers.

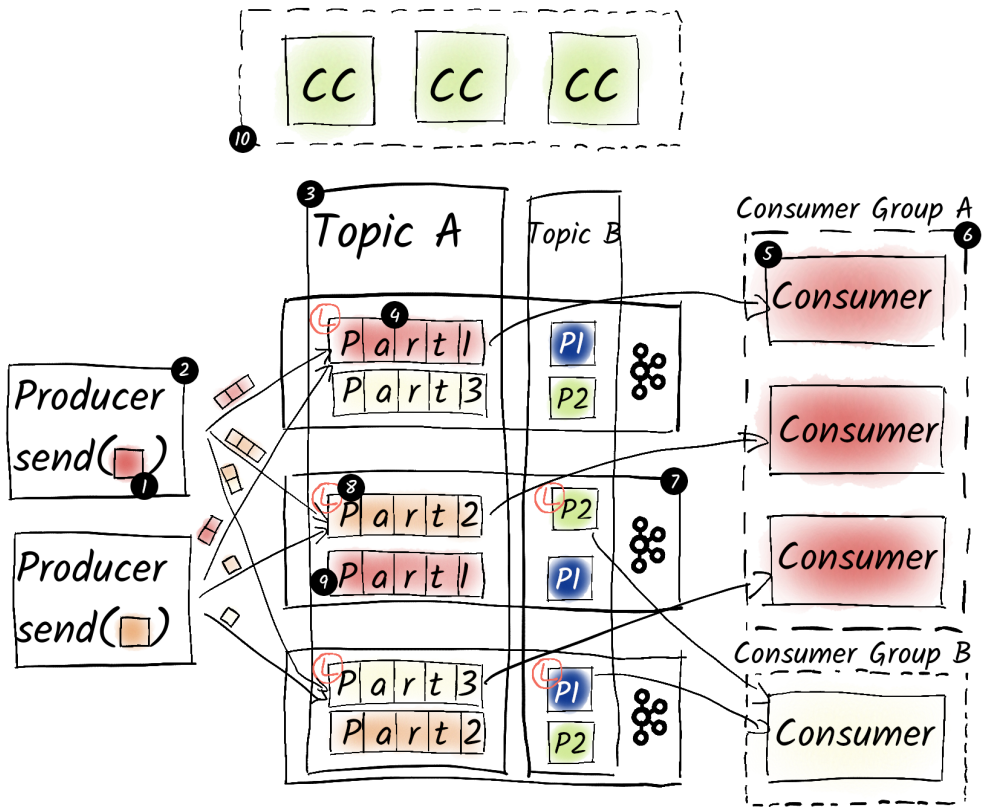


Figure 1.2 The components and data flow of Kafka

5 *Consumers* receive and process messages from Kafka and can read from multiple partitions, as well as from multiple topics.

6 *Consumer groups* allow parallel processing and scalable message consumption by dividing partitions and messages among consumers. If one consumer fails, the others in the group take over its tasks, ensuring fault tolerance.

7 *Brokers* are Kafka servers. They share replicas and tasks evenly among themselves, which improves performance. If one broker fails, another takes over, increasing reliability.

8 *Leaders* are the brokers responsible for read and write operations of a partition. Leaders are distributed as evenly as possible among all brokers.

9 *Followers* are the brokers to which the partitions are copied from the leader to increase resilience.

10 *Coordination cluster* (Kafka Raft [KRaft], previously ZooKeeper ensemble) is used by Kafka to coordinate itself.

Let's dive into an example to explore how Kafka's components work together. Picture a scenario where a bank is handling fund transfers Imagine the bank transfer application as the producer. Its role is to generate messages for each fund transfer event. These messages are like packets of information containing details such as source account, destination account, amount, and timestamp.

Now, these messages aren't sent directly to their destination. Instead, they are directed to a Kafka topic named `bank-transfers`. Think of a topic as a category where related messages are grouped together.

Within this topic, there are multiple partitions. Each partition handles a subset of messages, allowing for parallel processing. In our case, partitions might manage different types or groups of fund transfers.

The Kafka cluster, composed of brokers, is the backbone of this operation. Brokers are servers that store and manage the data. They work together to form a resilient and scalable Kafka cluster. Each broker oversees one or more partitions of each topic, ensuring efficient data distribution.

In traditional Kafka setups, ZooKeeper was used to coordinate brokers and manage metadata, and it played an essential role in ensuring Kafka's distributed nature. However, in more recent versions, Kafka has moved away from ZooKeeper and now uses the KRaft protocol, as mentioned in the preceding list, for internal coordination. This change allows Kafka to manage coordination directly without the need for an external system.

> **NOTE** The details of this shift from ZooKeeper to KRaft will be covered in more depth in a later chapter, but for now, it's enough to know that Kafka has transitioned to KRaft for improved scalability and simplicity.

Now let's talk about consumers. These are applications or services responsible for reading messages from Kafka topics. To handle this reading efficiently, consumers are organized into consumer groups. Each group can have multiple consumers, and each partition is consumed by only one consumer within a group. This parallel processing ensures that messages are processed swiftly.

Each partition has a leader broker and multiple followers. The leader takes charge of handling reads and writes, while followers replicate the data for fault tolerance. If a leader fails, a follower steps in to maintain a smooth flow of data.

Bringing it all together, as fund transfer messages traverse through Kafka, they are efficiently processed by consumer groups, ensuring that accounts are updated accurately and in a timely fashion. The Kafka cluster, with its distributed architecture and fault tolerance mechanisms, forms the backbone of this reliable and scalable data flow. Whether it's managing leaders and followers, partitioning messages, or coordinating brokers, Kafka orchestrates this symphony of data seamlessly.

1.4 Running and using Kafka

To effectively run Kafka, several components and prerequisites are essential. First, you need a reliable and properly configured set of servers to host the Kafka cluster. Each

server in the cluster acts as a broker and collaborates to manage the distributed processing of data streams. Additionally, adequate network infrastructure with low latency and high bandwidth is necessary to facilitate seamless communication between Kafka brokers. Users must have a clear understanding of their data requirements and design appropriate topics and partitions within Kafka to organize and distribute data effectively. It's also essential to have well-designed producer and consumer applications that interact with Kafka, allowing for the ingestion and processing of real-time data. Lastly, a comprehensive monitoring and management strategy is vital to keep track of Kafka's performance and troubleshoot any problems promptly. Running Kafka successfully demands a combination of well-configured infrastructure, robust network capabilities, and a deep understanding of Kafka's architecture and associated components.

Managing Kafka yourself is doable but shouldn't be underestimated. If you want to use Kafka in a cloud environment, multiple vendors offer managed Kafka services. Some cloud providers, such as AWS with Amazon Managed Streaming for Apache Kafka (Amazon MSK) and Azure HDInsight, offer Kafka solutions. Additionally, there are specialized Kafka vendors such as Confluent and Aiven. Besides these classical Kafka vendors, other competitors in this space offer Kafka-compatible services such as Redpanda and WarpStream, promising better performance, reduced costs, or additional features. Although many of these offerings sound interesting, we don't have enough experience with them to give a recommendation. We want to emphasize that managing the Kafka infrastructure itself is just one part of a successful streaming architecture, so even if you use a managed service, you need expertise in your team and company to make the most out of it.

> **TIP** We recommend both Confluent and Aiven, as we've had good experiences with both.

In addition to the infrastructure components mentioned earlier, running Kafka involves specific programming languages and tools. Kafka itself is implemented in Scala and Java, so a Java Runtime Environment (JRE) is required on the servers hosting Kafka brokers. To interact with Kafka and develop producer and consumer applications, users typically use programming languages such as Java, Scala, Python, or others supported by Kafka clients. Kafka provides official client libraries for various programming languages, enabling developers to integrate Kafka seamlessly into their applications.

1.5 Our learning path

Throughout the upcoming chapters, we'll embark on a comprehensive journey to understand Kafka, progressing seamlessly from fundamental concepts to a deep dive into its intricate workings. The learning experience is enhanced with a wealth of graphics that visually illustrate key Kafka principles, architecture, and processes, facilitating a clearer grasp of the underlying concepts. Whether exploring Kafka's distributed architecture or delving into the intricacies of topics, partitions, and brokers, you can rely on visual aids in this book to deepen your understanding.

The learning approach is hands-on, with simple yet effective examples that allow us to dive right in and start working with Kafka. Through practical scenarios and step-by-step instructions, we'll explore how to apply what we've learned to build and run Kafka applications. From producing and consuming messages to setting up and managing Kafka clusters, the examples are designed to be accessible, encouraging you to experiment and strengthen your skills through real-world use. With clear graphics and practical examples, we'll guide you through Kafka's complexities in a way that's both enlightening and directly applicable to your own work.

Summary

- Kafka is a powerful distributed streaming platform operating on a publish-subscribe model, allowing seamless data flow between producers and consumers.
- Widely adopted across industries, Kafka excels in real-time analytics, event sourcing, log aggregation, and stream processing, supporting organizations in making informed decisions based on up-to-the-minute data.
- Kafka's architecture prioritizes fault tolerance, scalability, and durability, ensuring reliable data transmission and storage even in the face of system failures.
- From finance to retail and telecommunications, Kafka finds applications in real-time fraud detection, transaction processing, inventory management, order processing, network monitoring, and large-scale data stream processing.
- Beyond its core messaging system, Kafka offers an ecosystem with tools such as Kafka Connect and Kafka Streams, providing connectors to external systems and facilitating the development of stream processing applications, enhancing its overall utility.
- Kafka can serve as a central hub for diverse system integration.
- Producers send messages to Kafka for distribution.
- Consumers receive and process messages from Kafka.
- Topics organize messages into channels or categories.
- Partitions divide topics to parallelize and scale processes.
- Brokers are Kafka servers managing storage, distribution, and retrieval.
- KRaft/ZooKeeper coordinates and manages tasks in a Kafka cluster.
- Kafka ensures data resilience through replication.
- Kafka scales horizontally by adding more brokers to the cluster.
- Kafka can run on general-purpose hardware.
- Kafka is implemented in Java and Scala, but there are clients for other programming languages as well, for example, Python.

2 First steps with Kafka

This chapter covers

- Introducing the book's use case
- Creating a topic with the `kafka-topics.sh` command
- Producing messages with the `kafka-console-producer.sh` command
- Consuming messages with the `kafka-console-consumer.sh` command

In this chapter, we'll delve into the realm of Apache Kafka, gaining our initial insights into its functionality. Our journey begins with the creation of a primary topic, followed by the production of messages directed to this topic. Ultimately, we'll complete the loop by consuming these messages once more, providing a comprehensive exploration of Kafka's fundamental operations. We assume that Kafka is already installed. We've described the exact installation instructions to create a Kafka cluster with three brokers in appendix B.

2.1 Introducing our use case

Throughout this book, we'll use a consistent use case to illustrate how Kafka can be effectively used in real-world scenarios. Our chosen use case revolves around an e-commerce platform, as shown in figure 2.1, which serves as a practical example to demonstrate Kafka's capabilities.

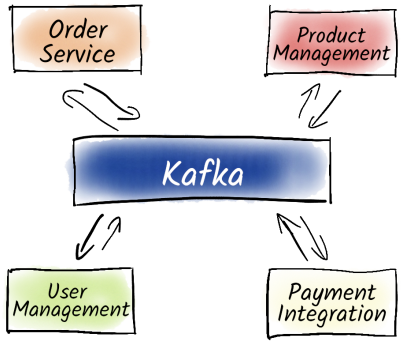


Figure 2.1 The high-level IT architecture for our online shopping platform

While our focus isn't on teaching how to build the e-commerce platform itself, this example will help contextualize Kafka's key concepts and techniques. We'll cover various functionalities such as user management, product management, order processing, and payment integration, showing how Kafka can enhance data streaming, real-time processing, and system scalability in these areas.

2.2 Producing messages

Let's take a closer look at our product management system. We aim to record all product price updates in Kafka, starting by creating a dedicated topic. Recording these price updates is crucial for maintaining an accurate history of pricing changes, enabling real-time monitoring, performing analytics, and making decisions based on current and historical pricing data. To begin, we'll initiate the process by creating a dedicated Kafka topic. Topics are similar to tables in databases in that we store a collection of data on a certain subject in topics. Following the usual naming conventions of using lowercase letters and separating components with dots for clarity and consistency, we name our topic `products.prices.changelog`. Here's how to create the topic:

```
$ kafka-topics.sh \
    --create \
    --topic products.prices.changelog \
    --partitions 1 \
    --replication-factor 1 \
    --bootstrap-server localhost:9092
Created topic products.prices.changelog.
```

We use the `kafka-topics.sh` command to manage our topics in Kafka. This script and many more, which we'll use in our examples, are shipped with Kafka itself. Here, we tell Kafka to create the topic `products.prices.changelog` (`--topic products.prices.changelog`) with the `--create` argument. First, we start with one partition (`--partitions 1`) and without replicating the data (`--replication-factor 1`) to keep it simple for now. Last, we specify the Kafka cluster we want to connect to. In our case, we use our local cluster, which by default listens on port `9092` (`--bootstrap-server localhost:9092`). The command confirms the successful creation of the topic. If we

get errors here, it's often because Kafka hasn't been started and thus is inaccessible or because the topic already exists.

> **NOTE** Confluent Kafka scripts don't have the .sh extension. For Windows users, .bat versions of these scripts are available; however, we strongly recommend using Windows Subsystem for Linux (WSL) for an improved experience.

So, now we have a place to store our price updates. Whenever a price for a product has changed, we produce a new message to that topic. For our practical exercise, we'll use the `kafka-console-producer.sh` command-line tool. The producer connects to Kafka, takes data from the command line, and sends it as messages to a topic (configurable via the `--topic` parameter). Let's write the message `coffee pads 10` into our topic `products.prices.changelog`:

```
$ echo "coffee pads 10" | kafka-console-producer.sh \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092
```

> **NOTE** The Kafka console producer doesn't print a confirmation message upon successfully sending a message. Because we're piping the output of `echo` directly into the producer, there's no interactive input mode, and no feedback is displayed.

> **NOTE** If you're a coffee lover, you might notice "coffee pad" in the code listings versus the more often used "coffee pod." While they do have their differences, both pad and pod result in coffee in our cups—and isn't that what matters?

2.3 Consuming messages

Our analysis component now needs to read this data. This component could analyze the effect of price changes on orders in real time, enabling timely adjustments to pricing strategies and inventory management. To retrieve the message we just sent, we'll initiate `kafka-console-consumer.sh`, an integral member of the Kafka command-line toolkit:

```
$ kafka-console-consumer.sh \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092
# Press Ctrl-C to cancel
Processed a total of 0 messages
```

When we start `kafka-console-consumer.sh`, it continues to run by default until we actively cancel it (e.g., with Ctrl-C). For `consumer`, we have to specify again which topic it should use (`--topic products.prices.changelog`).

Somewhat surprisingly, no message is displayed. This is because, by default, `kafka-console-consumer.sh` starts reading at the end of the topic and only prints new messages. To display already written data we have to use the flag `--from-beginning`:

```
$ kafka-console-consumer.sh \
    --topic products.prices.changelog \
    --from-beginning \
    --bootstrap-server localhost:9092
coffee pads 10
# Press Ctrl-C to cancel
Processed a total of 1 messages
```

This time, we see the message `coffee pads 10`! What happened? We used the `kafka-topics.sh` command to create the topic `products.prices.changelog` in Kafka and used the `kafka-console-producer.sh` command to produce the message `coffee pads 10`. We then read this message again with `kafka-console-consumer.sh`. This data flow is shown in figure 2.2. Without anything else, `kafka-console-consumer.sh` always starts reading at the end, which means that if we want to read all messages, we have to use the `--from-beginning` flag.

Figure 2.2 In our example, we produce data with the `kafka-console-producer.sh` command to be stored in the `products.prices.changelog` topic, and then we can read this data again with the `kafka-console-consumer.sh` command.

2.4 Consuming and producing messages in parallel

Interestingly, unlike many messaging systems, in Kafka, we have the flexibility to read messages multiple times as needed. This capability allows us to connect several independent consumers (representing separate systems) to the same topic, enabling them to access the data concurrently. For instance, we can have both an analysis system and an inventory management system consuming the price changes, as shown in figure 2.3. Moreover, there might be occasions where we need to analyze sales retrospectively, necessitating access to historical prices. In such cases, we can rerun the consumer multiple times, retrieving the same data each time for further evaluation.

However, we now want to see the current price in our inventory management system in such a way that the price is updated immediately when there's new data. To do this, we start `kafka-console-consumer.sh` in a terminal window to simulate our inventory management system. As soon as new data is available, the consumer fetches it from Kafka and displays it on the command line:

```
# Don't forget: use Ctrl-C to stop the consumer
$ kafka-console-consumer.sh \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092
```

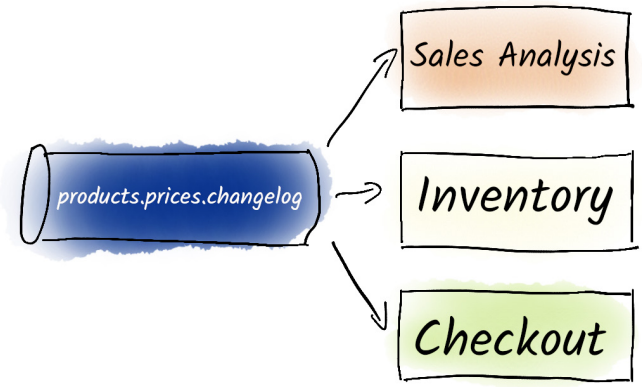


Figure 2.3 The price changes for our products can be consumed by multiple systems.

To simulate the price changes, we now start `kafka-console-producer.sh`. The command doesn't stop until we press Ctrl-D, sending the `EOF` (End of File) signal to the producer:

```
# Don't forget: use Ctrl-D to stop the producer
$ kafka-console-producer.sh \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092
```

The `kafka-console-producer.sh` command sends one message to Kafka per line we write. That means we can now type messages into the terminal with the producer:

```
# Producer window
> coffee pads 11
> coffee pads 12
> coffee pads 10
```

We should also see these promptly in the consumer window:

```
# Consumer window
coffee pads 11
coffee pads 12
coffee pads 10
```

Now, we also want to consume those messages with our sales analysis tool, which independently starts its consumer. We can simulate this by starting a `kafka-console-consumer.sh` in another terminal window, which displays all data from the beginning. Therefore, we're seeing `coffee pads 10` twice as this was also our first message produced at the beginning of this chapter:

```
# Consumer 2 window
$ kafka-console-consumer.sh \
    --topic products.prices.changelog \
```

```
    --from-beginning \
    --bootstrap-server localhost:9092
coffee pads 10
coffee pads 11
coffee pads 12
coffee pads 10
```

We see here that once data is written, it can be read in parallel by multiple consumers without the consumers having to talk to each other. Kafka keeps data for seven days by default, but we can also set it to not delete the data. This way, we can still start a consumer later that needs historical data.

Let's consider a scenario where price changes can be triggered by different departments that are responsible for different product categories. Kafka seamlessly handles this variability and efficiently processes data from multiple producers simultaneously, each representing a different product category, shown in figure 2.4. To demonstrate this capability, let's initiate another producer in a separate terminal window:

```
# Producer 2 window
$ kafka-console-producer.sh \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092
> pillow 30
> blanket 40
```

We see all the messages from all the producers show up in all our consumers in the order the messages were produced:

```
# Consumer 1 window
[...]
pillow 30
blanket 40
# Consumer 2 window
[...]
pillow 30
blanket 40
```

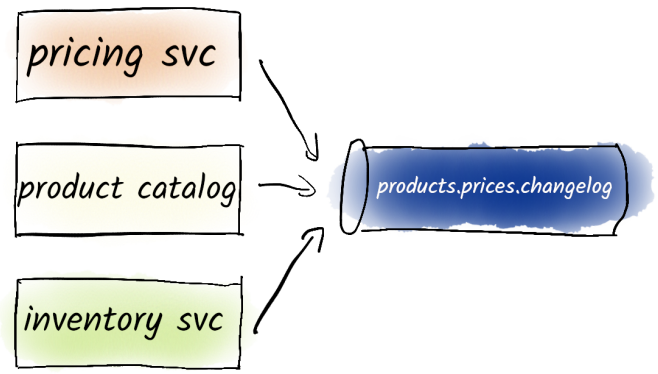


Figure 2.4 Multiple producers representing different departments are writing in parallel to the `products.prices.changelog` topic.

We've now successfully completed our first Kafka use case, which involved both writing and reading data to and from Kafka. Initially, we created a topic named `products.prices.changelog` using the `kafka-topics.sh` command-line tool, where we stored all the price changes for our products. Next, we used the `kafka-console-producer.sh` command to write data into this topic, capturing the price updates. Subsequently, we were able to read and display this data using the `kafka-console-consumer.sh` command. Furthermore, we advanced our exploration by producing data in parallel using multiple producers and simultaneously reading data using multiple consumers. By employing the `--from-beginning` flag in `kafka-console-consumer.sh`, we accessed historical data. Through this process, we've gained familiarity with both writing and reading data to and from Kafka by using its command-line tools.

After gaining this experience, we can now close all open terminals. We close producers with Ctrl-D and consumers with Ctrl-C. This example shouldn't hide the fact that Kafka is used wherever larger amounts of data are processed (in the order of several dozens of gigabytes per second).

2.5 Graphical user interfaces for Kafka

Kafka GUIs play a crucial role in simplifying the management and monitoring of Kafka clusters. For example, the open source Kafbat UI (https://github.com/kafbat/kafka-ui; shown in figure 2.5) or Kadeck (www.kadeck.com; proprietary license) provide intuitive interfaces for interacting with Kafka. These GUIs enable users to visualize data streams; monitor and manage topics, partitions, and consumer groups; and perform administrative tasks with ease. The Kafbat UI, in particular, offers a comprehensive view of Kafka clusters, simplifying the process of tracking message flow, inspecting audit logs, and configuring system settings. This accessibility enhances the overall user experience, making it easier for developers and administrators to ensure the efficient operation of their Kafka ecosystems.

> **WARNING** We strongly advise against using GUIs for writing data or modifying configurations in a production environment. Producing messages via a GUI in production can lead to inconsistencies, insufficient error handling, and poor scalability. Kafka is designed for system-to-system data exchange, not human interaction. Humans should only produce or consume data in production as administrators in absolute emergencies.

After getting an initial overview of Kafka in this chapter and going through a practical scenario, we'll go deeper in the next chapter and examine the Kafka architecture in more detail. We'll look at how Kafka messages are structured and how exactly they are organized into topics. In this context, we'll also look at the scalability and reliability of Kafka. We'll also learn more about producer, consumer, and the Kafka cluster itself.

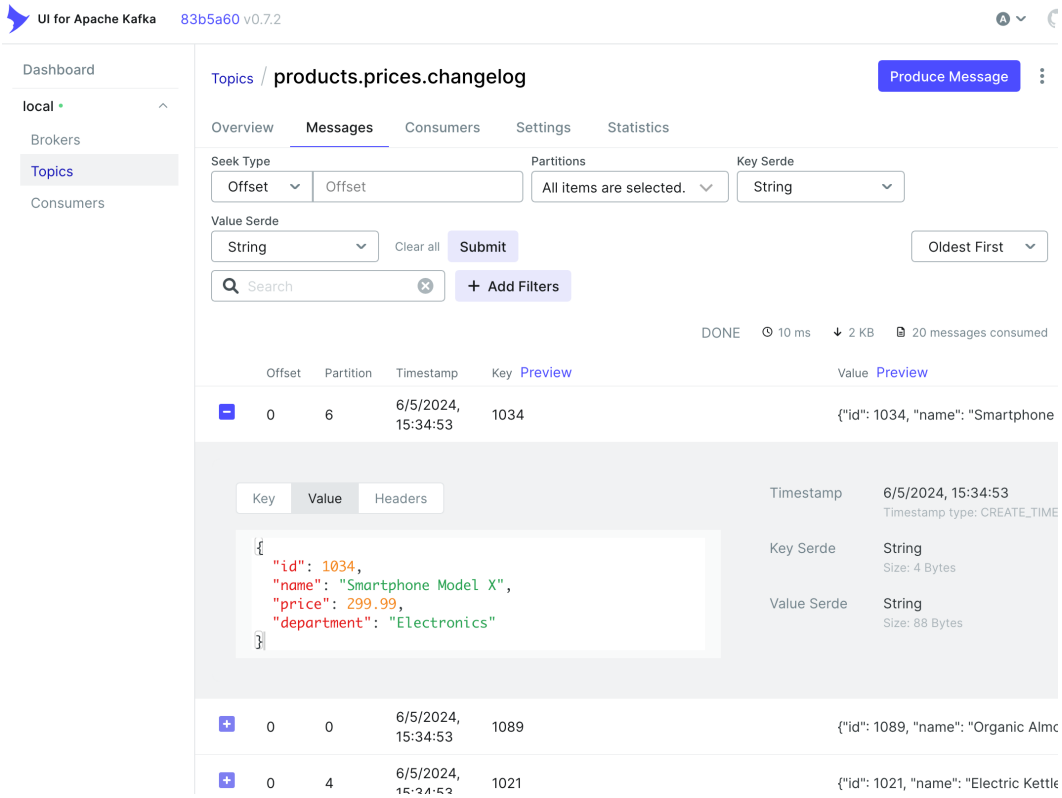


Figure 2.5 This Kafbat UI screenshot displays the messages within a specific topic, providing a clear view of the data being produced and consumed. This UI allows users to inspect individual messages, including their key, value, and timestamp, facilitating real-time monitoring and troubleshooting of data streams.

Summary

- Kafka includes many useful scripts for managing topics and producing or consuming messages.
- Kafka topics can be created with the `kafka-topics.sh` command.
- Messages can be produced with the `kafka-console-producer.sh` command.
- Topics can be consumed with the `kafka-console-consumer.sh` command.
- We can consume topics again from the beginning.
- Multiple consumers can consume topics independently and at the same time.
- Multiple producers can produce into topics in parallel.
- Kafka GUIs enable users to view real-time messages within a topic, displaying details such as message key, value, and timestamp.
- These GUIs aid in effective monitoring and troubleshooting of Kafka data streams.

Part 2

Concepts

In part 2, we dive into key concepts of Apache Kafka, focusing on how its architecture and core features empower efficient, scalable data streaming. This part provides a detailed examination of Kafka topics, messages, distributed logs, reliability, and performance optimization.

Chapter 3 explores Kafka topics and messages, including their structure, flow, and data formats. Chapter 4 examines Kafka as a distributed log, highlighting its partitioning and replication mechanisms while exploring its real-world applications. Chapter 5 discusses Kafka's reliability, focusing on acknowledgment settings, data durability, and fault tolerance, including its transactional capabilities. Chapter 6 focuses on performance optimization, covering partitioning strategies, producer and consumer tuning, and broker configurations.

3 Exploring Kafka topics and messages

This chapter covers

- Working with Kafka topics
- How topics structure the flow of data in Kafka
- Messages, the basic units of data in Kafka

In this chapter, we'll delve into the foundational elements of Apache Kafka: topics and the intricacies of messages. Kafka *topics* are the channels through which data is organized and distributed, while *messages* are the individual units of data that flow through these channels. We'll explore the various types of messages, their structures, and data formats, as well as understand how these elements contribute to efficient data streaming and processing. By the end of this chapter, you'll have a comprehensive understanding of how to effectively manage Kafka topics and the detailed composition of Kafka messages.

3.1 Topics

Kafka topics are key to organizing data flow, much like tables in a database. They help manage different types and amounts of information in an orderly way. Complex

data might be spread across multiple topics, so it's important to manage them efficiently for smooth streaming and processing.

3.1.1 Viewing topics

The creator of a Kafka topic is seldom its sole user. In scenarios where multiple users interact with the same topic, it becomes valuable for others to gain insights into the topic's general configuration. Understanding these configurations can enhance collaboration and ensure that all users are aligned on essential parameters governing the topic's behavior. For instance, let's take a closer look at the topic `products.prices.changelog` from the preceding chapter:

```
$ kafka-topics.sh \
    --describe \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog TopicId: GGYA9u_aRPSd0JRaGn2eBA
PartitionCount: 1 ReplicationFactor: 1 Configs:
  Topic: products.prices.changelog Partition: 0 Leader: 3 Replicas: 3
    Isr: 3 Elr: LastKnownElr:
```

We use the `kafka-topics.sh` command to manage our topics in Kafka. With the argument `--describe`, we can take a closer look at topics. The `--topic` and `--bootstrap-server` arguments are also familiar from the previous chapter and are used to select the topic `products.prices.changelog` (`--topic products.prices.changelog`) on our local Kafka cluster (`--bootstrap-server localhost:9092`).

Now let's take a look at the output. In the first line, we find general information about our `products.prices.changelog` topic. The field `TopicId` contains a unique identifier, which Kafka automatically creates for every topic. `PartitionCount` indicates the number of partitions in our topic and corresponds to the value set when the topic was created (`--partitions 1`). *Partitions* in Kafka serve as a means to parallelize data processing and enhance the scalability and throughput of the system by allowing multiple producers to write to different partitions concurrently and allowing consumers to read from them concurrently. The `ReplicationFactor` defines how often messages are stored redundantly. A replication factor of 1 means that messages are not replicated, so there is no failover. Both partitions and replication will be covered in detail in later chapters.

Further configuration settings for the topic can be found under `Configs`. In our example, the field is empty because we didn't change anything in the default configuration. `Configs` will also be covered in more detail in the course of the book. The last line contains information about the single partition and its replicas (only one here). If we had more than one partition, the output would be one line per partition.

To understand what the other numbers are all about, we have to take a short detour via the architecture of Kafka. So far, we've always talked about our Kafka cluster. What do we actually mean by this? Kafka consists of several components, which are shown in figure 3.1.

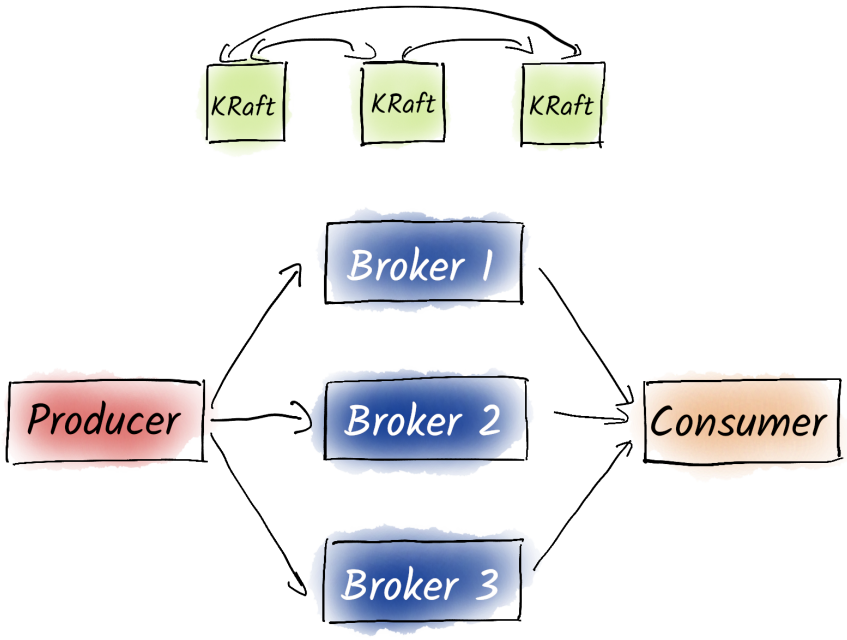


Figure 3.1 A typical Kafka environment consists of the Kafka cluster itself and the producers and consumers that write and read data to Kafka. This is joined by a coordination cluster based on Apache Kafka Raft (KRaft).

We learned about producers and consumers in the previous chapter. *Brokers* are responsible for processing and storing messages in the Kafka cluster. The coordination cluster is responsible for managing our brokers, but we'll ignore this for now.

The other numbers in the last line of the output (`Topic: products.prices .changelog Partition: 0 Leader: 3 Replicas: 3 Isr: 3, Elr:, astKnownElr:`) each refer to the ID of one of our three brokers. `Replicas` indicate on which brokers the partition is replicated. In this case, it's only available on the broker with ID 3. The abbreviation `ISR` stands for *in-sync replicas* and indicates which brokers are up-to-date for this partition. The field `Leader` indicates which broker has primary responsibility for the partition. Because we only have a `ReplicationFactor` of `1` and therefore the partition isn't replicated, the respective entries also only contain the ID of one of our three brokers. When creating a new topic, Kafka tries to distribute the partitions and replicas evenly among all brokers to ensure an even load distribution. Because we've only created one topic so far, the choice of broker is arbitrary. In our example, the broker with ID 3 was selected.

`ELR` stands for *eligible leader replicas* and contains the list of brokers that are also ISR eligible to become the leader for that partition if the current leader fails. `LastKnownElr` contains the list of previously eligible replicas that had an unclean shutdown. We'll explain the concepts of ISR and ELR in more detail in chapters 5 and 8.

> **NOTE** We'll omit some fields of the `--describe` command in the future to shorten the output if they aren't needed for explanation.

So far, we've ignored one important question. How do we know which topics exist in our Kafka cluster? For our example, the question is relatively easy to answer because we've only created the topic `products.prices.changelog` so far. In practice, a Kafka cluster can consist of many topics created by different people. To keep track of them, we can use the command `kafka-topics.sh` to display all topics in our Kafka cluster:

```
$ kafka-topics.sh \
    --list \
    --bootstrap-server localhost:9092
__consumer_offsets
products.prices.changelog
```

For this, we use the `--list` argument and again specify a Kafka broker (`--bootstrap-server localhost:9092`). It doesn't matter which broker actually manages the topic. So, alternatively, we could talk to our other two brokers (`--bootstrap-server localhost:9093` and `--bootstrap-server localhost:9094`, respectively). As a result, the topic `__consumer_offsets` stands out. This topic was created automatically by Kafka, which can be seen by the double underscores at the beginning of its name. This topic stores the current reading position for each consumer. We can use *offsets* as a kind of bookmark for our consumers. In fact, we already used offsets in our first example, as displayed again here:

```
# Window Consumer 2
$ kafka-console-consumer.sh \
    --topic products.prices.changelog \
    --from-beginning \
    --bootstrap-server localhost:9092
coffee pads 10
coffee pads 11
coffee pads 12
coffee pads 10
```

By default, the `kafka-console-consumer.sh` starts reading from the end of a topic, that is, it reads only new messages. By using the flag `--from-beginning`, we start reading from the earliest offset instead, which also reads all messages already produced.

> **WARNING** While theoretically the manual creation of topics with double underscores is possible, we strongly advise against doing so. Kafka reserves topic names with double underscores for internal topics and special purposes. Using such a naming convention for your own topics can lead to conflicts with Kafka's internal topics, future compatibility problems, and administrative confusion.

3.1.2 Create, customize, and delete topics

Now that we've learned how to view topics and their properties, let's take a closer look at how to create and customize topics. To start fresh, we first delete our topic `products.prices.changelog` by using the command `kafka-topics.sh` with the argument `--delete`:

```
$ kafka-topics.sh \
    --delete \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092
```

If we now display the topics in our Kafka cluster, we notice that the topic `products.prices.changelog` is no longer present:

```
$ kafka-topics.sh \
    --list \
    --bootstrap-server localhost:9092
__consumer_offsets
```

> **NOTE** The offsets for the deleted topic inside the `__consumer_offsets` topic have been cleaned up after deleting the `products.prices.changelog` topic.

After our Kafka cluster is empty again, except for the `__consumer_offsets` topic, we can re-create our `products.prices.changelog` topic:

```
$ kafka-topics.sh \
    --create \
    --topic products.prices.changelog \
    --replication-factor 2 \
    --partitions 2 \
    --bootstrap-server localhost:9092
Created topic procuts.prices.changelog.
```

We again use the well-known command `kafka-topics.sh` in combination with the argument `--create`. However, the `ReplicationFactor` and the number of partitions are interesting. Compared to our introductory example, we choose a value of `2` instead of `1`, which on the one side, increases reliability (replication factor), and on the other side, increases parallel processing (partitions). We'll dive into all the details about reliability and performance later on. Now, let's take a closer look at our new topic `products.prices.changelog` using `--describe`:

```
$ kafka-topics.sh \
    --describe \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog TopicId: 2VvA24HxRwqhF-znSY1tAQ
PartitionCount: 2 ReplicationFactor: 2 Configs:
  Topic: products.prices.changelog Partition: 0 Leader: 3
```

```
    Replicas: 3,2 Isr: 3,2
Topic: products.prices.changelog Partition: 1 Leader: 1
    Replicas: 1,3 Isr: 1,3
```

Not surprisingly, we now see a `PartitionCount` and a `ReplicationFactor` of `2`, and a new `TopicId`. More interesting are the following two lines. Each line represents a partition, and the partitions are numbered starting with 0. Partition 0 is on the brokers with the IDs 3 and 2 (`Replicas: 3,2`), whereby broker 3 takes the role of the leader (`Leader: 3`) and thus the main responsibility for partition 0. Partition 1 is also located on Broker 3 and additionally on Broker 1 (`Replicas: 1,3`), whereby Broker 1 has the role of the leader this time (`Leader: 1`), as shown in figure 3.2.

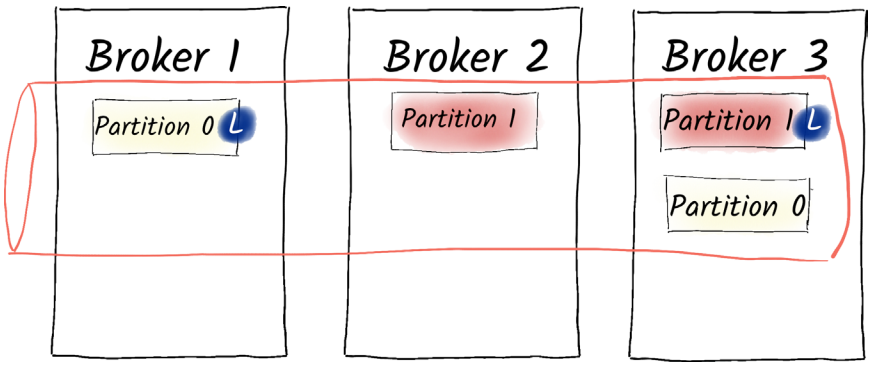


Figure 3.2 The replicas and partitions of the `product.prices.changelog` topic are distributed among our brokers. Broker 3 is the leader of partition 0, and broker 1 is the leader of partition 1.

The distribution of the individual partitions and replicas isn't pure chance because Kafka tries to distribute the load evenly among all brokers. No matter how many times we delete and re-create the topic in our small example, the same broker will never be leader of both partitions at the same time. However, which broker leads which partition may change. The distribution of replicas among brokers is similar. With three brokers, all four replicas wouldn't be distributed on only two brokers. Only the exact distribution among the brokers can change.

But what happens if we find out afterward that we need to change the number of partitions and replicas because, for example, performance is no longer sufficient or reliability is insufficient? Deleting and re-creating the topic each time is, apart from being impractical, not always possible, because we would lose all the data. Fortunately, Kafka has a solution for this as well. Let's first increase the number of partitions to three:

```
$ kafka-topics.sh \
    --alter \
    --topic products.prices.changelog \
```

```
    --partitions 3 \
    --bootstrap-server localhost:9092
```

To customize the topic, we use the `kafka-topics.sh` command with the `--alter` argument. We select the topic `products.prices.changelog` (`--topic products.prices.changelog`) and set the number of partitions to `3` (`--partition 3`). After running the command, we can take a closer look at the changes:

```
$ kafka-topics.sh \
    --describe \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog PartitionCount: 3 ReplicationFactor: 2
Configs:
  Topic: products.prices.changelog Partition: 0 Leader: 3
    Replicas: 3,2 Isr: 3,2
  Topic: products.prices.changelog Partition: 1 Leader: 1
    Replicas: 1,3 Isr: 1,3
  Topic: products.prices.changelog Partition: 2 Leader: 2
    Replicas: 2,3 Isr: 2,3
```

We can see that the `PartitionCount` has been increased to three. We also see a new line at the end of the output, which contains information about the newly created partition 2. The selection of leaders and replicas of the already-existing partitions isn't changed by `--alter`. In our example, Broker 2 was selected as the leader for the new partition, and Broker 3 was selected as another replica, as shown in figure 3.3.

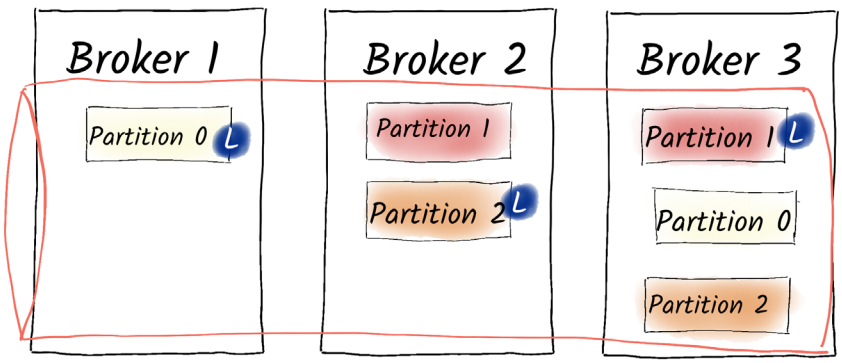


Figure 3.3 The replicas and partitions of the `product.prices.changelog` topic after we increased the partitions to three. Broker 2 is now the leader of the additional partition 2.

Because we already have another topic, the `__consumer_offsets` topic, which contains 50 partitions, one broker was assigned as leader for 17 partitions, while the other two brokers were assigned as leaders for 16 partitions each. Due to this existing distribution, the first two partitions of our `products.prices.changelog` topic were assigned to

the brokers with fewer leader partitions. However, any of the brokers could have been assigned as the leader for partition 3. If we had increased the number of partitions to five (adding three partitions), each broker would have been assigned as leader for one additional partition, resulting in a more balanced distribution.

> **WARNING** Reducing the number of partitions in Kafka isn't possible as it would result in data loss. The inability to move data between partitions is key to Kafka's fundamental principle of maintaining message order within partitions, a concept we'll delve into further in chapter 6.

If we want to change the replication factor, we have to either use the `--replica-assignment` argument in the `kafka-topics.sh` command or resort to the `kafka-reassign-partitons.sh` command. Additionally, it's important to remember that the replication factor can never be higher than the number of brokers.

> **NOTE** If you're only looking to reassign partitions for a single topic, the `kafka-topics.sh` command might do the trick. However, for more intricate reassignments, especially when dealing with multiple topics simultaneously, we recommend using the `kafka-reassign-partitions.sh` tool. This tool offers additional flexibility, allowing for complex reassignments and even the option to automatically redistribute assignments for multiple topics concurrently.

3.2 Messages

In the previous section, we got to know topics. The next question is, what do we produce into those topics? In the first chapter, we sent strings to Kafka and received strings from Kafka. Kafka is agnostic about data types, meaning it internally operates solely with byte arrays. A byte array is simply a sequence of bytes, which are the basic units of data storage in computers. Because Kafka works with byte arrays, it can handle all kinds of messages, regardless of their format or structure, making it highly flexible and capable of handling diverse types of data.

The only important thing to remember is that Kafka is optimized to handle many small messages. The maximum size of messages is set to 1 MB. While we can adjust this, we should avoid it because performance and resource usage can suffer greatly from larger messages. But what exactly does *many small messages* mean? In the extreme, it means that companies such as LinkedIn processed about 7 trillion (not billion) messages a day with Kafka in 2019. Note that LinkedIn didn't have just one Kafka cluster, but about 100 Kafka clusters at that time. But even smaller installations allow us to process many messages very quickly. We should just make sure that our messages stay below the 1 MB limit. For example, it makes little sense to use Kafka to transfer large files between systems. For such use cases, other solutions are preferable.

3.2.1 Message types

What data do we usually write to Kafka? This depends very much on the use case, but from experience, four types of messages are common: states, deltas, events, and

commands. In practice, most systems will use a mix of these message types. For example, an ERP system might emit an *event* indicating a product price change due to a promotion. A separate service monitoring promotions could then send a *command* to a notification service to inform customers about the promotion. This notification service operates independently and can be used by various other services to send notifications. We'll briefly describe each of these types in the following subsections.

States

We describe in the message the current state of a subject or object. They contain complete information about an object. The question behind it describes what we're interested in, such as how the object to be observed looks right now or historically. Additionally, if we're only interested in the latest state for an object, we can use log compaction together with keys to reduce the amount of storage needed. This will be covered in chapter 10. Let's imagine that we also want to store the current stock next to the price of our product in the `products.changelog` topic:

```
# One message per line
{"id": 123, "name": "coffee pad", "price": "10", "stock": 101010}
{"id": 234, "name": "cola", "price": "2", "stock": 52}
{"id": 345, "name": "energy drink", "price": "3", "stock": 42}
```

Deltas

Often it's not necessary to rewrite the entire state for every small change. In these cases, it may be useful to put only the change in state in the message. In terms of our inventory management system, we can think of a `products.stocks.changes` topic in which we collect the changes of the stock of our products, which are either triggered when a customer is buying something or when we receive a subsequent delivery of one of our products:

```
# One message per line
{"id": 123, "stock": 1000}
{"id": 234, "price": 2}
{"id": 345, "stock": 60, "price": 3}
{"id": 234, "name": "cola zero"}
```

We can save a significant amount of data volume by avoiding the frequent repetition of unchanged values. In addition, it's easier to find out what has changed since the last message with the help of deltas.

Events

Events describe what happened by adding *context* to a message. For example, we can describe that we got a new delivery of a product or that a customer bought something. Logs are a special kind of event. Kafka is widely used for collecting logs, providing a reliable and scalable solution for aggregating and processing log data from various sources. Kafka itself shares many characteristics with logs, a subject that will be covered in detail in the next chapter.

Often we use states and deltas as raw data, based on which, we can produce our events in another topic. If we think back to our first steps with Kafka, we can consider our `products.prices.changelog` topic as an event topic where we could also write the following messages:

```
# One message per line
{"id": 123, "event": "vat_adjustment", "payload": {"price": 2.19}}
{"id": 234, "event": "data_correction", "payload": {"name": "Coke Zero"}}
{"id": 345, "event": "storehouse_delivery", "payload": {"stock": 100"}}
{"id": 234, "event": "promotion_start", "payload": {"price": 1.99}}
```

If we look at these messages with what we know now, we can recognize the events in them. This is where we answer the "what just happened?" question.

NOTE Kafka records always contain a timestamp.

COMMANDS

Sometimes, we need to instruct another system to perform a specific action. Unlike events, which describe something that has happened, commands request an action to be performed in the future. For instance, a message might direct a notification service to inform a customer about a promotion:

```
# One message per line
{"command": "notify_customer", "customer_id": 123456,
"message": "The promotion for Coke Zero started. It's now 1.99"}
{"command": "notify_customer", "customer_id": 123457,
"message": "The promotion for Coke Zero started. It's now 1.99"}
{"command": "notify_customer", "customer_id": 123458,
"message": "The promotion for Coke Zero started. It's now 1.99"}
```

This introduces more coupling, as we rely on another system to execute the command. In contrast to events, where the sender doesn't concern itself with who the recipient is or what happens next, commands require a response or action from the recipient system.

3.2.2 Data formats

Now that we have an idea of what types of messages we want to store in Kafka, we should think about what data format we want to use. Kafka processes data solely as byte arrays and doesn't interpret the data we send, which means there are no built-in functions to manipulate it. This design choice is a key factor in Kafka's high performance.

One of the most popular data formats in the Kafka world is *JSON*. It has the advantage that both humans and machines can easily read this data format. But similar to XML, the amount of data can inflate a lot, and we can't get far even when using message compression. If minimizing the data size is important, we can use binary formats such as *Google Protocol Buffers* (*Protobuf;* https://developers.google.com/protocol-buffers) or the *Apache Avro* (http://avro.apache.org/), which is widely used in the Kafka community. In our experience, almost every topic needs to be read by a human at

least once in its life, which is more cumbersome with binary data formats. Selecting a consistent data format holds greater importance than delving into the specifics of the format itself. It's advisable to avoid using a different data format for each topic. The significance of this decision increases as the number of topics in a Kafka cluster grows and the cluster ages. In this context, documenting data formats becomes paramount. For binary data formats, a *schema* is imperative for interpretation. Essentially, a schema acts as a blueprint or set of rules defining the structure and format of data. This applies not only to binary formats but also to JSON (using *JSON Schema* [http://json-schema.org/]) and XML (with *XSD* [www.w3schools.com/xml/schema_intro.asp] or *Document Type Definition* [DTD; www.w3schools.com/xml/xml_dtd_intro.asp]). The real strength lies in the use of schemas, ensuring uniformity across topics and facilitating seamless interpretation within a specified data format. Schema management and schemas themselves will be discussed in detail later in the book.

NOTE You may have noticed that the message format appears different in our deltas example, which might seem contradictory at first. However, this difference is perfectly valid because a schema can include optional fields. Depending on the use case, these optional fields can either be included or omitted, as demonstrated in our example.

3.2.3 Message structure

Upon closer inspection of messages in Kafka, it's evident that they are more than a singular entity. In Kafka, a message comprises the following components: an optional key, value, optional custom headers, and a timestamp (shown in figure 3.4), each fulfilling a unique function in data exchange and processing within the Kafka ecosystem. If we don't explicitly specify keys using `kafka-console-producer.sh`, messages will be produced without keys.

CAUTION Headers should be viewed similarly to HTTP headers, as they are meant for technical metadata, such as sending tracing IDs. Business data should never be sent in headers, and it's perfectly fine not to use headers at all.

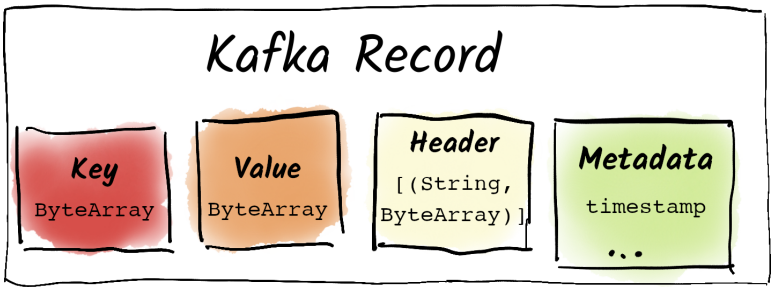


Figure 3.4 A Kafka message or record consists of a timestamp, a value, an optional key, and optional custom headers to represent metadata.

In Kafka, a *key* is an optional attribute associated with a message, serving to categorize messages. Keys influence how data is distributed across partitions. Messages with the same key are usually meant to belong together (more about this later). Meanwhile, the value represents the primary payload of the message, carrying the actual information intended for consumption by consumers. Topics themselves are agnostic of keys, meaning that we don't need to specify anything extra when creating a topic intended to contain messages with keys.

A key could be the name of the product, for example. Let's first create the `products.prices.changelog.keys` topic:

```
$ kafka-topics.sh \
    --create \
    --topic products.prices.changelog.keys \
    --replication-factor 2 \
    --partitions 2 \
    --bootstrap-server localhost:9092
Created products.prices.changelog.keys.
```

Now, let's start `kafka-console-producer.sh` so that we can produce messages with keys:

```
# Previously we have created the topic products.prices.changelog.keys
$ kafka-console-producer.sh \
    --topic products.prices.changelog.keys \
    --property parse.key=true \
    --property key.separator=: \
    --bootstrap-server localhost:9092
```

In addition to the usual specifications, such as the topic and Kafka cluster, we pass two further parameters. With `--property parse.key=true`, we activate the parsing of keys in `kafka-console-producer.sh`, and with `--property key.separator=:`, we set the colon (`:`) as separator between keys and values. Now let's produce some messages:

```
# Producer window:
> coffee pads:10
> cola:2
> coffee pads:11
> coffee pads:12
```

If we now run `kafka-console-consumer.sh` as before, we only see prices because the consumer doesn't display keys by default:

```
$ kafka-console-consumer.sh \
    --from-beginning \
    --topic products.prices.changelog.keys \
    --bootstrap-server localhost:9092
10
2
11
```

```
12
Processed a total of 4 messages
```

We cancel the command again with Ctrl-C. To see the keys, we have to switch this on explicitly by adding `--property print.key=true` to the command:

```
$ kafka-console-consumer.sh \
    --from-beginning \
    --topic products.prices.changelog.keys \
    --property print.key=true \
    --property key.separator=":" \
    --bootstrap-server localhost:9092
coffee pads:10
cola:2
coffee pads:11
coffee pads:12
Processed a total of 4 messages
```

Now the consumer shows us the messages, including the key. But what do we need keys for? In key-value databases, this is easy to explain. We always store data to a concrete key, under which we can retrieve the written data again or manipulate it. In Kafka, this isn't so obvious. First, the key is optional, and as we can see from the output of `kafka-console-consumer.sh`, old messages with the same key aren't deleted.

We'll learn about Kafka as a distributed log in the next chapter and realize that usually topics consist of more than one partition. We can use keys to ensure that messages consistently land in the same partition, as Kafka uses keys to determine the partition to which a message is produced. We'll delve into this concept further in a later chapter. Because Kafka guarantees the order of messages only within one partition, if we created our topic `products.prices.changelog` with more than one partition, we couldn't guarantee that the message `coffee pads 10` would be read before the message `coffee pads 11`. This would lead to wrong prices in our shop and—worse—it could lead to inconsistencies between different systems depending on the order in which messages have been read. If we now created the topic `products.prices.changelog.keys` with more than one partition, we would use keys to guarantee that the messages for `coffee pads` would always arrive in the correct sequence. But we would have no guarantee regarding which of the two products would be read first:

```
cola:2
coffee pads:10
coffee pads:11
coffee pads:12
```

In the vast majority of cases, we can accept that the messages for different keys aren't in the correct order. It's usually sufficient to be able to guarantee that the messages for a key are sorted correctly. If we don't specify a key, messages are distributed across different partitions in a round-robin fashion.

NOTE The properties to handle keys are set on the client and not on the broker or topic.

Summary

- Kafka organizes data in topics.
- Topics can be distributed among partitions for increased performance.
- Topics can be replicated among different brokers to improve reliability.
- Topics can be created, viewed, altered, and deleted with the `kafka-topics.sh` script.
- The number of partitions of a topic can never be decreased.
- Partitions can be reassigned among brokers to redistribute load.
- Complex partition reassignments can be performed with the `kafka-reassign-partitions.sh` script.
- Kafka is optimized to exchange many (trillions of) small messages of 1 MB or less.
- Kafka messages can be classified into states, deltas, events, and commands.
- States contain the complete information about an object.
- Deltas consist only of the changes and are therefore very data-efficient, but a single delta is often not very useful. They require either a context or a complete state.
- Events add context to a message and describe a business event that happened.
- Commands are used to instruct other systems to perform actions.
- Data formats and schemas play a crucial role in Kafka to ensure consistency.
- Messages in Kafka consist of technical metadata, including a timestamp, optional custom headers (metadata), an optional key, and a value, which is the main payload.
- Messages with the same key are produced to the same partition, so the order of those messages is guaranteed for a single producer.
- Keys can also be used for log compaction to clean up deprecated data.

4 Kafka as a distributed log

This chapter covers

- The uses and properties of logs
- Considering Kafka as a log
- Kafka as a distributed system: partitions and replication
- Taking a closer look at Kafka cluster components
- Kafka in corporate use

In this chapter, we'll explore the concept of logs and how Kafka operates using a log-based approach. We'll examine Kafka's architecture as a distributed system and delve into its core components. Finally, we'll discuss the practical applications of Kafka in a corporate environment, highlighting its benefits for enterprise data management.

4.1 Logs

We like to describe Kafka as a log. Although the term *log* is new to many in the context of Kafka, we encounter logs all the time in everyday life in the world of IT.

4.1.1 What exactly is a log?

Our operating systems produce system logs. If we're responsible for these systems, we use the logs regularly to check the status of the systems and, in the event of errors, to understand how these errors could have occurred. We may even use log monitoring systems that sound alarms when certain events occur and, if necessary (but hopefully not), wake us up in the middle of the night so we can respond to the errors.

The same applies to our applications and services. They produce log events in which we can see what is happening, that is, whether errors are occurring or whether all systems are running properly. We use both, system logs and application logs, to analyze errors and monitor the state of the corresponding system. Here, logs are useful and essential, but not the core of our systems.

The situation is different for database systems. Here, the commit log is hidden deep in the system, but it's one of the main components of the system. Databases store any changes to data in the commit log. If data is added, changed, or deleted, the database first writes this change to the commit log and only then to the corresponding tables. If the database fails, the commit log guarantees that we can restore it to its last clean state without losing any data. This also applies to replication, such as in a clustered database setup. Often this replication is based on keeping the commit log up-to-date between cluster units, and each server then builds the tables independently based on the data in the replicated commit log.

As different as these logs may be, they all answer the "What happened?" question. The system and application logs answer these questions for the operator of the systems, and the commit logs in databases answer this question for the other servers in the database cluster or even for themselves if the system crashes or other errors occur.

Thinking back to our example from the first chapter, we can also see log-like data structures. For example, here follows the content of our topic `products.prices.changelog`:

```
coffee pads 10
coffee pads 11
coffee pads 12
coffee pads 10
```

The reason for the similarities is simple. Kafka is based on logs as the central data structure. Unlike databases, Kafka doesn't hide the log, but the log becomes the core element of our system. Our data is stored in logs in Kafka, and we can take the same patterns and insights from the world of system, application, and database logs and apply them to Kafka!

4.1.2 Basic properties of a log

We also use logs very often outside of IT. Probably the simplest example is a diary. If we start at the front of the diary and write a page every day, for example, without leaving

any space in between, using a ballpoint pen instead of a pencil, we recognize the characteristics of a log.

- *Order and sorting*—Messages in a log are sorted by time. The oldest message is at the beginning of the log, and the newest message is at the end. It's the same in our diary. We know (if we don't leave any pages blank) that the entry on page 10 is older than the entry on page 11.
- *Writing and reading direction*—We always write new entries at the end of the log. When we read the log, we usually start on one page and read the older entries first and then the newer ones. Of course we can scroll back in the log, but still, the natural reading direction is from old to new.
- *Immutability*—Once we've written an entry, we can't easily change or remove it. In the diary, this is still possible by adding annotations to pages, crossing out words, or tearing out pages. But at least these changes are usually noticeable.

Another interesting property of logs is that we can easily understand how the part of the world described by the log has changed over time, and thus we can also find out what this part of the world looked like at a certain point in time.

We often look at system or application logs when questioning how a certain error happened. If we read the entries in the log chronologically, we can often understand what has changed over time and thus find a clue as to what went wrong. The same is true for databases. To find out what a table looked like at a particular time, we start at the very beginning in the commit log, working our way forward entry by entry, seeing how the table has changed over time and why we see exactly the entries in the table that it shows us.

In addition, logs allow us to travel back in time! If we want to restore the state of our database to the day before yesterday, we can go through the log from the beginning to the entries of the day before yesterday and see how the database world looked back then.

With this knowledge, we can define the log a bit more formally. A log is a list of events on which we define the following two operations (see the visual representation of these operations in figure 4.1):

- *Write operation*—We add new entries to the end of the list.
- *Read operation*—We start at a particular entry and read from old to new, usually to the end of the log.

Most of the time, our logs are too large and too long to read in one piece. Application logs can be several gigabytes in size. So how do we remember which entries we've already read and which we haven't yet? We number our entries. The first entry gets the position 0, the second the position 1, and so on.

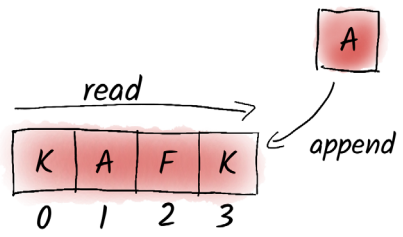


Figure 4.1 A log is a sequential list where we add elements at the end and read them from a specific position (offset). For example, we read from offset 0, then choose to read from offset 4, and so on.

In Kafka, this position is the offset, which we already learned about in the previous chapter when we wanted to read our topic from the beginning. The offset does two things. It shows us where a message is in the log, like a page number, and also points to the entry we want to read next, similar to remembering a page number in a book. Just as we start a book from the beginning (offset 0), read some pages, and remember the page (or offset) for the next read, Kafka uses something like bookmarks.

> **NOTE** Kafka brokers automatically assign an offset to a record at the moment it's written to a topic, ensuring precise positioning within the log.

The systems reading the logs can store these offsets in their memory (RAM), but that's not very reliable. If the system fails, it might have to start reading from the beginning. Instead, Kafka helps us reliably remember these offsets by storing them inside the `__consumer_offsets` topic, which we encountered in the previous chapter when listing all the topics in our cluster. This means Kafka not only keeps track of the logs but also provides consumers the means to manage their offsets and thus their progress.

4.1.3 Kafka as a log

Because logs in IT systems usually don't have a static number of entries, but constantly add new entries, they also usually don't have an end. In Kafka, we note the offset we want to read next and continuously query the Kafka cluster to see if there are new entries. If there are new entries, we get those entries back and note the next offset we want to read. If there are no new entries, then we don't get any data, and we don't have to adjust our offset.

The interesting thing about an offset is that it only describes the position of a message, but not its content. Just like page numbers in a book, it's a consecutive numbering without any further meaning. This is important to understand because in logs, the offset is the only way to address data. We can't access concrete elements in a log.

This has a huge effect on the architecture of our services that use Kafka. We shouldn't use Kafka to answer queries about data in an ad hoc fashion. We also shouldn't use Kafka to access data with a specific key in a manner similar to a key-value store. Any of these operations would possibly provoke a search over all the data in the log.

Logs, or Kafka, aren't the cure-all for replacing all of our databases, caches, and analytics tools (with one system), but Kafka can help us organize and share data between systems more effectively across our enterprise, as shown in figure 4.2.

Rather than misusing Kafka as a central database for our services, we use it as a central data hub, while choosing the most suitable technology within each service to store data according to our specific use cases. For example, if we want to build a search service over our products, we write the data from Kafka to a search engine such as *Elasticsearch*. If we want to evaluate data ad hoc, we could use a relational database such as *PostgreSQL* to do this and write data from Kafka to the database.

We've learned about Kafka as a log in the last few pages. We know that a log is a list in which we append data to the back for writing and read from old to new. Once the data is written in the log, we don't change or delete the data. One reason logs are used

in many places is that this is one of the simplest data structures and they don't contain any magic. Instead of storing the current state of the world as in a database, logs usually store the history of events from the beginning. It's important not to be tempted to think of logs as a panacea and to misuse Kafka as a database. We'll learn about Kafka as the core of a streaming platform in a later chapter and see how Kafka can best interact with other systems in our IT landscape.

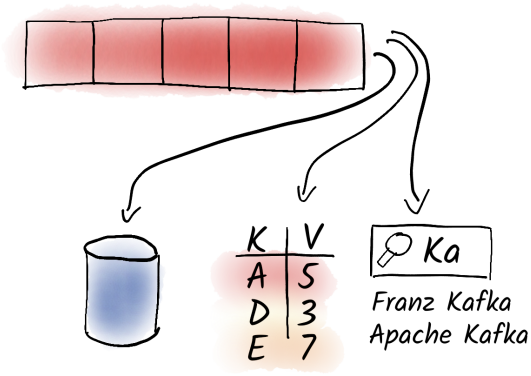


Figure 4.2 A log is a perfect data structure to exchange data between systems. Typically, we don't work directly with the data in the log, but store it in a data format best suited for our particular use case. For example, we can use relational databases to perform complex queries over our data. To access prepared data quickly, we can use an in-memory key-value store such as *Redis*, and to provide a search function over the data in the log, we can use a search engine.

4.2 Kafka as a distributed system

Looking at Kafka only as a log already gives us some insights, but this isn't enough to really understand Kafka. We have to go one step further and understand Kafka as a *distributed log*. Because it's difficult for us to achieve the necessary speed, scalability, and resilience with just one log, we need to learn how to efficiently distribute logs across multiple servers.

Computer systems are relatively unreliable, but mass-produced servers are fairly cheap. So instead of running Kafka on expensive, specialized hardware (e.g., mainframes or special-purpose hardware), we typically run Kafka on off-the-shelf systems and compensate for the increased vulnerability to failure by not running Kafka on just one machine. Instead of avoiding errors at all costs, we learn to accept errors and plan our systems accordingly. Practice proves us right in this. When we run a single system, reliability is usually good enough. However, hardware failures in data centers are a common concern. For example, the company Backblaze keeps statistics on the reliability of its deployed hard disks (www.backblaze.com/b2/hard-drive-test-data.html) and found that the hard disks used have an annual probability of failure of about 1%. In computer science, we can scale systems either vertically or horizontally as visualized in figure 4.3.

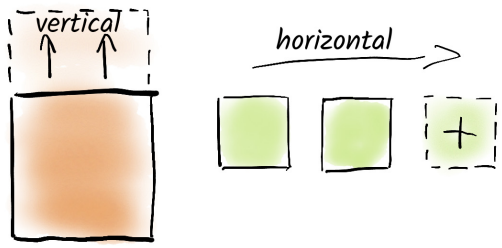


Figure 4.3 Scaling vertically means adding more resources to a single instance. Scaling horizontally means adding more instances to a system.

When we scale a system vertically, we get more powerful systems. Instead

of the database server with eight CPU cores and 16 GB RAM, we get the server with 64 cores and 256 GB RAM, hoping to achieve better performance. This is an easy way to increase the performance of our system, and it works very well for a while. Unfortunately, our reliability doesn't improve. Most of the time, when we replace a server with a stronger one, the probability of failure doesn't decrease significantly.

Horizontal scaling means that instead of buying bigger and more powerful servers, we use multiple servers instead of one. This is very tempting because we can scale seemingly endlessly. When our servers are at capacity, we simply add another one. We hope to see a significant increase in performance from this parallelization and also improved reliability and data durability from the increased redundancy.

Unfortunately, this is often thought of too simply. Distributing a system across multiple servers is anything but simple. The software must be able to coordinate. Someone has to decide which server handles which request. We have to think about what happens if a server goes down. How do we determine that it has stopped working or, worse, is responding to requests incorrectly? How do we replace the failed servers?

The fact that these aren't just theoretical problems, but that we as developers and operators of distributed systems have to think about them seriously is also shown by the regular failures even at the large IT corporations such as Google, Amazon, Netflix, and others. Working with distributed systems is often challenging, especially for people who have previously used primarily the vertical way of scaling, and requires a different mindset.

Instead of assuming that our systems are reliable, we not only accept errors but also build them into our everyday work to learn how to handle them better. We don't just accept these assumptions at the time of operation, but start thinking about how failures will affect us during system design. A very important point here is to design and develop the distributed systems as simply as possible. While Kafka is a complex software system, the basic concepts and underlying technologies are kept as simple as possible. For example, the log itself is one of the simplest data structures for storing large amounts of data. A log is easy to replicate (copy message by message) and to split (take multiple logs instead of one).

4.2.1 Partitioning and keys

Among Kafka's design goals from the beginning was to be able to handle huge amounts of data and process these data volumes at a very high speed. Moreover, in numerous scenarios, it's crucial for us to ensure the preservation and integrity of our data, avoiding both loss and corruption. Therefore, we can configure Kafka to reliably write messages and maintain message order. In the rest of this chapter, we'll learn the basics of how Kafka achieves exactly this performance and reliability.

We just discussed that one of the hopes of spreading a system across multiple servers is improved performance. Simply talking about improved performance when scaling horizontally is too short-sighted. Most of the time, initially, the speed at which messages

are processed actually decreases when we start scaling horizontally. It's like multi-core processors. If a task can't be parallelized, it does no good to have more cores available. That's why we don't just talk about improved performance when we scale horizontally, but we also talk about parallelization. We can use it to process more messages per unit of time than a single server could. For this to happen, however, the way we process the messages must be parallelizable.

The simplest way to divide data among multiple subsystems is to partition the data, also known as *sharding* in database systems. For example, let's imagine that we want to build a worldwide cab exchange. We could now store all data on all trips in a central database. This gives us all kinds of advantages. We can identify common movement patterns, optimize our operations, and thus improve our services. But if our service is successful, we'll receive and process much more data than a single server is capable of handling. As a result, everything will slow down, and our customers will go to our competitors. However, when we take a closer look at the data, we realize that it's not so important for us to keep all the data on all the trips in all the cities in one central database. It's sufficient for us to keep data from one city together, but it's not important to us that the order of the data across cities is correct, for example. This also allows us to scale horizontally more easily. We can start with one server, and as we get more and more customers, we can add more servers and offload some cities to them.

We can also proceed similarly in our online store example from the previous chapters. We don't really need all the data for all the products in a single log. It's actually sufficient for us to keep the data for one product in one log. If the order of data from different products isn't correct, it's not dramatic. The most important thing is that the data for a single product is sorted correctly, as shown in figure 4.4.

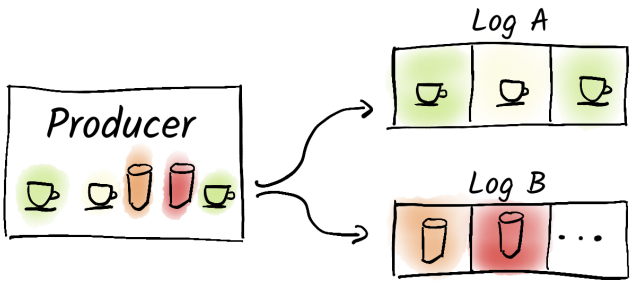


Figure 4.4 Log A holds all the data for coffee pads, and log B holds all the data for cola.

In Kafka, we do the same. Instead of writing all data of a topic into a log, we divide this topic into several partitions. When we create a topic, we have to specify how many partitions we want to have. Now let's create a topic with multiple partitions. For this, we use our well-known `kafka-topics.sh` command and only increase the number of partitions to two (`--partitions`):

```
$ kafka-topics.sh \
    --create \
    --topic products.prices.changelog.multi-partitions \
    --partitions 2 \
    --replication-factor 1 \
    --bootstrap-server localhost:9092
Created topic products.prices.changelog.multi-partitions.
```

Now let's produce some data into this topic by using `kafka-console-producer.sh`, as mentioned earlier:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog.multi-partitions \
    --bootstrap-server localhost:9092
> coffee pads 10
> cola 2
> energy drink 3
> coffee pads 11
> coffee pads 12
> coffee pads 10
# Press Ctrl-D to cancel
```

With the `kafka-console-consumer.sh` command, we can consume the data again:

```
$ kafka-console-consumer.sh \
    --topic products.prices.changelog.multi-partitions \
    --from-beginning \
    --bootstrap-server localhost:9092
cola 2
coffee pads 11
coffee pads 10
energy drink 3
coffee pads 10
coffee pads 12
# Press Ctrl-C to cancel
Processed a total of 6 messages
```

We notice that suddenly the order of the messages is no longer correct! This may not be the problem in some cases, but in our online store example, this would lead to wrong prices in our shop and—worse—could lead to inconsistencies between different systems depending on the order in which messages are read, as mentioned in the previous chapter. In fact, this is a problem in most cases, as data inconsistencies and incorrect processing can have significant negative effects on the reliability and accuracy of any system relying on Kafka for data streaming and processing. The reason is that the `kafka-console-producer.sh` command automatically distributes the data across multiple partitions, and Kafka guarantees the order of messages only within one partition.

NOTE We may need to stop and restart the producer to ensure that messages are sent to different partitions, especially because we're working with a very

low number of messages here. Additionally, it might be necessary for us to consume the messages multiple times to truly observe the problem with the message order.

To guarantee the correct order between certain messages, we need to make sure that these messages end up in one partition. We can guarantee that two messages are produced into the same partition by assigning the same key to both messages. Keys are optional, simple byte arrays just like values. When we use keys, the Kafka library in the producer decides which partition to produce based on the hash value of the key:

```
partition_number = hash(key) % number_of_partitions;
```

For this, the Kafka library first calculates the hash value of the key and then calculates it as modulo of the number of partitions. If we change the number of partitions of a topic in between, the order guarantee of Kafka is no longer given.

CAUTION By default, the `librdkafka` library uses a different hash algorithm than the Java client library for partitioning messages. In environments where different systems use different programming languages or client libraries (e.g., Java-based backend systems and Python-based applications using `librdkafka`), messages may be partitioned inconsistently, leading to wrong calculations, especially when using stream processing libraries such as Kafka Streams. To ensure correctness, we should set the partitioner to *murmur2_random* in `librdkafka` producers. This aligns the partitioning strategy between Java and `librdkafka` clients, promoting even distribution across partitions.

If the order of the messages isn't important for us, we can omit the key. In this case, the producers produce the messages into partitions using the round-robin method. That is, the first message is sent to partition 0, the second message to partition 1, and so on. You may have noticed in the example without keys that while the messages were grouped, every even message was consumed first and then every odd message, shown in figure 4.5, although this grouping could have happened the other way around by chance.

To reduce the number of network requests and thus increase the data throughput, the producer in Kafka uses a better optimized round-robin procedure since version 2.4. Instead of selecting a different partition after each message, Kafka accumulates the messages into the batches that exist anyway and selects another partition only after a batch has been sent. Thus, we minimize the number of network requests and fill our batches better at the same time so that we benefit from better batching and, if enabled, better compression.

So now, our producers distribute our messages to the different partitions. Ideally, our partitions are distributed evenly across all brokers, and thus the load is also distributed reasonably evenly across these brokers.

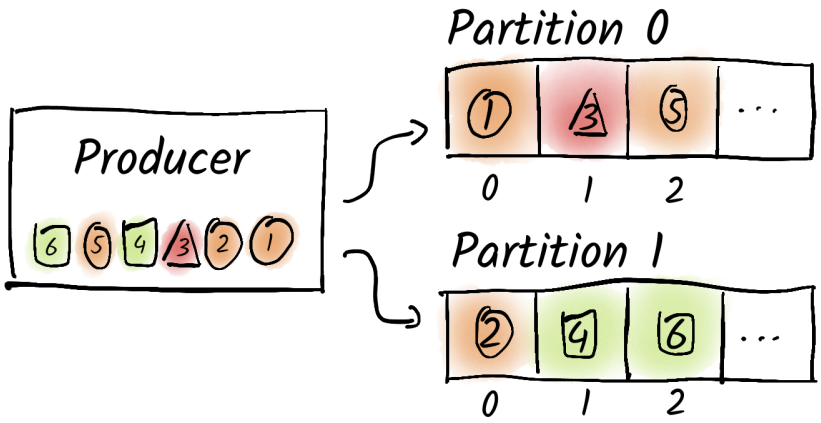


Figure 4.5 Every odd message was produced to partition 0, and every even message was produced to partition 1.

To specify keys, we use the `--parse-keys` and `key.separator` properties in `kafka-console-producer.sh`. As we did earlier, we create a new topic with two partitions and produce some messages with the product names as keys:

```
# First create the topic products.prices.changelog.multi-partitions-keys as
above
$ kafka-console-producer.sh \
    --topic products.prices.changelog.multi-partitions-keys \
    --property parse.key=true \
    --property key.separator=":" \
    --bootstrap-server localhost:9092
> coffee pads:10
> cola:2
> cola:1
> energy drink:3
> coffee pads:11
> coffee pads:12
> energy drink:4
> coffee pads:10
# Press Ctrl-D to cancel
```

In the consumer, we can enable the `print.key` property to display keys:

```
$ kafka-console-consumer.sh \
    --from-beginning \
    --topic products.prices.changelog.multi-partitions-keys \
    --property print.key=true \
    --property key.separator=":" \
    --bootstrap-server localhost:9092
coffee pads:10
cola:2
cola:1
```

```
coffee pads:11
coffee pads:12
coffee pads:10
energy drink:3
energy drink:4
# Press Ctrl-C to cancel
Processed a total of 8 messages
```

We see here that the data is in a different order than when it was produced, but we see that messages with the same key show up in the correct order, even if other keys are seen in between, as shown in figure 4.6.

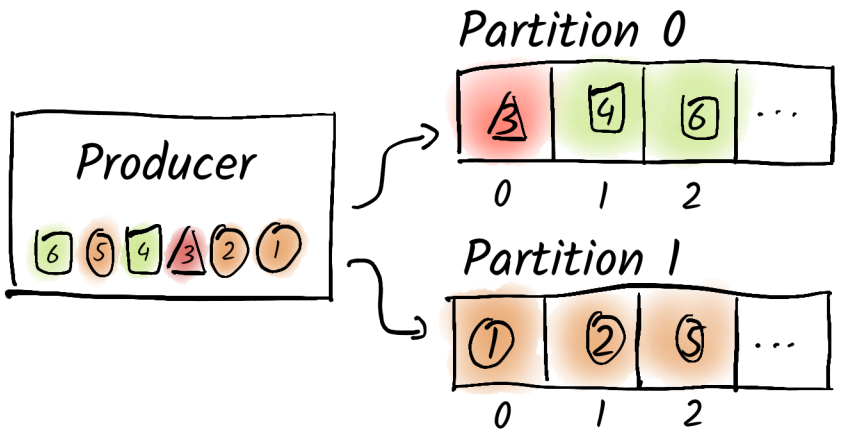


Figure 4.6 Messages with the same key (here, the form) are produced to the same partition.

> **CAUTION** Unequal distribution of keys in Kafka leads to imbalanced data distribution across partitions. As keys determine the partition placement, an uneven key distribution results in certain partitions handling a disproportionate number of messages, affecting workload balance and overall cluster performance. Balancing key distribution is crucial for optimizing workload and resource utilization in Kafka.

Normally, key distribution isn't too much of a problem if you have significantly more keys than partitions. For instance, if your key is the customer ID and you have thousands of customers, the distribution will likely be sufficiently balanced across the partitions. However, caution is needed if a small number of keys are responsible for a significant amount of traffic. In such cases, the distribution might become skewed, leading to potential bottlenecks. For example, social networks need to handle top members, who generate over 90% of the traffic, differently from the average user to ensure the system remains efficient and responsive.

4.2.2 Consumer groups

We can now distribute these partitions to different brokers and thus balance the load between the brokers. The producers decide independently which data should be distributed to which partitions. If we don't use keys, our producers distribute the data to the partitions in a round-robin fashion, that is, first to partition 0, then partition 1, and so on. If we use keys, then we distribute the messages to the partitions in such a way that messages with the same key end up on the same partition.

This usually works very well in terms of performance. However, if we have only one consumer that needs to read data from all partitions, it may not be able to keep up, and we may not be able to process the data in a timely manner. Because Kafka doesn't push data, but the consumer fetches the data itself, the consumer can't be overloaded if programmed correctly. It just processes the data correspondingly slower. Here, we don't yet know any way to scale our consumer. This can quickly lead to our consumer becoming the bottleneck and our system not being able to take full advantage of Kafka, as illustrated in figure 4.7.

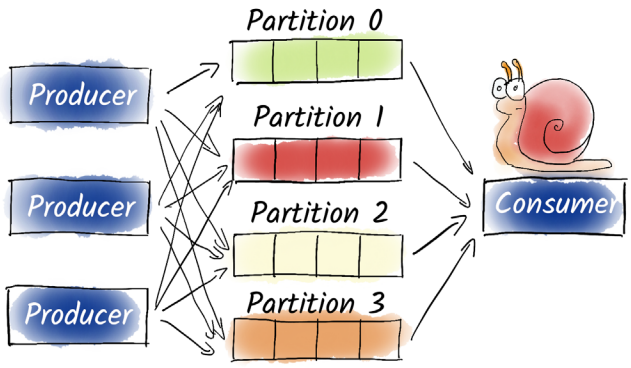


Figure 4.7 If we have only one consumer that needs to read data from all partitions, it may not be able to keep up, and we may not be able to process the data in a timely manner.

Most of the time, we also don't have just one type of consumer that wants to consume the same data. In our online store example, we can imagine having an analysis service that reads data from the `products.prices.changelog` topic and then updates the price in a database. For this service, it would be perfectly sufficient to use only one consumer, as this is a very simple and resource-efficient task. In addition, we have another service that wants to scientifically analyze the data from the same `products.prices.changelog` topic and maybe enrich it with data from our topics or data sources. For this, one consumer is probably not enough for us because it would take way too long to evaluate this data with only one consumer.

For this scenario, we need two things: first, we need a way for these different services to independently read data from the partitions without affecting each other. For example, the scientific analytics service should be able to read data even though our metrics

service has already read it. Fortunately, this is easy to do in Kafka, as consumers decide on their own which data to read using offsets.

Second, we need a way to scale our services horizontally so that the partition data is distributed among the different instances of the service in such a way that all the data is not just read, but read in the correct order. To ensure this, each partition may only be consumed by exactly one instance per service. A partition can't be consumed by two instances of the same service while still maintaining the order guarantees.

To solve both, in Kafka, we use *consumer groups*. Different consumers form a consumer group if their `group.id` is the same. Consumers in a consumer group distribute the load as evenly as possible across the members of the group and write their offsets to Kafka. This means that if we start a consumer with a consumer group, stop it again, and then start it again, the consumer can pick up where it left off, thanks to Kafka remembering the offsets. For a visual representation, see figure 4.8.

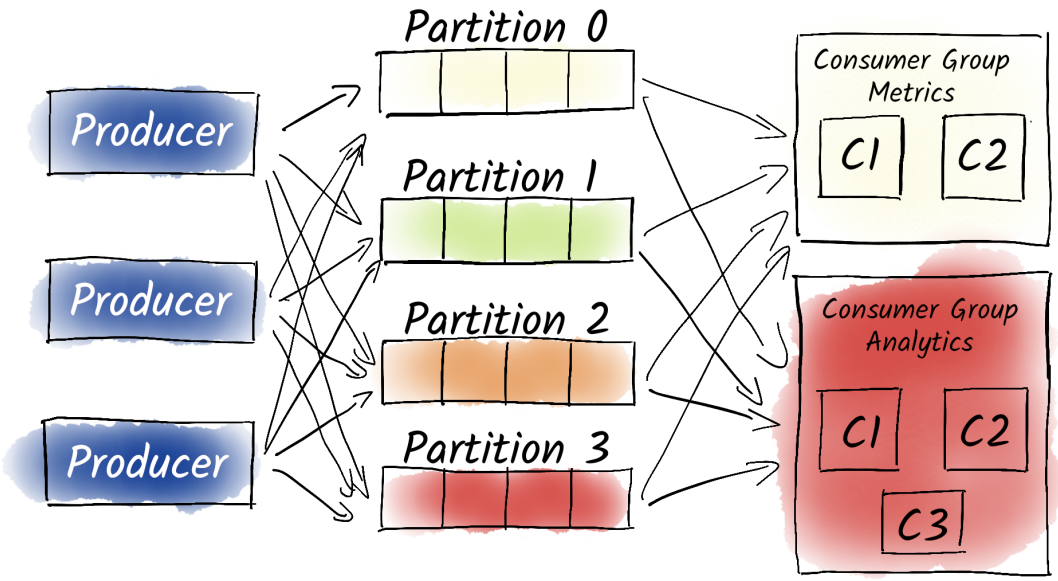


Figure 4.8 Consumer groups allow us to split the processing of multiple partitions between different instances of the same service. Often, several consumer groups, not just one, consume the data from a topic. Consumer groups are isolated from each other and don't influence one another.

Let's try it for our `products.prices.changelog.multi-partitions-keys` topic from the previous section:

```
$ kafka-console-consumer.sh \
    --from-beginning \
    --topic products.prices.changelog.multi-partitions-keys \
```

```
    --property print.key=true \
    --property key.separator=":" \
    --group products \
    --bootstrap-server localhost:9092
coffee pads:10
energy drink:3
coffee pads:11
coffee pads:12
energy drink:4
coffee pads:10
cola:2
Cola:1
# Press Ctrl-C to cancel
Processed a total of 8 messages
```

The only thing we changed here in the command is to specify the consumer group (`--group products`). When we run the command again, we don't see any messages, and after pressing Ctrl-C, we get the following information:

```
Processed a total of 0 messages
```

To see how the consumers in a consumer group share the load among themselves, we start multiple consumers at the same time with the same group ID. To do this, we can run the command we just used in two terminals and display both terminal windows side by side so we can see the output at the same time. In addition, let's start a producer that produces some more data:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog.multi-partitions-keys \
    --property parse.key=true \
    --property key.separator=: \
    --bootstrap-server localhost:9092
>energy drink:2
>energy drink:3
>cola:2
>cola:5
>energy drink:1
>cola:2
# Press Ctrl-D to cancel
```

Now we should see a picture similar to table 4.1 among our consumers.

Table 4.1 Consumer windows

Consumer 1	Consumer 2
cola:2	energy drink:2
cola:5	energy drink:3
cola:2	energy drink:1

This means that the consumers of the consumer group successfully divided the work among themselves. Consumer 1 was responsible for the partition containing the messages for `cola`; Consumer 2 took care of the other partition. Inside a consumer group, a partition will at most be read by one consumer, but a consumer can read multiple partitions. Because we created the topic with only two partitions, it doesn't make sense to have more than two consumers in the group. We'll look at consumer groups again in more detail in the performance chapter.

NOTE If we had used `coffee pads` instead of `energy drink`, then all messages would end up with the same consumer as they also end up in the same partition.

4.2.3 Replication

After learning about partitioning as a way to reliably distribute data across multiple systems, we next want to improve the reliability of our system. Maybe we know from our own experience that computer systems fail for no apparent reason, hard disks suddenly break, someone deletes the wrong folder, and so on. There are many reasons why we can lose data. The most reliable way to prevent data loss is replication. On a small scale, we do this (hopefully) in private by setting up backups so we don't lose all of our data in the event of a system failure. We also use backups in a corporate context. The disadvantage of using backups is that it takes time to restore them. We can't guarantee uninterrupted operation with backups, but we have the option of restoring data in the event of a disaster. That's why we also call backups *cold replication*. The data is replicated, but in some circumstances, it can take a long time before we can access that data again. Kafka itself has no support for such cold replication. If we want backups, we have to take care of it ourselves.

However, we can call Kafka's replication strategy *warm*. Instead of storing data on a single server, Kafka places data on multiple running servers. If one Kafka broker fails, another broker can take over almost immediately and continue to accept requests from both producers and consumers.

In Kafka, we configure replication at the topic level. For each topic, we can set the replication factor independently of other topics. We usually do this when we create a topic. For example, let's create the topic `products.prices.replication` with three partitions and a replication factor of three:

```
$ kafka-topics.sh \
    --create \
    --topic products.prices.replication \
    --partitions 3 \
    --replication-factor 3 \
    --bootstrap-server localhost:9092
Created topic products.prices.changelog.replication.
```

Logically, our replication factor can't be larger than the number of brokers we have available. In our test environment, described in appendix A, we have three Kafka brokers. If we try to create a topic with four replicas, we get an error message:

```
Error while executing topic command : Unable to replicate the partition
4 time(s): The target replication factor of 4 cannot be reached because
only 3 broker(s) are registered.
[…] ERROR org.apache.kafka.common.errors.InvalidReplicationFactorException:
Unable to replicate the partition 4 time(s): The target replication factor
of 4 cannot be reached because only 3 broker(s) are registered.
 (org.apache.kafka.tools.TopicCommand)
```

Let's take a closer look at what happened after the topic was created by again using the command-line tool `kafka-topics.sh` with the flag `--describe`:

```
$ kafka-topics.sh \
    --describe \
    --topic products.prices.replication \
    --bootstrap-server localhost:9092
Topic: products.prices.replication PartitionCount: 3 ReplicationFactor: 3
Configs:
Topic: products.prices.replication Partition: 0 Leader: 2 Replicas: 2,1,3
Isr: 2,1,3 Elr:
Topic: products.prices.replication Partition: 1 Leader: 3 Replicas: 3,2,1
Isr: 3,2,1 Elr:
Topic: products.prices.replication Partition: 2 Leader: 1 Replicas: 1,3,2
Isr: 1,3,2 Elr:
```

We have three partitions and three replicas as intended.

Provides detailed information about all three partitions

We see that for partition 0, the leader is broker 2. But what does this mean? Kafka's replication strategy follows the *one leader–several followers principle*. In concrete terms, this means that there's one broker, which is the leader, for each partition. In our example output, this means that broker 2 is the leader for partition 0 of the `products.prices.replication` topic. Depending on the replication factor, there may be other brokers that are *followers*. We refer to all of these brokers on which this partition is stored as replicas. That is, with a replication factor of 1, we normally have one replica, one leader, and no follower. With a replication factor of 3, as in our example, we have three replicas, one leader, and two followers in the normal state.

With this knowledge, we can decipher the rest of the `kafka-topics.sh` command. `Replicas: 1,3,2` simply says that the replicas are on brokers 1, 3, and 2. By default, the broker whose ID is first in the list of replicas is the leader for that partition. We can see from the output that this is also true for the other two partitions. Broker 3 is first for partition 1 and is also the leader. For partition 2 this also matches with broker 1 as the leader.

Finally, the statement `Isr: 2,1,3`, which refers to *in-sync replicas* (ISRs). These are the replicas that are in-sync, that is, up-to-date. If the ISR list doesn't match the list of replicas, then we know something's wrong, and a broker may have failed. We'll explore ISR in detail in a subsequent chapter.

We already talked about the fact that Kafka follows a one leader–several followers principle, which means there is one leader per partition, and all read and write

operations are performed only via the leader. The followers are there solely to replicate the data from the leader to themselves as quickly as possible and to be available in case the leader fails, as shown in figure 4.9.

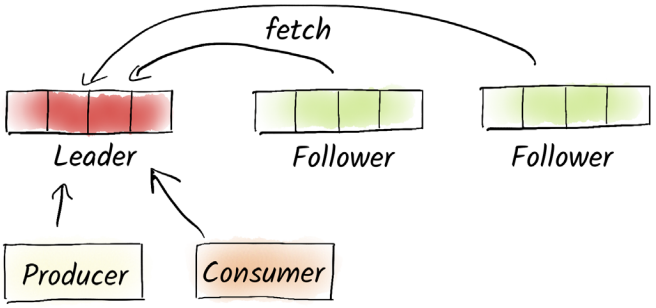


Figure 4.9 Consumer and producer communicate exclusively with the leader (with rare exceptions). Followers are only there to continuously replicate new messages from the leader. If the leader fails, one of the followers takes over.

If the leader fails, one of the followers who are in-sync (ISR) or at least an eligible leader replica (ELR) takes over the leader's job and becomes the new leader themselves. The producers and consumers then automatically switch to the new leader.

A big advantage of using logs in Kafka is that replication can be implemented relatively simple and reliably. Kafka's followers, similar to consumers, request new messages from the leader, starting from a specific offset, and write this data to their own local log. This not only ensures high performance with relatively simple code but also helps maintain the order of messages.

However, the guarantee that messages are processed in the correct order is more complex. Kafka achieves this through a combination of techniques, including ensuring that the leader is always the authoritative source of data for each partition, and the followers are kept in sync by continuously pulling data from the leader. This approach provides strong consistency for data replication, ensuring that all consumers, regardless of which replica they read from, will receive messages in the correct order.

Now the following question arises: If we have only one leader, then isn't the load unevenly distributed across the brokers? That would be a problem if we had only one partition in our Kafka cluster, but usually we have many partitions. One broker isn't the leader for all partitions, but there is one leader for each partition. Kafka, when creating partitions, tries to distribute the leaders as evenly as possible across the brokers. We have to pay a lot of attention to this even distribution of the load during the operation of production systems because it's a guarantee for Kafka's performance.

4.3 Components of Kafka

In principle, a Kafka cluster consists of only three different components: *coordination cluster*, *brokers*, and *clients*. The coordination cluster and the brokers have already been mentioned several times on the previous pages, but what about clients, and what role

do consumers and producers play? The answer to this question is simple because by *clients*, we mean all applications that connect to our Kafka cluster, including the consumers and producers we already know. A typical Kafka cluster usually consists of a coordination cluster and three additional brokers for our actual cluster, as shown in figure 4.10. We'll explain the reason for this in more detail later in the chapter.

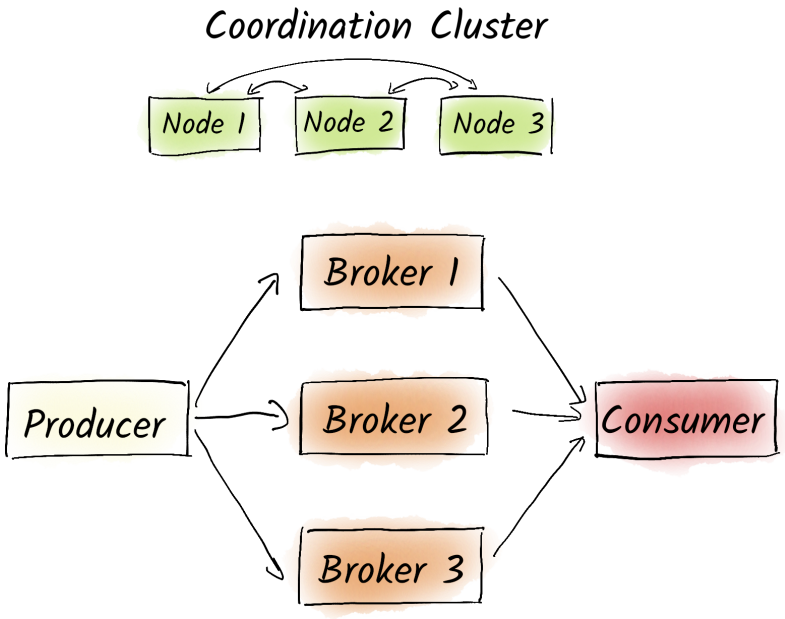


Figure 4.10 A typical Kafka environment consists of the Kafka cluster itself and the clients that write and read data to Kafka. Before the Kafka Raft (KRaft)-based coordination cluster, a ZooKeeper ensemble was used as a coordination cluster. Without ZooKeeper, brokers can take over the task of the coordination cluster themselves or outsource it to a standalone coordination cluster.

4.3.1 Coordination cluster

A distributed system such as Kafka requires some level of orchestration for coordination. In Kafka, this task is performed by the coordination cluster, shown in figure 4.11. The coordination cluster continuously monitors the current state of the cluster and checks whether all brokers are still reachable and functional. As part of this task, it also manages the list of all

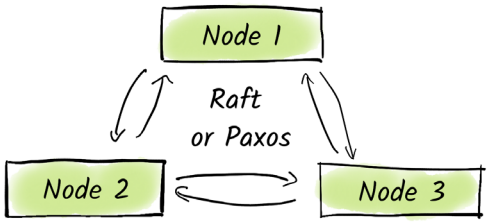


Figure 4.11 Kafka uses either a KRaft-based or a ZooKeeper-based coordination cluster. Both should consist of an odd number of nodes (usually three or five) and use a protocol to find a consensus.

active brokers and is responsible for adding new brokers and removing existing brokers from the cluster.

The coordination cluster stores metadata about topics and manages access to topics. As part of topic management, it also stores the assignment of partitions to brokers and the leader broker for each partition. Using the coordination cluster, one of the brokers is designated as the *controller*. The controller is responsible for managing the state of individual partitions and replicas. In this activity, for example, it takes care of assigning brokers to partitions and designating a partition leader. If one of the brokers fail, the controller automatically reassigns the partitions and selects new leaders to maintain availability and consistency.

Prior to KRaft, the role of the coordination cluster was filled by a *ZooKeeper ensemble*. ZooKeeper itself isn't part of Kafka, but an independent project of the Apache Software Foundation. It's a distributed coordination service that helps manage configuration, naming, and synchronization across distributed systems. ZooKeeper provides a centralized registry for storing metadata and offers primitives such as distributed locks that distributed applications need. In the context of Kafka, ZooKeeper was used to maintain cluster metadata, manage broker leadership elections, track partition assignments, and store access control lists (ACLs).

However, ZooKeeper's general-purpose nature also introduced several challenges. As a separate system, it required additional operational expertise and maintenance overhead. Its generic data model, while flexible, wasn't optimized for Kafka's specific needs, leading to performance bottlenecks at scale.

These limitations led to the development of Kafka Raft (KRaft), mentioned earlier, which integrates the coordination directly into Kafka. KRaft provides several advantages: it simplifies the architecture by eliminating the external dependency on ZooKeeper, improves performance through a more specialized metadata management system, and reduces operational complexity by having a single system to maintain. KRaft also introduces a more efficient protocol for handling metadata updates and leader elections, which is particularly beneficial for large clusters with frequent topology changes.

Starting with Kafka 3.3, KRaft became production-ready and is now the recommended coordination mechanism. For smaller clusters, one significant advantage of KRaft is its ability to run within the Kafka cluster itself—some brokers can be designated to handle both regular Kafka operations and coordination duties. However, for larger production environments with many brokers, running a dedicated coordination cluster remains the recommended practice to ensure optimal performance and clear separation of concerns.

For coordination to work reliably, whether using KRaft or ZooKeeper, the coordination cluster must consist of an odd number of nodes. This requirement enables the cluster to reach consensus through majority decisions. With three nodes, the cluster can continue operating even if one node fails; with five nodes, it can tolerate two node failures while maintaining quorum.

TIP For smaller Kafka clusters, we recommend a coordination cluster consisting of three KRaft brokers or ZooKeeper nodes. For larger clusters, we can also launch five instances to increase resilience.

4.3.2 Broker

Brokers form the actual cluster in Kafka because they are responsible for receiving, storing, and sending messages and data. A Kafka cluster usually consists of multiple brokers because otherwise we can't replicate the data, which is one of the core concepts of Kafka. If our Kafka cluster consisted of two brokers, we could already replicate but would run into problems relatively quickly if in doubt. For example, let's imagine that we want to maintain our cluster, so one of the brokers is offline. If the second and final broker were to fail at the same time, this could quickly lead to data loss. Therefore, we should always start with at least three brokers. If the load becomes greater, we can add more brokers to increase performance.

4.3.3 Clients

In Kafka, a *client* is any external application connecting to our cluster. Strictly speaking, clients aren't considered components of Kafka, at least in the context of a Kafka cluster. Our now well-known `kafka-console-consumer.sh` and `kafka-console-producer.sh` are nothing but Kafka clients. Both are Java applications that access the Kafka API. Clients that aren't written in Java usually use the `librdkafka` library (https://github.com/edenhill/librdkafka/). We go into more detail about how clients work in part 3 of this book.

The simplest form of Kafka clients are producer and consumer, which we've already met. While producers write data to Kafka, consumers read data from Kafka. We can also describe it as producers producing data and consumers consuming data, which explains the naming.

To simplify the processing of data streams, Kafka includes the *Kafka Streams* library (https://kafka.apache.org/documentation/streams/) with which we can filter, transform, and merge data in Kafka, plus much more. To do this, Kafka Streams takes on the role of both consumer and producer at the same time. *Kafka Connect* (https://kafka.apache.org/documentation/#connect) is a framework that is used to connect Kafka with other systems. For example, entire databases can be populated from a Kafka cluster via Kafka Connect. In part 4 of the book, we'll look into how exactly external systems can be connected to Kafka and how Kafka Streams and Kafka Connect work in detail.

TIP Using Kafka Connect instead of custom consumer or producer code offers advantages such as scalability, seamless integration with various data sources, built-in connectors, modularity, and simplified error handling. It promotes a standardized, maintainable, and scalable approach to data integration.

4.4 Kafka in corporate use

We've learned about Kafka as a distributed log in the last few pages. In use, we find Kafka as a distributed system with multiple brokers, a coordination cluster, and of course, all the producers and consumers we need to exchange data with Kafka to solve our business problems. Although this architecture alone brings immense value to our business, we often need more than just a central place to store and route data. Kafka as the center of a streaming platform can help us move our business toward a data-driven enterprise, enabling us to make near-real-time decisions. However, Kafka alone isn't enough to do this (see figure 4.12 for reference).

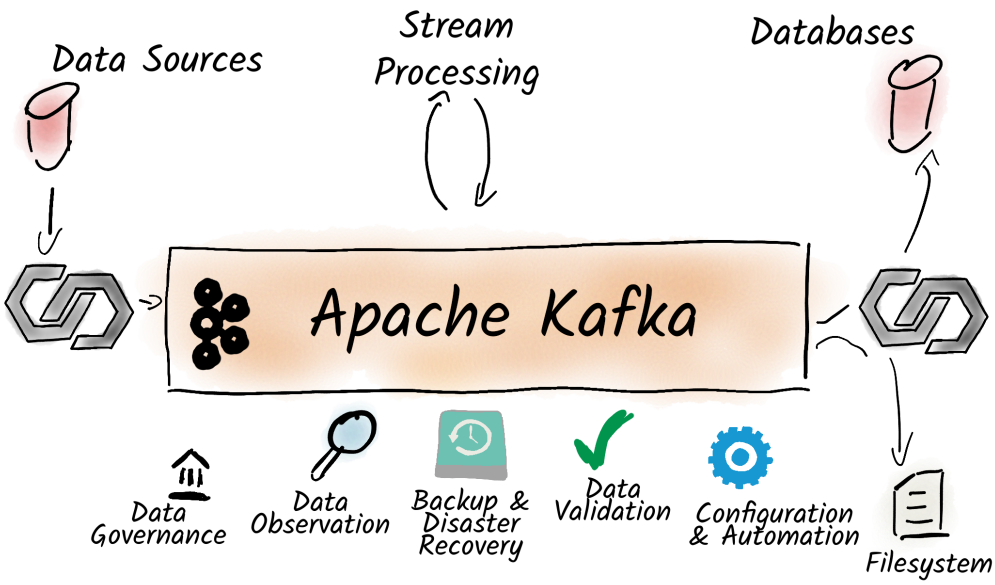


Figure 4.12 Kafka alone is usually not enough. The Kafka ecosystem offers numerous components to integrate Kafka into our enterprise landscape and thus build a streaming platform.

Our Kafka cluster doesn't live in isolation, so it should be well connected to our existing systems in the enterprise such as the numerous database systems that store our core data, messaging solutions that connect proprietary services and external service providers, legacy applications that no one dares touch anymore, and more. Of course, we could write a separate Kafka consumer or producer for each of these services to connect them to Kafka. But fortunately, Kafka provides Kafka Connect, a tool and framework to efficiently and reliably dock third-party systems to Kafka. Kafka Connect ships directly with Apache Kafka and is under the same Apache 2.0 license. We may still have mainframes in enterprise use for which there are no Kafka libraries and where it would be very cumbersome to write custom connectors for Kafka Connect. To connect

these systems, we can, for example, connect one of the REST proxies available on the market. This allows systems that communicate via HTTP/REST to interact with Kafka.

> **NOTE** Confluent offers the REST Proxy, a proprietary REST proxy for its Confluent Enterprise platform. Alternatively, there is an open source variant of the REST Proxy from Aiven: https://github.com/aiven/karapace.

Now, we may have numerous systems connected to Kafka, and we can exchange data between the systems. Often, in near real time, we not only want to send the data but also evaluate and manipulate it. Because we can view the data in Kafka as a stream, a stream processing system, that is, a system that operates directly on data streams, is suitable for this purpose. There are a large number of competing frameworks to implement stream processing with Kafka, such as Kafka Streams, Apache Flink, Scala's Akka, and the like. With the help of stream processing libraries, it's relatively easy to implement scalable and robust stream processing operations even on large data sets, that is, operations such as filtering data, manipulating individual data sets and joining, or connecting, different data streams. There are also some offerings on the market that allow stream processing even without programming knowledge.

The more data we store in Kafka and the longer we want to keep this data, the more important it is that the data is stored in a consistent data format. Especially when more than one team accesses the Kafka cluster, schema management is highly relevant. However, no schema management is shipped with Apache Kafka, as Kafka itself isn't interested in the data format. There are a few possible approaches to schema management ranging from schema definitions in a Git repository up to components such as Confluent's *Schema Registry* or the schema registry implemented in *Karapace*.

> **NOTE** Confluent's Schema Registry is part of the Confluent Enterprise platform. Karapace, consisting of the REST Proxy discussed earlier and the Schema Registry, can be found on GitHub at https://github.com/aiven/karapace.

Depending on how important the data we store in Kafka is, we should think about whether it makes sense to mirror it to additional data centers. With Kafka *MirrorMaker 2*, there's now a tool included with Kafka that greatly simplifies the multiple data center operation of Kafka. In some cases, it's still worth considering a backup and recovery approach. This is more demanding and extensive, especially for larger Kafka installations, than it seems at first glance.

In addition, any Kafka installation that's intended to be more than a playground for trying out the technology needs other components. The streaming platform must be extensively monitored using a monitoring tool. At best, Kafka shouldn't be set up and configured manually, but with the help of automation tools. In addition, it's important, especially in larger companies, to pay attention to certain compliance rules and to enforce them.

Looking at Kafka as a streaming platform in great detail is far beyond the scope of this book, and this area is evolving dynamically at the time of writing. We address some

of these problems in greater detail than we do here in chapter 18, but we recommend that you ask a Kafka expert about the current state of play if you have further questions.

Summary

- A log is a sequential list where we add elements at the end and read them from a specific position.
- Kafka is a distributed log in which the data of a topic is distributed to several partitions on several brokers.
- Offsets are used to define the position of a message inside a partition.
- Kafka is used to exchange data between systems; it doesn't replace databases, key-value stores, or search engines.
- Partitions are used to scale topics horizontally and enable parallel processing.
- Producers use partitioners to decide which partition to produce to.
- Messages with the same keys end up in the same partition.
- Consumer groups are used to scale consumers and allow them to share the workload, and one partition is always consumed by one consumer inside a group.
- Replication is used to ensure reliability by duplicating partitions across multiple brokers within a Kafka cluster.
- There is always one leader replica per partition that's responsible for the coordination of the partition.
- Kafka consists of a coordination cluster, brokers, and clients.
- The coordination cluster is responsible for orchestrating the Kafka cluster—in other words, for managing brokers.
- Brokers form the actual Kafka cluster and are responsible for receiving, storing, and making messages available for retrieval.
- Clients are responsible for producing or consuming messages, and they connect to brokers.
- There are various frameworks and tools to easily integrate Kafka into an existing corporate infrastructure.

5 Reliability

This chapter covers

- Kafka's acknowledgment settings
- Data availability and fault tolerance in Kafka
- Kafka's delivery guarantees
- Kafka's transactional capabilities
- Leader-follower principle in Kafka

Now we know how topics and messages look in Kafka and how Kafka relates to a log. In the next sections, we'll explore some crucial aspects of Kafka's operation. Before we take a closer look at how we can influence the performance of our Kafka cluster in the next chapter, we'll examine the parameters we can adjust to improve reliability in this chapter.

When we consider the topic of reliability, we can roughly divide it into three subcategories. The first category is data durability, which deals with how Kafka ensures that data is stored correctly in perpetuity. The second category is availability, which refers first to ensuring that consumers can access our written data at any time if

possible, and second to ensuring that producers can also write data at any time. Both data durability and availability can be achieved relatively well with replication in Kafka.

However, it becomes somewhat more problematic when we consider consistency, the third category in Kafka's reliability concept, and how it comes into play. Consistency itself also has various manifestations, which we'll explore in more detail later in this chapter, but ultimately, the goal is to ensure that the data in the cluster is persisted exactly as desired. Achieving consistency in a distributed system such as Kafka is challenging. To ensure that messages sent actually arrive, Kafka makes uses *acknowledgments* (ACKs), idempotent writes, and transactions, which allow producers to send messages atomically and ensure data integrity across multiple writes.

But before we get into the details of ACKs, transactions, and replication in Kafka, let's briefly look at the kind of problems that can arise with respect to data durability, availability, and consistency.

Let's imagine that we have a Kafka cluster consisting of three brokers, as in our previous examples. As long as nothing goes wrong, there are no problems with data durability, availability, and consistency. But what happens if one of the brokers suddenly fails? Are our topics still readable? What does that mean for the consistency of our data? Can we continue to produce and write data? What happens if the broker suddenly comes back online or another broker fails? Is persistent storage of our data still guaranteed? Can we continue to read data, or does that cause consistency problems? Here, we can already see that sometimes we also have to set priorities with regard to the individual properties. What is more important to us—being able to write and read data as long as possible, even when there are bad failures, or ensuring that the data is correct? We'll answer all of these questions in the next sections.

5.1 Acknowledgments

ACKs are used by many network protocols, such as TCP or IEEE 802.11 (Wi-Fi), to confirm successful data transmission. However, the mechanisms of lower communication layers aren't always sufficient to guarantee successful transmission of data. For example, if the data line is heavily congested or even completely disrupted, even TCP can no longer guarantee successful transmission. In this case, TCP aborts the data transmission after a few unsuccessful transmission attempts. Even if the transmission was successful, it doesn't guarantee that the application handled it successfully. Many applications therefore use their own ACKs, including Kafka.

ACKs are used by producers in Kafka. Consumers use a different method to ensure that all data has been read successfully, which we'll also discuss in detail in a later chapter. In Kafka, ACKs are not just used to ensure that the broker has received the data from the producer, but they also serve another purpose as part of the replication strategy. ACKs in Kafka are completely independent of the configuration of a topic and are configured in the producer. However, topic configuration in combination with ACKs does have a major effect on reliability and performance.

5.1.1 ACK strategies in Kafka

Before we go into more detail about the consequences of ACKs in Kafka, let's take a look at the different ways ACKs can be configured in Kafka in the first place and what choices we have. ACKs are controlled directly through the producer. There are three options here, which are shown in figure 5.1. We discuss them in descending order with an eye toward overall system reliability.

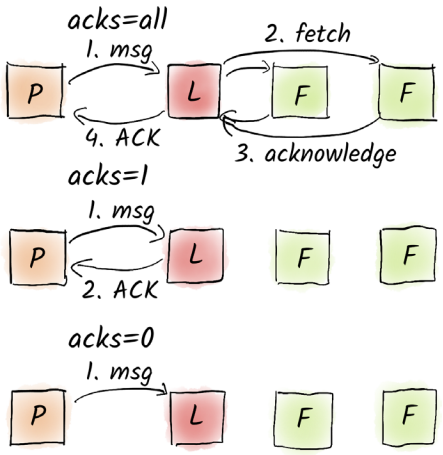


Figure 5.1 Different ACK configurations. With `acks=all` (or `acks=-1`), the producer doesn't get feedback until all in-sync replicas have replicated the message. With `acks=1`, the producer gets a response as soon as the leader has received the message. With `acks=0`, the producer sends messages and doesn't wait for confirmation that they have actually arrived.

Our `kafka-console-producer.sh` uses the `--producer-property` argument for this purpose, via which we can additionally specify properties such as ACKs:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092 \
    --producer-property acks=all
```

In the first case, we set our ACK strategy to `all`. This means that after successfully receiving a message, the broker or leader of the partition won't send an ACK back to the producer until the message has been successfully distributed to all in-sync replicas (ISRs). Replicas are in sync if they have fetched the current data within the past 30 seconds.

NOTE The setting `acks=-1` is equivalent to `acks=all`.

Another possibility is to set `acks=1`:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092 \
    --producer-property acks=1
```

This setting can be compared very well with TCP. As soon as the leader has received the message, it sends the ACK to the producer. Compared to `acks=all`, the producer receives the ACK faster, which in turn slightly improves latency. However, if the leader fails immediately after sending the ACK and before distributing the messages to the followers, data loss must be expected. Additionally, a message may be written to the topic twice if no additional precaution is taken. We'll discuss this later in the chapter. The last variant is `acks=0`:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog \
    --bootstrap-server localhost:9092 \
    --producer-property acks=0
```

Here, no ACK is sent from the broker to the producer. This configuration is comparable to User Datagram Protocol (UDP). The producer sends its data, but if the data doesn't arrive, the producer doesn't care. Messages are thus sent by the producer only once, and Kafka doesn't guarantee that the message will be successfully persisted. This doesn't mean that messages don't arrive, and besides, Kafka still uses TCP in the transport layer. Nevertheless, this ACK strategy obviously provides the lowest reliability and should only be used if data loss can be tolerated.

In return, this configuration offers higher performance compared to the other two strategies. Let's imagine a scenario where we have a temperature sensor that measures the temperature every millisecond and sends it to a Kafka cluster. Unless this is a sensor for monitoring pharmaceutical products, it would probably matter little if we lost some values because we're only interested in the current value anyway, and repeating old, failed transmissions would unnecessarily burden our network and brokers. At this point, it seems that `acks=0` gives the best performance among the ACK strategies if the data loss is tolerable, and `acks=all` gives the best data consistency.

> **NOTE** Since Kafka 3.0, the default setting for Kafka is `acks=all`. The default up until Kafka 3.0 was `acks=1`.

5.1.2 ACKs and ISRs

The topic configuration property `min.insync.replicas` takes effect together with the producer configuration `acks=all`. As we know, with `acks=all`, the leader doesn't send an ACK to the producer until all ISRs have successfully persisted the data. With `min.insync.replicas`, we specify that at least a certain number of replicas must be in sync before an ACK is sent to the producer on `acks=all`. If not enough replicas are reachable or in sync, the producer receives an appropriate error message and tries to resend the message.

For this, let's create a new topic named `products.prices.changelog.min-isr-2` as a test and try to write data on the topic as a test:

```
$ kafka-topics.sh \
    --create \
    --topic products.prices.changelog.min-isr-2 \
    --replication-factor 3 \
    --partitions 3 \
    --config min.insync.replicas=2 \
    --bootstrap-server localhost:9092
```

With the `--config` argument, we can configure the topic more precisely, and we set the minimum number of ISRs to 2 with the `min.insync.replicas=2` parameter. Let's take a quick look at the newly created topic:

```
$ kafka-topics.sh \
    --describe \
    --topic products.prices.changelog.min-isr-2 \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog.min-isr-2 PartitionCount: 3
    ReplicationFactor: 3 Configs: min.insync.replicas=2
  Topic: products.prices.changelog.min-isr-2 Partition: 0
    Leader: 1 Replicas: 1,2,3 Isr: 1,2,3
  Topic: products.prices.changelog.min-isr-2 Partition: 1
     Leader: 2 Replicas: 2,3,1 Isr: 2,1,3
  Topic: products.prices.changelog.min-isr-2 Partition: 2
    Leader: 3 Replicas: 3,1,2 Isr: 1,2,3
```

As expected, we have three partitions, each with three replicas that are in sync. As long as our three brokers are reachable, we won't notice any differences in terms of ACK strategies, so we shut down two of our brokers at this point.

> **NOTE** If the broker with ID 2 is our active controller, then we won't receive any answer from our cluster anymore as the remaining controller alone can't elect a new active controller. We can solve that problem easily by starting one of the offline brokers again and stopping the same broker again after a short time.

In appendix A, you'll find the `kafka-broker-stop.sh` script that we'll use at this point to shut down our brokers with ID 2 and ID 3. After that, let's take a quick look at our topic:

```
$ kafka-broker-stop.sh 3
$ kafka-broker-stop.sh 2
$ kafka-topics.sh \
    --describe \
    --topic products.prices.changelog.min-isr-2 \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog.min-isr-2 PartitionCount: 3
    ReplicationFactor: 3 Configs: min.insync.replicas=2
  Topic: products.prices.changelog.min-isr-2 Partition: 0
    Leader: 1 Replicas: 1,2,3 Isr: 1
  Topic: products.prices.changelog.min-isr-2 Partition: 1
    Leader: 1 Replicas: 2,3,1 Isr: 1
  Topic: products.prices.changelog.min-isr-2 Partition: 2
    Leader: 1 Replicas: 3,1,2 Isr: 1
```

As we can see, Broker 1 has taken over from Broker 3 as the leader for Partition 2 and from Broker 2 as the leader for Partition 1. Additionally Broker 2 and Broker 3 no longer appear in the list of ISRs. Now let's try to send messages to the topic with `acks=0`. But first we start a `kafka-console-consumer.sh` in another terminal to read the messages directly:

```
$ kafka-console-consumer.sh \
    --topic products.prices.changelog.min-isr-2 \
    --from-beginning \
    --bootstrap-server localhost:9092
```

After that, we start our `kafka-console-producer.sh`:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog.min-isr-2 \
    --bootstrap-server localhost:9092 \
    --producer-property acks=0
>cola 0
```

Although we haven't encountered any problems with brokers being unreachable or sending messages, the message still doesn't appear in our `kafka-console-consumer.sh`. As we used `acks=0`, we can't be sure if the message was successfully persisted because consuming messages requires that the minimum ISR is met. The same is true if we try to send messages with `acks=1`:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog.min-isr-2 \
    --bootstrap-server localhost:9092 \
    --producer-property acks=1
>cola 1
```

However, even though our message isn't shown in our consumer window, we can be sure the message was successfully persisted at the leader replica of the partition because we used `acks=1`. Now let's look at what happens when we try to produce messages with `acks=all`:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog.min-isr-2 \
    --bootstrap-server localhost:9092 \
    --producer-property acks=all
>cola all
>[…] WARN [Producer clientId=console-producer] Got error produce response
with correlation id 10 on topic-partition
products.prices.changelog.min-isr-2-2, retrying (2 attempts left).
Error: NOT_ENOUGH_REPLICAS
(org.apache.kafka.clients.producer.internals.Sender)
[…] WARN [Producer clientId=console-producer] Got error produce response
with correlation id 11 on topic-partition
products.prices.changelog.min-isr-2-2, retrying (1 attempts left).
Error: NOT_ENOUGH_REPLICAS
(org.apache.kafka.clients.producer.internals.Sender)
[…] WARN [Producer clientId=console-producer] Got error produce response
with correlation id 12 on topic-partition
products.prices.changelog.min-isr-2-2, retrying (0 attempts left).
Error: NOT_ENOUGH_REPLICAS
(org.apache.kafka.clients.producer.internals.Sender)
[…] ERROR Error when sending message to topic
products.prices.changelog.min-isr-2 with key: null, value: 8 bytes with
error: (org.apache.kafka.clients.producer.internals.ErrorLoggingCallback)
org.apache.kafka.common.errors.NotEnoughReplicasException: Messages are
rejected since there are fewer in-sync replicas than required.
```

We first get warnings that there aren't enough replicas (`NOT_ENOUGH_REPLICAS`), and the producer tries to resend the message. After a few failed attempts, we get an error message that the message was rejected because there aren't enough ISRs as only one replica is in sync when we require at least two to be in sync.

If we would have only stopped one broker, then we would still be able to produce and consume messages with `acks=all`. Let's start our brokers again:

```
$ ~/kafka/bin/kafka-server-start.sh -daemon ~/kafka/config/kafka2.properties
$ ~/kafka/bin/kafka-server-start.sh -daemon ~/kafka/config/kafka3.properties
```

We should now finally be able to consume our messages `cola 0` and `cola 1`, which we have sent via `acks=0` and `acks=1`, respectively, because our restarted brokers have caught up with the leader and are now in sync again.

> **NOTE** In previous Kafka versions, it wasn't required that `min.insync.replicas` were in sync to consume a message. It was sufficient that all remaining ISRs had the message persisted.

This example clearly shows the enormous effect that the ACK strategy in conjunction with the `min.insync.replicas` property of topics can have on our Kafka cluster. We'll take a detailed look at how exactly a producer sends data and how it behaves in such a case in a later chapter. If we don't set `min.insync.replicas` or set it to `1` and the number of ISR is `1`, we may experience data loss despite `acks=all` because `acks=all` behaves equivalent to `acks=1` in this case. Therefore, it's important to set a reasonable value for `min.insync.replicas`. Now we can start the brokers again.

> **TIP** In general, we recommend setting `min.insync.replicas` to `2` with a replication factor of `3`, but we need to consider our failure scenario carefully. If we need more resilience, for example, being able to lose two brokers without effect and still have one more to spare, then we set the replication factor to `4` and `min.insync.replicas` to `2`.

5.1.3 Message delivery guarantees in Kafka

We've already discussed that with `acks=0`, we can't make sure that a message arrives at the brokers at all. But we at least know that we won't get any duplicate messages. Thus, with `acks=0`, we get *at-most-once* delivery guarantees, as illustrated in figure 5.2.

Usually, we prefer to have a more reliable configuration, so we set `acks=all`. However, there's one problem that we haven't even considered yet: What happens if a message was successfully persisted, but the ACK didn't reach the producer? In this case, the producer resends the message

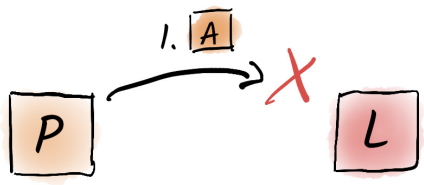


Figure 5.2 We get at-most-once delivery guarantees with `acks=0`.

because it must assume that the message couldn't be persisted successfully, as illustrated in figure 5.3. The result is that we unknowingly store a message twice in our cluster. If we again consider the example of a temperature sensor, this probably isn't a big problem, but now, let's imagine that the message is a banking transaction. This would, in case of doubt, lead to amounts being credited or debited twice, which obviously isn't a desirable behavior.

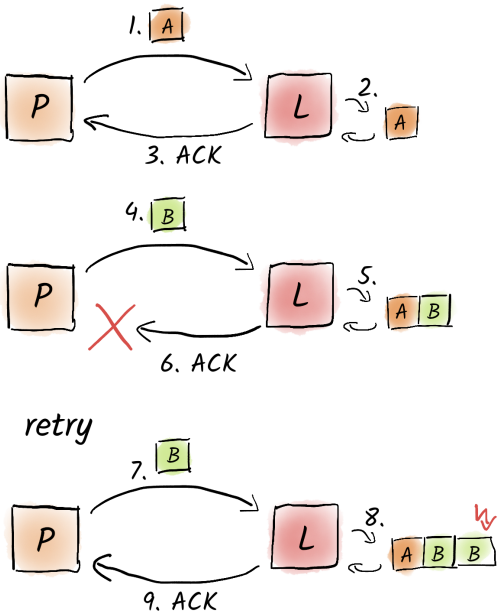


Figure 5.3 Problems arise when the producer is expecting ACKs, but they are lost on the way from the leader to the producer, although the data was written successfully. The producer sends the message again in this case, and the message is in the log twice, possibly even in the wrong order (not shown).

At this point, it's useful that we first look at the message delivery guarantees in distributed systems: at most once, at least once, and exactly once.

At most once guarantees that a message will be delivered at most once, and thus there can be no duplicates. However, it doesn't guarantee that the message will arrive at all. This is the case when `acks=0`. This is useful, for example, when readings are sent continuously in very large quantities and we want to ensure that no reading shows up twice. The loss of measured values isn't relevant here.

The *at least once* guarantee is achieved by setting `min.insync.replicas` to a reasonable value (e.g., `2`) and `acks` to `all`. In this case, a message is guaranteed to arrive in every case. However, duplicates can occur, which can lead to the problem just described. Before exactly-once semantics were introduced in Kafka, this was the preferred guarantee for most applications.

This leads us to *exactly once*, which guarantees that a message is persisted exactly once and moreover in the correct order, establishing the mathematical property of

idempotence. *Idempotence* means that if we call a function or operation several times with the same inputs, we always get the same result. Kafka implements this by having the producer assign a sequence ID to a message. By default, this feature wasn't enabled for producers until Kafka 3.3, and messages were sent without sequence IDs:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog.min-isr-2 \
    --bootstrap-server localhost:9092 \
    --producer-property acks=all \
    --producer-property enable.idempotence=true
```

Equivalent to ACKs, we can enable idempotence with the `--producer-property` argument (`enable.idempotence=true`).

> **NOTE** Idempotence requires `acks=all`, `max.in.flight.requests.per.connection` not greater than `5` (default is `5`), and `retries` greater than `0`. The default number of retries is 2,147,483,647 (`MAX_LONG`), but don't worry, there is a delivery timeout preventing the message from being sent this often. The setting `max.in.flight.requests.per.connection` controls the number of unacknowledged requests sent to a broker.

If idempotence is enabled, the leader of a partition checks when messages are received and whether they arrive in the correct order. If the message has already been persisted, only the ACK is resent, as shown in figure 5.4.

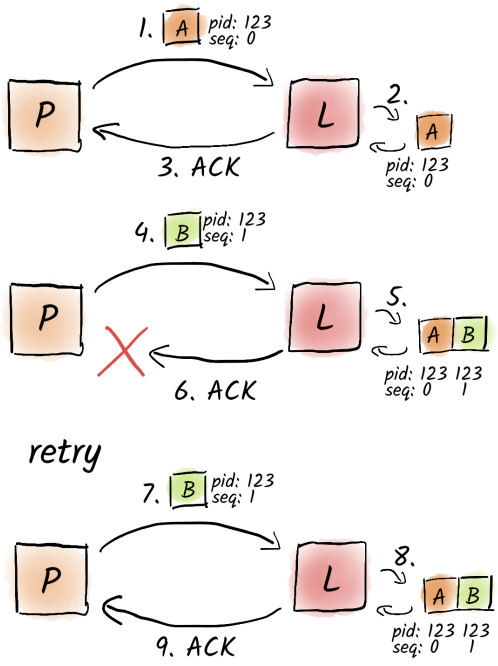


Figure 5.4 If idempotency is enabled, the producer sends its producer ID with each message and also a sequence ID for each message. This allows the broker to detect if messages are missing, appear in the wrong sequence, or have been sent multiple times, and then to respond accordingly.

If the messages arrive in the wrong order, the leader sends a *negative ACK* (NACK) to the producer, which then tries to send the message again, as shown in figure 5.5.

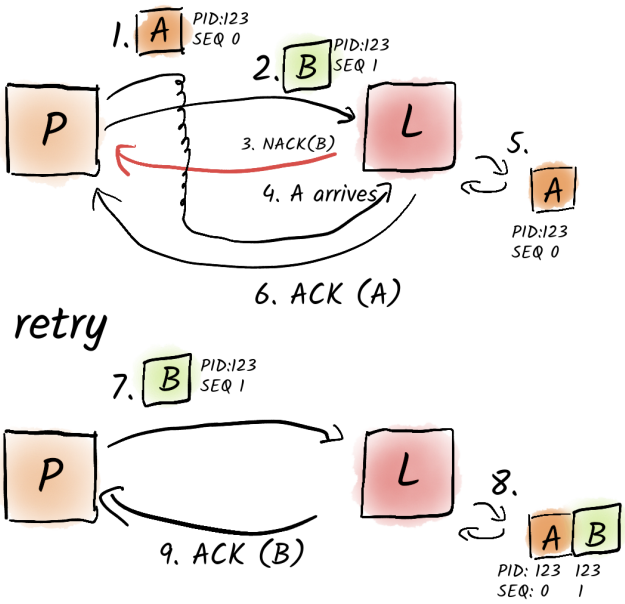


Figure 5.5 If a message was received out of order, then the broker sends a NACK to the producer. This way, we can ensure the correct order of messages and prevent, for example, message B from arriving before message A.

With idempotence, we can now ensure that messages are written in the correct order within a partition and are also present exactly once. The reason idempotency was disabled by default until Kafka 3.0 was to give the community time to update their clients. There's a negligible performance loss associated with idempotency.

> **TIP** At this point, we strongly recommend keeping idempotence enabled. If you encounter performance problems, they likely aren't caused by idempotence, so address those problems first.

5.2 Transactions

In this section, we delve into the core principles of transaction management within Kafka. Transactions in Kafka are vital for ensuring data consistency and reliability, allowing for atomic writes across multiple partitions and correct processing of messages. We'll explore the essential steps for initializing, executing, and maintaining transactions, providing insights into Kafka's robust mechanisms for guaranteeing transactional integrity, even in challenging operational scenarios.

Now it's certainly nice that the messages are correctly stored in a partition, but often we also want to process this data further. Producer idempotence has no effect on our consumers or even third-party applications. Along the way, message duplicates or losses

can still occur if we aren't careful. If we take a closer look at the consumption process in Kafka, we find that every read operation is also a write operation because we need to commit our offset. Kafka offers us the option to store our offsets in Kafka, but this isn't mandatory.

5.2.1 Transactions in databases

If we want to transfer data from Kafka to a database system exactly once, we have two options to ensure exactly once semantics. One option is that we can continue to use our Kafka consumer as usual and commit our offsets to Kafka by first writing the data to the database and then committing the offset. However, we must guarantee that the write operation to the database is idempotent. This means we check before each write operation whether the data already exists, and we only write the data if it's not already there. The easiest way to achieve this is through `UPSERT` statements. In PostgreSQL, this could look like the following:

```
INSERT INTO products_price_changelog (id, timestamp, name, price)
    VALUES (…)
    ON CONFLICT DO NOTHING;
```

The second option is to store the offsets in the database itself rather than in Kafka. If we now write the data along with the offset in a transaction, we can guarantee that each message from Kafka appears exactly once in the database because we write the offsets together with the data atomically. In PostgreSQL, we could implement this as follows:

```
BEGIN TRANSACTION;
    INSERT INTO products_price_changelog (id, timestamp, name, price)
        VALUES (…);
    INSERT INTO offsets (topic, partition, offset)
        VALUES ('products_price_changelog', 0, 123);
COMMIT;
```

But what can we do if we want to read data from a Kafka topic, process it, and then write it to another Kafka topic? Here, we also need a way to atomically write to multiple partitions, namely, the partitions of our target topics and the partitions of the `__consumer_offsets` topic, as shown in figure 5.6. This process of reading data, processing it, and writing it back to another topic is so common that Kafka Streams provides a library to make this as simple as possible.

5.2.2 Transactions in Kafka

To implement transactions reliably in Kafka, we need two ingredients. First, we need a method to produce messages reliably to individual partitions. We achieve this using idempotent producers. Second, we need a way to atomically write messages to multiple partitions simultaneously. This means that either all messages are successfully written in a transaction or none at all.

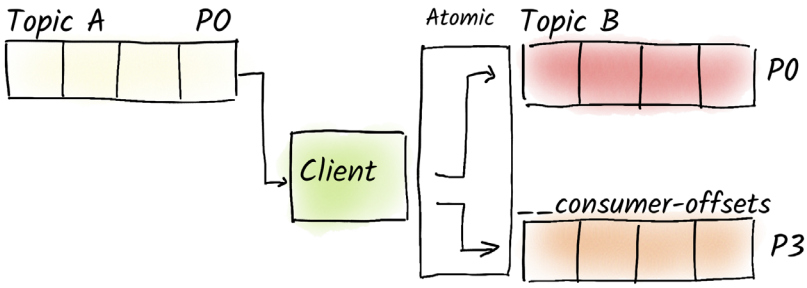


Figure 5.6 Transactions in Kafka allow us to atomically write to two or more partitions. Either all messages are successfully written or none are.

We can use transactions in our code similar to how we use them in a database. First, we inform the producer that we want to use transactions. Then, we start the transaction, write our messages, and possibly commit our offset. Finally, we commit the transaction itself to successfully complete it. Only then can consumers process the written messages. If we abort a transaction, we don't delete the written messages from the partitions, but mark them so that consumers ignore them.

A classic example of transactions is when a user (let's say, Bob) wants to transfer a sum of money to another user (Alice). For this, we need to decrease Bob's account balance by the transferred amount and increase Alice's account balance by the same amount. Both actions should occur atomically, meaning either the transfer is successful and both accounts are adjusted, or something goes wrong and neither account is adjusted. In Kafka, we could implement this, for example, with the following Python code:

```
from confluent_kafka import Producer

producer = Producer({
    'bootstrap.servers': 'localhost:9092',
    'acks': 'all',
    'enable.idempotence': True,

    'partitioner': 'murmur2_random',

    'transactional.id': 'transaction-1',
})

producer.init_transactions()
producer.begin_transaction()

producer.produce("customer.balance",
    key="bob", value="-10")
producer.produce("customer.balance",
    key="alice", value="+10")

producer.commit_transaction()
```

Imports the confluent_kafka Python library

Our recommended settings for reliable producers

Makes the producer compatible with Java producers

ID of the transactional producer

Initializes the transactional producer

Begins an actual transaction

Produces as usual

Commits the transaction

We import our Kafka library for our programming language, which is `confluent-kafka`, the Python library officially supported by Confluent. The library is based on `librdkafka`, our preferred C library for Kafka. First, we initialize our producer with the familiar `bootstrap.servers` and the `murmur2_random` partitioner to remain compatible with Java. Additionally, we need to specify a unique `transactional.id` for this producer. We must ensure that at no time will two producers run with the same `transactional.id`, because they will interfere with each other. Kafka uses this ID to mark a previous producer as a zombie if there is a newer producer with the same `transactional.id`. Before we can use transactions, we must first initialize the producer with `init_transactions()`. Then, we start the transaction (`begin_transaction()`), write our messages (`produce()`), and finally, successfully end our transaction (`commit_transaction()`). To abort a transaction, we call `abort_transaction()`.

If we want to write an application consisting of both a consumer and a producer that guarantees exactly once processing of messages, it becomes more complex. In Kafka, only producers can participate in transactions. Therefore, if we commit an offset directly in the consumer part of the application, it won't be included in the transaction, which could lead to message loss or duplication.

To solve this, we need to prevent the consumer from automatically committing offsets by setting the configuration option `enable.auto.commit` to `False`. This ensures the consumer doesn't commit offsets as messages are processed. Instead, after consuming a message, we use the `send_offsets_to_transaction()` function in the producer to commit the offset within the same transaction as the produced message. This way, the offset commit happens atomically along with the message production, guaranteeing that both are either successfully written or rolled back together. By doing this, we ensure that both the message and its offset are handled within the transaction on the producer side, enabling exactly once processing.

5.2.3 Transactions and consumers

It's crucial that we set the `isolation.level` to `read_committed` in the consumer. Otherwise, our consumer would read uncommitted messages, rendering our use of transactions meaningless.

> **TIP** We recommend setting the `isolation.level` of all consumers to `read_committed` even if we're not planning to use transactions soon. We might change our opinion in the future, and then we'll be very thankful we don't need to update all of our consumers.

But why do we have to set the isolation level, and how do these transactions work? Kafka doesn't reinvent the wheel here; it uses a variation of the *Two-Phase Commit protocol* to atomically write messages to multiple partitions. We can imagine the process akin to booking a trip. We want to book a hotel and a flight simultaneously but know this process can sometimes be error-prone. We don't want to end up with a booked hotel but no flight or vice versa. To handle this, we first call the hotel and reserve a room with

the option to cancel. Of course, the hotel doesn't want to wait indefinitely, so we tell them we'll call back within 10 minutes to confirm the reservation. The hotel agrees. Then, we book the flight ticket as usual. If this fails, we call the hotel to cancel the room or simply wait 10 minutes until it's automatically canceled. If the flight booking is successful, we call the hotel to confirm the reservation.

In essence, this is a similar approach to what Kafka does with transactions. First, we register the partitions we want to write to with one of the brokers coordinating our transaction. Then, we write normally to these partitions but mark the messages as tentative. Once we're done with the transaction, we inform the transaction coordinator, and it writes *commit markers* to the partitions to confirm the transaction. If we abort the transaction, the transaction coordinator writes abort markers.

> **NOTE** The transaction coordinator guarantees that transactions work reliably and that even if our Kafka brokers or producers crash, the guarantees aren't violated.

The broker ensures that only committed messages are being read by our `read_committed` consumer. Figure 5.7 depicts three simple cases. First, we have messages of a started but not yet committed or aborted transaction. In this case, the broker stops sending the messages to our consumer, and it appears as if no new messages arrive. Even messages not belonging to any transaction can't be read, as the broker needs to still guarantee the ordering of messages.

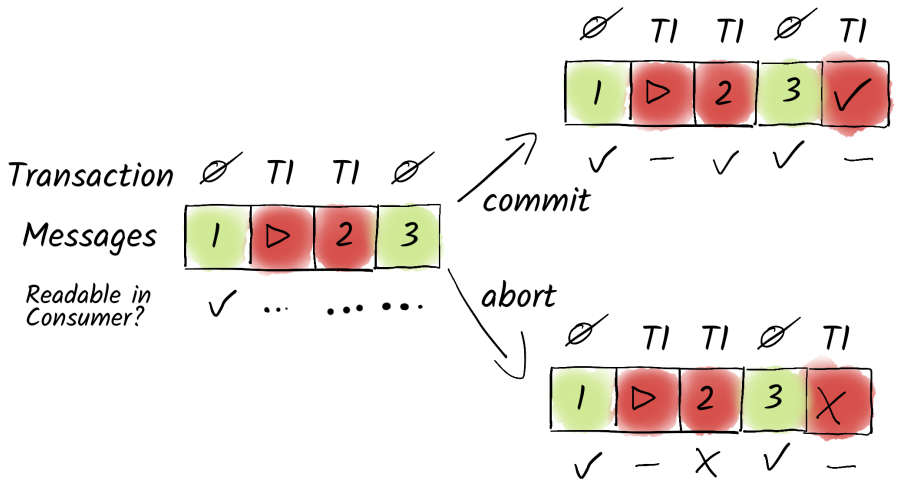


Figure 5.7 Our consumer stops receiving new messages as soon as a transaction remains unfinished because the broker only sends messages up to the start of the unfinished transaction and holds back all subsequent messages, even those unrelated to the transaction. The ordering guarantee still holds. When a transaction-commit is received, all messages are released and sent to our consumer. If a transaction is aborted, our consumer doesn't receive messages belonging to the aborted transaction, but messages not belonging to the transaction are still received normally.

> **NOTE** Our consumer is agnostic of any transaction, including ongoing, committed, and aborted, as the broker is solely responsible for handling those.

In the second case, if the transaction is committed, all messages between the first message of the transaction and the commit marker are released and can be consumed. Finally, in the third case, if a transaction is aborted, all messages belonging to the aborted transaction are skipped. Messages not belonging to any transaction are, of course, forwarded in the correct order to our consumer and can be processed as usual.

> **NOTE** If there's another ongoing transaction before the commit marker, then only messages until the first message of the other transaction can be consumed. We have to wait until that transaction is also either committed or aborted.

This brings certain performance implications. Because the transaction coordinator must write additional messages per partition, the overhead depends on the number of partitions involved in the transaction, but not on the number of written messages. This means that for very short transactions in the millisecond range, the overhead is significant and decreases with longer transactions. For example, when transactions were introduced in Kafka, the developers measured that for transactions lasting 100 ms and producers producing at maximum speed, the overhead was about 3%. But for shorter transactions lasting just 10 ms, the overhead was also about 3 ms, which meant an overhead of 30%.

5.3 Replication and the leader-follower principle

In past chapters, we've already learned quite a bit about replication and partitioning in Kafka. We already know that Kafka uses the *leader-follower principle* to manage its replicas. In this section, we'll look deeper behind the scenes of the leader-follower principle in Kafka. We'll look at how a broker becomes a partition leader and the tasks that go along with it. We'll also look at how followers and leaders interact with each other and what happens if a leader fails.

Now let's look at the whole thing with an example. First, we create the topic `products.prices.changelog.replication-3`:

```
$ kafka-topics.sh --create \
    --topic products.prices.changelog.replication-3 \
    --replication-factor 3 \
    --partitions 3 \
    --config min.insync.replicas=2 \
    --bootstrap-server localhost:9092
Created topic products.prices.changelog.replication-3.
```

Next, we look at how the partitions have been distributed to each broker:

```
$ kafka-topics.sh --describe \
    --topic products.prices.changelog.replication-3 \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog.replication-3 PartitionCount: 3
```

```
  ReplicationFactor: 3 Configs: min.insync.replicas=2
Topic: products.prices.changelog.replication-3 Partition: 0
  Leader: 2 Replicas: 2,3,1 Isr: 2,3,1 Elr:
Topic: products.prices.changelog.replication-3 Partition: 1
  Leader: 3 Replicas: 3,1,2 Isr: 3,1,2 Elr:
Topic: products.prices.changelog.replication-3 Partition: 2
  Leader: 1 Replicas: 1,2,3 Isr: 1,2,3 Elr:
```

Because we have a replication factor of 3 and also only three brokers in total, each of our three partitions is replicated to each of our three brokers. The partition leaders are evenly distributed among the brokers to balance the load. Broker 1 is leader of Partition 2, Broker 2 is leader of Partition 0, and Broker 3 is leader of Partition 1, as visualized in figure 5.8.

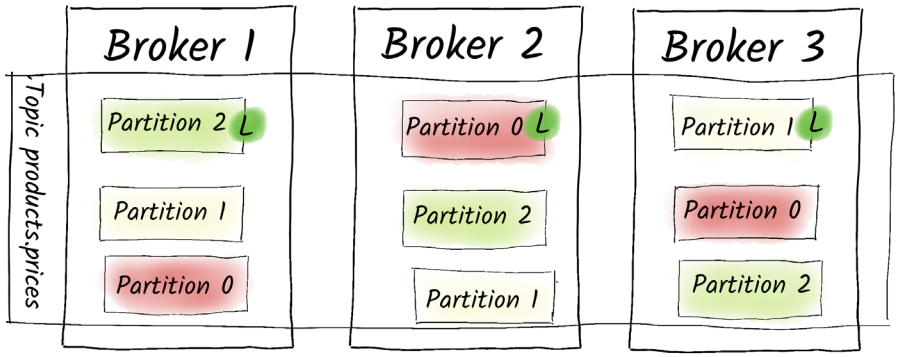


Figure 5.8 The partitions of our topic are distributed among our three brokers. Broker 2 is the leader for Partition 0, Broker 3 is the leader for Partition 1, and Broker 1 is the leader for Partition 2. Because we have a replication factor of 3 and also three brokers, each partition is found on each broker.

In section 5.1 on ACKs, we already learned that a producer always sends messages to the leader of a partition and that the leader, depending on the ACK strategy, waits until the followers have also received the message before confirming receipt of the message to the producer. But how do the messages get from the leader to the followers? Kafka brokers outsource as much work as possible to clients. From Kafka's point of view (or that of a partition leader, as the case may be), followers are consumers that query messages using a special fetch request, as shown in figure 5.9.

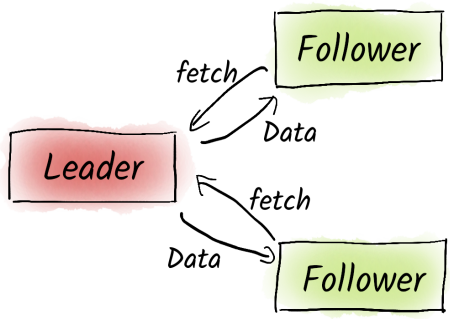


Figure 5.9 Followers continuously fetch data from the leader to stay current and to be ready to take over the leader's role if the leader fails.

Like normal consumers, followers also inform the leader of their current offset, that is, the current read position in the log. In this way, the leader knows when a follower has consumed a message and can then send confirmation to the producer accordingly.

It gets exciting when we look at what happens in the event of a broker's failure and the broker is no longer accessible. For this, we first stop Broker 3 with our `kafka-broker-stop.sh` script and then describe the topic again:

```
$ kafka-broker-stop.sh 3
$ kafka-topics.sh --describe \
    --topic products.prices.changelog.replication-3 \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog.replication-3 PartitionCount: 3
    ReplicationFactor: 3 Configs: min.insync.replicas=2
  Topic: products.prices.changelog.replication-3 Partition: 0
    Leader: 2 Replicas: 2,3,1 Isr: 2,1 Elr:
  Topic: products.prices.changelog.replication-3 Partition: 1
    Leader: 1 Replicas: 3,1,2 Isr: 1,2 Elr:
  Topic: products.prices.changelog.replication-3 Partition: 2
    Leader: 1 Replicas: 1,2,3 Isr: 1,2 Elr:
```

NOTE Depending on the distribution of partitions in our cluster before creating the `products.prices.changelog.replication-3` topic, it's possible that not every broker is a leader of one of the partitions. If this is the case, we need to make sure that the broker we stop is actually assigned as a leader for our topic to be able to follow our example.

We see that Broker 3 is still in the list of replicas, but no longer appears in the list of ISRs. If a broker that is the leader of partitions fails, Kafka (or the controller) must appoint a new leader for the corresponding partitions; otherwise, the topic will no longer be accessible. Therefore, Broker 1 has taken over the leader role for Partition 1, as shown in figure 5.10. Furthermore, not every replica is eligible to become a new leader; only ISRs or eligible leader replicas (ELRs) can be chosen. Why this is the case and how exactly it's determined whether a replica is in sync or eligible as a leader will be discussed in detail in chapter 8.

In Kafka, we can use `kafka-topics.sh` to display all topics or partitions that don't currently have the desired number of replicas. For this we pass the arguments `--describe` and `--under-replicated-partitions`:

```
$ kafka-topics.sh --describe \
    --bootstrap-server localhost:9092 \
    --under-replicated-partitions
  Topic: products.prices.changelog.replication-3 Partition: 0
    Leader: 2 Replicas: 2,3,1 Isr: 2,1
  Topic: products.prices.changelog.replication-3 Partition: 1
    Leader: 1 Replicas: 3,1,2 Isr: 1,2
  Topic: products.prices.changelog.replication-3 Partition: 2
    Leader: 1 Replicas: 1,2,3 Isr: 1,2
```

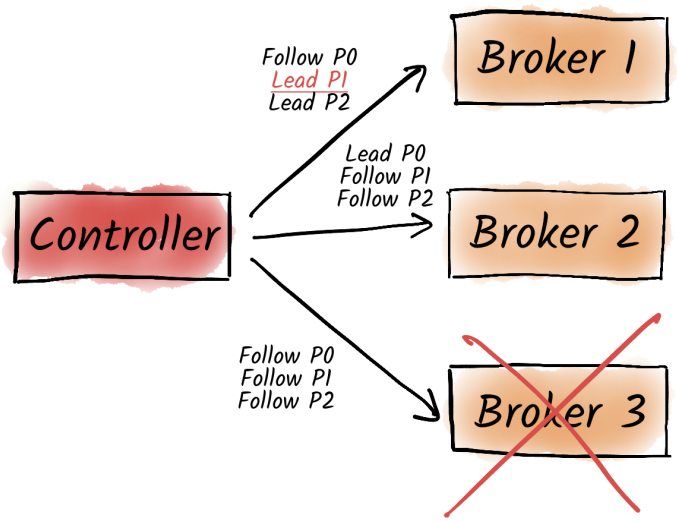


Figure 5.10 The controller has detected the failure of Broker 3. Now that the leader for Partition 1 has failed, the controller picks another broker (e.g., Broker 1) that is in sync for this partition to become the leader.

Unsurprisingly, we see that all partitions of our topic `products.prices.changelog.replication-3` are underreplicated. Equivalently, we can also use `--under-min-isr-partitions` to display all partitions that currently have too few ISRs, that is, fewer ISRs than required by `min.insync.replicas`. In this case, however, we won't get any result (or rather, no partitions will be displayed) because we still have two ISRs, which corresponds to our `min.insync.replicas` requirement. Both of these commands can be extremely helpful for debugging our Kafka cluster, as we can easily determine across topics if something is wrong.

Now that we've seen what happens when a broker goes down, let's take a look at what happens when the broker is available again. To restart the broker, we can simply run `kafka-server-start.sh` again. Then, we look again at the current status of the topic:

```
$ kafka-server-start.sh ~/kafka/config/kafka3.properties
$ kafka-topics.sh --describe \
    --topic products.prices.changelog.replication-3 \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog.replication-3 PartitionCount: 3
    ReplicationFactor: 3 Configs: min.insync.replicas=2
  Topic: products.prices.changelog.replication-3 Partition: 0
    Leader: 2 Replicas: 2,3,1 Isr: 2,1,3
  Topic: products.prices.changelog.replication-3 Partition: 1
    Leader: 1 Replicas: 3,1,2 Isr: 1,2,3
  Topic: products.prices.changelog.replication-3 Partition: 2
    Leader: 1 Replicas: 1,2,3 Isr: 1,2,3
```

Broker 3 reappears in the list of ISRs, but Broker 1 remains the leader of Partition 1, and Broker 3 has to settle for the role of follower, as illustrated in figure 5.11.

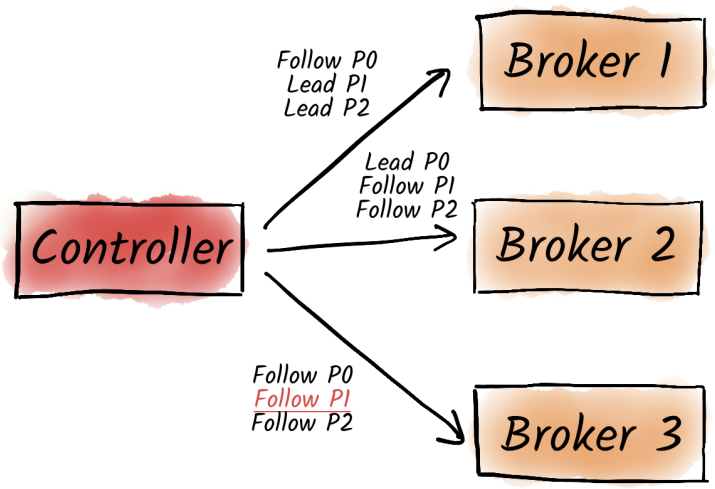


Figure 5.11 Although Broker 3 is accessible again, it doesn't immediately become the leader for Partition 1.

Broker 3 must first sync itself; then, by default, it automatically becomes the leader again after a few minutes for the partition for which it's a preferred leader. We recognize the preferred leader by the fact that it's at the top of the list of replicas. Depending on how we've configured our cluster, either a leader rebalancing takes place automatically after some time or we have to rebalance the leaders manually. With the script `kafka-leader-election.sh`, we can trigger this rebalancing of the leaders manually:

```
$ kafka-leader-election.sh \
    --election-type=preferred \
    --all-topic-partitions \
    --bootstrap-server localhost:9092
Successfully completed preferred leader election for partitions
products.prices.changelog.replication-3-1
```

The command output (`products.prices.changelog.replication-3-1`) indicates the partitions and topics where the preferred leader election was executed. Each partition number is hyphenated with its corresponding topic.

We have to be especially careful with the `election-type` argument. Here, we can choose between *preferred* and *unclean*. The former means that we try to reinstate the original and thus preferred leader of a partition as the leader. We should only perform an unclean leader selection in an extreme emergency. But what exactly is such an emergency situation?

As mentioned earlier, when a partition leader fails, Kafka determines the new leader from the list of ISRs. But what happens if there is no other ISR? In this case, Kafka wouldn't be able to determine a new leader by default, which means that no new messages can be written to the partition and no messages can be consumed from the partition.

With the unclean leader election, replicas that aren't in sync can also be designated as the new leader. Because these replicas most likely don't have all messages, data loss is to be expected, so this method is also called `unclean`. We can also configure unclean leader election in the cluster itself in the broker configuration (`unclean.leader.election.enable`). However, we strongly advise against this. Furthermore, we can apply the `kafka-leader-election.sh` command either to a specific topic only (`--topic products.prices.changelog.replication-3`) or to all topics (`--all-topic-partitions`). Especially in the case of an `unclean-leader-election`, a specific topic should definitely be selected!

WARNING We should only make an unclean leader selection in extreme emergencies. Even then, it should only be done for specific topics.

Finally, let's look at the topic again:

```
$ kafka-topics.sh \
    --describe \
    --topic products.prices.changelog.replication-3 \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog.replication-3 PartitionCount: 3
    ReplicationFactor: 3 Configs: min.insync.replicas=2
  Topic: products.prices.changelog.replication-3 Partition: 0
    Leader: 2 Replicas: 2,3,1 Isr: 2,1,3
  Topic: products.prices.changelog.replication-3 Partition: 1
    Leader: 3 Replicas: 3,1,2 Isr: 1,2,3
  Topic: products.prices.changelog.replication-3 Partition: 2
    Leader: 1 Replicas: 1,2,3 Isr: 1,2,3
```

As we can see, Broker 3 is again leader of partition 1, and Broker 1 has been appointed as a normal follower (see figure 5.12).

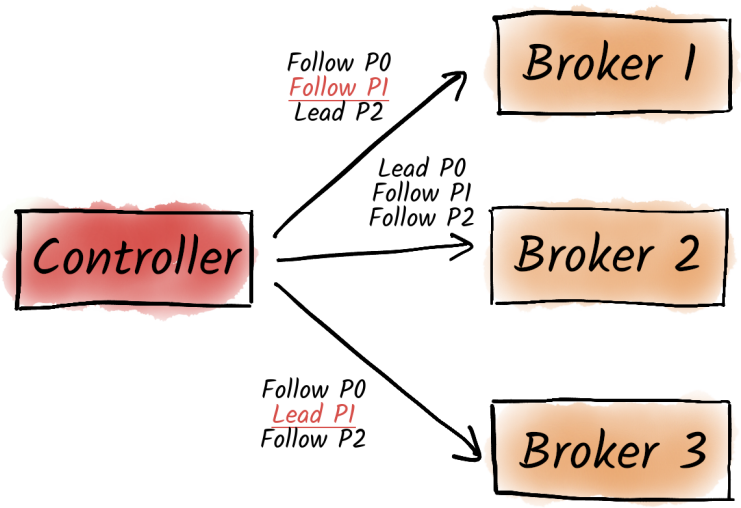


Figure 5.12 After the preferred leader selection (manual or automatic) is performed, Broker 3 again becomes the leader for Partition 1, and Broker 1 becomes the follower for that partition.

Summary

- ACKs are pillars of Kafka's reliability.
- ACKs are optional and configurable for each producer.
- Producers can opt for no ACK (`acks=0`) for speed, but this sacrifices resilience.
- Producers can choose to receive ACKs after the leader receives the message (`acks=1`) or after replication to all ISR (`acks=all`), with the latter now the default and most reliable choice.
- Kafka offers three message delivery guarantees: at most once, at least once, and exactly once. At most once is achieved with `acks=0`; at least once is achieved with `acks=all` and sufficient minimum ISRs; and exactly once requires enabling idempotence and setting `acks=all`.
- Transactions in Kafka enable atomic writes across multiple partitions, ensuring all messages in a transaction are written together or not at all, maintaining data consistency.
- Kafka uses idempotent producers for reliable message production and a variation of the Two-Phase-Commit protocol to manage transaction commit markers, ensuring messages are processed exactly once.
- Consumers must set the `isolation.level` to `read_committed` to avoid reading uncommitted messages, ensuring only fully completed transactions are processed to maintain data integrity.
- Kafka's transaction coordinator ensures reliability, even in the event of broker or producer crashes, although it introduces some performance overhead.
- For each partition, there's a broker acting as the leader, handling all requests, while followers replicate data by fetching from the leader.
- If a partition leader becomes unavailable, an in-sync replica (ISR) takes over the leader role.
- When the previous leader is back in sync, it can become the leader again, as Kafka aims to reinstate the original leader, known as the preferred leader.
- In critical failures, Kafka can perform an unclean leader election by allowing non-ISRs to become leaders, which can lead to data loss.
- Kafka's leader-follower principle ensures high availability and fault tolerance in the event of broker failures.

6 Performance

This chapter covers

- Topic settings to increase performance
- Determining the best number of partitions for a topic
- Kafka broker settings that influence performance
- Producer and consumer performance tuning

In the previous chapter, we saw how to reliably produce messages with Kafka. In this chapter, we'll address the unfinished business of optimizing Kafka. How do we achieve the promised performance with Kafka?

Similar to reliability, when we talk about performance, we first need to clarify what this means to us. We can define performance in different ways. Are we concerned with bandwidth? The bandwidth that the system offers us is often crucial. We measure bandwidth in bytes per second. How many kilobytes/megabytes/gigabytes per second of data throughput do we achieve with our system? When we only talk about bandwidth, we often neglect latency, although it's usually of greater importance to us.

We should never underestimate the bandwidth of a truck fully loaded with microSD cards. The bandwidth of such a truck is enormous. A microSD card,

weighing less than 1 g, now has the capacity to store over 1 TB of data. This translates to a remarkable 1,000 TB (1 PB) per kilogram, and an astonishing 1,000,000 TB (1 EB) per ton, showcasing the extraordinary data storage density achieved by modern technology. But mostly that's not what we want. In the end, we care much more about the end-to-end latency of a system than the theoretical maximum bandwidth. Especially these days, our users expect instant responses from us. In 2008, Amazon estimated that for every 100 ms of additional load time, up to 1% of revenue is lost (https://mng.bz/QD16).

While bandwidth and latency are enormously important for IT systems, there's another aspect that we shouldn't neglect: the resource friendliness of a system. Normally, we don't have unlimited resources at our disposal. We want to be very careful with our resources to avoid overspending and to minimize the environmental effect of our IT systems.

Especially in a system as extensively configurable as Kafka, it's important to understand how we optimize our systems. In our Kafka training courses, participants have a lot of fun testing Kafka's limits. Participants enjoy optimizing their systems from 0.2 MB/s throughput to more than 200 MB/s on a single moderately sized machine.

But where do we configure performance in Kafka? The short answer is that Kafka is intrinsically tuned for performance in many places (see figure 6.1), but we still need to pay attention to the configuration settings in all components.

Kafka's performance optimization starts with the basic assumptions it makes. Kafka is a child of the 21st century. We assume that hard disks are relatively cheap and that main memory can also be installed in larger quantities. But much more important is another assumption: we know that individual IT systems aren't particularly reliable. Instead of trying to avoid hardware failures at all costs, Kafka takes a different approach in that it's okay if subsystems fail because Kafka can still continue to function. Kafka also tries to avoid the one-size-fits-all approach, and we can also make different guarantees about performance and reliability in a single Kafka cluster.

Architecturally, Kafka is trimmed for performance from the ground up. It starts with the log as the central data structure. A log, as we learned in the first chapters, is one of the simplest data structures. The access patterns of logs are very predictable, which means that Kafka runs at a high performance level even on classic spinning disks. This is because although SSDs beat classic hard disks in almost all categories, the price of the latter is still significantly lower.

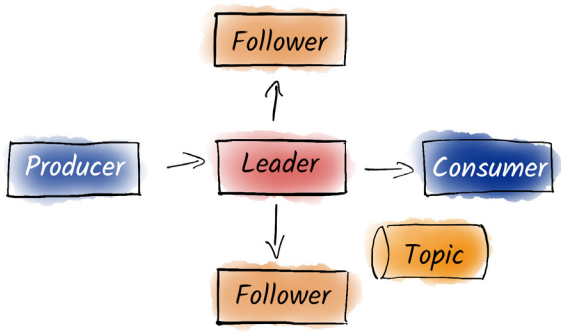


Figure 6.1 We can configure performance in Kafka in many places: in the topic configuration, the brokers, and all clients.

Recall that Kafka is capable of transporting arbitrary data formats. This not only gives us greater flexibility but also forces Kafka to forgo common features of other messaging systems. By not even trying to interpret messages, Kafka doesn't lose any performance. Kafka's whole approach of having the messaging system perform as few tasks as possible and outsourcing the actual work to the clients serves to further improve the performance of the overall system.

6.1 Configuring topics for performance

Before we start tweaking the configuration settings of our brokers and clients, we need to ensure that our topics are configured correctly. In the first chapters, we learned that we partition our topics primarily for horizontal scaling, which enables parallel processing within our system by alleviating the consumer as the bottleneck. Moreover, this approach facilitates handling larger data volumes than a single server could manage, although it's important to bear in mind that this approach can affect the ordering of messages.

6.1.1 Scaling and load balancing

The number of partitions not only affects how our producers send messages and how the load is distributed across brokers, but more importantly, how we scale our consumers. In principle, many independent consumers can consume messages from the same topic. Let's recall our example from chapter 4, section 4.2.2, where we have one consumer analyzing the effect of prices on sales by consuming the `products.prices.changelog` topic, while another updates the prices in our online shop.

On the one hand, we don't want these independent consumers to get in each other's way. We can easily ensure this because the consumers explicitly query Kafka for all new prices starting at a certain offset. If each independent consumer manages the offsets for itself, they won't get in each other's way either.

Now we have the situation where we have many partitions but only one consumer that consumes all messages. To scale our consumers and also manage the offsets in Kafka, we use consumer groups, which we've already learned about. We do this by launching multiple instances of the same consumer, and Kafka helps these consumers divide the work among themselves. However, we can't distribute the messages arbitrarily to consumers of a consumer group, but we distribute whole partitions to consumers to guarantee the order of our messages as shown in figure 6.2.

Partitions not only help us distribute the load among different broker but also allow us to scale our consumer groups to process data in parallel. However, this partitioning introduces some limitations. Kafka brokers ensure that the messages are stored in the partition in the same order as they arrived. The order isn't guaranteed across partitions. This means that even though it's perfectly fine for multiple producers to write to the same topic, ordering among messages produced by different producers isn't guaranteed. To ensure message ordering, the messages must come from the same producer and go to the same partition.

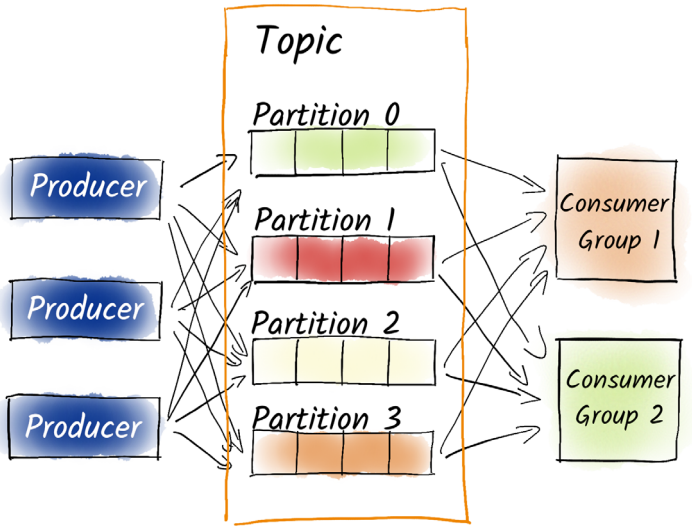


Figure 6.2 In Kafka, we partition topics and distribute these partitions to different brokers for better load balancing and parallelization. Producers decide on which partition to produce the message based on the key or round-robin. On the consumer side, we use consumer groups to process the partitions in parallel. Different consumer groups are isolated from each other.

Furthermore, ordering guarantees only hold if `enable.idempotence=true` is set, as discussed in chapter 5. This setting ensures message uniqueness based on a combination of the producer ID and sequence ID, which prevents Kafka from getting confused while receiving and processing messages. Without this configuration, even a single producer may encounter problems with ordering in the event of retries or failures.

In the context of partitions, it's also always important to think about how many partitions are right for the particular use case. We might get the idea that it's advantageous to have many partitions because we can then both scale better and parallelize our work better. But the question of the right number of partitions is unfortunately not a simple one, and there's usually no perfect answer to it.

6.1.2 Determining how many partitions are needed

We begin by identifying where our bottleneck lies: Is it the consumer or Kafka itself? In many cases, the consumer sets the upper limit on performance. We determine the number of partitions based on the required number of consumers to process messages, excluding big data use cases. For example, if a single consumer needs approximately 100 ms to process a message (e.g., writing to a slow data store), it can handle 10 messages per second. To process 100 messages per second at peak, we'd need at least 10 consumers and therefore 10 partitions.

> **NOTE** Sometimes, the bottleneck originates from external systems rather than Kafka itself. For instance, if a consumer's performance is constrained by a slow data store, increasing Kafka consumers or partitions may not significantly enhance performance. It's crucial to analyze and pinpoint whether the bottleneck lies within Kafka or stems from external dependencies.

For flexibility, having a few more partitions than required by our consumer's performance is advisable. We aim for easily divisible numbers, facilitating seamless scaling of consumers. Ideally, this divisibility extends to the number of brokers, though the total partitions generally outnumber brokers. From our experience, starting with 12 partitions is typically sufficient for most scenarios.

To address these considerations, we must answer two critical questions: First, is our current partition count too small, and will we require more than 12 parallel consumers within the same group processing data concurrently? This scenario indicates very high throughput. If needed, we double the partition count. We continue doubling until we determine that 24 parallel consumers, or another number, sufficiently meet our needs.

Second, what are the implications of having too many partitions? While additional partitions can increase throughput, they introduce complexities. Each partition demands client resources such as RAM, especially when clients interact with multiple partitions across diverse topics. In the past, managing numerous partitions under ZooKeeper led to prolonged periods of inaccessibility during outages, with leader redistribution taking considerable time per operation. This could accumulate to minutes for hundreds or thousands of partitions, temporarily rendering some inaccessible.

Moreover, an excessive number of partitions can overload our Kafka cluster, straining CPU, memory, and file handling resources. System complexity also escalates, potentially degrading performance and lengthening recovery times during broker failures. Excess partitions complicate operational tasks such as monitoring and troubleshooting. Therefore, an optimal partition balance is crucial to avoiding unnecessary overhead while effectively meeting performance demands.

Next, we consider whether to optimize the number of partitions for each use case or to adopt a default number. We'll see later, in chapter 12, that we need the same number of partitions in two topics to join them effectively.

> **TIP** Considering these factors, we often recommend adopting a default partition number, such as 12, across all use cases, even if initially this is more than required. For high-throughput topics, increasing partition numbers may be necessary, but caution is advised regarding cost implications. Notably, some cloud providers (e.g., Confluent) base pricing on partition count, making tighter calculation economically advantageous.

When configuring Kafka clusters, understanding partitioning limitations and best practices is critical for achieving optimal performance and resource utilization. While Kafka itself lacks strict partitioning limits, adhering to general guidelines is essential, particularly in ZooKeeper-managed setups. It's recommended to have at most 4,000 partitions per broker, with Kafka clusters accommodating up to 200,000 partitions. KRaft, however, allows for significantly larger clusters because it eliminates the dependency on ZooKeeper and manages metadata more efficiently.

Assessing the necessity of additional partitions is vital, as even a few partitions at maximum throughput can exceed other system capacities such as relational databases or

MongoDB. Therefore, while Kafka's partitioning flexibility is advantageous, practical considerations beyond Kafka's capacity are paramount for maximizing overall system efficiency.

At the time of writing, there's a new feature being developed called *Queues for Kafka* that introduces *share groups*, allowing cooperative consumption of records from topics. With shared groups, the number of consumers could exceed the number of partitions, potentially reducing the need for over-partitioning to achieve high parallelism. This approach is particularly beneficial for scenarios with relatively slow consumers, where queue-like behavior is desired.

However, one critical question arises: How would order be guaranteed within a shared partition? In this model, while Kafka ensures ordering within a partition for traditional consumer groups, shared groups may not maintain strict order because multiple consumers can pull messages from the same partition. This behavior mirrors traditional queue systems, where order is typically less critical.

> **WARNING** While shared groups could provide more flexibility in certain use cases, they aren't intended to replace traditional partitioning for high-throughput scenarios, especially where strict order guarantees are required. For such cases, traditional consumer groups are still be necessary.

6.1.3 Changing the number of partitions

We discuss the number of partitions in detail here because it can't be changed easily, at least if we want to rely on the correct order of messages. In Kafka, the number of partitions can only be increased. It's not possible to reduce them for a topic because Kafka would then face the unsolvable task of deciding what to do with the messages in the deleted partitions. Simply moving messages to other partitions would break Kafka's guarantee of message order.

When we increase the number of partitions, the newly created partitions are initially empty, causing an immediate imbalance in data distribution. This imbalance may not be a problem if we retain data for only a short period (e.g., a few days). Once the retention time expires and older messages are deleted, the amounts of data in the partitions will naturally equalize again.

A much bigger problem arises when increasing partitions because it disrupts the order of messages. This happens because Kafka uses the hash of the message key, modulo the number of partitions, to determine the target partition for each message. If the number of partitions changes, the modulo calculations will yield different results, causing messages with the same key to end up in different partitions. The disruption in message order persists until the old data, which adheres to the previous partitioning scheme, is deleted after the retention period. Figure 6.3 illustrates how this re-partitioning affects message distribution.

Of course there are use cases where this isn't a problem, but still we should think very carefully if we want to change the number of partitions afterward. If we don't use

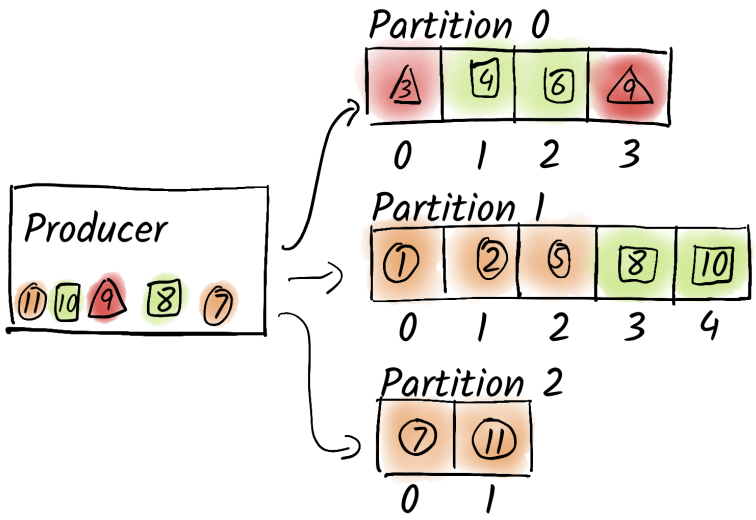


Figure 6.3 After changing the number of partitions, the partition number for a particular message key usually changes. In this example, messages with the key circle are now produced to partition 2 instead of partition 1, and messages with the key square are now produced to partition 1 instead of partition 0.

keys because we don't care about the order of the data in the topic in general, it's no problem to increase the number of partitions, except for the uneven distribution of the messages to the partitions.

If we can't allow the order of messages to get lost and we still want to increase the number of partitions (or if we miscalculated and want to reduce the number of partitions), we have no choice but to create a new topic and delete the old topic. How we best approach this depends on the use case. In the simplest case, our system processes messages immediately after they are written, and we have the luxury of being able to set up maintenance windows. In this case, we could stop our producer at the beginning of the maintenance window, wait until all messages have been processed, and then delete and recreate the topics in question.

If we can't afford a noticeable downtime, but don't store data in the topic indefinitely, we can create a new topic with the target number of partitions and let the producer produce into the new topic. The consumers need to finish the consumption of the old topic and then migrate to the new one. As soon as all consumers are migrated, we can delete the old one.

If we want to keep the data indefinitely, such a migration is more costly. Usually, we first create a new topic with the desired configuration. Then, we use, for example, Kafka Streams to copy the data from the old topic to the new topic to guarantee that the order of the messages is respected when we use keys. After creating the topic and copying the data, we need to move our consumer from the old topic to the new topic. This alone

isn't a trivial process because we have to figure out how to translate the offsets because the offsets for the new partitions will be different, as shown in figure 6.4. Otherwise, our consumers might either miss messages or consume messages twice, which can lead to problems depending on our application. Finally, we migrate all the producers to the new topic and delete the old topic.

TIP Idempotent consumption can be achieved by including an event ID. This allows systems to determine if an event has already been processed, preventing duplicate processing.

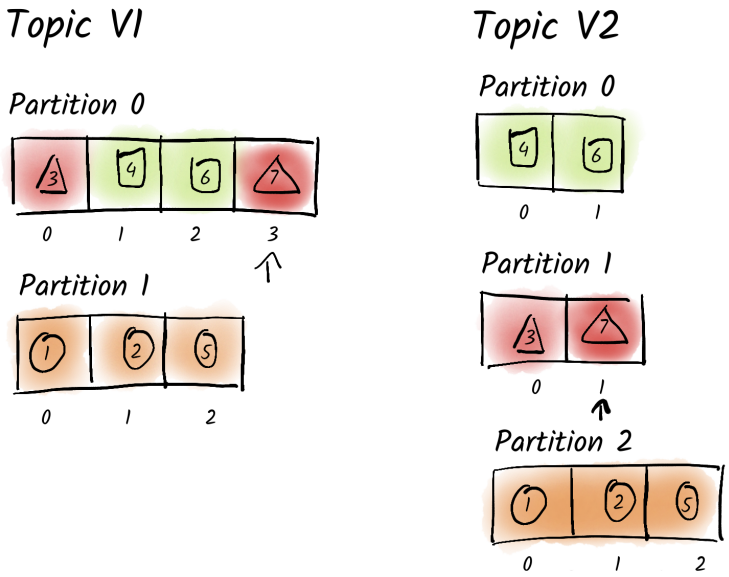


Figure 6.4 The message 7 with the key triangle was previously stored at offset 3 of partition 0. After creating a new topic with three partitions and migrating the data, this message is now stored at offset 1 of partition 1.

WARNING As we can see, changing the number of partitions afterward is time-consuming and error-prone. Therefore, it's recommended that we think carefully in advance about the requirements for our topics. However, creating a new topic and migrating data is manageable. It also provides an opportunity to review and improve configuration, naming conventions, and payload formats, as well as often transitioning from JSON to Avro or Protobuf.

6.2 Producer performance

Now that we know how to optimize performance for our topics, we can move on to take a closer look at our route from producer to broker to consumer. Because the

producer is the point where messages are produced into the Kafka cluster, we can have a big effect on the performance and resource utilization of the overall system here.

6.2.1 Producer configuration

To produce messages, producers usually use either the Java Kafka library or, if another programming language is used, a library based on `librdkafka`. The program code typically passes only the message with a value and an optional key. The library then decides into which partition this message should be produced. It's also possible to implement custom partitioning algorithms, but this should be avoided in general. Letting the producer decide into which partition to produce a message might seem counterintuitive, but this ensures less coordination overhead on the brokers and thus higher performance. This approach gives producers the flexibility to implement their own partitioning algorithms for the rare cases where this is needed.

Kafka improves data throughput through *batching*, that is, grouping multiple messages into a batch before sending them to a broker. Larger batches mean fewer network requests and better efficiency. Batching can be configured independently on each producer using two settings: `batch.size` and `linger.ms`. The `batch.size` setting specifies the maximum size of a batch, with a default value of 16 KB (16,384 bytes). The `linger.ms` setting specifies how long the producer should wait for additional messages before sending a batch. By default, this is set to 0, meaning the producer will send messages as quickly as possible.

Even with the default setting of 0, batching often occurs because messages aren't always sent as quickly as they are produced. This allows us to benefit from batching naturally when multiple messages fit into a batch. For scenarios where immediate delivery is less critical, we can increase throughput by setting a larger `batch.size` and a higher `linger.ms` value. Note that `batch.size` is measured in bytes, and `linger.ms` is in milliseconds. Figure 6.5 illustrates the batching process.

We believe the default `batch.size` in Kafka, set at 16 KB, is too small for most use cases. Increasing the batch size can significantly enhance the efficiency of data processing and storage, as larger batches reduce the overhead associated with individual message handling. The maximum permissible `batch.size` is determined by `max.message.bytes`, allowing for tuning up to 1 MB by default. Similarly, the default `linger.ms` is often set too low at 0 ms. Setting `linger.ms` to approximately 10 ms typically improves both throughput and latency, especially for producers sending more than 100 messages per second.

For lower message rates, `linger.ms` can be increased further, though this might degrade latency. Generally, `linger.ms` adjustments are less critical unless dealing with a high-throughput cluster. However, increasing `batch.size` can drastically enhance performance and reduce cluster load by minimizing the number of network messages that need processing.

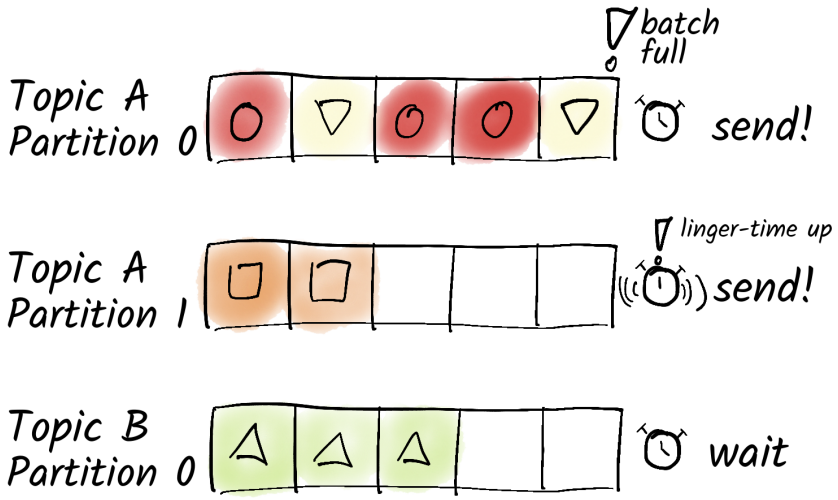


Figure 6.5 The producer batches messages for each topic and partition. A message is sent as soon as either the batch is full or the linger time is up.

> **TIP** For most use cases, we recommend setting `batch.size` to up to 1MB and `linger.ms` to around 10ms. Always monitor your system's performance metrics after adjusting these settings to ensure optimal configuration for your specific workload.

We've already dealt with acknowledgments (ACKs) in chapter 5 and saw that we can use ACKs to control how reliable or performant messages are delivered from the producer to Kafka. We can improve performance at the expense of reliability with ACK values of 1 or even 0, but the ACK value mostly depends on our reliability requirements and not on our performance requirements.

Instead, there's another set screw by means of compression, which can not only significantly improve the performance of our Kafka environment but also reduce its resource requirements. We can use the `compression.type` setting to influence the type of compression. By default, compression is turned off, but Kafka supports some algorithms, so we can choose between `none`, `gzip`, `snappy`, `lz4`, or `zstd`.

Depending on the requirements, such as compression ratio, compression time, and decompression time, one algorithm may be better suited than the others. Usually, `zstd` or `lz4` provide the best balance between throughput and CPU overhead. Instead of compressing individual messages, Kafka compresses whole batches along the entire route from producer to consumer. This is possible because a producer uses a separate batch for sending messages for each partition; that is, all messages for one partition are grouped in one batch, and messages for another partition are collected in a separate batch.

Even better, more active batching usually leads to improved compression. The producer compresses the entire batch once and sends the compressed batch to the leader. The leader stores it unchanged on the local hard disk and transmits this compressed batch unchanged to the followers. Only when the consumer receives the batch is it unpacked and split into individual messages, as shown in figure 6.6. This approach ensures that compression not only provides a one-time reduction in size but also decreases network load at multiple points and minimizes the hard disk space required for storage.

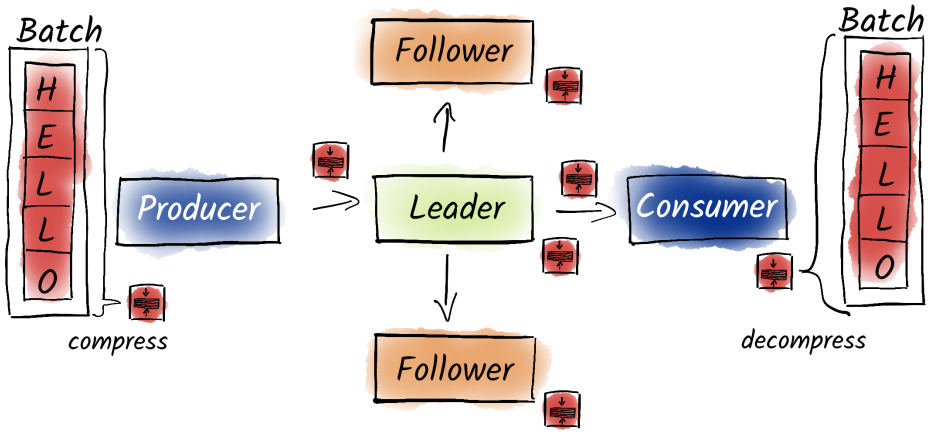


Figure 6.6 When we turn on compression, the producer compresses entire batches. Not only do we send these compressed batches over the network, but they are also stored on the brokers in compressed form. Only the consumer has to finally unpack the batch to extract the individual messages.

It's important to note how Kafka handles offsets within compressed batches. When a consumer needs to start at an offset within a compressed batch, it will receive the entire batch, decompress it, and begin processing from the correct message within that batch. This design aligns with Kafka's philosophy of treating data as immutable logs and avoiding the need for the broker to parse or modify message contents.

Additionally, if some producers write compressed messages and others don't for the same topic, Kafka handles this gracefully. Each producer specifies its own compression settings, and Kafka doesn't enforce uniform compression across producers. Therefore, one producer can send compressed batches while another sends uncompressed messages, and both can coexist within the same partition. However, note that mixed compression configurations may lead to inconsistencies in throughput and storage efficiency, as compressed batches consume less space and reduce network overhead compared to uncompressed ones. For optimal performance, it's generally recommended to standardize compression settings across producers for a given topic.

6.2.2 Producer performance test

The best way to test the performance of our producer is to use the Kafka-provided tool called `kafka-producer-perf-test.sh`. We can use this tool to test different configuration settings and message types for performance effect. Always keep in mind when doing these tests that this isn't a substitute for performance testing with real producers. For such end-to-end performance tests, tools such as *Apache JMeter* (https://jmeter.apache.org/) are better suited.

> **WARNING** The `kafka-producer-perf-test.sh` tool will, depending on its configuration, produce many messages and therefore create multiple gigabytes of data on your disk.

The easiest way to start `kafka-producer-perf-test.sh` is with the following command. For this, we assume that we've already created a topic `performance-test`.

```
$ kafka-producer-perf-test.sh \
    --topic performance-test \
    --num-records 1000000 \
    --record-size 10000 \
    --throughput -1 \
    --producer-props bootstrap.servers=localhost:9092
```

Here, we produce 1,000,000 messages (`--num-records`), each 10,000 bytes in size (`--record-size`), without limiting throughput (`--throughput -1`)—so as fast as possible. We produce the data into the topic `performance-test` (`--topic`). We have to set the bootstrap server via the `--producer-props` flag. After running this command, we get an evaluation that looks similar to the following:

```
39481 records sent, 7896.2 records/sec (75.30 MB/sec),
238.1 ms avg latency, 399.0 ms max latency.
62616 records sent, 12523.2 records/sec (119.43 MB/sec),
164.1 ms avg latency, 253.0 ms max latency.
[...]
1000000 records sent, 12317.546345 records/sec (117.47 MB/sec),
165.43 msavg latency, 399.00 ms max latency, 157 ms 50th,
211 ms 95th, 287 ms 99th, 353 ms 99.9th.
```

Let's take a closer look at the output. The command gives us the current statistics every few seconds. It's not uncommon for `kafka-producer-perf-test.sh` to start slower at the beginning and for the throughput and latency to improve after a few outputs. This behavior is often attributed to the Java Virtual Machine (JVM) warming up, meaning that the JVM gradually optimizes the code execution and needs some time to allocate buffers and other objects at the beginning.

In our example, after the desired number of messages are produced, we can look at the results. We see that Kafka sent approximately 12,317 messages per second at a total throughput of 117.47 MB/s. Our average latency was 165 ms.

If we run the performance test only once, we should be careful about interpreting too much into the result. To run performance tests correctly, we need to run several repetitions and then evaluate them. But it gives us a first clue regarding how we could further improve the performance.

For example, we know that increasing the size of the batches and simultaneously increasing the linger time leads to improved throughput. We set this with the `--producer-props` flag. Let's give it a try:

```
$ kafka-producer-perf-test.sh \
    --topic performance-test \
    --num-records 1000000 \
    --record-size 10000 \
    --throughput -1 \
    --producer-props bootstrap.servers=localhost:9092 \
                     batch.size=100000 linger.ms=100
...
1000000 records sent, 45879.977978 records/sec (437.55 MB/sec),
64.61 ms avg latency, 279.00 ms max latency, 58 ms 50th,
101 ms 95th, 126 ms 99th, 178 ms 99.9th.
```

We can see here that instead of about 117 MB/s as at first, we achieve a whole 437.55 MB/s data throughput. Interestingly, even the latency drops from the original 165 ms to 65 ms. The reason for the improvement in latency here is that the batch size was probably chosen too small for the use case. We remember that the batch size is set to 16 KB by default. So if we generate messages of about 10 KB each, we don't use batching at all, and the messages pile up. This leads to degraded latency.

This example should make it clear once again that it's essential to test the performance in your own cluster with the given use cases. Table 6.1 provides information on what should be considered when tuning the performance of a producer, but it can't replace a real performance test, using production-like data under conditions similar to those in a live environment. Every use case is different, and the settings should always be chosen carefully.

Table 6.1 Producer settings

Configuration Setting	Description
`acks`	`0`: Higher throughput, data loss possible `1`: Slightly lower throughput, data loss less likely `-1/all`: Even slightly lower throughput, higher reliability
`enable.idempotence`	`false`: Not recommended `true`: Guarantees the order and uniqueness of messages and only compatible with `acks=-1`
`compression.type`	`none`: No compression `gzip`: Better compression, more CPU load `zstd`, `snappy`, `lz4`: Can be more performant than `gzip`

Table 6.1 Producer settings (*continued*)

Configuration Setting	Description
`batch.size`	Batch size in bytes: In general, the higher the batch size, the higher the latency and throughput. Don't be afraid of setting `batch.size` to big numbers (maximum 1 MB).
`linger.ms`	Time that is waited to fill a batch: In general, larger values lead to higher latency but also higher data throughput.

6.3 Broker configuration and optimization

The brokers comprise the central entity in Kafka. All data is stored in the brokers, the clients communicate with the brokers, and the brokers manage themselves using the coordination cluster. The brokers try to do as little as possible themselves and outsource as many performance-critical operations as possible to the clients. At the end of the day, Kafka brokers only push bytes from network sockets to disk (when producing) and from disk to network sockets (when consuming).

Many database systems, for example, try to trick and bypass the operating system where they can to get better access speeds to the data they need. Kafka, on the other hand, works *with* the operating system and not against it. Kafka tries to make it as easy as possible for the operating system to hold the data that Kafka needs. The core idea is sequential reading and writing. Kafka's access patterns are very predictable. We either write to the end of a file or read sequentially in a file. Even on classical hard disks with rotating disks, these modes of access are performant.

However, Kafka goes further in its optimization. We've already discussed that Kafka doesn't commit the written data to the filesystem. This means that Kafka writes data only to the page cache in main memory and doesn't instruct the operating system to write this data to disk. Kafka relies on the operating system to do this on its own later in the background. Nevertheless, Kafka sends an ACK to the producer after writing the data to main memory, even though the data would be lost in the event of a power failure. This may seem disastrous at first glance, but Kafka's reliability comes not from writing data to disk immediately, but from replicating data across multiple brokers. This assumes that all brokers don't crash at once and lose the data. If the brokers are running on different machines in different parts of a data center, this is usually a legitimate assumption.

Because Kafka's data format is the same everywhere, that is, the data produced by the producer is transferred over the network in the same way, written to disk, and thus received by the consumer, the brokers can use a feature of the Linux kernel called *zero-copy transfer*. Actually, the data produced by a producer should be written from the network socket in main memory to the page cache in main memory. But because the data is identical, the Linux kernel only has to change a few pointers, and the data ends up in the page cache without having been moved. However, this optimization is applicable

only in scenarios where Transport Layer Security (TLS) isn't used, as encryption processes introduce additional steps that preclude zero-copy benefits.

6.3.1 Optimizing brokers

So how can we influence the broker performance ourselves? Kafka's default settings on the broker side are already trimmed for performance. Before we run Kafka, it's recommended to adjust some operating system settings. Of particular importance is to increase the maximum number of open *file descriptors* because the brokers keep every log segment in every partition as an open file, which adds up quickly.

We should also make some settings regarding virtual memory so that the operating system starts writing data to disk early in the background and doesn't block our broker processes if possible. We also want to disable or minimize the *swappiness* (page swapping) of the operating system and make some settings at the network level.

For example, if securing data traffic with TLS is required, it's advisable to double the network threads (`num.network.threads`) from three to six. If multiple disks are used, the I/O threads (`num.io.threads`) should match the number of disks to optimize throughput. For high numbers of producers and consumers, increasing the maximum queued requests (`queued.max.requests`) beyond the default of 500, aligning with the number of client connections, can help maintain performance. Generally, these adjustments are recommended primarily for production environments handling large data volumes. Seeking professional guidance for complex configurations is advisable, as covering every use case is beyond the scope of this text.

TIP At the time of writing, Cloudera provides a good overview of the recommended parameters, available at https://mng.bz/XxQ9.

6.3.2 Determining broker count and sizing

Note that it's not possible to influence these parameters in every scenario; for example, if we want to run Kafka in a Kubernetes environment, we usually have no influence on these values. Instead, we can monitor extensively to at least notice when certain limits are reached and to give us time to figure out how to handle the situation.

The basic rule for any kind of optimization is to first understand, then measure, and then optimize. This is especially true for Kafka, as the broker configuration is already quite performant by default. However, the number of brokers is something we should think about early on. Most smaller environments start with three brokers. This allows us to use a replication factor of 3 for our partitions, and in the event of broker maintenance, we can survive the failure of another broker without any downtime. We need to estimate the number of brokers based on what types of computers we have available: How much main memory and hard disk space do we need? What data throughput do we expect? What kind of network connection do we have? How long do we want to store data?

Basically, we need to find a good balance between brokers that are too big and brokers that are too small. The more brokers we have, the less we're bothered by the failure of a single broker, but the more management overhead we have to deal with. In addition, we shouldn't choose the size of the machines as too small because Kafka itself consumes some CPU and memory, which becomes less relevant as the machines grow.

For example, if we use computers with 2 TB of disk space, and we process about 1 TB of data a day that needs to be stored for seven days, we would need at least 4 brokers to store the 7 TB of data. But we haven't replicated that data yet. With a replication factor of 3, we would now need brokers to store a total of 21 TB of data, or 11 brokers. With this amount of data, we should also consider what data rates we can expect and what our network interface can do.

TIP For smaller installations, it's common to start with three brokers first and add more brokers if the brokers become overloaded with data volume.

6.4 Consumer performance

Unlike many traditional messaging systems, consumers in Kafka are fetch-based. That is, Kafka brokers don't proactively send data to consumers; instead, consumers fetch data from Kafka on their own. Consumers decide on their own when to fetch, what data to get, and how much. The advantage of this approach is that our consumers can't be overloaded, unless a bug has crept into the program code.

6.4.1 Consumer configuration

Similar to the producer, we can improve the bandwidth or the latency for the consumers. For this, we have two configuration settings in the consumer. First up, the `fetch.min.bytes` setting tells the broker to wait for at least this number of bytes before responding to the consumer. By default, `fetch.min.bytes=1`, so the broker sends a response to the consumer as soon as there's at least 1 byte of new messages. The broker also doesn't send single messages to the consumer, but always whole batches. It's even possible to receive several batches in one response.

Of course, the broker doesn't wait forever before sending a response to the consumer, so the consumer can also use the `fetch.max.wait.ms` setting to configure the maximum amount of time the broker can wait before sending a response, even if there's less than `fetch.min.bytes` of data. By default, the value is set to 500 ms. This means that even if there are no new messages, the consumer will receive a response to its request after 500 ms. After the consumer has received the response and the messages have been processed, the consumer requests the next messages. Of course, this is done with a higher offset if we've processed messages.

WARNING Never set `fetch.min.bytes` or `fetch.max.wait.ms` to 0, as this could potentially overload your brokers with requests. With every fetch request being immediately answered by the broker and the consumer promptly requesting new data, the configuration could lead to excessive load. Considering that

a request is sent in parallel for every partition and the latency between client and broker may be as low as 1 ms, even with just a single consumer and topic, this setup could result in up to 10,000 requests per second.

6.4.2 Consumer performance test

Similar to the producer, we can also test the performance of a consumer. For this purpose, it's best to take the topic in which we produced with `kafka-producer-perf-test.sh`. The tool for consumer performance tests is called `kafka-consumer-perf-test.sh`:

```
$ kafka-consumer-perf-test.sh \
    --topic performance-test \
    --messages 1000000 \
    --bootstrap-server localhost:9092
```

This command's behavior is slightly different behavior from `kafka-producer-perf-test.sh`. We can't simply affect the consumer configuration via the command line, but must pass a configuration file. For example, if we want to set the `fetch.min.bytes` high to 1 MB and the `fetch.max.wait.ms` explicitly to 500 ms, we create the `consumer.properties` file with the following content:

```
fetch.min.bytes: 1000000
fetch.max.wait.ms: 500
```

Now, we start `kafka-consumer-perf-test.sh` again, including the configuration file:

```
$ kafka-consumer-perf-test.sh \
    --topic performance-test \
    --consumer.config ./consumer.properties \
    --messages 1000000 \
    --bootstrap-server localhost:9092
```

The output is unfortunately very confusing:

```
start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg,
nMsg.sec, rebalance.time.ms, fetch.time.ms, fetch.MB.sec, fetch.nMsg.sec
2025-02-01 10:59:37:208, [...]
```

We suppressed the entire output—it's long—and it's easier if we look at the output of the two consumer tests in table 6.2.

Even in table form, we need to look closely to understand what the performance test is trying to tell us. The lines with `start.time` and `end.time` show us how long the commands took. With our customized `consumer.properties`, the command takes only 13 seconds instead of 17 seconds. Both commands process the same number of messages (`data.consumed.in.nMsg=1000000`) and the same amount of data (`data.consumed.in.nMB=9536.7432`), which is about 9.5 GB. We thus achieve about 734 MB/s read with

Table 6.2 Consumer performance output

Name	Results with Default Values	Results with Our Consumer Properties
start.time	2025-02-01 10:44:18:30	2025-02-01 10:59:37:208
end.time	2025-02-01 10:44:35:742	2025-02-01 10:59:50:200
data.consumed.in.MB	9536.7432	9536.7432
MB.sec	547.0512	734.0473
data.consumed.in.nMsg	1,000,000	1,000,000
nMsg.sec	57362.4735	76970.4433

our configuration file and 547 MB/s (`MB.sec`) with the default values. The other values are only relevant if we use consumer groups. If we don't use them (as in our case), then we can ignore them because they show numbers of no interest to us.

We observe a substantial enhancement in consumer performance through a few configuration adjustments. Nonetheless, it's essential to remember that consumer performance is typically constrained not by Kafka, but by the speed at which the consumer processes data. Additionally, our performance tests reveal that our consumer significantly outpaces our producer. While our producer attained speeds of up to 437 MB/s, our consumer now reaches speeds of up to 734 MB/s.

> **NOTE** The actual values and the difference in performance strongly depends on our hardware configuration. Especially in a local Kafka setup like ours, the difference might be relatively low.

The tests we've performed here aren't representative and show only exemplary values, so take the results with caution. For real performance tests, we should also use real-looking data and run Kafka in a production-like environment.

We've run our tests here for one topic at a time with one partition and no replication. This way, we have no additional effort to replicate data, but we can't parallelize our operations because we would need more than one partition.

We've seen that individual consumers can often receive the data seamlessly. The bottleneck is usually not Kafka itself, but either the network connection or the further processing in our own code. To make our consumers really performant, a single consumer is often not enough. Instead, we use consumer groups to scale our consumers and parallelize data processing. Additionally, we need consumer groups to store our offsets using Kafka's on-board resources. Typically, this means that we use consumer groups even when we have only one consumer, which means we don't have to worry about our offsets. Let's now delete the `performance-test` topic as this uses a lot of storage on our disk:

```
$ kafka-topics.sh \
    --delete \
    --topic performance-test \
    --bootstrap-server localhost:9092
```

Summary

- High throughput doesn't imply low latency, but both can be equally important.
- Partitioning allows distributing the load and thus increasing performance.
- Partitioning strategy involves identifying performance bottlenecks in consumers or Kafka and adjusting partitions accordingly.
- Consider balancing partition counts to manage client RAM usage and operational complexity.
- Start with a default of 12 partitions, scaling up as needed for high throughput, while considering operational and cost implications.
- The number of partitions can never be decreased.
- Increasing the number of partitions can lead to consuming messages in the wrong order.
- A consumer group distributes load across its members.
- Batching can increase the bandwidth but also the latency.
- Batching can be configured with `batch.size` and `linger.ms`.
- Producers can compress batches to reduce the required bandwidth, but this might increase latency.
- The usage of `acks=all` reduces the performance of producers by a bit; the same goes for `idempotence`.
- Brokers won't decompress batches; this is the task of the consumer.
- In most cases, brokers don't require any further fine-tuning.
- Brokers open file descriptors for every partition.
- Kafka heavily depends on the operating system, necessitating specific operating system optimizations to maximize its performance.
- Consumer performance depends mostly on the number of consumers in a consumer group but can be also fine-tuned by setting `fetch.max.wait.ms` and `fetch.min.bytes`.

Part 3

Kafka deep dive

In part 3, we take a deeper look at some of the more advanced topics surrounding Apache Kafka's architecture and functionality. This section digs into cluster management, the process of producing and persisting messages, the mechanics of consuming messages, and the critical aspects of cleaning up old or outdated messages. By exploring these topics, we can understand how Kafka achieves its scalability, fault tolerance, and efficiency in managing real-time streams of data. Whether you're looking to optimize your Kafka setup or solve more complex operational challenges, this section provides the tools and knowledge you need.

In chapter 7, we dive into Kafka cluster management, exploring how Kafka ensures stability, scalability, and failover within the cluster. Chapter 8 delves into how Kafka handles message production and persistence, from serialization to replication. In chapter 9, we examine Kafka's consumption model, offset management, and how consumer groups handle distributed workloads. Finally, chapter 10 covers Kafka's message cleanup mechanisms, detailing how log retention and compaction work to maintain performance and ensure data integrity.

7 Cluster management

This chapter covers

- Kafka cluster management with KRaft and ZooKeeper
- Migrating from ZooKeeper to KRaft
- How clients connect to Kafka

In the first chapters, we got to know Kafka as a distributed system and addressed multiple times that coordination in such systems is always associated with challenges. These problems affect not only Kafka itself but also our consumers when using consumer groups. Fortunately, Kafka takes a lot of work off our hands, and we don't have to worry about reliably distributing partitions to our consumers.

But how is Kafka actually managed? And what do we need cluster management for? The core challenge that cluster management systems solve is finding consensus in distributed systems. As long as we only have one broker, this is simple. We don't have to reach an agreement with anyone; this broker is the leader for all partitions, always in sync, the group coordinator for all consumer groups, and so on. But how do we ensure a consistent state when adding more brokers? Who then distributes the partitions to the brokers, monitors the health status of the brokers, and removes

brokers from the cluster if they don't report? Who stores the current configuration of the cluster?

In Kafka, there is a broker that acts as a controller for this metadata and issues instructions to all brokers. As far as how we select this controller, we could set a controller when starting Kafka, of course, but what happens if it fails, is too slow, or encounters other problems?

Kafka has undergone significant changes in recent years. The previous method of managing metadata and achieving consensus in Kafka underwent a massive transformation. The previous approach of using Apache ZooKeeper (https://zookeeper.apache.org/) has been replaced by a new approach. Because we believe that older Kafka versions based on ZooKeeper will remain in use for a longer time, we discuss both approaches here.

In the final section, we'll examine the process of connecting to a Kafka cluster. This includes handling metadata requests and establishing robust connections in a distributed environment, ensuring seamless integration and communication within the Kafka ecosystem.

7.1 Apache Kafka Raft cluster management

With Kafka 3.3.0, Kafka Raft (KRaft; https://kafka.apache.org/documentation/#kraft), the new coordination method of Kafka is production ready, and since Kafka 3.5.0, ZooKeeper has been deprecated for coordinating Kafka. Please note that the support for ZooKeeper-backed clusters will be dropped in Kafka 4.0.

As the name KRaft implies, it's based on the Raft protocol. This protocol was first described in the 2014 paper, "In Search of an Understandable Consensus Algorithm," by Diego Ongaro and John Ousterhout. Consensus in distributed systems is hard, and it took the Kafka community a long time to implement this new approach. Before KRaft, ZooKeeper was used, which we describe in more detail in section 7.2.

ZooKeeper kept people awake at night for several reasons. First, teams operating Kafka needed to operate two distributed systems, Kafka and ZooKeeper, which increased the burden on the operators. But the more relevant reason is that ZooKeeper is too slow for Kafka and limits Kafka's ability to scale the number of partitions, reducing its reliability. This isn't because ZooKeeper is bad software, but because ZooKeeper provides useful abstractions on top of its internal log-data structure. But because Kafka itself is already excellent at handling logs, these additional abstractions are unnecessary.

Therefore, the Kafka community needed a new architecture. Kafka brokers can now be configured in two different processor roles: *controller* and *broker*. We don't like these names, as they can mean different things in the world of Kafka. If a broker is configured for the controller role, it becomes a member of the coordination cluster. As with ZooKeeper, we need an odd number of controller nodes. For local development environments, one controller-broker is enough, but it provides no failover guarantees. If we want to survive the failure of one controller-broker, we need three controller-brokers, and if we have five controller-brokers, even two can fail.

This coordination cluster is now responsible for electing an active controller (known just as *controller* in the ZooKeeper-Kafka world) that monitors all normal brokers, manages partitions and leader assignments, and changes these assignments when something fails. The inactive controllers are called standby controllers. If the active controller fails, one of them takes over the work and can immediately continue managing the cluster. In a ZooKeeper-backed cluster, the failure of a controller could, depending on the cluster size, cause problems for many minutes.

For coordination, there is now a new topic called `__cluster_metadata`. This special topic stores all the metadata that was stored previously in ZooKeeper and is necessary for operating Kafka. This topic is replicated across all the brokers, no matter which processor role each one has, and this way, all the brokers know what they need to do.

The normal brokers are the brokers that do the routine work we've been discussing so far in this book. They accept producer and consumer requests, replicate the partitions, and are responsible for storing all the data in Kafka. Figure 7.1 describes the recommended minimal cluster for production.

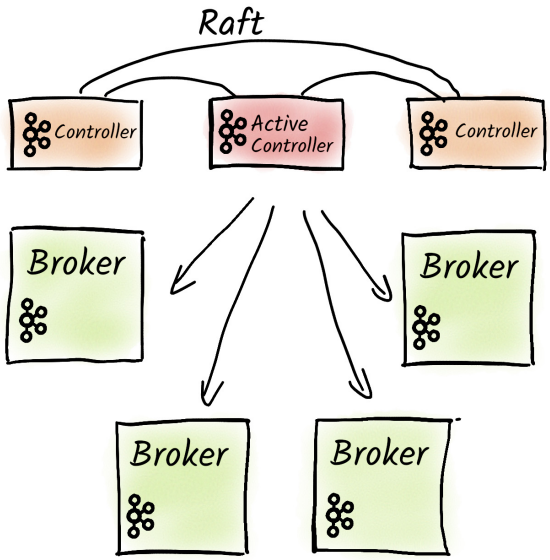


Figure 7.1 With Kafka 3.3, ZooKeeper was replaced by a coordination cluster based on the Raft consensus protocol.

TIP We recommend having three or five controller nodes for better reliability, and at least three normal brokers. (Having four controllers doesn't bring any benefit compared to three controllers as we could still only compensate the failure of one controller.) The separation of controllers and brokers makes it easier to scale the brokers up and down and makes operations simpler in general. For nonproduction environments, it's also possible to have brokers with both the controller and broker role (i.e., controller-brokers) to save resources,

but you also lose flexibility. For local development environments, it's even possible to have just one Java Virtual Machine (JVM) process running both a single broker and a controller in combined mode.

7.2 ZooKeeper Cluster Management

Until KRaft became production ready, Kafka relied on ZooKeeper for cluster management. Even though support for ZooKeeper-based cluster management will be removed in Kafka 4.0, we assume a significant number of Kafka clusters will still rely on ZooKeeper for the time being.

TIP We recommend migrating such clusters as soon as possible to KRaft.

ZooKeeper is a standalone Apache project used by numerous software products such as *Apache Hadoop MapReduce* and *Neo4j*. ZooKeeper itself initially feels like a hierarchical key-value store to use. Similarly to directories and files in operating systems, we can create, read, and modify *ZNodes*.

However, the key feature of ZooKeeper is that it guarantees that all ZooKeeper nodes always have exactly the same state. For example, if we try to write to a ZooKeeper ZNode with multiple clients simultaneously, there will be only one winner, and all our clients will see exactly the same result. We use this in Kafka to determine the controller, for instance. If there is no controller ZNode in the ZooKeeper key-value store, all brokers try to write their own ID to this ZNode, and whoever convinces ZooKeeper becomes the new controller.

Furthermore, ZooKeeper can also monitor which clients are currently still connected, allowing us to detect when a broker fails. To coordinate these changes reliably, ZooKeeper is based on a Paxos-variant consensus protocol, called ZooKeeper Atomic Broadcast (ZAB). To reach consensus, the majority of the ZooKeeper nodes must agree on a value, and the subordinate nodes with different or no opinion accept the prevailing opinion.

Kafka uses ZooKeeper to determine the controller and also to store metadata related to Kafka clusters in ZooKeeper—specifically, information on which partitions are assigned to which brokers and which broker is the leader for which partitions. Partition leaders write to ZooKeeper which followers are currently in-sync. However, because Kafka also stores access control lists (ACLs) and other security-related information in ZooKeeper, it's necessary to further secure ZooKeeper.

Figure 7.2 illustrates an example Kafka cluster with ZooKeeper that has a ZooKeeper ensemble of three ZooKeeper nodes and four Kafka brokers. The controller is merely one of these brokers and also takes over certain management tasks. This is different from the KRaft-backed Kafka. There, the active controller is always part of the controller cluster. If this controller fails, another broker takes over. Note that other brokers also communicate directly with ZooKeeper, and here the arrows only indicate the main communication paths.

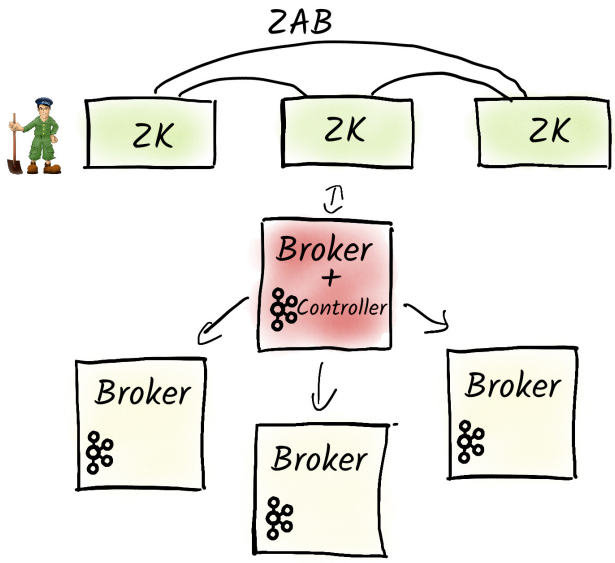


Figure 7.2 Before KRaft, a Kafka cluster consisted of multiple brokers and a ZooKeeper ensemble consisting usually of three or five ZooKeeper nodes. The controller was simply one of the brokers. If the controller failed, another broker took over these tasks.

Without ZooKeeper, it's doubtful whether Apache Kafka would have emerged in its current form, as ZooKeeper handles many complex tasks that would be difficult to implement independently. However, relying on ZooKeeper also introduces significant challenges. ZooKeeper's design prioritizes reliability and consensus over speed, which makes it relatively slow and can hinder Kafka's performance in certain areas. For administrators, ZooKeeper adds complexity, as maintaining and understanding this additional and often unfamiliar system requires considerable effort.

One of the main contradictions lies in Kafka's philosophy of using the log as an elegant data structure for exchanging messages between systems while simultaneously relying on ZooKeeper to store its own metadata. This reliance introduces both performance bottlenecks and consistency problems. To mitigate performance effects, brokers often retain metadata locally instead of querying ZooKeeper for every operation. However, this approach risks inconsistency, as the locally cached data may become outdated if brokers fail to synchronize properly with ZooKeeper.

When brokers act on outdated information, they can make decisions based on stale data, leading to potential problems. For example, the controller might assign a leader partition to a broker that is no longer available or make improper replica assignments based on outdated data from ZooKeeper. This can occur because certain broker operations, such as updating partition states or managing topic configurations, are performed directly in ZooKeeper, bypassing the controller entirely.

To maintain performance, the controller doesn't continuously monitor ZooKeeper for these changes; instead, it periodically polls for updates. This polling mechanism can create a time lag, during which the controller might miss critical updates or react too late to changes in the cluster state. These delays lead to situations where decisions are

made based on outdated information, compromising the consistency and reliability of the Kafka cluster.

7.3 Migrating from ZooKeeper to KRaft

By now, the advantages of KRaft over ZooKeeper are clear so the next step is to migrate from ZooKeeper to KRaft. Luckily, the migration works without any downtime, but it still requires careful planning and execution. At a high level, we need to first upgrade our existing Kafka cluster to the latest version that still supports ZooKeeper. For this, we can use the normal update procedure as described in the Apache Kafka documentation.

Then, migrate from ZooKeeper to KRaft as described in this section. Finally, it's time to upgrade to the latest Kafka version, again using the normal update procedure. There's one limitation when migrating from ZooKeeper to KRaft: only migration to a dedicated KRaft-controller cluster is supported, thus the number of machines we need to operate stays the same. However, this shouldn't stop us from migrating to KRaft.

A detailed description of the migration procedure can be found in the official documentation (https://mng.bz/yW6p). Here, we'll cover the procedure on a conceptual level only.

First, we need to provision the number of controller-brokers we'll need later. So usually, we start with three or five additional brokers configured as KRaft controllers. It's important that these controllers have the same cluster ID as the ZooKeeper cluster, that the migration mode is enabled (`zookeeper.metadata.migration.enable=true`), and that we pass the connection details to ZooKeeper (`zookeeper.connect=<Zookeeper address>`).

Now it's time to add the migration-relevant configuration to the existing brokers. For this, we need to enable the migration mode (`zookeeper.metadata.migration.enable=true`) and also provide the connection information to the new controller nodes. Once all brokers are configured for migration, the migration begins. The new controller nodes will replicate the data from ZooKeeper, and after some time, the brokers should log the following message:

```
Completed migration of metadata from ZooKeeper to KRaft
```

Note that the normal brokers are still in ZooKeeper mode, and we can revert to ZooKeeper mode at this point without any problems. Check the documentation for the instructions.

Now, the normal (noncontroller) brokers need to be migrated to KRaft mode. This is done by removing migration mode and the ZooKeeper connection from the configuration and by setting the `process.roles=broker` on each normal broker.

> **TIP** We should keep the cluster running in this state for a week or two to check whether everything works as expected. We can still revert to ZooKeeper mode at this point.

When everything works as expected, we can switch to KRaft mode. But be careful—this is the point of no return. Disable the metadata migration mode on the controller nodes, and remove the ZooKeeper connection from the configuration. Then, do a rolling restart of the controllers, and we're done with the migration. Kafka is now running in KRaft mode.

7.4 Connection to Kafka

Our exploration of Kafka's architecture now brings us to the crucial process of connecting to a Kafka cluster. While this may seem uncomplicated, it's intricately linked to the complex mechanisms of cluster coordination and metadata management we've discussed. By examining these connections, we'll gain deeper insight into Kafka's distributed operations. This analysis will bridge the gap between theory and practice, revealing how Kafka's design principles manifest in real-world scenarios.

In a traditional database system, this is straightforward. We provide the connection details of the database, and we're connected. However, Kafka is a distributed system, consisting of multiple Kafka brokers and a coordination cluster. In addition, usually not all leaders of all partitions are located on a single broker. The information regarding which leader for which partition on which topic is stored by Kafka in the coordination cluster. Each broker of the cluster caches this information. In the early versions of Kafka, clients queried this information directly from the ZooKeeper ensemble. Since Kafka 0.10.0, this information is queried directly from any Kafka broker in a *metadata request*. Let's take a look at the following command, which outputs a list of all topics:

```
$ kafka-topics.sh \
    --list \
    --bootstrap-server localhost:9092
```

We pass the URL of one of our Kafka brokers using the `--bootstrap-server` parameter. In the case of our test environment from appendix A, this is Broker 1. We could instead specify any other broker, and the command would still work. In a production environment, however, this approach of addressing only a single broker directly isn't recommended, as our clients will no longer start if the broker we specified goes down. Instead, the recommended approach is to specify all brokers in the `--bootstrap-server` parameter:

```
$ kafka-topics.sh \
    --list \
    --bootstrap-server \
      localhost:9092,localhost:9093,localhost:9094
```

As the number of brokers grows, manually specifying each broker becomes cumbersome. Because it doesn't matter which broker a client initially connects to, we can instead use a load balancer for this metadata request or handle the load balancing

through our domain name system (DNS). It's important that the brokers still have their own reachable IP address or DNS name.

Our client sends a metadata request to one of the bootstrap servers, asking for information on a set of topics. For example, if a consumer wants to consume the topic `products.prices.changelog`, the metadata request's response would include information about all the brokers in our cluster and which broker is the leader for each partition of that topic.

In the next step, our client can connect to the respective leaders of our partitions and fetch messages. The client doesn't need to constantly update the metadata. If a critical change occurs, such as a broker failure where the leader for one of the consumed partitions goes down, the client will detect this when it tries to fetch data from the partition leader and encounters an problem (e.g., the leader is unavailable). At this point, the client sends a new metadata request to one of the bootstrap servers. Because it doesn't matter which broker receives this request, we can simplify the connection process using DNS, a load balancer, or virtual IP addresses, rather than specifying as many broker addresses as possible in all our clients.

When a broker failure occurs, Kafka handles it by promoting one of the partition's replicas to be the new leader. The controller in the Kafka cluster detects the failure and triggers the leader election process. Once the new leader is elected, metadata is updated, and clients will be able to retrieve the correct metadata and reconnect to the new leader. During this brief period of leader election, clients may experience a short delay in fetching messages, but once the new leader is established, communication resumes as usual. After the metadata request, the clients then communicate directly with the respective brokers. The workflow is visualized in figure 7.3.

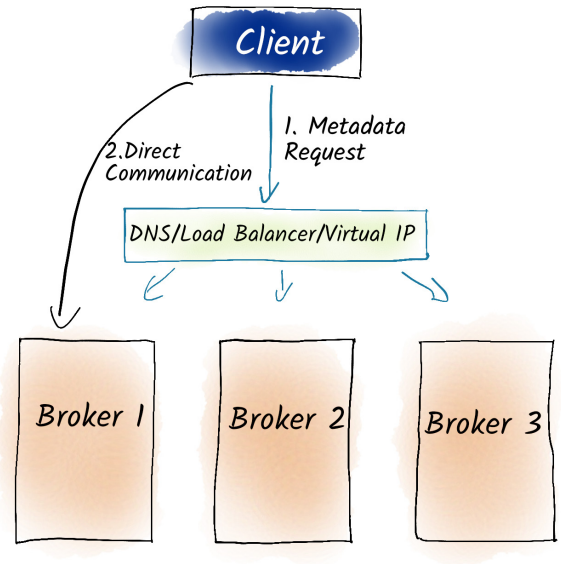


Figure 7.3 Before our clients can produce or consume messages, they need to know which broker is responsible for what. This is achieved through a metadata request.

Summary

- Kafka 3.3.0 introduced KRaft, a new coordination method based on the Raft protocol, replacing ZooKeeper from Kafka 3.5.0 onward.
- The Raft protocol resolves the complexities of achieving consensus in distributed systems, previously handled by ZooKeeper in Kafka.
- KRaft integrates coordination directly within Kafka brokers, eliminating the need for a separate ZooKeeper system, thereby improving scalability and reducing operational complexity.
- Controllers in KRaft manage partition assignments, leader elections, and failover mechanisms, ensuring cluster stability and facilitating efficient scaling and management of Kafka clusters.
- KRaft introduces the `__cluster_metadata` topic, ensuring consistent metadata storage across all brokers, which was previously managed by ZooKeeper, thereby streamlining cluster operations and enhancing reliability.
- Despite ZooKeeper being phased out in Kafka 4.0, a significant number of Kafka clusters currently rely on ZooKeeper for essential functions.
- It's advisable to migrate these clusters to KRaft promptly to use improved performance and simplified management offered by the Raft-based coordination.
- ZooKeeper serves critical roles in Kafka, such as electing the controller and storing metadata (e.g., partition assignments and leader information), ensuring consistent state across all nodes.
- While ZooKeeper's reliability in maintaining a consistent state is crucial, its inherent performance limitations and operational complexities have prompted Kafka's shift toward the more integrated KRaft solution.
- Migrating metadata from ZooKeeper to KRaft in Kafka involves careful planning and execution without downtime. We start by upgrading our Kafka cluster to the latest version that supports ZooKeeper.
- We provision three or five additional brokers as KRaft controllers with identical cluster IDs as the ZooKeeper cluster. We enable migration mode and configure ZooKeeper connection details.
- We configure existing brokers for metadata migration by enabling migration mode and specifying new controller-node connections. We verify successful metadata migration.
- We transition normal brokers to KRaft mode by adjusting configurations, testing stability, and finalizing the migration. We disable migration mode on controllers and remove ZooKeeper connections for a full transition to KRaft mode.
- Connecting to a Kafka cluster involves querying metadata to understand broker roles and partition leadership distribution across the cluster.

- Clients initiate connectivity through a bootstrap server, typically one of several Kafka brokers listed in the `--bootstrap-server` parameter, which provides initial metadata.
- To ensure resilience, production setups should specify multiple brokers in `--bootstrap-server` parameters or use DNS for load balancing.
- If a broker failure occurs, clients can send a new metadata request to a bootstrap server to discover new partition leaders, enabling uninterrupted message production or consumption by establishing connections with the newly elected leaders.

8 Producing and persisting messages

This chapter covers

- Serialization and partitioning in Kafka
- Acknowledgment handling and broker interactions
- Message reception and persistence
- Optimization within Kafka brokers
- Kafka's data and file structures
- Replication mechanisms and system performance

This chapter delves into the intricacies of producing and persisting messages in Apache Kafka, which are crucial components of its distributed data architecture. We'll explore how Kafka manages data serialization, partitioning, acknowledgment handling, and broker interactions, which are essential for ensuring reliability and scalability in real-time data processing. Understanding these aspects is key to optimizing message reception, persistence, and overall system performance within Kafka's ecosystem. By examining Kafka's data and file structures, replication

mechanisms, and their effect on system efficiency, we gain insights into how these foundational elements contribute to Kafka's robustness and operational excellence in modern data pipelines.

8.1 Producer

Typically, our producers use either the official Kafka Java library or, if our producer isn't running in the Java Virtual Machine (JVM), a library that is based on the C library `librdkafka` (https://github.com/confluentinc/librdkafka).

> **TIP** We generally advise against using other libraries because, although they may sometimes be easier to use, they often lack many features and optimizations.

Delving deeper into the various aspects of development for Kafka would fill an entire book, but because this book isn't a developer's guide, we'll only briefly go into the most important details. For the sake of clarity, we mostly use Python in our code examples. In Java and other programming languages, the basic concepts for communicating with Kafka are very similar.

8.1.1 Producing messages

With the `confluent-kafka` Python library, we can use the following code to send messages to Kafka:

```
from confluent_kafka import Producer
producer = Producer({
    'bootstrap.servers': 'localhost:9092',
    'acks': -1,
    'enable.idempotence': True,
    'partitioner': 'murmur2_random',
})

def delivery_report(err, msg):
    if err is not None:
        print(f"Delivery failed for record {msg.key()}: {err}")
        return
    print(f"Record {msg.key()} successfully produced to {msg.topic()}
        [{msg.partition()}] at offset {msg.offset()}")

producer.produce(
    "products.prices.changelog",
    key="cola",
    value="2",
    on_delivery=delivery_report)
```

Recommended configuration for reliable producers

Make sure the producer is compatible with Java.

Defines the callback function

Error handling

Topic to produce to

Uses the callback function

Before we can communicate with Kafka, we create a producer. We pass it `bootstrap.servers` as the most important parameter. If we use both Java libraries and libraries based on `librdkafka`, we should manually set the hash function of the partitioner in the `librdkafka` libraries to `murmur2_random`, which is used by the Java library. `librdkafka`

uses a different hash function by default. In the worst case, different partitioners mean that we can no longer give any guarantees about sequences, as the messages for the same keys are distributed to different partitions.

As soon as we've initialized our producer, we can produce messages. In Python, we use the `produce()` method for this. In Java, we first manually create the message we want to send and then pass it to the `send()` method.

We pass the message that we intend to produce and the topic to which it should be written to the producer. Optionally, we can pass a callback (here, `delivery_report`), which is called as soon as we've received an acknowledgment (ACK) for the message. Our message consists of the value and the key (optional).

To produce data, we have to give the `produce()` method at least the topic where the message is to be written and a value. But what happens between calling the `produce()` method and calling the callback? Quite a lot happens here, but fortunately the Kafka library does most of the work for us. A high-level overview of the whole producer workflow is shown in figure 8.1.

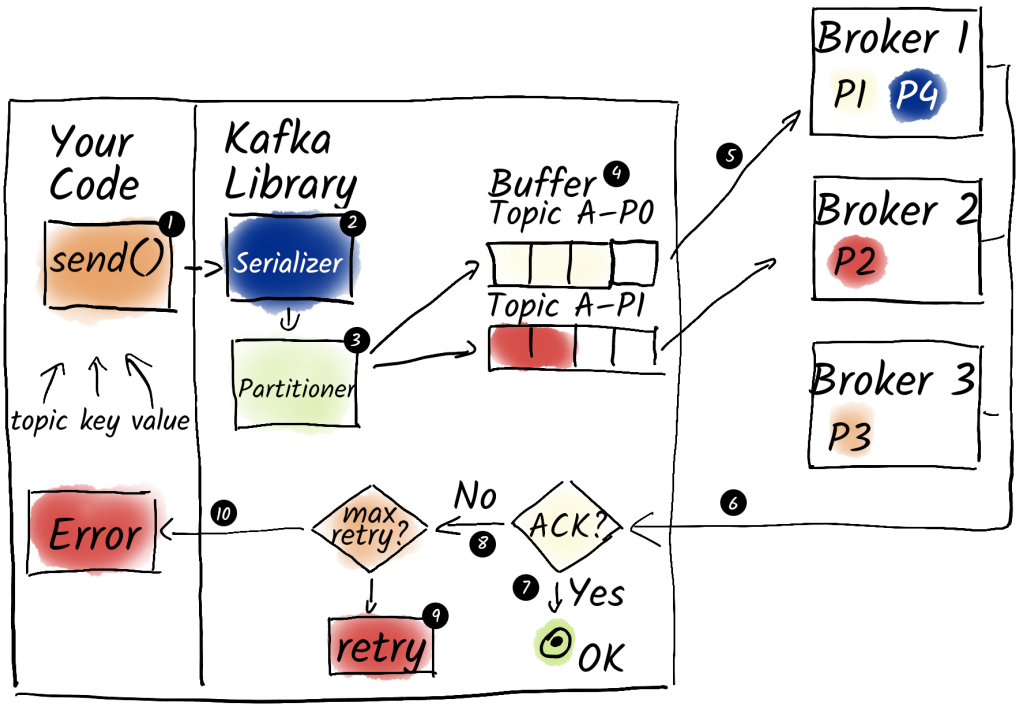


Figure 8.1 What happens when a message is produced? After calling the `produce()` (1) method, the data must first be serialized using a serializer (2). The partitioner (3) then decides which partition to write to and stores the message in a buffer (4) for this partition. Once there are enough messages, the producer sends (5) entire batches to the corresponding brokers and, depending on the ACK setting, waits for an ACK (6). If we receive an ACK in time (7), the success callback is called. If we don't receive an ACK (8), we try (9) again for a while. If an error occurs, the Kafka library throws an exception (10).

8.1.2 Production process for messages

We've already discussed in detail that Kafka doesn't interpret the content of messages, but can only deal with byte arrays. This means that our producer is responsible for serializing the data. In our Python implementation, we don't see the magic that happens because the Python library expects a string or byte array as the input value. Strings are then converted into byte arrays using the string serializer.

Kafka libraries usually support several different serializer classes by default, such as *JSON*, *Protocol buffers* ([Protobuf], https://developers.google.com/protocol-buffers/) or *Apache Avro* (http://avro.apache.org/). If we use a different data format, we can also implement our own serializers (and deserializers) with a manageable amount of effort. We'll discuss these formats in more detail in chapter 13, where we explore their role in schema management.

As soon as our keys and values are available as byte arrays, the partitioner can decide into which partition the message should be produced. We can, of course, also give the `produce()` method a fixed partition, but unless there's a solid reason to do so, we should rely on the partitioning methods provided. We've already discussed several times that the partitioner determines a partition based on the hash value of the key or, if no key is available, uses a round-robin mechanism.

For each partition and for each topic we want to produce to, there is a buffer into which the partitioner writes the messages. By default, the entire buffer is 32 MiB in size (`buffer.memory` setting in the producer). Batches with a maximum `batch.size` (16 KiB by default) are then created from this buffer. By default, producers don't wait for batches to fill up before sending data to the broker, but instead Kafka sends the messages it's supposed to produce to the brokers as quickly as possible. We can use batching if we produce data faster than we can send individual messages to the brokers.

As soon as the messages have arrived in the buffer, we can send them to the brokers. Remember that we only send messages to the leader brokers of the partitions. In many cases, we have more partitions than brokers. Instead of sending individual messages to the broker, the producer groups messages into larger partition-based batches for each broker and sends them together.

8.1.3 Producer and ACKs

What happens next depends on the ACK settings. If we've set `acks=0` in the producer, the producer sends the message and isn't interested in what happens. The message will usually arrive, but we can't guarantee this. In most cases, especially if the data is important to us, we set `acks=all` to ensure that the data not only arrives at the leader (as with `acks=1`) but also has been successfully replicated. (We looked at ACKs in great detail in chapter 5.)

If we instruct the producer to wait for an ACK, then it waits for the ACK until the sent request leads to a timeout. If no ACK is received during this time or an error is received from the broker, the producer assumes that the delivery wasn't successful. Our producer then tries to send the message again, as shown in figure 8.2.

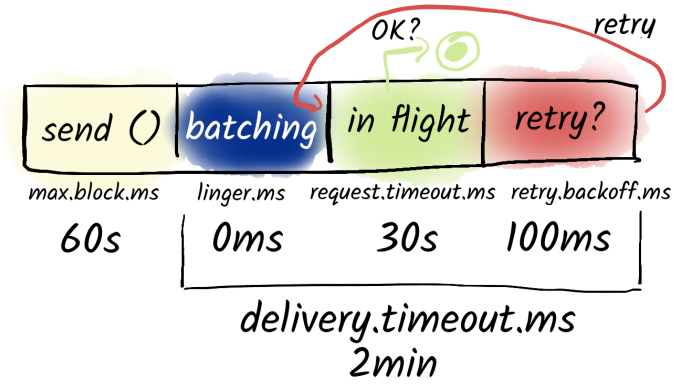


Figure 8.2 We can define numerous timeouts in the producer. The most important are `linger.ms`, which we can use to configure batching, and `delivery.timeout.ms`, which we can use to configure when the Kafka library stops redelivering messages and throws an error.

Let's take a closer look at the timeouts. The `produce()` (or `send()`) method is responsible for writing messages to the buffer. If the buffer becomes full, the producer thread will wait for space to become available. The amount of time it waits before giving up is governed by the `max.block.ms` setting, which defaults to 60 seconds. This means that if the buffer is full, the producer thread will block and try to send messages again until either the buffer has space or the timeout of 60 seconds is reached.

Additionally, Kafka batches messages before sending them, and the batching mechanism waits up to `linger.ms` (default is 0 ms) for more messages before sending the batch. Once the batch is ready, it's sent to the broker, and the producer waits for a response. The response timeout is controlled by `request.timeout.ms` (default 30 seconds). If a response isn't received within that time frame or if an error occurs, the producer will retry the send operation after waiting for `retry.backoff.ms` (default 100 ms).

If the producer continues to fail to send the messages within the time allowed by `delivery.timeout.ms` (default 2 minutes), it will give up and throw an exception, which the application must handle appropriately.

But what do we do if we get such an exception? We shouldn't blindly resend the messages, as our Kafka cluster is probably in a bad state. We should leave the retry mechanism to the Kafka library, as it also takes care of the correct order of the messages, for example, if we've activated `enable.idempotence`. In the event of an exception, we should use a circuit breaker (www.martinfowler.com/bliki/CircuitBreaker.html) or another design pattern to intercept this error case, for example.

8.2 Broker

In the previous section, we took a closer look at how our Kafka producer library sends messages and deals with errors. Kafka outsources a lot of work to the clients so that our brokers have to do as little as possible themselves. In the event that we produce messages, our leader must receive the messages correctly, possibly check whether we're authorized to write to the partitions at all, and then write the messages to disk as

quickly as possible. Then, these messages have to be distributed to the followers, but fortunately, the followers themselves are responsible for this. If we've configured our producer in such a way that the reliability of the messages is important to us, our leader sends an ACK back to the producer at the end.

8.2.1 Receiving and persisting messages

Although this may not sound like much effort, the Kafka brokers nevertheless must be very complex systems to fulfill these tasks as efficiently and reliably as possible. A high-level overview of the processes that happen in the broker is shown in figure 8.3.

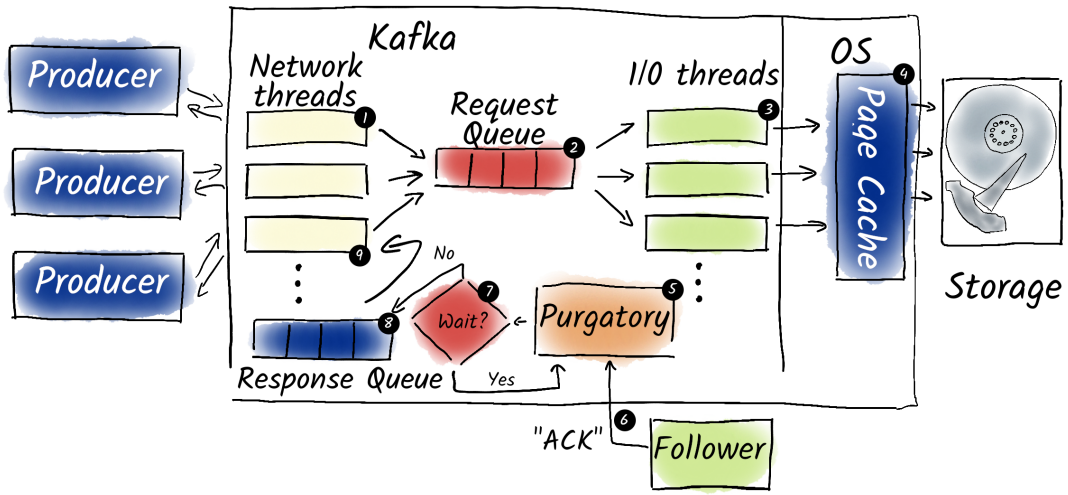


Figure 8.3 When a broker receives a produce request (1), it's put on the request queue (2) and later handled by the I/O threads (3) and written to the page cache (4). The operating system flushes it to disk later. Then, the message stays in purgatory (5) until all in-sync followers acknowledge the message (6). When the wait (7) is over, the response is put on the response queue (8) and sent back to the producer by one of the network threads (9).

First of all, the network threads take care of receiving the messages from the producers to write them to the request queue after successful authorization. If our I/O threads aren't overloaded, the messages only remain in the request queue for a very short time and are then written by the I/O threads to the correct location in the filesystem, that is, to the end of the current log segment for the corresponding partition. Kafka doesn't ensure that these messages are successfully persisted to disk; this is the task of the operating system. Log segments are files where Kafka stores messages for a partition. We'll look into the details of Kafka's data structure in the next section.

> **TIP** Avoid blocking synchronization as much as possible. Instead, the operating system should independently write the data from the page cache to disk in the background.

Incidentally, the statement that Kafka doesn't care at all about persisting messages on the hard drive isn't 100% correct. In Kafka, we do have the option of influencing the operating system in this respect. There are two setting options for this in the configuration. First, we can use `flush.ms` to specify after how many milliseconds Kafka should trigger a manual `fsync`, and second, we can use `flush.messages` to specify after how many messages an `fsync` is triggered. By default, both settings are set to the highest possible number (long datatype). (So if we were to run our cluster long enough, Kafka would trigger an `fsync`. In the case of `flush.ms`, for example, this would only be a paltry 300 million years!) Even if we could theoretically improve the reliability of our cluster with these two configuration options, we strongly advise against doing so at this point due to considerable performance losses as a result. We also achieve reliability in Kafka through replication.

The performance optimization that we don't commit data to disk independently comes with the price that we have to think about where and how we deploy the brokers. If we deploy all replicas of a partition on the same VM host, for example, and this host then crashes, we'll definitely lose data. We should therefore distribute our brokers as evenly as possible across the systems that are available to us. Additionally, we should use the rack-awareness feature of Kafka, which we'll explain in chapter 9 in more detail.

8.2.2 Brokers and ACKs

Once the messages are stored on the filesystem, the broker needs to determine whether to respond to the producer and whether to wait for other brokers to confirm they have received the message. This is where a mechanism called *purgatory* comes into play. Purgatory acts as a temporary holding area, where responses are stored while the broker waits for additional information, such as confirmation from follower brokers. When producing messages, the broker waits for the followers to acknowledge the message. Once all confirmations are received, the broker writes the response to the response queue, and the network threads send an ACK back to the producer.

8.3 Data and file structures

Having looked in detail at how messages are produced and then processed by the broker, in this section, we'll focus entirely on Kafka's data and file structures. We'll take a closer look at the different types of data Kafka stores and how this data is structured in our brokers.

8.3.1 Metadata, checkpoints, and topics

When setting up our test environment, we created a directory for each of our three brokers under `~/kafka/data/kafka<Broker ID>`. This is exactly where our brokers store all of their data. For a better overview, we've completely reorganized our Kafka cluster for this chapter by simply deleting the corresponding directories and creating new ones. Let's first take a look at the directory corresponding to our broker with ID 1 using `ls`:

```
$ ls -1 ~/kafka/data/kafka1/
__cluster_metadata-0
bootstrap.checkpoint
cleaner-offset-checkpoint
log-start-offset-checkpoint
meta.properties
recovery-point-offset-checkpoint
replication-offset-checkpoint
```

In our fresh Kafka cluster, there are a total of six files and one folder. The `bootstrap.checkpoint` file as well as the `__cluster_metadata-0` folder, which contains the data for Partition 0 of the `__cluster_metadata` topic, are used by Kafka Raft (KRaft); we'll omit those in the following outputs as we won't go more into detail. The `meta.properties` file contains the ID of our broker, the ID of our Kafka cluster, an ID for the directory, and the metadata version so that Kafka knows how to interpret the corresponding files. All other files contain only two zeros. The first zero stands for the metadata version, and the second number indicates how many lines of actual information follow. As our cluster is freshly set up and we don't have any actual topics (besides `__cluster_metadata`), this value is 0 everywhere.

Before we get to exactly what information is stored in these checkpoint files, we first create a topic called `products.prices.changelog.file-structure`. We select our standard replication factor of 3 for this and distribute the topic across three partitions:

```
$ kafka-topics.sh \
    --create \
    --topic products.prices.changelog.file-structure \
    --partitions 3 \
    --replication-factor 3 \
    --bootstrap-server localhost:9092
Created topic products.prices.changelog.file-structure.
```

Now that we've created the topic, let's take a closer look at the contents of the `replication-offset-checkpoint` file:

```
$ cat ~/kafka/data/kafka1/replication-offset-checkpoint
0
3
products.prices.changelog.file-structure 0 0
products.prices.changelog.file-structure 2 0
products.prices.changelog.file-structure 1 0
```

The metadata version is still `0` (line 1 of the file), but our file now contains three lines of information (line 2). The other lines of our file each begin with the name of the topic we've just created. The second column refers to the partition, and the third column contains an offset. The offsets refer to the position in the log of the respective partition up to which our messages have already been successfully replicated to our follower replicas. We'll take a closer look at exactly what this means in the next section

of this chapter. As we haven't yet produced any messages, the position in the log up to which replication has taken place is `0`.

The file `recovery-offset-checkpoint` has a similar structure, only here the offsets refer to the position in the log up to which our messages have been successfully persisted, that is, written from the main memory to the hard drive by the operating system.

The `log-start-offset-checkpoint` file contains information on the offset of the first message in our log or on our partitions, as the first message in our log doesn't necessarily have to have the offset 0. This is due to the fact that Kafka can continuously clean up our log and thus automatically delete old and no longer required messages, for example; otherwise, our log would grow ad infinitum.

The last remaining file—`cleaner-offset-checkpoint`—contains the offsets of our partitions up to which the Kafka log cleaner has compacted messages. How exactly Kafka cleans up our log and what options we have for doing this are covered in detail in chapter 10.

8.3.2 Partitions directory

With the creation of our `products.prices.changelog.file-structure` topic, not only was metadata for this topic added to our existing files but a subdirectory was also created for each partition in the data directory of our broker (`products.prices.changelog.file-structure-0`, `productsprices.changelog.file-structure-1`, `products.prices.changelog.file-structure-2`):

```
$ ls -1 ~/kafka/data/kafka1/
cleaner-offset-checkpoint
products.prices.changelog.file-structure-0/
products.prices.changelog.file-structure-1/
products.prices.changelog.file-structure-2/
log-start-offset-checkpoint
meta.properties
recovery-point-offset-checkpoint
replication-offset-checkpoint
```

Strictly speaking, a file isn't created for each partition in each broker, but only for each partition that has been assigned to our broker. As we have a replication factor of 3 and our cluster consists of a total of three brokers, a subdirectory was created in each of our brokers or in each of our broker file paths for all three partitions of our `products.prices.changelog.file-structure` topic. Let's now take a look at the files of our Partition 0:

```
$ ls -1 ~/kafka/data/kafka1/products.prices.changelog.file-structure-0
00000000000000000000.index
00000000000000000000.log
00000000000000000000.timeindex
leader-epoch-checkpoint
partition.metadata
```

There are a total of five files in our partition directory. The `partition.metadata` file contains the metadata version and the ID of our topic (`topic_id`):

```
$ cat ~/kafka/data/kafka1/\
products.prices.changelog.file-structure-0/partition.metadata
version: 0
topic_id: q4RDr2lTR8y1Wy6eYUpl6g
```

The `leader-epoch-checkpoint` file has a similar structure to our other checkpoint files. It contains the metadata version in the first line, followed by the number of lines of information in line 2. Depending on whether the broker in whose file directory we're currently located is the leader of the partition, the file looks slightly different. Let's therefore first take a brief look at which broker is the leader of which partition:

```
$ kafka-topics.sh --describe \
    --topic products.prices.changelog.file-structure \
    --bootstrap-server localhost:9092
Topic: products.prices.changelog.file-structure
    TopicId: q4RDr2lTR8y1Wy6eYUpl6g PartitionCount: 3
    ReplicationFactor: 3 Configs:
  Topic: products.prices.changelog.file-structure Partition: 0
    Leader: 1 Replicas: 1,3,2 Isr: 1,3,2
  Topic: products.prices.changelog.file-structure Partition: 1
    Leader: 2 Replicas: 2,1,3 Isr: 2,1,3
  Topic: products.prices.changelog.file-structure Partition: 2
    Leader: 3 Replicas: 3,2,1 Isr: 3,2,1
```

Broker 1 is the leader of Partition 0, so we look at the file `leader-epoch-checkpoint` in Partition 0 of Broker 1:

```
$ cat ~/kafka/data/kafka1/\
products.prices.changelog.file-structure-0/leader-epoch-checkpoint
0
1
0 0
```

NOTE The broker that is the leader of Partition 0 could be a different broker.

The metadata version is also 0 (line 1) and our file contains one line of information (line 2). The information itself in this file is just two zeros. The first zero represents the number of previous partition leaders; the second zero represents the current log offset at the time of the leader selection. The other partitions in Broker 1 only contain two zeros at this point. The leader epoch is incremented with each leader change and is used to ensure that two brokers don't accidentally claim the leader role for a partition.

Let's take a look at the following example to see exactly under what circumstances this could happen and how the leader epoch prevents this. Using the `kafka-broker-stop.sh` script from appendix A, we first stop our broker with ID 3, which is the leader of Partition 2. We then take another look at our `leader-epoch-checkpoint`:

```
$ kafka-broker-stop.sh 3
$ cat ~/kafka/data/kafka1/\
products.prices.changelog.file-structure-0/leader-epoch-checkpoint
0
1
1 0
```

Kafka realizes that one of the brokers is no longer accessible and starts a leader election to determine a new leader for Partition 2. The leader epoch is incremented by 1 in each partition. We could use `kafka-topics.sh --describe` to see which broker has taken over the leader role for Partition 2, but this isn't important for our example. Let's now restart our broker with ID 3:

```
$ ~/kafka/bin/kafka-server-start.sh -daemon ~/kafka/config/kafka3.properties
```

This is where our leader epoch becomes important. The leader epoch ensures that only the current leader of a partition can distribute messages. Let's imagine, for example, that Broker 3 didn't even realize that it had been unavailable. It would then still think that it's the leader of Partition 2. When replicating messages, a broker always sends the current leader epoch. If a broker sends an outdated leader epoch, the other brokers ignore its messages. As the leader epoch is still 0 from the perspective of Broker 3, its messages are rejected because the current, actual leader epoch has the value 1.

Of course, Broker 3 isn't condemned to send messages that are rejected forever. After a short time, it receives the current partition leader and leader epochs, and Broker 3 itself becomes the leader of Partition 2 again after a while, as Kafka attempts to reinstate the preferred leaders as actual leaders every 5 minutes by default (`leader.imbalance.check.interval.seconds`). We can also speed that up by running `kafka-leader-election.sh` from chapter 5:

```
$ kafka-leader-election.sh \
    --election-type=preferred \
    --all-topic-partitions \
    --bootstrap-server localhost:9092
```

To conclude this example, let's take another look at our `leader-epoch-checkpoint` in Partition 2 (`products.prices.changelog.file-structure-2`) of Broker 3 (`kafka3`):

```
$ cat ~/kafka/data/kafka3/\
products.prices.changelog.file-structure-2/leader-epoch-checkpoint
0
1
2 0
```

For Partition 2, the value has incremented again, as Broker 3 has taken over the leader role from Broker 2 again. Incidentally, the values for Partition 1 and Partition 2 haven't changed, as there is neither a new leader nor has a new leader selection been carried out.

Let's first produce a few messages in our `products.prices.changelog.file-structure` topic before we look at the remaining files in the partition directory:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog.file-structure \
    --bootstrap-server localhost:9092
> cola 2
> coffee pads 8
[…]
> cola 1
```

But first, let's take another quick look at our replication offsets:

```
$ cat ~/kafka/data/kafka1/replication-offset-checkpoint
0
3
products.prices.changelog.file-structure 0 8
products.prices.changelog.file-structure 2 2
products.prices.changelog.file-structure 1 2
```

As shown, the messages have already been successfully replicated. The messages haven't been distributed evenly across the partitions because they were sent by the producer in batches, and depending on how fast we were typing our messages, several messages ended up in one batch and therefore one partition. This resulted in a slight imbalance in terms of load distribution.

> **NOTE** If we had used keys, the distribution of our messages would have been deterministic. Without keys, we may even need to restart the producer to produce into multiple partitions.

8.3.3 Log data and indices

We now know that our data has been replicated, but where and how is it stored? This is where our remaining files in the partition directory come into play. The file with the extension `.log` contains our messages. With the `kafka-dump-log.sh` script, Kafka also provides us with a tool with which we can take a closer look at our log. All we have to do is pass the log segment to be inspected using the `--files` argument. Let's now take a look at the file `00000000000000000000.log` in Partition 0 of the `products.prices.changelog.file-structure` topic. We've selected this partition because most of the messages have ended up there:

```
$ kafka-dump-log.sh \
    --files ~/kafka/data/kafka1/\
products.prices.changelog.file-structure-0/0[…]000.log
Dumping ~/kafka/data/kafka1/
products.prices.changelog.file-structure-0/0[…]000.log
Log starting offset: 0
baseOffset: 0 lastOffset: 1 count: 1 […] position: 0
CreateTime: 1738455138490 size:  68 […] crc: 1092464612
[…]
```

The output of our command shows us some metadata about our log file. For a better overview, we've shortened the output of this command somewhat and limited ourselves to explaining the most interesting data for us at this point. We can see the offset at which the log segment starts (`Log starting offset`), and then we get an overview of the individual batches. We can see the offset of the first message and the last message of the respective batch (`baseOffset` and `lastOffset`) and how many messages the respective batch contains in total (`count`). We can also see the timestamp (`CreateTime`) and the checksum of the batch (`crc`). The size of the batch in bytes is hidden behind the `size` attribute, and `position` indicates the byte position in the segment at which the respective batch begins. A batch also contains some metadata, totaling 60 bytes, which isn't surprising given the wealth of information that we receive from `kafka-dump-log.sh`. Here, we can also see how batches significantly improve the performance of Kafka, especially with many small messages!

The payload of the messages themselves is only a few bytes in size (depending on our message) because the messages in the individual batches contain further metadata. If we also pass the `--print-data-log` argument to our `kafka-dump-log.sh`, we get all the information about our individual messages, including the actual message! However, we'll omit the output at this point.

The file with the extension `.index` is used to find our messages in the log more quickly, as Kafka simply saves all messages or batches one after the other in the log. Without the index, we would have to decode the entire segment every time a consumer wants to consume messages and then search for the corresponding offset. As this wouldn't be very efficient, Kafka saves the byte position associated with the respective offsets in the log instead.

Note that this isn't done for every offset, but only if at least 4,096 additional bytes have been added to our log since the last entry in the index. This value can also be adjusted in the topic configuration using the `index.interval.bytes` setting. Alternatively, we can also adjust the default value for our Kafka cluster via the configuration of our brokers using the setting `log.index.interval.bytes`. We can also display the content of the index file with `kafka-dump-log.sh`.

> **TIP** If we reduce the index interval, we can jump closer to the desired position in the log, but the index will grow faster. The default value of 4 KiB offers us a very good trade off at this point in the vast majority of cases.

The file with the extension `.timeindex` has a similar task and function to our index file, but instead of matching between offset and byte position, a matching between timestamp and offset takes place here. This allows us to simply consume messages from a specific day. A common use case for this is resetting the consumer offsets to the start of the day and thus consuming all messages for that day again.

8.3.4 Segments

We've already talked about segments several times in the course of this chapter. We already know that messages are assigned in Kafka topics and that topics are in turn

divided into partitions. Partitions are in turn divided into segments, each of which consists of the log and index files we've just learned about. The overall data structure is visualized in figure 8.4.

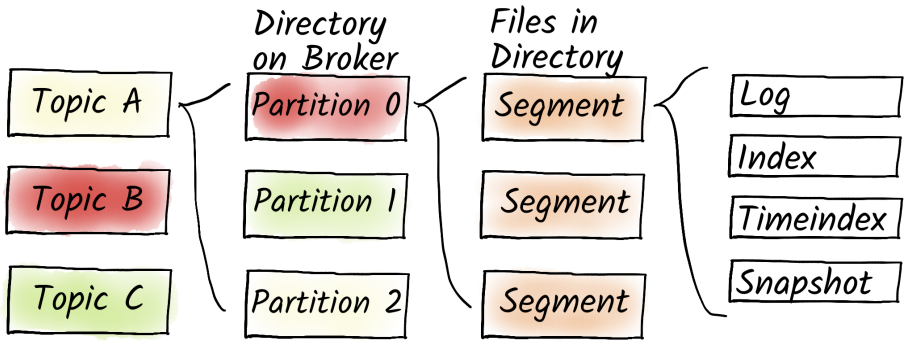


Figure 8.4 Topics are divided into partitions. Partitions create segments as messages are produced. Segments consist of the actual log files, index files, and a snapshot file.

The reason for this subdivision is that partitions naturally grow over time and can sometimes contain dozens of gigabytes or even terabytes of data. A single log or index file can quickly become very inefficient. However, this relates less to the consumption of messages and more to how messages are cleaned up in Kafka. We'll take a closer look at this in chapter 10.

At this point, it also makes sense to take a closer look at the filenames of the log and index files, as the filename is nothing other than the first offset in the respective segment. But when exactly does Kafka create a new segment? There are two important parameters for this. Kafka creates a new segment when either the segment has exceeded a certain size (1 GiB by default) or has reached a certain age (seven days by default).

We can adjust these settings again via the topic configuration. With `segment.ms`, we specify the time in milliseconds after which a new segment is created. We can influence the maximum segment size in bytes using `segment.bytes`. Of course, we also have the option of adjusting the corresponding default settings for topics in our broker configuration (`log.segment.bytes` or `log.roll.ms`).

Let's test this directly by reducing the maximum age of a segment to 60 seconds. To do this, we can use the `kafka-configs.sh` script to adjust the configuration of our `products.prices.changelog.file-structure` topic accordingly:

```
$ kafka-configs.sh \
     --alter \
    --topic products.prices.changelog.file-structure \
    --add-config segment.ms=60000 \
    --bootstrap-server localhost:9092
```

After we've customized our topic configuration, we still need to produce a few messages so that Kafka creates a new segment. For this, we use our `kafka-console-producer.sh` again:

```
$ kafka-console-producer.sh \
    --topic products.prices.changelog.file-structure \
    --bootstrap-server localhost:9092
> cola 2
> energy drink 5
[…]
> cola 3
```

Kafka should now have created a new segment. Let's just check the whole thing by taking a look at the files in our partition:

```
$ ls -1 ~/kafka/data/kafka1/products.prices.changelog.file-structure-0
00000000000000000000.index
00000000000000000000.log
00000000000000000000.timeindex
00000000000000000003.index
00000000000000000003.log
00000000000000000003.snapshot
00000000000000000003.timeindex
leader-epoch-checkpoint
```

As expected, there are now more log and index files in our partition. However, another file with the extension `.snapshot` has been added. In our example, this file only refers to the start offset of the current segment. Apart from that, this file is used to manage idempotent producers in Kafka. Idempotence was discussed in detail in chapter 5.

8.3.5 Deleted topics

Let's delete our topic again, not to clean it up, but to see how Kafka deletes topics. To do this, we use the `kafka-topics.sh` script again:

```
$ kafka-topics.sh \
    --delete \
    --topic products.prices.changelog.file-structure \
    --bootstrap-server localhost:9092
```

However, Kafka doesn't delete files directly, but first marks them for deletion:

```
$ ls -1 ~/kafka/data/kafka1/
cleaner-offset-checkpoint
products.prices.changelog.file-structure-0.<UniqueID>-delete/
products.prices.changelog.file-structure-1.<UniqueID >-delete/
products.prices.changelog.file-structure-2.<UniqueID>-delete/
log-start-offset-checkpoint
meta.properties
recovery-point-offset-checkpoint
replication-offset-checkpoint
```

Kafka simply changes the name of the partition directories by adding a special extension in the form `.<UniqueID>-delete` to the original directories. After 1 minute, the directories are then permanently deleted. During this period, we could theoretically undo the deletion of our topic and thus save our data if we've accidentally deleted the wrong topic. Once again, we have the option of adjusting this time period. To do this, we can either adjust the topic configuration using the `file.delete.delay.ms` parameter or change the setting globally for our entire cluster using the `log.segment.delete.delay.ms` broker configuration.

8.4 Replication

In this section, we explore Kafka's replication mechanisms, with a particular focus on *in-sync replicas* (ISRs). Replication in Kafka ensures fault tolerance and scalability by maintaining redundant copies of data across multiple brokers. We examine the concept of ISR, which plays a critical role in ensuring data consistency and availability.

Additionally, we discuss the significance of the High Watermark (HWM) in determining message commit points and analyze the implications of replication delays on data consumption and production. This section provides a comprehensive understanding of how Kafka manages replication to maintain reliability and performance in distributed data environments.

8.4.1 In-sync replicas

We replicate data in Kafka by having followers fetch data from the leader at regular intervals. The leader thus knows which follower has fetched data at what time. We already know that with `acks=all` we wait until all ISRs have fetched this message. But we don't wait until all followers have collected the message because we want to avoid slowing down the entire cluster if a broker is slow or unavailable. To do this, we define the ISR.

A follower is in sync if it has retrieved all messages from the leader within the past 30 seconds. We can adjust this time span using `replica.lag.time.max.ms` in the broker configuration. Leaders are always in sync. By default, a replica asks the leader for new messages every 500 ms. We can also adjust this time by changing the value for `replica.fetch.wait.max.ms` in the follower broker.

If a replica hasn't managed to consume all messages up to the *Log End Offset* (LEO) of the leader at least once in the maximum lag time, it's removed from the list of ISRs in the partition by the leader. The LEO marks the position of the last message received on each partition and each replica and thus points to the end of the corresponding log. This serves to identify replicas that are temporarily unable to replicate fast enough and thus prevent a slow broker from negatively affecting the performance of the entire cluster.

A replica can also be immediately removed from the ISR list if it stops responding to heartbeats. *Heartbeats* are periodic messages sent by brokers to confirm that they are alive and functioning correctly. If a replica fails to send a heartbeat within the expected

interval, such as in the case of a broker failure, it's considered out of sync and removed from the ISR list. This is part of Kafka's fault tolerance, ensuring that the system remains operational even if individual brokers fail or become unresponsive.

8.4.2 High Watermark

But how does the leader know that the messages have been successfully replicated to the follower, and what role does the LEO play in this? As soon as a partition leader or follower receives a new message, it adjusts the corresponding LEO. When a follower sends a fetch request to the leader, this request also contains the current LEO of the follower, equivalent to how normal consumers communicate their current offset. This tells the leader whether the follower has already received all messages or which messages are still missing and need to be sent to them.

At this point, another offset comes into play: the *High Watermark* (HWM). The HWM is the position in the log up to which all ISRs have already consumed messages, as shown in figure 8.5. It's therefore the smallest LEO committed to the leader among all ISRs. In addition, in Kafka, a message is only considered successfully committed when all ISRs have received the message, that is, when the HWM is greater than or equal to the position of the message in the log, regardless of the ACK strategy. The current HWM is also propagated by the leader to its followers during the fetch.

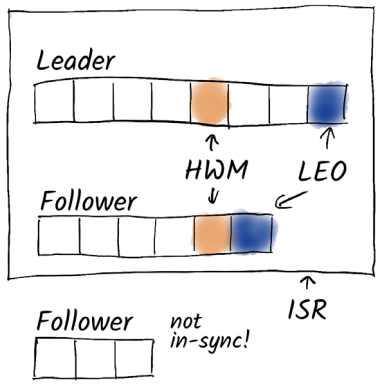


Figure 8.5 The High Watermark is the offset that all ISRs have replicated and committed to the leader. The lowest follower isn't in sync and isn't taken into account when calculating the HWM.

However, the HWM serves another purpose in Kafka. Consumers can only read messages that are committed from Kafka's point of view. This limit in the log is marked precisely by the HWM. As replication in Kafka is completely asynchronous, it can take several seconds before a newly produced message can be consumed, depending on the settings. In the normal state, communication with Kafka takes place in near real-time, and we're talking about latencies in the millisecond range.

> **WARNING** If there are less than `min.insync.replicas` in sync, then the HWM doesn't advance and therefore consumption of new messages is blocked until the minimum number of replicas are in sync again.

Let's take a brief look at the steps involved in replicating a message. When a partition leader receives a new message, the LEO of the leader first increases. With the next fetch request from the followers, the leader realizes that the followers are missing a message and sends the message to the respective followers. With the next fetch request, the leader determines that all followers have received the messages, as the LEO of the

followers corresponds to the LEO of the leader, and the leader increments the HWM and propagates the new value to the followers with the response to the fetch request.

Let's now look at the whole thing from the perspective of a follower, as shown in figure 8.6. If a follower wants to read messages, it sends a fetch request with its current position in the log (its own LEO) to the leader of the corresponding partition. The leader then sends the messages to the follower and updates the *Fetch Offset* (*Last Committed Offset* [LCO]) stored for this follower to the offset the follower sent with the fetch request. At this point, it doesn't matter to the leader whether the messages will arrive successfully or not. If a follower hasn't received a message, it simply sends a new request to the broker with the same offset, and then the broker resends the messages. This reduces complexity for brokers, as they don't have to worry about error management at this point.

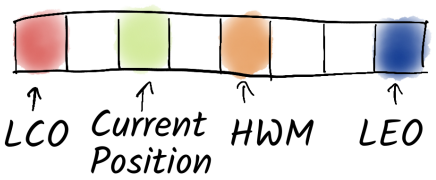


Figure 8.6 This shows the follower's LCO before and after the fetch request (Current Position). The HWM indicates the latest offset committed by all ISRs. Because the follower is behind, its LCO is lower than the HWM. The LEO is a bit ahead of the HWM, as new messages are arriving.

8.4.3 Effects of delays during replication

To conclude this section, let's look at the effects on the consumption and production of data if an ISR starts to lag behind—that is, it can no longer replicate messages from the leader fast enough.

As long as a replica is in sync, it influences the HWM and therefore which messages can be consumed. As a result, if an ISR starts to lag behind, all consumers are inevitably slowed down or are no longer able to consume the messages produced in near-real-time. After 30 seconds, or what we've configured as the maximum lag, the replica will disappear from the list of ISRs, the HWM will probably make a big leap, and the consumers can catch up again. Apart from that, we can consume messages as long as there's a partition leader and at least `min.insync.replicas`. This also means that if the current leader of a partition drops out, we can't consume messages for a brief moment (until a new leader is determined).

We might think about reducing the maximum lag to 2 seconds, but this could introduce new problems. Asynchronous replication can create a small, temporary lag between the followers and leaders, leading to replicas being removed from the ISR list and added back shortly after, which would generate unnecessary overhead.

Moreover, removing replicas from the ISR list too quickly results in tracking overhead because the ISRs are managed by the controller, and we might risk falling below `min.insync.replicas`. If the number of ISRs falls below the configured minimum, we can't produce new messages with `acks=all`, as there aren't enough ISRs available.

Additionally, the HWM won't advance, so even if we don't use `acks=all`, our consumers can't consume new messages. Although it might seem tempting to adjust the minimum ISR configuration for performance reasons, we strongly advise against this. It's crucial to set the minimum ISR to a reasonable value to ensure the reliability of our Kafka cluster.

Kafka doesn't handle the actual persistence of messages but leaves this task to the operating system. If the leader is the only ISR and fails suddenly, there's a risk that data will be consumed that hasn't been persisted in the filesystem yet.

TIP To improve reliability, the minimum number of ISRs should always be set to a reasonable value such as 2 when we have a replication factor of 3.

Kafka recently introduced a feature called *eligible leader replicas* (ELRs). When the number of ISRs falls below the `min.insync.replicas` threshold, the HWM doesn't advance. In this case, replicas that aren't in sync but have caught up to the HWM are considered eligible to become the leader of the partition in the event of a failure. While this feature doesn't guarantee that messages won't be lost with `acks=0` or `acks=1`, it ensures that no messages previously consumed are lost during a failure.

Summary

- Producers in Kafka typically use either the official Kafka Java library or `librdkafka`. Avoid using other libraries due to potential lack of features and optimizations.
- The producer workflow involves serialization, partitioning, and buffer management.
- Handling acknowledgments (ACKs) in the producer includes timeout settings and retry mechanisms.
- Kafka brokers delegate much of their work to clients, focusing on message reception and efficient message persistence.
- Upon receiving a produce request, the broker writes data to the operating system's page cache, potentially waiting for followers to replicate before sending an ACK.
- Network threads manage message reception, queuing them for I/O threads to write to the filesystem.
- Kafka relies on the operating system to persist messages to disk, with options to influence disk flush timing.
- Broker components can be optimized and configured for specific use cases, with professional support advised for complex environments.
- Kafka's data structures—including metadata, checkpoints, and topics—organize and manage data within brokers.
- Partitions divide topics into segments, each with log and index files for efficient data retrieval.

- Log data and indices optimize message storage and retrieval within log segments.
- Segments, based on size or time, manage partition growth and optimize data storage efficiency.
- Replication involves followers fetching data from leaders, ensuring all brokers stay up-to-date.
- In-sync replicas (ISRs) are followers that have received all messages from the leader within a specified time frame.
- The Log End Offset (LEO) marks the last received message position, determining ISR status.
- The High Watermark (HWM) indicates the offset replicated and committed to by all ISRs, affecting message consumption and availability.
- Delays in replication can slow down or halt message consumption and production, with ISR lag affecting HWM and consumer availability.
- Adjusting parameters such as `replica.lag.time.max.ms` and `acks=all` can affect replication efficiency and system performance, requiring careful configuration for balance.

9 Consuming messages

This chapter covers

- Kafka's fetch-based consumption model
- Offset management
- The role of consumer groups
- How Kafka coordinates task distribution
- The effect of Range Assignor and Round Robin Assignor
- Static memberships and Cooperative Sticky Assignor

In this chapter, we embark on a detailed exploration of Kafka's message consumption process. We begin by examining the fundamental principles of Kafka's fetch-based consumption model, shedding light on how consumers interact with the Kafka cluster to retrieve messages. As we delve deeper, we unravel the mechanisms behind offset management, discussing how consumers specify partitions and manage their reading progress within Kafka.

Furthermore, we delve into the Kafka rebalance protocol, elucidating how Kafka coordinates the distribution of partitions among consumer group members. Throughout our discussion, we explore strategies such as Range Assignor and Round Robin Assignor, which are essential for efficient partition assignment and workload distribution.

Additionally, we investigate advanced features such as static memberships and Cooperative Sticky Assignor, which are crucial components for optimizing rebalance behavior and ensuring seamless operation within Kafka consumer groups. By the end of this chapter, you'll have a comprehensive understanding of Kafka's message consumption workflow and the intricacies involved in consuming messages efficiently within Kafka.

9.1 Fetching messages

In the simplest case, we have a consumer that wants to consume data from one or more topics. The consumer has already successfully connected to the Kafka cluster, and thanks to the response to its metadata request, our consumer knows which brokers are the leaders for which partitions.

9.1.1 Fetch requests

To consume messages, the consumer sends a fetch request to the leaders with the partition it wants to consume from and the offset from which the broker should start returning messages. We can simulate this using the `kafka-console-consumer.sh` script as follows.

> **NOTE** In the previous chapter, we recreated our Kafka cluster, so we first need to create the topic `products.prices.changelog` again and produce some messages. Additionally, we might need to adapt the partition number in our command, as we don't know in which partition our messages will be. Alternatively, we can create the topic with just one partition.

```
$ kafka-console-consumer.sh \
     --topic products.prices.changelog \
     --offset 0 \
     --partition 0 \
     --bootstrap-server localhost:9092
coffee pads 10
coffee pads 11
coffee pads 12
coffee pads 10
```

For this, we need to explicitly specify the desired partition and the offset from which we want to start receiving messages. Usually, we don't specify the partition or the offset explicitly but rely on the Kafka library to handle this. We already encountered the `--from-beginning` flag in the first chapter. By default, `kafka-console-consumer.sh` starts reading at the end of a partition, and we use the `--from-beginning` flag to read from the beginning. The Kafka library uses the same default setting. When we start a consumer, we begin reading the latest messages. To start reading from the beginning

of the partitions, we need to set the configuration option `auto.offset.reset` to `earliest` (default is `latest`).

9.1.2 Fetch from the closest replica

So far, we've always said that a consumer fetches messages from the leader of a partition. This isn't entirely correct because since Kafka Improvement Proposal 392 (KIP-392), consumers can consume from the nearest replica.

> **NOTE** A Kafka Improvement Proposal (KIP) is a structured document used by the Apache Kafka community to propose and review new features or changes, ensuring organized, community-driven development. KIPs go through a process of submission, community discussion, and approval before implementation in Kafka.

But how does Kafka know which is the nearest replica? In Kafka, there's an option to assign a rack ID or location to brokers (`broker.rack`). A *rack* typically refers to a physical or logical grouping of servers, often within the same data center or availability zone, allowing Kafka to be aware of broker locations. Originally, this configuration was only intended to inform Kafka of the brokers' locations so that Kafka could distribute replicas evenly across fault domains to increase fault tolerance. With KIP-392, an equivalent configuration option was added for clients (`client.rack`), allowing clients to select a specific replica for fetching messages, as visualized in figure 9.1.

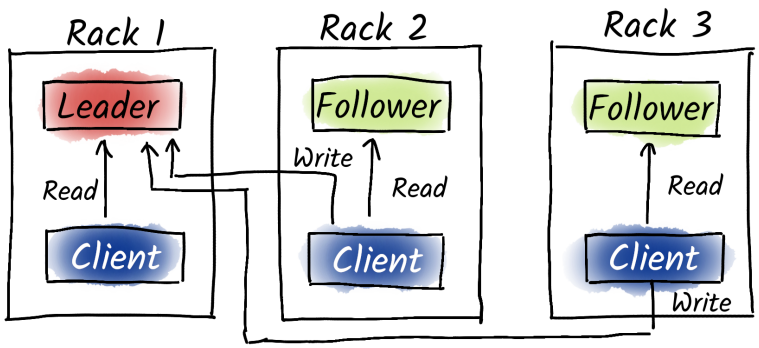


Figure 9.1 The consumers are consuming from the nearest replica (in the same rack), but producers still have to produce to the leader of the partition.

9.2 Broker handling of consumer fetch requests

Before we look in detail at how consumer groups help us persist offsets, let's first examine what the broker needs to do to provide our consumer with a satisfactory response. Essentially, the process a broker handles for a fetch request is very similar to a produce

request, with the main difference being that instead of writing data to the filesystem, the broker reads it from the filesystem. The broker starts with the same network threads that receive the request, placing it in the request queue for buffering. However, in this case, the I/O threads don't write batches to the filesystem, but instead read batches from it. Ideally, the broker doesn't even need to access the disk, as the messages may still be in the page cache and can be fetched from there.

When consuming messages, the request purgatory is also used, but instead of waiting for other brokers, it waits for additional messages if the consumer requests them. This behavior is configured on the consumer side because different consumers may have different latency and bandwidth requirements. By default, the broker sends an immediate response to the consumer when at least one new message is available, as the `fetch.min.bytes` setting is set to 1.

If there are no new messages, the broker won't wait indefinitely to respond. Instead, it will wait up to a maximum of `fetch.max.wait.ms`, which is set to 500 ms by default. As discussed in chapter 6, these parameters can be optimized to improve consumer performance.

After the broker decides to send a response, it places the response in the response queue, from where a network thread can send the response to our consumer. The whole process is shown in figure 9.2.

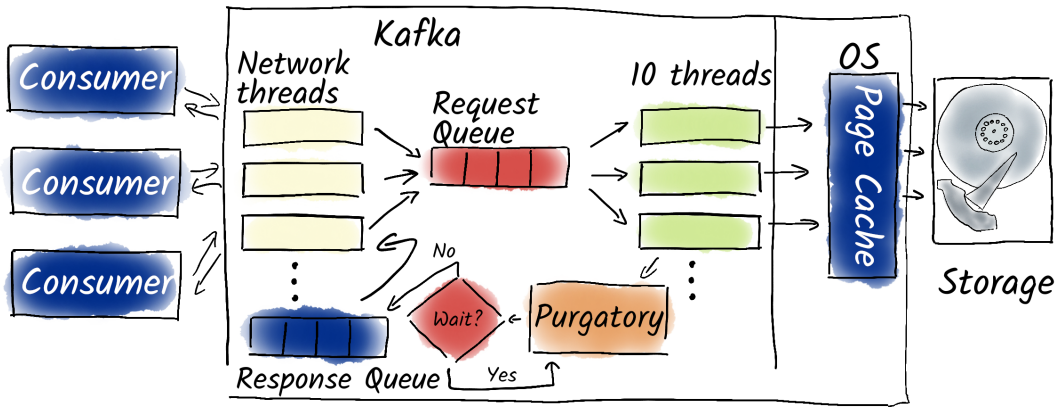


Figure 9.2 The broker undergoes a similar process when receiving a fetch request as that of when receiving a produce request. However, instead of writing data to the disk, we need to read from it. Additionally, we don't need to wait for other brokers, but rather for more messages depending on the `fetch.max.wait.ms` and `fetch.min.bytes` settings.

9.3 Offsets and Consumer

In traditional messaging systems, the system itself manages which messages are sent to which consumers. However, in Kafka, the consumer is responsible for tracking its

own messages from each partition. This means that the consumer has to keep track of which messages it has already read and which ones it hasn't. In Kafka, this is done using offsets.

When a message is produced, it gets assigned a unique offset starting from 0 for the first message in a partition, 1 for the second, and so on. The Kafka broker increments the offset for each new message as they are produced. Even if a message is later deleted (e.g., due to cleanup policies), the offsets of the other messages remain unchanged. As a result, there might be gaps in the sequence of offsets. However, this doesn't affect the processing of messages, as Kafka will simply send the next available message with its current offset.

9.3.1 Offset management

For the consumer, the offset indicates which message to read next. In section 9.1, we learned about the `auto.offset.reset` setting, which allows us to configure what the consumer should do if it doesn't know an offset for a partition yet, such as whether it should start reading from the oldest message (`auto.offset.reset=earliest`), that is, the one with the lowest offset, or from the end of the partition (`auto.offset.reset=latest`).

But where can our consumers store the offsets so that we can resume reading from where the consumer left off after a restart? There are cases where we don't want to store our offsets at all. For example, let's say our consumer populates an in-memory cache with data from Kafka. If this consumer crashes, the data is lost, and we need to refill this cache from scratch. We don't need to store offsets here.

In most cases, we want to persist, process, or forward data from Kafka somewhere else, which means we need to store our offsets. One way to do this is by storing the offset locally on the service's disk. Every time the service restarts, it reads the stored offset from the disk and requests messages starting from that point. This approach is useful, for example, if we want to store Kafka data in a local database or cache.

However, in many cases, we want to keep our services stateless, which eliminates the need for persistent storage of offsets within the service. Storing offsets in the service also complicates horizontal scaling, as new instances of the service will have no knowledge of the last processed offset, making it difficult to ensure all consumers start from the correct point.

Another option is to store the offsets in an external system. For example, if we write the data from Kafka to a database, we can create an additional table for offsets and even write the data with the offsets in a transaction. This even gives us exactly-once guarantees for our consumer! Because either the read message with the offset is written to the database, or neither one. By the way, this is exactly the approach many Kafka Connect connectors use to provide exactly-once guarantees.

But actually, we already have a system in which we store data, and offsets are nothing but data as well. Instead of managing offsets independently, Kafka provides built-in support for handling them through a compacted topic called `__consumer_offsets`.

This compacted topic ensures that old offsets for each consumer group are cleaned up regularly using compaction to save space. We'll learn more about compaction in the next chapter.

Kafka itself doesn't know which offsets belong to which consumers, so we must specify this. Rather than storing offsets separately for each consumer, Kafka associates them with a consumer group. This allows us to scale our consumers horizontally, with multiple instances of the same consumer group sharing responsibility for consuming messages from the same set of partitions.

9.3.2 Understanding offsets in Kafka

Now, let's create a topic named `products.prices-offsets` with two partitions. The following code illustrates this:

```
$ kafka-topics.sh \
    --create \
    --topic products.prices-offsets \
    --partitions 2 \
    --replication-factor 3 \
    --bootstrap-server localhost:9092
Created topic products.prices-offsets.
```

Afterward, we produce some messages into this topic:

```
$ kafka-console-producer.sh \
    --topic products.prices-offsets \
    --property parse.key=true \
    --property key.separator=: \
    --bootstrap-server localhost:9092
> coffee pads:10
> cola:2
> energy drink:4
> coffee pads:11
> coffee pads:12
> energy drink:4
```

Now, let's start a Kafka consumer and specify a group:

```
$ kafka-console-consumer.sh \
    --topic products.prices-offsets \
    --bootstrap-server localhost:9092 \
    --group products.prices.monitoring \
    --from-beginning \
    --property print.key=true \
    --property key.separator=":"
energy drink:4
energy drink:3
coffee pads:10
cola:2
coffee pads:11
coffee pads:12
```

> **NOTE** We can see that the order of messages consumed in our consumer is different compared to the order of messages produced in our producer. This is because the messages for `energy drink` are produced in one partition while the messages with `cola` or `coffee pads` are produced into the other. Additionally, due to the nature of our simple example, our consumer will very likely first print all messages from one partition and afterward from the other.

If we interrupt this command with Ctrl-C and start it again, we won't see any new messages. That means the correct offsets have been remembered somewhere. To check this, we can use the `kafka-consumer-groups.sh` tool, which shows us information about this group. Let's start it in a new terminal:

```
$ kafka-consumer-groups.sh \
    --group products.prices.monitoring \
    --describe \
    --bootstrap-server localhost:9092
```

For clarity, table 9.1 shows the information provided by the `kafka-consumer-groups.sh` tool about our consumer group `products.prices.monitoring`.

Table 9.1 Describing consumer group `products.prices.changelog-monitoring`

Group	products.prices.monitoring	products.prices.monitoring
Topic	`products.prices-offsets`	`products.prices-offsets`
Partition	0	1
Current Offset	2	4
Log End Offset	2	4
Lag	0	0
Consumer ID	`console-consumer-<ID>`	`console-consumer-<ID>`
Host	`/127.0.0.1`	`/127.0.0.1`
Client ID	`console-consumer`	`console-consumer`

We can see that our consumer group is consuming the topic `products.prices-offsets` as expected, with one entry for each partition. Partition 0 has a Log End Offset (LEO) of 2, indicating that the offset of the next written message will be 2. This LEO is identical to the current offset of our group, indicating that the group has read all messages from this partition. As a result, the lag is 0. The most important metric for our consumers is this lag, which indicates how many messages a consumer group lags behind. If this is 0, we know that all messages have been read. For Partition 1, the values are similar, except we only have one message in the partition.

The consumer ID is automatically generated to identify members of a consumer group. Because we've only started one consumer, the ID of the consumer for both partitions is identical. The host corresponds to the IP address of the consumer.

NOTE The `client.id` is configurable from the consumer application itself. This is useful for logging, metrics, and tracking purposes.

If we now stop our consumer, produce some new messages, and then rerun the `kafka-consumer-groups.sh` command, we'll see that the LEO and the lag have changed depending on the new messages (see table 9.2).

Table 9.2 Describing consumer group `products.prices.changelog-monitoring` after writing more messages

Group	products.prices.monitoring	products.prices.monitoring
Topic	products.prices-offsets	products.prices-offsets
Partition	0	1
Current Offset	2	4
Log End Offset	4	7
Lag	2	3

If we were to restart the consumer, we could read the new messages, and the lag would again reduce to 0. Even if we don't need to scale our consumers horizontally, consumer groups provide us with a very convenient and reliable method for storing offsets.

WARNING The group ID determines which consumer group is responsible for consuming and committing specific offsets. If you accidentally use the wrong group ID, whether by mistakenly copying code or forgetting to change the group parameter, it can result in one consumer overwriting the offsets of another. This can cause one consumer to steal the offsets, leading it to consume messages that were intended for a different consumer. While this may cause problems during development or testing, it can have severe consequences in production, potentially leading to data loss or processing inconsistencies.

9.4 Understanding and managing Kafka consumer groups

We've encountered consumer groups several times already. We use consumer groups to horizontally scale our consumers, and we learned in the previous section that we use consumer groups to conveniently manage offsets in Kafka.

9.4.1 Consumer group management

To use consumer groups, we don't need any additional components; we just need to set the group ID in our consumers. One or more consumers with the same group ID form a consumer group.

Now, our goal in Kafka is to allow the same messages to be read multiple times by different services. For example, in the introduction of the book, we created the topic `products.prices.changelog` and ran two services, a sales analytics service and the

price update service. We saw that these two consumer groups independently consumed the topic and distributed the partitions within the group, as shown in figure 9.3.

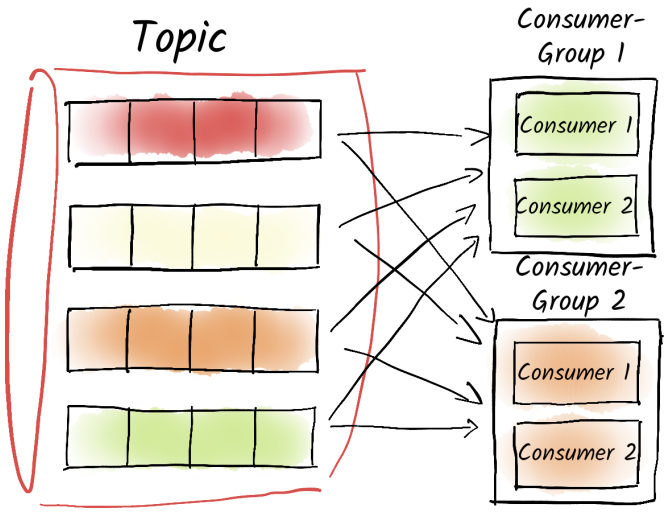


Figure 9.3 A consumer group consists of any number of consumers that divide the desired partitions among themselves. Different consumer groups are isolated and unaware of each other. The Kafka brokers assist consumers in coordinating the task distribution.

But how do the consumers in a group know about each other, and how do they divide the partitions among themselves? The consumers need to coordinate their work somehow. We recall from our introduction to distributed systems in chapter 4 that coordination is a very challenging problem. However, we already have a fault-tolerant distributed system at our disposal, namely the Kafka brokers themselves. Instead of operating a separate coordination cluster for each consumer group, the Kafka brokers assist us in coordinating the consumers.

We'll learn later that consumer groups aren't the only component supported by Kafka in this manner. Similarly, Kafka brokers assist in coordinating our Kafka Connect clusters and Kafka Streams applications.

For this purpose, there's even a dedicated protocol called the *Kafka Rebalance Protocol*. This protocol enables the distribution of resources (e.g., partitions, Kafka Streams tasks, or Kafka Connect tasks) among members of a group. Here, too, Kafka follows the philosophy of offloading as much work as possible onto the clients. Kafka itself isn't concerned with the specifics of what's being coordinated.

NOTE At the time of writing, there's ongoing work to implement the next generation of the Consumer Group Rebalance Protocol as described in KIP-848. The new version will vastly simplify the operations of consumer groups and Kafka Streams. Even though the current version will still be available for specific use cases, we advise you to use the next-generation rebalance protocol as soon as it's released in version 4.0.0.

But how does this Kafka Rebalance Protocol work? First, on the Kafka side, we need a central instance responsible for a specific group. This broker is called the *group coordinator*. For better load distribution, there isn't just one broker acting as the group coordinator; instead, we distribute the load as evenly as possible. To achieve this, we use a familiar pattern: depending on the group ID, Kafka distributes the group coordinators across the brokers. The protocol is visualized in figure 9.4.

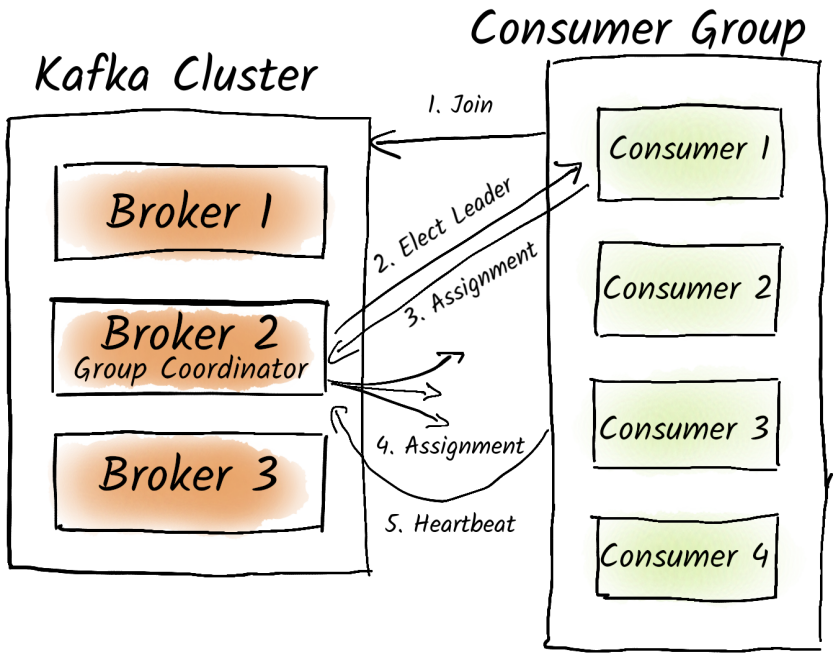


Figure 9.4 The Kafka Rebalance Protocol allows a group to distribute certain resources among its members, for example, consumer partitions. In this process, a broker acts as the group coordinator, and one of the consumers serves as the group leader.

When consumers want to join a consumer group, they first send a join request (1) to the group coordinator. The group coordinator manages the group and its members. The join request acts as a synchronization point, meaning that no response is sent until all potential group members have reported to the coordinator. If there are already existing members in the group, they are temporarily removed by the group coordinator to rejoin with a new join request. During this process, the consumers aren't yet members of the group and can't consume any messages.

To identify the group coordinator, the consumer must first send a `FindCoordinator` request to any broker in the Kafka cluster. The broker then responds with the identity (host and port) of the group coordinator for the requested consumer group. Once the

consumer knows which broker is the group coordinator, it can directly communicate with that group coordinator for group management operations. This communication includes the `JoinGroup` request (to join the group and initiate a rebalance), `Heartbeat` signals (to indicate the consumer is still active), and `SyncGroup` (to receive partition assignments after a rebalance).

When the group members have joined the group, the first member to join is designated as the group leader (2). This leader is responsible for coordinating the distribution of work within the group, particularly which group member will handle which partition. The leader sends this partition distribution plan to the group coordinator (3). The coordinator then passes these assignments back to the other group members (4). At this point, the group members are allowed to resume consuming messages from their assigned partitions.

Once the group is formed, the group coordinator must monitor the members to ensure they are still active. To do this, each consumer sends heartbeat signals at regular intervals (5). If a member fails to send heartbeats within a specified period (usually three intervals), the group coordinator considers the member dead and removes it from the group. The intervals can be adjusted using the `heartbeat.interval.ms` setting, and the timeout after which a consumer is considered inactive is controlled by `session.timeout.ms`. However, for most use cases, the default settings are usually sufficient.

In summary, the group coordinator doesn't disassemble the group when it receives a join request, but it coordinates the addition of new members to the group and ensures that they are assigned partitions. When a consumer group is formed, the first consumer to join becomes the leader, responsible for partition assignments. The group coordinator helps manage the members of the group by ensuring they stay alive via heartbeat signals and managing partitions across them.

9.4.2 Distribution of partitions to consumers

Kafka offers several ways to distribute partitions to consumers. We can configure this through the setting `partition.assignment.strategy` in the consumer. The group leader is responsible for determining partition assignments during a rebalance and communicating them to the group coordinator. All other consumers simply follow the leader's assignments.

> **WARNING** It's crucial that all consumers in the same group use the same partition assignment strategy. If they use different strategies, the rebalance can cause unpredictable changes in partition assignments, leading to potential data processing inconsistencies and performance problems.

By default, Kafka uses the *Range Assignor*. With the Range Assignor, we assume that all our consumers want to consume topics that have the same number of partitions. The Range Assignor now guarantees that Partition 0 of all topics will be assigned to the same consumer, Partition 1 of all topics to the same consumer, and so on. For example,

if we have two topics, each with two partitions and three consumers, as shown in the example in figure 9.5, we assign Partition 0 of both topics to Consumer 1 and assign Partition 1 of both topics to Consumer 2. Consumer 3 isn't assigned any partitions.

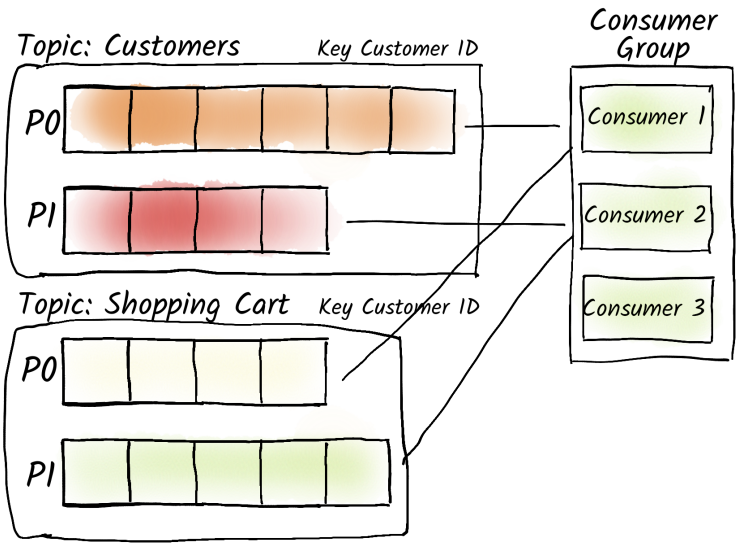


Figure 9.5 By default, partitions in a consumer group are assigned in such a way that the same partition across different topics is always consumed by the same consumer. While this may not be the most resource-efficient allocation, it allows us to easily perform joins across multiple topics.

This assignor is useful when we want to process data from multiple topics together. For instance, we could enrich the data from the `products.prices.changelog` topic with information about sales for our analytics service from the `products.sales` topic. We use the name of the product as the key in both topics, so we know that messages for the same product always land on the same partition, whether in the `products.prices.changelog` topic or the `products.sales` topic. This means that if we read data about a product from Partition 0 of the `products.prices.changelog` topic, we'll also find the data about the product in Partition 0 of the `products.sales` topic. This is called a *stream join*, and Kafka Streams, in particular, makes extensive use of it.

> **NOTE** Even though it's possible to implement a naive version of joins using the Range Assignor, we highly advise not to do it in a consumer. If you need to join data from multiple topics, use a stream processing library such as Kafka Streams.

Usually, when consuming from multiple topics with a simple consumer, we may not need to correlate the data between topics, and we might not even use keys. In this case, we can avoid the limitations of the Range Assignor and instead use the *Round Robin*

Assignor. This assignor evenly distributes the partitions of all topics among the consumers, as shown in figure 9.6.

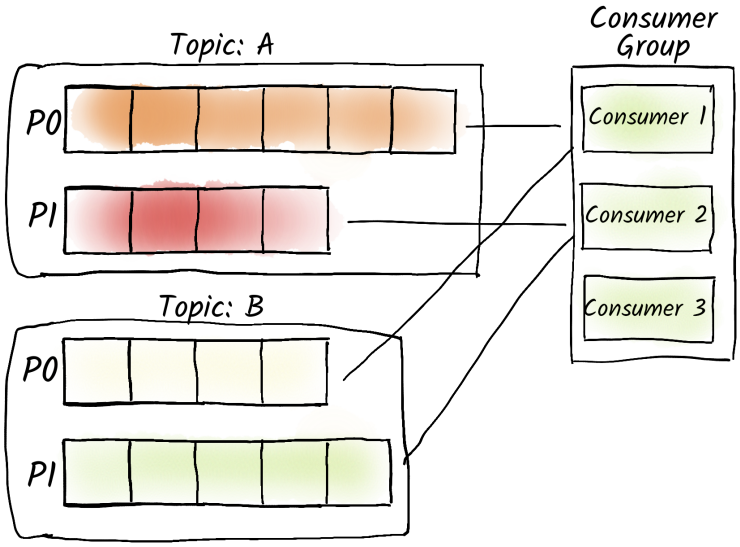


Figure 9.6 The Round Robin Assignor (and the more advanced Sticky and Cooperative Sticky Assignors described later) distributes the load evenly across the consumers of our consumer group.

We remember that during a rebalance, we must stop consuming, and our consumer might get assigned different partitions after a rebalance. This means that consumers need to clear any internal state, such as caches, and commit their offsets before a rebalance occurs. In our example image, if Consumer 1 fails, each consumer will be assigned entirely different partitions. We can avoid this by using the *Sticky Assignor*. Instead of recalculating the partition assignment from scratch each time, the Sticky Assignor tries to minimize changes as much as possible. Otherwise, it works just like the Round Robin Assignor.

Even though the Sticky Assignor is a major improvement over the Round Robin Assignor because it doesn't try to reassign everything, it still requires stopping the world for the time of the rebalance. To combat this, Kafka 3.0 introduced an improved assignor called the *Cooperative Sticky Assignor*. Instead of needing to stop the world during the rebalance, the cooperative protocol allows consumers to continue consuming the partitions that don't change.

Let's take a closer look at how this works. In figure 9.7, we have a consumer group consisting of two consumers consuming these three partitions. Next, a new consumer joins the group. Using the Sticky Assignor, all consumers would need to stop consuming messages, then the reassignment happens, and then normal consumption continues.

With the cooperative protocol, the group instead continues normal consumption. During the first rebalance, the assignment "takes away" the third partition from Consumer 2. Of course, now the consumption for the third partition stops. But this rebalance immediately triggers the next rebalance where the currently dangling Partition 3 is assigned to the new Consumer 3. This reduces the need to stop consumption for all partitions just because one partition needs to be reassigned.

> **TIP** Until the next-generation Consumer Group Rebalance Protocol is released in version 4.0.0 and available in your Kafka installation, we recommend using the Cooperative Sticky Assignor in consumer groups.

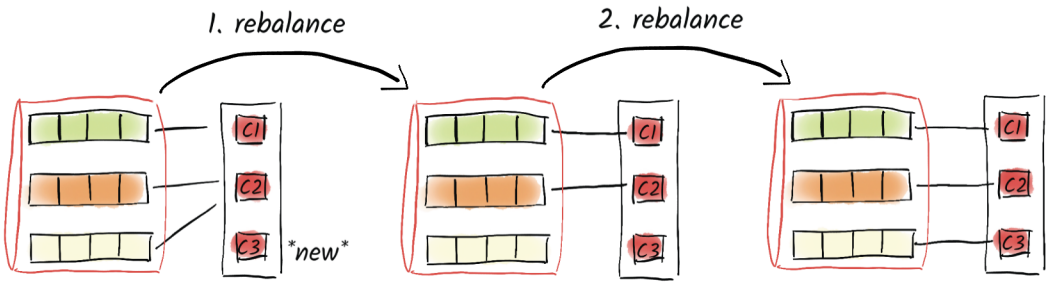


Figure 9.7 The Cooperative Sticky Assignor avoids stopping the world and rebalances by replacing a single rebalance by two consecutive ones. During the first rebalance, consumers are asked to let go of the partitions that should be moved to another consumer; in the second rebalance, the now dangling partitions are assigned to the consumers.

So, the Kafka Rebalance Protocol allows us to distribute resources such as partitions or connector tasks among members of a group. However, because we need to halt all processes during the execution of the protocol, rebalances are very costly. Rebalances are triggered when new members want to join the group, when members intentionally or unintentionally leave the group, or when the number of partitions of a consumed topic changes, for example. In our fully automated environment, for example, performing a rolling restart of three consumers in a consumer group would trigger six rebalances: one each when shutting down a consumer and again when starting it up.

9.4.3 Static memberships

To avoid such excessive rebalances, Kafka 2.3 introduced the concept of *static memberships*. We assume that our infrastructure is fully automated, and if a consumer fails, it's automatically restarted immediately. This is often the case in cloud and Kubernetes environments. Instead of triggering a rebalance every time a consumer restarts, we increase the `session.timeout.ms` to a significantly larger value (e.g., several minutes).

Additionally, we need to provide each consumer with its identity. For this purpose, we set the `group.instance.id` for each consumer to a unique value per consumer. We

only need to ensure that the consumer gets the same ID after a restart. We can ensure this, for example, by using Kubernetes StatefulSets or setting the ID to the hostname in the cloud if it remains constant across restarts. With these settings, a rebalance only needs to be performed when new consumers are added, a consumer doesn't report for several minutes, or the number of partitions changes. This significantly improves the utilization of our consumers and avoids major interruptions in data processing.

Summary

- Kafka uses a fetch-based approach for consumers to retrieve messages.
- Kafka brokers handle most of the coordination work, minimizing overhead.
- Kafka brokers support rack IDs for fault tolerance and load distribution.
- Consumer groups help manage offsets and distribute workload among consumers.
- Offsets are stored in Kafka as part of the consumer group's metadata in a special internal topic called `__consumer_offsets`.
- Offsets are crucial for consumers to keep track of which messages they have already consumed.
- The Kafka Rebalance Protocol coordinates task distribution among consumer group members.
- Consumer groups facilitate parallel processing by distributing partitions among consumers.
- Rebalances are triggered by changes in group membership, topic partitions, or consumer failures.
- Range Assignor and Round Robin Assignor are strategies for partition assignment.
- Range Assignor ensures that the same consumer handles the same partitions across topics.
- Round Robin Assignor evenly distributes partitions among consumers when joins across topics aren't required.
- Static memberships and Cooperative Sticky Assignor optimize rebalance behavior.
- Static memberships reduce rebalance frequency by extending session timeouts and using unique group instance IDs.
- Cooperative Sticky Assignor improves rebalance efficiency by iteratively approaching the desired state.

10 Cleaning up messages

This chapter covers

- Mechanisms behind message cleanup in Kafka
- Options for managing message retention
- How Kafka handles cleanup of outdated data

In Kafka, managing the life cycle of messages is crucial for maintaining system performance and ensuring data integrity. This chapter delves into two key approaches: log retention and log compaction. Log retention focuses on deleting messages based on age or size, offering simplicity in implementation and catering to various use cases such as compliance requirements and data management. On the other hand, log compaction selectively removes outdated data based on keys, ensuring that only the latest message for each key is retained. By understanding the principles and configurations of log retention and log compaction, Kafka users can effectively manage message retention policies tailored to their specific needs, optimizing storage usage and ensuring data accuracy throughout the system.

10.1 Why clean up messages?

Before we get into the details of how Kafka cleans up messages, let's briefly consider why we should clean up messages in Kafka and what consequences will arise if we never clean up messages. One reason is our storage capacity. Theoretically, we could simply store all messages forever, but this would also cause our log to grow infinitely and quickly reach the limits of our available storage.

Another reason is performance. The larger our log, the longer it will take for us to process all messages, and depending on the use case, we probably process a large number of irrelevant messages as well. Perhaps the most important reason for deleting data is that we no longer need the data or are no longer allowed to retain it for legal reasons. However, we also must ensure that we don't accidentally delete messages that we still need.

10.2 Kafka's cleanup methods

Kafka pursues two different approaches for this purpose: log retention and log compaction. With *log retention*, all messages that have reached a certain age, meaning messages produced before a certain time, are simply deleted. *Log compaction*, on the other hand, deletes outdated data, as shown in figure 10.1. To determine if a message is outdated, Kafka uses the keys of our messages. In this process, only the latest message is retained for each key. Log compaction is therefore only possible when we assign keys to our messages.

Each method has its own advantages and disadvantages, and they can be combined. Log retention has the advantage of being relatively easy to implement and causing little overhead for our brokers, as we only need to check the age of our messages to delete them. At the same time, log retention serves important use cases in a very straightforward manner. For example, we can ensure that messages which need to be deleted after a certain time for data protection reasons are automatically deleted.

Another important class of use cases for log retention includes sensor data, which may become irrelevant after a certain time and can be deleted, or logs from programs that we don't need to keep indefinitely. Perhaps Kafka is only used as

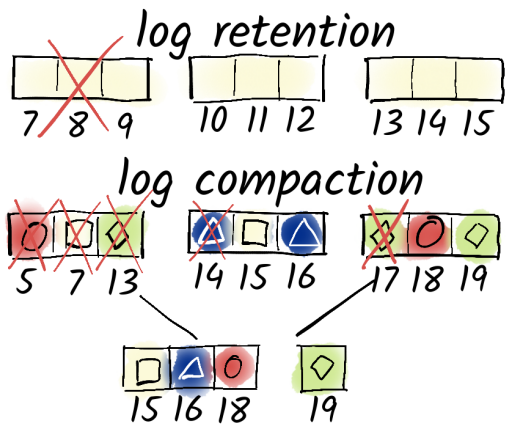


Figure 10.1 The log cleaner always operates on a segment-by-segment basis. With the delete policy, an entire segment is deleted once the newest message in the segment is older than a predefined value. If we use the compact policy, the log cleaner deletes only messages with a key for which there is a newer message with the same key. It's also possible to employ both methods simultaneously.

a messaging system, in which case, we can also safely delete messages after a short time. However, with log retention, we can't selectively delete data and must be careful not to accidentally delete important data just because it's old.

This is where log compaction comes into play. For example, imagine we have a topic where we store customer data, such as customer addresses. Every time the address changes, a new message with the corresponding key for the customer is generated. The old address will likely become irrelevant from that point on, but at the same time, we need to ensure that we don't delete the current address just because the message has reached a certain age.

Log compaction is ideal for such use cases because it ensures that we always keep the latest message for a key and automatically delete outdated information or messages. The downside of log compaction compared to log retention is the relatively high overhead; with log compaction, we need to search the entire log to determine which messages can be deleted.

Whether we use log retention, log compaction, or even both simultaneously to clean up messages, we can set it either per topic using `cleanup.policy` or set a default value for all our topics via `log.cleanup.policy`. By default, log retention is activated in Kafka (`log.cleanup.policy=delete`). In the following two sections, we'll take a closer look at how exactly log retention and log compaction work in Kafka.

10.3 Log retention

In this section, we'll examine with examples when Kafka exactly deletes data and how we can configure log retention according to our needs. Let's start by creating a topic called `products.prices-retention`:

```
$ kafka-topics.sh \
    --create \
    --topic products.prices-retention \
    --partitions 3 \
    --replication-factor 3 \
    --bootstrap-server localhost:9092
Created topic products.prices-retention.
```

Afterward, we'll produce a few messages into the newly created topic so that we have messages to delete as well:

```
$ kafka-console-producer.sh \
    --topic products.prices-retention \
    --bootstrap-server localhost:9092
> cola 2
> coffee pads 8
[…]
> energy drink 5
```

If we take a look into one of our partitions, we'll see the files familiar to us from chapter 8:

```
$ ls -1 ~/kafka/data/kafka1/products.prices-retention-0
00000000000000000000.index
00000000000000000000.log
00000000000000000000.timeindex
leader-epoch-checkpoint
partition.metadata
```

10.3.1 When is a log cleaned up via retention?

We can configure log retention in Kafka in two different ways. First, we can set the size of a partition at which log retention is triggered. If the partition size exceeds this value, the oldest segment is deleted. The parameter to configure the partition size at which this happens is `retention.bytes`, defining the size in bytes. By default, this value is set to `-1`, which means that log retention based on partition size is disabled, as this is only useful in rare cases.

The second option is to delete messages or segments after a defined period called the *retention period*. The parameter for this is `retention.ms`, so the period is defined in milliseconds. By default, Kafka deletes messages or segments when the newest message in a segment is older than seven days, as shown in figure 10.2.

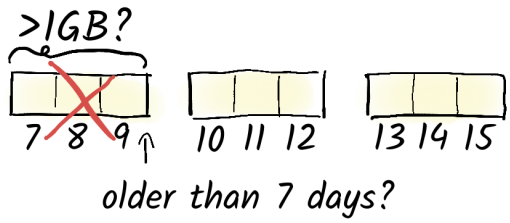


Figure 10.2 Log retention checks whether a segment is larger than 1 GB or whether the newest message is older than seven days. If so, the whole segment is deleted.

With `log.retention.bytes` and `log.retention.ms`, we can adjust the default values for our cluster. Both options can also be combined. This way, we can ensure that we delete both old messages and prevent our topic or partition from growing infinitely.

Let's now try out log retention by adjusting the configuration of our `products.prices-retention` topic using the `kafka-configs.sh` script and setting the value for `retention.ms` to `60` seconds:

```
$ kafka-configs.sh --alter \
    --topic products.prices-retention \
    --add-config retention.ms=60000 \
    --bootstrap-server localhost:9092
Completed updating config for topic products.prices-retention.
```

If we then take a look into our partition, we'll notice that some changes have occurred:

```
$ ls -1 ~/kafka/data/kafka1/products.prices-retention-0
00000000000000000004.index
00000000000000000004.log
00000000000000000004.snapshot
00000000000000000004.timeindex
leader-epoch-checkpoint
partition.metadata
```

> **NOTE** If we don't see the new files, we were either too fast in checking or we should try another partition. Additionally, we might see our old files with a `.deleted` extension.

As we can see, our original files containing the messages we just produced have disappeared. This is because our last message was produced more than 60 seconds ago, so the last message in our segment, and thus the segment itself, was older than our retention period of 60 seconds, leading to the deletion of the entire segment. Because a partition always consists of at least one segment, a new segment was created with the current offset as the filename.

Additionally, the *leader epoch* and *recovery checkpoints* were adjusted to the current offsets. Subsequently, when we produce messages, they land in our new segment. After 60 seconds, our segment will be deleted again, and a new segment will be created.

Log retention depends on how often we create a new segment. This dependency is best illustrated by an example. Let's say, for regulatory reasons, we're allowed to retain a message for a maximum of seven days. It might seem obvious to simply set `retention.ms` to a value corresponding to seven days. However, this isn't sufficient because the retention period refers to the newest message in a segment. As long as we regularly write messages into a segment, it can't be deleted, so all messages in this segment remain.

Therefore, we first need to ensure that we regularly roll out a new segment, which we can do, as mentioned in chapter 8, by using `segment.ms`. Second, we must ensure that `retention.ms` is chosen such that the oldest message in a segment is deleted no later than the maximum allowed retention period. Ultimately, it's important that the sum of `segment.ms` and `retention.ms` isn't greater than seven days, although this isn't yet 100% accurate, as Kafka defaults to checking only every 5 minutes whether segments can be deleted, so we need to subtract another 5 minutes. This property can be adjusted via `log.retention.check.interval.ms`.

How we divide `segment.ms` and `retention.ms` into these seven days, or seven days minus five minutes, affects how long our messages are actually retained. For example, if we only roll out a new segment every six days, we could set `retention.ms` to a maximum of 23 hours and 55 minutes. This would result in our segment containing messages that can be between one and seven days old at the time of deletion.

Alternatively, if we roll out a new segment every day and adjust `retention.ms` to five days and 23 hours and 55 minutes accordingly, we delete messages that are between six and seven days old. Therefore, the more frequently we roll out a new segment, the more finely we can delete our messages, and the higher we can set `retention.ms`, which allows us to retain messages longer.

> **TIP** To delete all messages in a topic, it was recommended in old Kafka versions to set the retention period to 0, wait until it's deleted, and then reset the period to the initial version. While this approach worked, we recommend using the Kafka Admin API to delete messages in a topic instead.

10.3.2 Offset retention

As offsets for consumer groups are also stored in a Kafka topic, it makes sense to clean up the offsets of old consumers too. This is also done by the log cleaner and configured using the broker setting `offsets.retention.minutes`. This setting configures after what time offsets of inactive consumer groups should be deleted. By default, offsets of inactive consumer groups are deleted after seven days.

We don't recommend changing it to a shorter period, as consumer groups might lose their offsets if you need to stop them because you encountered a bug and you need longer than a week to fix this. In addition, the offsets consume hardly any storage, and it might even make sense to increase the value.

10.4 Log compaction

Log compaction allows us to identify and delete outdated data based on their keys, as shown in figure 10.3. Unlike log retention, log compaction guarantees that we always have at least the latest entry for a key in our log.

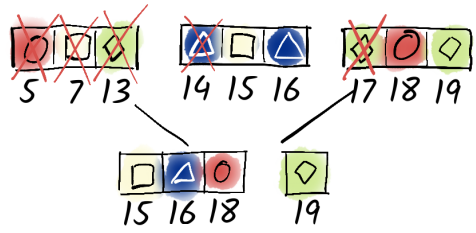


Figure 10.3 Log compaction reduces the amount of disk space used by deleting an old message for which there is a newer message with the same key.

Log compaction is best explained with a practical example. Let's start by creating a new topic called `products.prices-compaction`:

```
$ kafka-topics.sh \
    --create \
    --topic products.prices-compaction \
    --partitions 3 \
    --replication-factor 3 \
    --config cleanup.policy=compact \
    --bootstrap-server localhost:9092
Created topic products.prices-compaction.
```

This time, we explicitly set the compaction policy using the `--config` parameter (`--config cleanup.policy=compact`), as topics typically have log retention set as the cleanup policy by default. Let's now proceed as usual and try to produce a message:

```
$ kafka-console-producer.sh \
    --topic products.prices-compaction \
    --bootstrap-server localhost:9092
> cola 2
> ERROR when sending message to topic products.prices-compaction with
key: null, value: 6 bytes with error: (…)
org.apache.kafka.common.InvalidRecordException: Compacted topic cannot
accept message without key in topic partition products.prices-compaction-1
```

This time, Kafka doesn't allow us to simply send messages. Because we chose log compaction as the cleanup policy, which inherently requires keys, all messages without keys

are rejected by our brokers. Let's try again to send messages by using the producer properties `parse.key` and `key.separator` and then assigning keys to our messages:

```
$ kafka-console-producer.sh \
    --topic products.prices-compaction  \
    --property parse.key=true \
    --property key.separator=: \
    --bootstrap-server localhost:9092
>cola:2
>coffee pads:10
[…]
>energy drink:5
```

Then, we take a brief look at the log for our Partition 0 by using the `kafka-dump-log.sh` script. Using `--print-data-log`, this command also displays the contents of each batch, allowing us to see which keys and values are present in our messages:

```
$ kafka-dump-log.sh \
    --print-data-log \
    --files ~/kafka/data/kafka1/products.prices-compaction-0/0[……] 000.log
Dumping ~/kafka/data/kafka1/products.prices-compaction -0/0[……]000.log
Log starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 […] position: 0
CreateTime: 1738488072332 size: 81 […] crc: 906930835
| offset: 0 CreateTime: 1738488072332 keysize: 12 valuesize: 1 sequence: 0
headerKeys: [] key: energy drink payload: 5
```

In our example, only the messages consisting of the key `energy drink` landed in Partition 0. Let's now produce another message with the key `energy drink`:

```
$ kafka-console-producer.sh \
    --topic products.prices-compaction \
    --property parse.key=true \
    --property key.separator=: \
    --bootstrap-server localhost:9092
>energy drink:3
```

Then, we take another look at the log, and, as expected, the new message with the same key also lands in Partition 0:

```
$ kafka-dump-log.sh \
    --print-data-log \
    --files ~/kafka/data/kafka1/products.prices-compaction-0/0[……]000.log
Dumping ~/kafka/data/kafka1/products.prices-compaction-0/0[……]000.log
Log starting offset: 0
[…]
baseOffset: 3 lastOffset: 3 count: 1 […] position: 243
CreateTime: 1738488316569 size: 81 […] crc: 3247897454
| offset: 0 CreateTime: 1738488316569 keysize: 12 valuesize: 1 sequence: 0
headerKeys: [] key: energy drink payload: 3
```

10.4.1 When is a log cleaned up via compaction?

Despite log compaction, the outdated message still remains in our log. Shouldn't the older message have been deleted?

> **NOTE** At this point, we could've also used our `kafka-console-consumer.sh` to check which messages we were still able to consume and which ones were deleted by log compaction.

The reason for the outdated messages remaining is that log compaction, much like log retention, doesn't occur continuously. First, our log cleaner defaults to checking every 15 seconds (`log.cleaner.backoff.ms`) whether messages can be deleted. Second, searching the entire log for outdated messages every 15 seconds is both inefficient and practically impossible. Therefore, the log cleaner uses additional parameters to determine if a log can be compacted.

With `min.cleanable.dirty.ratio`, we can specify the minimum ratio between the dirty log and the entire log for our log cleaner to take action. A segment is considered *dirty* if it has never been compacted. By default, the ratio between dirty segments and the entire log must be at least 0.5 for a partition to be compacted again. We can adjust this default value with `log.cleaner.min.cleanable.dirty.ratio`. The lower we set this value, the more frequently our log is compacted, and the less maximum storage space is occupied by outdated messages, as only as many messages can be outdated as there are new messages in the uncompacted segments. For example, with a value of 0.5, twice as much storage space is used as necessary, and with a value of 0.2, the overhead in storage space is at most 25%.

So why not simply set the value to zero or almost zero? After all, this could potentially save a lot of storage space. We don't do this because log compaction is very resource-intensive, as every log compaction requires reading the entire partition, and our brokers would then be solely occupied with log compaction. However, the log dirty ratio isn't the only parameter by which the log cleaner determines whether a log can be compacted. With the parameters `max.compaction.lag.ms` and `min.compaction.lag.ms`, we can specify after what maximum period a message should be compacted and how long messages shouldn't be compacted. The former is used to regularly compact logs with low data throughput, while the latter ensures that messages are retained for at least a certain period, allowing consumers to read all messages, even if they are theoretically outdated. By default, the minimum period is set to 0 (`log.cleaner.min.compaction.lag.ms=0`), and the maximum period is practically unset with the default value of approximately 300 million years (`log.cleaner.max.compaction.lag.ms=9223372036854775807`).

As illustrated in figure 10.4, a partition is considered suitable for compaction by the log cleaner if there are uncompacted messages older than the maximum compaction lag or if the dirty ratio exceeds the threshold and there are uncompacted messages older than the minimum compaction lag. Furthermore, the log dirty ratio serves

another purpose: prioritizing log compaction if multiple partitions are simultaneously suitable for log compaction.

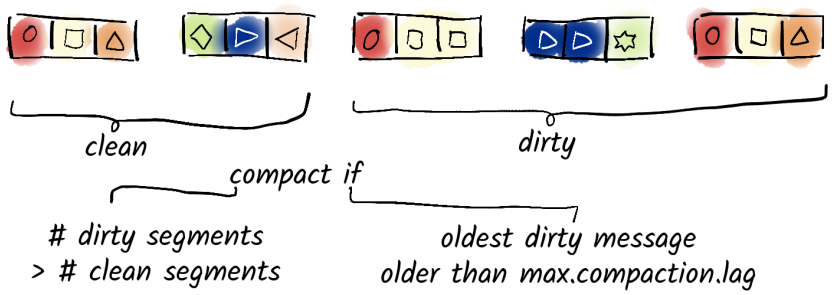


Figure 10.4 By default, compaction is triggered when the number of dirty segments is larger than the number of clean segments. Often, it's also useful to set the max compaction lag.

Now, let's return to our initial question of why the partition in the previous example hasn't been compacted yet. According to what we've just learned, our partition should have been compacted already, but the log cleaner, or log compaction in Kafka, always ignores the current segment. The reason for this is relatively simple: log compaction alters segments or fundamentally rebuilds our partition. Applying log compaction to the current segment, where we write all newly produced messages, could easily lead to inconsistencies.

Therefore, we must ensure that a new segment is created. We could use our familiar `segment.ms` parameter for this purpose, but we can achieve the same effect with `max.compaction.lag.ms`. This parameter automatically creates a new segment if the oldest message in a segment is older than this maximum delay time. Let's adjust the configuration of our `products.prices-compaction` topic accordingly and set `max.compaction.lag.ms` to `60` seconds:

```
$ kafka-configs.sh \
    --alter \
    --topic products.prices-compaction \
    --add-config max.compaction.lag.ms=60000 \
    --bootstrap-server localhost:9092
```

However, this alone isn't enough to create a new segment. Similar to `segment.ms`, we first need to produce a new message, as only then will there be a check of whether a new segment needs to be created. So, we'll need to use our `kafka-console-producer.sh` again and produce another message for our partition:

```
$ kafka-console-producer.sh \
    --topic products.prices-compaction \
```

```
    --property parse.key=true \
    --property key.separator=: \
    --bootstrap-server localhost:9092
>energy drink:6
```

Let's take another look at our partition now. Unsurprisingly, a new segment has been created:

```
$ ls -1 ~/kafka/data/kafka1/products.prices-compaction-0
00000000000000000000.index
00000000000000000000.log
00000000000000000000.timeindex
00000000000000000004.index
00000000000000000004.log
00000000000000000004.snapshot
00000000000000000004.timeindex
leader-epoch-checkpoint
partition.metadata
```

NOTE If we were fast, we would see additional files with a `.deleted` extension.

10.4.2 How the log cleaner works

Those who were particularly quick at this point might have even witnessed the log cleaner in action live and observed the original segment with the starting offset 0 being first marked for deletion (the `.delete` file extension was added) and then deleted. This is due to the functionality of the log cleaner. It initially looks into our uncompacted segments and saves the latest offset for each key. Then, it starts reading from the oldest message onward. The log cleaner ignores keys or messages that are outdated, meaning keys for which there is a newer entry, and writes the cleaned messages into a new segment. Once this new segment is full, the old ones are replaced by the new clean segment. This explains why the log cleaner requires only the additional storage space of one segment at most. Now, let's take a look at the content of our log:

```
$ kafka-dump-log.sh \
    --print-data-log \
    --files ~/kafka/data/kafka1/products.prices-compaction-0/0[……]000.log
Dumping ~/kafka/data/kafka1/products.prices-compaction-0/0[……]000.log
Log starting offset: 0
[…]
| offset: 3 CreateTime: 1738488316569 keysize: 12 valuesize: 1 sequence: 0
headerKeys: [] key: energy drink payload: 3
```

Our older messages were indeed deleted, but message `3` is still there. This is because the newest segment isn't compacted and thus entirely excluded. Let's produce two more messages in quick succession to examine this in more detail and then wait briefly for the log cleaner to run:

```
$ kafka-console-producer.sh \
    --topic products.prices-compaction \
    --property parse.key=true \
    --property key.separator=: \
    --bootstrap-server localhost:9092
>energy drink:5
>energy drink:6
```

When we check our partition, unsurprisingly, we'll find that our log has rotated again:

```
$ ls -1 ~/kafka/data/kafka1/products.prices-compaction-0
00000000000000000000.index
00000000000000000000.log
00000000000000000000.timeindex
00000000000000000005.index
00000000000000000005.log
00000000000000000005.snapshot
00000000000000000005.timeindex
leader-epoch-checkpoint
partition.metadata
```

Let's quickly use `kafka-dump-log.sh` to inspect the contents of our two segments:

```
$ kafka-dump-log.sh \
    --print-data-log \
    --files ~/kafka/data/kafka1/products.prices-compaction-0/0[……]005.log
Dumping ~/kafka/data/kafka1/products.prices-compaction-0/0[……]005.log
Log starting offset: 5
[…]
| offset: 5 CreateTime: 1738489003023 keysize: 12 valuesize: 1 sequence: 0
headerKeys: [] key: energy drink payload: 5
[…]
| offset: 6 CreateTime: 1738489005780 keysize: 12 valuesize: 1 sequence: 0
headerKeys: [] key: energy drink payload: 6
```

Our current segment contains the two messages we just produced. We can tell it wasn't compacted, because if so, we would only see the latest message there. Consequently, our newest segment can contain any number of messages for a key:

```
$ kafka-dump-log.sh \
     --print-data-log \
     --files ~/kafka/data/kafka1/products.prices-compaction-0/0[……]000.log
Dumping ~/kafka/data/kafka1/products.prices-compaction-0/0[……]000.log
Log starting offset: 0
[…]
| offset: 4 CreateTime: 1738488639215 keysize: 12 valuesize: 1 sequence: 0
headerKeys: [] key: energy drink payload: 6
```

Our compacted segment now contains the previously produced Message `6` and no other message for this key. Even if we had additional segments, in all compacted segments, we would find at most one message per key.

Another important aspect of log compaction that we haven't explicitly mentioned yet is that, while new segments are created during log compaction, the offsets and the order of our messages remain unchanged. This is logical, as during log compaction, we essentially only delete outdated entries and otherwise merge segments. However, this property is crucial for consistency reasons!

At this point, we inevitably need to ask ourselves what happens when a consumer tries to read nonexistent offsets or messages. Even in our small example, this is already the case because our first segment still has an offset of 0 as the start offset despite deleted messages. Kafka also has a simple solution for this. If an offset doesn't exist, it simply jumps to the next available message. So, if a consumer requests the message with Offset 0, this request is equivalent to requesting the message with Offset 1.

10.4.3 Tombstones

Log compaction brings another feature with it. We can not only overwrite outdated data but also even selectively delete data. For this, we just need to create a *tombstone* by generating a message with a `null` payload. Log compaction then deletes the outdated messages associated with that key as usual. The special thing here compared to normal log compaction is that the tombstone message itself will be deleted after some time, and thus we won't find any messages for that specific key in our log anymore.

The time after which a tombstone is deleted by Kafka or the log cleaner is one day by default and can be changed using the parameters `delete.retention.ms` for a topic or `log.cleaner.delete.retention.ms` for our entire cluster. The reason tombstones aren't immediately deleted is to give consumers the opportunity to process the deletion accordingly in their own processes. Kafka deletes tombstones because, otherwise, our cluster would have many segments consisting only of tombstones from different keys that are no longer used, unnecessarily inflating our log.

Summary

- Kafka offers two main approaches for message cleanup: log retention and log compaction.
- Log retention deletes messages based on age, while log compaction removes outdated data based on keys.
- Log compaction ensures that only the latest message for each key is retained, which is suitable for scenarios such as updating customer data.
- Log retention is simpler to implement, clearing messages based on age and supporting various use cases such as data privacy compliance and changelog data management.
- Both methods have tradeoffs: log retention is easier but less precise, while log compaction requires more effort but ensures data accuracy.
- Log retention can be configured based on partition size or time, offering flexibility in managing storage space.

- Log compaction parameters determine the threshold for segment cleaning and ensure that only outdated data is removed.
- Log compaction maintains message order and offsets, preserving data consistency.
- If a consumer tries to fetch messages that don't exist, the brokers simply return the next messages.
- Kafka's log cleaner periodically checks for outdated data and optimizes log storage by removing unnecessary segments.
- Regular segment rotation ensures efficient log management, preventing log bloat and optimizing storage usage.
- Kafka's flexibility in message cleanup allows for tailored data management strategies based on specific use cases.
- Tombstone messages facilitate selective deletion in log compaction, ensuring efficient data cleanup.
- Tombstones are eventually removed to prevent log inflation, with configurable retention periods.

Part 4

Kafka in enterprise use

In part 4, we examine how Apache Kafka can be integrated into enterprise environments, focusing on both foundational and advanced topics. This section covers practical aspects of using Kafka within larger systems, such as connecting external systems with Kafka Connect, performing stream processing with Kafka Streams, implementing governance strategies, and ensuring high-performance deployment using Kafka's reference architecture. We also dive into critical areas such as monitoring, alerting, disaster recovery, and Kafka's integration within modern, real-time, event-driven architectures. By understanding how Kafka fits into enterprise workflows and addressing these advanced considerations, organizations can harness Kafka's full potential to handle data integration, processing, and management at scale, with an emphasis on performance, reliability, and resilience.

In chapter 11, we explore Kafka Connect, a powerful tool for integrating external systems with Kafka through connectors for databases, filesystems, and other data sources. Chapter 12 introduces stream processing with Kafka Streams, focusing on real-time data transformation, analysis, and SQL-like queries to enable more dynamic data architectures. Chapter 13 delves into governance strategies, covering schema management, security practices, and resource allocation to ensure Kafka's reliability and security within an enterprise environment. In chapter 14, we provide a comprehensive reference architecture, outlining Kafka's deployment models, hardware requirements, and recommended tools for managing a Kafka cluster effectively.

In chapter 15, we examine Kafka monitoring and alerting, focusing on strategies to ensure Kafka's performance and reliability by monitoring key metrics across brokers, clients, and frameworks such as Kafka Streams and Kafka Connect.

Chapter 16 addresses disaster management strategies, emphasizing fault tolerance, replication, and recovery mechanisms to mitigate risks associated with data loss and system failures. Chapter 17 compares Kafka to other technologies, exploring its role in modern architectures and how it compares to tools such as REST APIs and relational databases. Finally, chapter 18 explores Kafka's role in modern enterprise architectures, specifically its integration within data mesh and event-driven systems, while highlighting common pitfalls and best practices for successful Kafka implementation.

11 Integrating external systems with Kafka Connect

This chapter covers

- Integrating Kafka with data sources and sinks
- Configuring connectors and workers for optimal data flow
- Exploring the REST API for managing Kafka Connect
- Creating and modifying connectors
- Using Java Database Connectivity Source and Debezium connectors

In most cases, we don't introduce Kafka independently of other systems in our company, but rather we want to connect systems such as databases and messaging systems to Kafka. Most of these use cases are quite similar. We want to transfer data from predefined database tables to specific topics or write data from certain topics into a file. Of course, we always have the option to manually write our own producers and consumers to move data to or from Kafka.

However, this is very time-consuming and often leads to error-prone systems that are also difficult to scale. Even with simple requirements, such as transferring data from one system to another, there are many special cases to consider. An alternative is to use Kafka Connect.

11.1 What is Kafka Connect?

Kafka Connect is a framework for writing data from third-party systems to Kafka and also for transferring data from Kafka to other systems, as shown in figure 11.1. It distinguishes between *source connectors*, which are basically producers to Kafka that first read data from an external system, and *sink connectors*, which act as consumers from Kafka that write data to external systems. As a framework, it's part of Apache Kafka and, like Kafka itself, is available as open source software under the Apache 2.0 license.

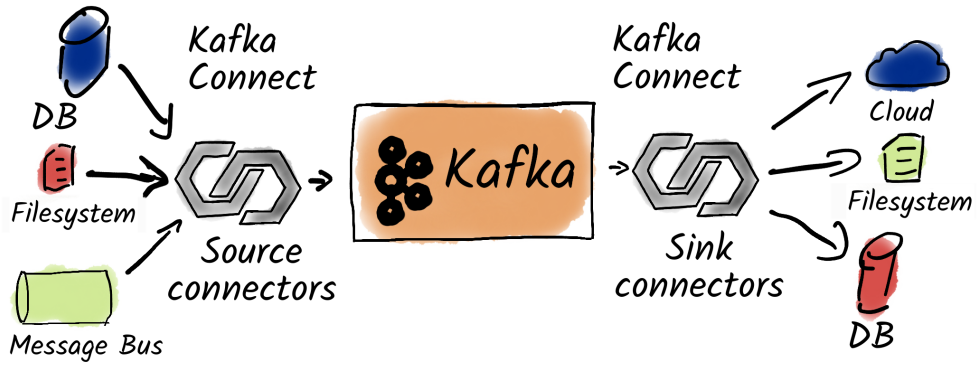


Figure 11.1 Kafka Connect integrates third-party systems with Kafka.

The goal of Kafka Connect is to provide a standardized tool for moving data. Like Kafka, Kafka Connect is scalable, fault-tolerant, and places a strong emphasis on correctness and performance. There are connectors available for a wide variety of systems, including databases such as PostgreSQL, SQLite, or MySQL; object storage systems in different cloud environments such as Amazon S3 or Azure Blob Storage; messaging protocols such as Message Queuing Telemetry Transport (MQTT) or Java Message Service (JMS); and data warehouse systems such as Snowflake or Amazon Redshift. As of the time of writing, the Confluent Hub alone lists around 200 connectors. Additionally, there are numerous connectors from other sources.

The core idea of Kafka Connect is to enable seamless integration with external data sources and sinks, particularly those that can be represented as partitioned streams. This means that data is divided into different streams, allowing us to scale and handle large volumes of data efficiently. Each stream contains related data that should remain

intact and not be subdivided further. In Kafka, topics are partitioned to achieve this, ensuring that data is appropriately distributed across multiple consumers for processing, as illustrated in figure 11.2.

In database systems, we can view individual tables as data streams that can't be easily split without losing guarantees such as order and other important properties, as shown in figure 11.3.

First, in a filesystem, these data streams could, for example, correspond to individual files. Second, for our partitioned streams, we need a way to track which data we've already read and which we haven't read. Because we expect these data streams to be very long, we can't simply remember which messages we've seen. Instead, as in Kafka, we work with offsets, positions within the data stream, to determine up to which point we've already read the data. In databases, we might use sequential IDs or timestamps to track when a row was last modified. In files, we could track the last line or byte position that we've read.

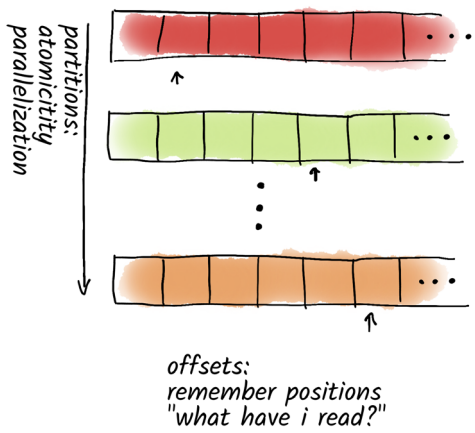


Figure 11.2 A partitioned stream uses partitions for parallelization and offsets to remember the last read message.

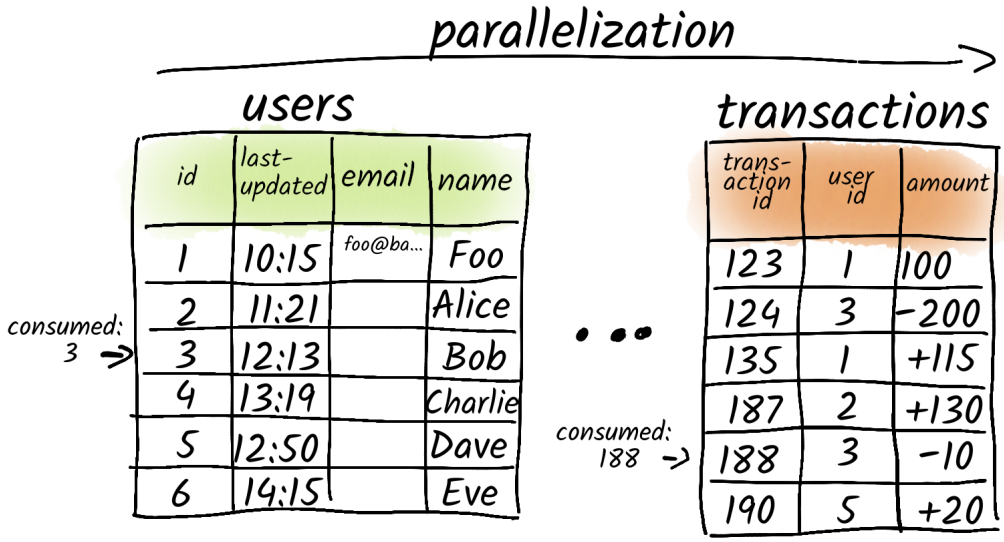


Figure 11.3 Partitioned streams in databases. We can view tables as independent data streams and use, for example, sequential id columns to track which rows have been already seen by Kafka Connect.

11.2 Kafka Connect cluster: Distributed Mode

To fully benefit from Kafka Connect, we typically run it in what's called *Distributed Mode*. In this mode, Kafka Connect stores all configuration data and internal offsets in Kafka topics. We then start as many Kafka Connect instances as needed. All Kafka Connect instances with the same `group.id` form a Kafka Connect cluster, where they coordinate and distribute the load among themselves using the same Kafka Rebalance Protocol that we discussed in chapter 9, section 9.4.

In development environments and certain specific cases such as when requiring access to local files, it may make sense to forgo these advantages and instead run Kafka Connect in what's known as *Standalone Mode*. In this mode, Kafka Connect doesn't store internal data such as offsets in Kafka, nor does it retain any configurations. We simply run Kafka Connect as a program on our local machine, and when we're done, we can stop it. This mode is ideal for testing Kafka Connect on development machines but doesn't offer scalability or the ability to store Kafka Connect offsets in Kafka.

11.2.1 Configuring a Kafka Connect cluster

First, we configure the Kafka Connect worker for Distributed Mode. A *worker* in Kafka Connect is a single process that runs as part of a Kafka Connect cluster, responsible for managing connectors and their tasks. In this context, a *task* is a unit of work, such as reading data from a source partition or writing it to a destination.

Each task runs independently and can be distributed across multiple workers for parallel processing. Kafka Connect workers coordinate with each other to ensure data is processed efficiently and in a fault-tolerant, scalable manner. To begin the configuration, we create a file named `worker.properties` with the following content:

```
bootstrap.servers=localhost:9092
group.id=connect
config.storage.topic=connect-config
offset.storage.topic=connect-offset
status.storage.topic=connect-status
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.storage.StringConverter
plugin.path=<PATH-TO-YOUR-USER-DIRECTORY>/kafka/libs/
```

NOTE We need to replace the placeholder in our `plugin.path` property.

We'll explain those parameters in detail in section 11.4, but for now, it's sufficient to just start Kafka Connect with the following command:

```
$ connect-distributed.sh worker.properties
```

For configuration, Kafka Connect provides a REST API that listens by default on port 8083. If the topics don't already exist, Kafka Connect will create them for us. Note that the recommended replication factor for the topics should be three. If we have fewer

than three brokers, Kafka Connect can't start. In test environments, we can reduce the replication factor by adjusting the `config.storage.replication.factor` setting (and similarly for the other topics).

11.2.2 Creating a connector

Now, while Kafka Connect is running, it's not doing anything yet. We first need to configure a connector. We create and manage connectors through a REST API. For example, let's say we have a file named `customers.txt` where we store the names of our customers. We want to import this data into Kafka. To accomplish this, we can use the `FileStreamSource` connector that comes with Kafka. We recommend not using this connector in production, as it's very minimalistic and has poor error handling. We prefer the `FilePulse` connector for cases like this. You can find this open source connector on GitHub: https://mng.bz/MDAQ. Nevertheless, let's create a file named `source-connector.json` for this purpose:

```
{
  "name": "customers-source",
  "config": {
    "connector.class": "FileStreamSource",
    "tasks.max": "1",
    "file": "/tmp/customers.txt",
    "topic": "customers"
  }
}
```

Here, we configure a connector named `customers-source` that will use the `FileStreamSource` class. Because we're only reading from one file, we can't scale this connector, so we set `tasks.max` to `1`. We want to write data from the file `/tmp/customers.txt` to the topic `customers`.

> **TIP** In production or real-world environments, we recommend explicitly specifying the converter classes in connector configurations rather than relying on the default configurations of the worker.

Using the following `curl` command, we send this configuration file via a `POST` request to Kafka Connect:

```
$ curl -X POST -H "Content-Type: application/json" \
    --data @source-connector.json \
    http://localhost:8083/connectors
```

11.2.3 Testing the connector

We can query all connectors in our Kafka Connect cluster at `http://localhost:8083/connectors`:

```
$ curl http://localhost:8083/connectors
["customers-source"]
```

To check the current status of our connector, we use the following command:

```
$ curl http://localhost:8083/connectors/customers-source/status | jq
{
  "name": "customers-source",
  "connector": {
    "state": "RUNNING",
    "worker_id": "127.0.0.1:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "RUNNING",
      "worker_id": "127.0.0.1:8083"
    }
  ],
  "type": "source"
}
```

TIP We use `jq` to prettify our output.

Here, we see that the connector is in the `RUNNING` state and is operating on worker `127.0.0.1:8083`. We also see all tasks. Because we defined only one task, it runs on the same worker as the connector itself. However, if we take a closer look at the Kafka Connect log output, we find that not everything is running smoothly, as indicated by the `/status` endpoint:

```
WARN [customers-source|task-0] Couldn't find file /tmp/customers.txt for
FileStreamSourceTask, sleeping to wait for it to be created
(org.apache.kafka.connect.file.FileStreamSourceTask:119)
```

So, let's write some data to the file `/tmp/customers.txt` in another terminal window:

```
$ echo "Linus Torvalds" >> /tmp/customers.txt
$ echo "Steve Jobs" >> /tmp/customers.txt
```

Now, we can start a consumer in parallel to observe what Kafka Connect is writing to the topic `customers`:

```
$ kafka-console-consumer.sh \
    --topic customers \
    --from-beginning \
    --bootstrap-server localhost:9092
```

If everything is working correctly, we should see our two customers:

```
Linus Torvalds
Steve Jobs
```

If not, we can check the `/status` endpoint or the Kafka Connect logs for error messages to get an idea of what went wrong. As we continue to write more data to the file,

those entries will also appear in the consumer. If we restart Kafka Connect, the connector won't start reading the data from the beginning of the file again; instead, it remembers the position in the file and only reads new entries.

> **WARNING** Because the connector remembers the last line it has read, changes to existing data in the file won't result in new messages in Kafka. This connector detects only appended data.

11.3 Scalability and fault tolerance of Kafka Connect

How does Kafka Connect enable us to scale our connectors and increase our fault tolerance? First, Note that this isn't easily achievable with the simple `FileStreamSource` connector, as it only reads files from the local disk, and those files are unlikely to be found on other Kafka Connect workers.

> **WARNING** We advise against using the FileStreamSource connector in production, as it functions more like a proof of concept than a fully developed, mature connector.

A Kafka Connect cluster consists of one or more workers running in Distributed Mode with the same `group.id`, as shown in figure 11.4. These workers coordinate with each other, similarly to consumers in a consumer group, using the Kafka Rebalancing Protocol. If a worker crashes, or we add a new worker, the remaining workers redistribute the tasks among themselves. In a Kafka Connect cluster, we can run multiple connectors simultaneously. For example, we can read data from an external database while simultaneously writing other data to a third-party system.

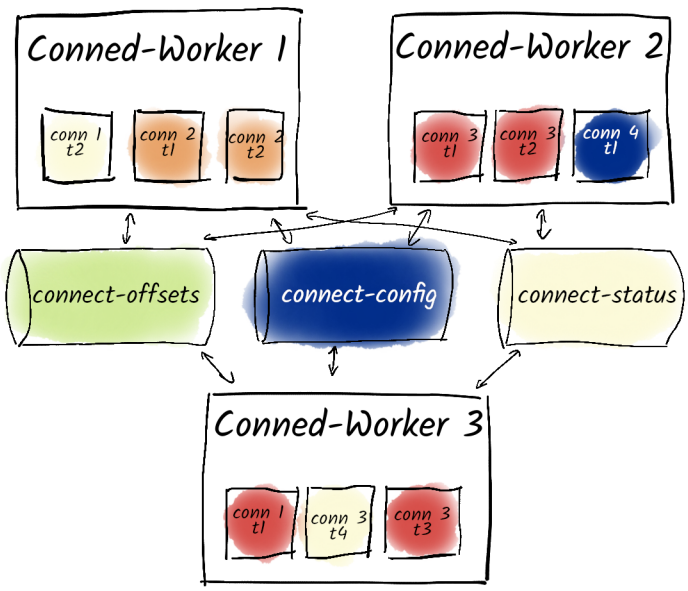


Figure 11.4 A Kafka Connect cluster consists of one or multiple workers. Tasks are distributed across workers, and Kafka Connect uses three topics to store its internal state.

If we want to scale our Kafka Connect cluster, we simply add more workers with the same `group.id`. The connectors themselves divide the work into different tasks that run independently. For instance, if we want to import data from 12 database tables, and we have `tasks.max` set to `4`, each task would be responsible for three tables. Kafka Connect distributes these tasks across the different workers, allowing for an even load distribution and enabling us to move larger volumes of data quickly to and from Kafka.

> **TIP** There's already a large number of different connectors available, and we highly recommend using them. If there's no connector for a specific use case, but the data structures can be effectively represented as partitioned streams, we generally recommend writing your own Kafka Connect connectors rather than attempting to recreate this functionality using producers and consumers.

11.4 Worker configuration

In our practical example for Distributed Mode, we've already encountered a minimalist configuration for Kafka Connect workers. In this section, we'll delve deeper into the options available for customizing a Kafka Connect cluster. Note that there are two kinds of configurations in Kafka Connect. First, the worker configuration configures the worker that Kafka Connect runs on and is valid for all connectors of that worker. Second, the connect configuration is provided via the REST API and configures the actual connectors that run on the workers. The Kafka Connect configuration is distributed across the workers using the internal configuration topic. Let's take a look at the configuration file `worker.properties` that we created earlier:

```
bootstrap.servers=localhost:9092
group.id=connect
config.storage.topic=connect-config
offset.storage.topic=connect-offset
status.storage.topic=connect-status
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.storage.StringConverter
plugin.path=<PATH-TO-YOUR-USER-DIRECTORY>/kafka/libs/
```

The `bootstrap.servers` property, as we know from previous chapters, defines the address of our Kafka cluster. A very important parameter is `group.id`. This parameter can be a bit misleading, as it doesn't relate to consumer groups in the traditional sense. Instead, `group.id` provides a name or unique identification for the cluster (`connect.id` would have probably been a better name). All Kafka Connect workers using the same `group.id` belong to the same Kafka Connect cluster.

The various storage topics are used to manage the Kafka Connect cluster. The configuration topic (`config.storage.topic`) stores the configurations of individual connectors, while the status topic (`status.storage.topic`) is used for monitoring the connectors and contains information about the individual tasks. The offset topic (`offset.storage.topic`) stores information about the offsets of the source connectors, such as the ID of a database entry.

The converter classes used for encoding and decoding messages are determined by `key.converter` and `value.converter`. In addition to the `StringConverter`, there are classes available for Avro, Protobuf, JSON Schema, simple JSON, and byte arrays. The converters for Avro, Protobuf, and JSON Schema require a Schema Registry, which we'll discuss in detail later.

> **TIP** While we recommend specifying converters in the connector configuration, note that these parameters are still required in the worker configuration. In production or real-world environments, we recommend using either the `JsonConverter` or `AvroConverter` as the default value converter instead of the `StringConverter`.

Certain connectors such as the `JSONConverter` provide additional properties. The `JSONConverter`, for example, also provides a setting `key/value.converter.schemas.enable` to enable or disable the writing of the schema in each message. This leads to very verbose messages, and thus we recommend using a converter supporting the schema registry such as the `io.confluent.connect.avro.AvroConverter`. This converter provides additional settings such as the address of the schema registry `key/value.converter.schema.registry.url`.

The final parameter from our introductory example is `plugin.path`. This parameter is used by Kafka Connect to locate plugins for connectors, converters, or transformations. It also allows us to specify multiple paths separated by commas.

Now, let's look at some additional important parameters. Kafka Connect is capable of automatically creating topics if they don't exist. Depending on how we manage our Kafka Connect cluster or our Kafka cluster in general, this behavior may not be desirable. We might want complete control over our Kafka topics and to manage them as Infrastructure as Code (IaC). We can control the automatic creation of topics with the `topic.creation.enable` parameter, which is enabled by default.

In chapter 5, we discussed exactly-once guarantees. Kafka Connect supports this for source connectors (producers to Kafka), but the support must be activated by setting `exactly.once.source.support` to `enabled`, as it's disabled by default. If you want to change this behavior for an existing Kafka Connect cluster, an intermediate step is necessary, in which you first set the parameter to `preparing` and then to `enabled`.

The consumer behavior of Kafka Connect can also be influenced by various parameters. With `session.timeout.ms`, we can specify how long a worker in the cluster is considered inactive. A lower value may reduce potential downtime, but it can also lead to unnecessary rebalancing of consumer groups, which takes time.

The `request.timeout.ms` parameter sets the maximum wait time for a response. If this time is exceeded, Kafka Connect assumes that something went wrong with the request and retries it. A lower timeout helps respond to errors more quickly. However, if network quality is poor or the load on our brokers is temporarily high, data transfer may take several seconds, especially if a lot of data needs to be transmitted in a single request. Setting `request.timeout.ms` too low can lead to unnecessary request retries

and may even result in requests never being marked as successful, causing the connector to fail to process data.

Similar effects can arise from the `offset.flush.interval.ms` and `offset.flush.timeout.ms` parameters. These settings allow us to configure how often Kafka Connect flushes messages and how much time it has for this operation.

> **TIP** If we have many small messages, flushing may take longer, so it's advisable to flush more frequently or increase the timeout. Otherwise, we risk creating an endless loop, which can be identified by the following log message from our Kafka Connect cluster: "Failed to flush, timed out while waiting for producer to flush outstanding x messages." We can also set general configurations for the producers and consumers of our workers, such as `fetch.max.bytes` or `fetch.max.wait.ms`.

A connector configuration also allows us to override the general parameters of our Kafka Connect cluster, which can be particularly useful for the parameters mentioned previously. We define which parameters can be overridden with `connector.client.config.override.policy`. Then, in the specific connector configuration, we can adjust parameters using the prefixes `producer.override` and `consumer.override`, for example, `consumer.override.fetch.max.bytes`.

There's also the option to use *config providers*, which allow us to use variables in individual parameters. This can be a significant advantage from a security perspective. In `config.providers`, we define a list of aliases for our config providers, and in `config.providers.<name>.class`, we reference the corresponding class, which can be customized with `config.providers.<name>.param.<param>`. This allows us to use the same class multiple times with different settings under different names.

We'll explore connector-specific configuration options further later in this chapter. The parameters specifically related to authentication and authorization will be addressed in chapter 13.

Finally, let's take a look at the `listeners` parameter. This allows us to configure which port, host, and protocol the REST API listens to. If nothing is specified, Kafka Connect uses port `8083` over `HTTP`.

11.5 The Kafka Connect REST API

After learning about the configuration options for our Kafka Connect cluster, we'll now look at how to manage the cluster and connectors. Kafka Connect provides a REST API for this purpose, which we briefly encountered at the beginning of this chapter.

It's crucial to mention that Kafka Connect's REST API doesn't provide any authentication and authorization out of the box. It's possible to configure basic authentication, but then everybody with access to the cluster can do all operations. We recommend that no humans have access to this API in production. Instead, a continuous integration/continuous deployment (CI/CD) pipeline should be used to manage connectors.

11.5.1 Status of a Kafka Connect cluster

At the root URL, we find only information about the Kafka version used by our Kafka Connect cluster and workers, along with the corresponding *Git commit ID* and the ID of the connected Kafka cluster:

```
$ curl http://localhost:8083 | jq
{
  "version": "3.9.0",
  "commit": "a60e31147e6b01ee",
  "kafka_cluster_id": "y9uk3gYxSu6sGKfGRDgjGQ"
}
```

Notice that the information is packaged in a *JSON object*. The REST API of Kafka Connect is completely JSON-based, meaning that Kafka Connect expects a JSON object in requests and responds with a JSON object as well. The cluster ID refers solely to the Kafka cluster used for managing our Kafka Connect cluster. Typically, this is also the cluster to which we produce data or from which we consume data.

We can check whether Kafka Connect has successfully loaded all of our desired plugins at the path `/connector-plugins`:

```
$ curl http://localhost:8083/connector-plugins | jq
[
  {
    "class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "type": "sink",
    "version": "3.9.0"
  },
  {
    "class": "org.apache.kafka.connect.file.FileStreamSourceConnector",
    "type": "source",
    "version": "3.9.0"
  },
 …
]
```

Here we see, for example, the `FileStream` plugins needed for our connectors. If a required plugin for a connector is missing, we'll also receive an appropriate error message when creating the connector, followed by a list of the installed plugins. We can obtain the list of currently running connectors at the path `/connectors`:

```
$ curl http://localhost:8083/connectors | jq
[
  "customers-source"
]
```

If we want to take a closer look at a connector, we can do so under `/connectors/<connector>`:

```
$ curl http://localhost:8083/connectors/customers-source | jq
{
  "name": "customers-source",
  "config": {
    "connector.class": "FileStreamSource",
    "file": "/tmp/customers.txt",
    "tasks.max": "1",
    "name": "customers-source",
    "topic": "customers"
  },
  "tasks": [
    {
      "connector": "customers-source",
      "task": 0
    }
  ],
  "type": "source"
}
```

In response, we receive information about the type (`source`) and configuration (`config`) of the connector. Additionally, the tasks are listed. We can also obtain this information for all connectors at once by calling `/connectors?expand=info`.

If we want only the current configuration, we can access it at `/connectors/<name>/config`. Detailed information about the current status of individual tasks and the workers on which the tasks are running can be found at `/connectors/<connector>/status`:

```
$ curl http://localhost:8083/connectors/customers-source/status | jq
{
  "name": "customers-source",
  "connector": {
    "state": "RUNNING",
    "worker_id": "127.0.0.1:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "RUNNING",
      "worker_id": "127.0.0.1:8083"
    }
  ],
  "type": "source"
}
```

The workers are identified here by the address at which they are accessible. We also receive information about the state of each task (`running`, `paused`, `failed`), including any error descriptions. Similar to the connector information, we can view the status of all connectors simultaneously at `/connectors?expand=status`. Comprehensive information about a connector's tasks can be found at `/connectors/<connector>/tasks`, and the status of an individual task at `/connectors/<connector>/tasks/<id>/status`.

Another useful feature of Kafka Connect is the ability to view the topics actually used by a connector at `/connectors/<connector>/topics`:

```
$ curl http://localhost:8083/connectors/customers-source/topics | jq
{
  "customers-source": {
    "topics": [
      "customers"
    ]
  }
}
```

We can also reset this status using `PUT /connectors/<connector>/topics/reset`. This can be useful after maintenance work, but it doesn't have any operational effect.

11.5.2 Creating, modifying, and deleting connectors

After learning in detail what connector information we can query via the REST API, we'll now look at how we can create, modify, and delete connectors, and thus configure the connection to our external data sources. First, let's delete our connector, `customers-source`:

```
$ curl -X DELETE http://localhost:8083/connectors/customers-source
```

If everything works, we receive an `HTTP 204` code in response. If the connector doesn't exist, we get an `HTTP 404` code. Now, let's recreate the connector. For this, we call `POST /connectors` and pass the connector as a JSON object, which consists only of the connector name and the connector-specific configuration:

```
$ cat source-connector.json
{
  "name": "customers-source",
  "config": {
    "connector.class": "FileStreamSource",
    "tasks.max": "1",
    "file": "/tmp/customers.txt",
    "topic": "customers"
  }
}
$ curl -X POST -H "Content-Type: application/json" \
    --data @source-connector.json \
    http://localhost:8083/connectors
```

If the connector is successfully created, we receive the same response that we would get if we called `GET /connectors/<connector>`. If we try to create a connector that already exists, we'll receive an appropriate error message.

We can also modify the configuration of a connector afterward. For example, if the source file for our connector should now be `/tmp/new_customers.txt` instead of `/tmp/customers.txt`, we can easily adjust the configuration by changing the `file` parameter. We create the file `source-connector-config.json` for this purpose:

```
{
    "connector.class": "FileStreamSource",
    "tasks.max": "1",
    "file": "/tmp/new_customers.txt",
    "topic": "customers"
}
```

Then we can adjust the configuration via `PUT` at `/connectors/<connector>/config`, and we receive the same response as when we initially created a connector:

```
$ curl -X PUT -H "Content-Type: application/json" \
    --data @source-connector-config.json \
    http://localhost:8083/connectors/customers-source/config
```

We can also pause connectors. This can be useful during maintenance of the third-party systems we connect to, as we risk our connector failing or unintentionally writing data to Kafka due to inconsistencies during maintenance:

```
$ curl -X PUT http://localhost:8083/connectors/customers-source/pause
$ curl http://localhost:8083/connectors/customers-source/status | jq
{
  "name": "customers-source",
  "connector": {
    "state": "PAUSED",
    "worker_id": "127.0.0.1:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "PAUSED ",
      "worker_id": "127.0.0.1:8083"
    }
  ],
  "type": "source"
}
```

We can then resume the connector:

```
$ curl -X PUT http://localhost:8083/connectors/customers-source/resume
```

A connector or its tasks can also fail. While Kafka Connect can compensate for most transient errors through retry mechanisms, sometimes this isn't enough. This can happen if there are prolonged network problems or problems with the data structure of the messages. In such cases, we need to restart our connector and its tasks via `POST` after resolving the problem:

```
$ curl -X POST http://localhost:8083/connectors/customers-source/\
    restart?includeTasks=true&onlyFailed=true
```

With the flags `includeTasks` and `onlyFailed`, we ensure that not only the connector itself but also all tasks that are in a failed state are restarted. Alternatively, we can also restart only individual tasks via `POST /connectors/<connector>/tasks/<taskId>/restart`.

11.6 Connector configuration

After learning how to configure a Kafka Connect cluster and manage connectors, this section will explore the configuration options we have for connectors. First, let's take another look at our `customers-source` connector:

```
{
  "name": "customers-source",
  "config": {
    "connector.class": "FileStreamSource",
    "tasks.max": "1",
    "file": "/tmp/customers.txt",
    "topic": "customers"
  }
}
```

The arguably most important and often the most difficult parameter to configure for a connector is its name (`name`). The name is used to identify the connector and is also part of the path in the Kafka Connect REST API. Because the name is the only metadata parameter of a connector, it's crucial that we use a reliable naming scheme, especially when our Kafka Connect cluster manages a large number of connectors.

11.6.1 General connector configuration

The connector class (`connector.class`) determines the type of connector. Connector classes often have specific parameters, such as the `file` parameter for the `FileStreamSource` connector. Other examples of connector classes include the following:

- `org.apache.kafka.connect.mirror.MirrorSourceConnector`
- `io.confluent.connect.jdbc.JdbcSourceConnector`
- `io.confluent.connect.jdbc.JdbcSinkConnector`
- `io.aiven.connect.jdbc.JdbcSourceConnector`
- `io.aiven.connect.jdbc.JdbcSinkConnector`
- `io.debezium.connector.postgresql.PostgresConnector`

Generally, there are two types of connectors. While *source connectors* are used to read data from external systems and produce it to Kafka, *sink connectors* are used to consume data from Kafka and write it to external systems. MirrorMaker is an exception, as it replicates data from one Kafka cluster to another.

The `tasks.max` parameter defines the maximum number of tasks into which a connector's workload can be split, allowing for greater parallelization. However, the actual number of tasks is often limited by the connector itself or the structure of the data

being processed, such as the number of partitions in the source topics. Setting `tasks.max` higher than this limit won't result in additional parallelization.

Sink connectors also have a `topics` parameter. This parameter defines the list of topics to consume. Alternatively, the `topic.regex` parameter allows us to use a regular expression to select our topics. This can be useful if we want to consume many topics, or if the topics to be consumed change over time. This requires a consistent naming scheme for the topics, highlighting the importance of naming conventions.

For sink connectors (consumed by Kafka), a consumer group with the prefix `connect` followed by the connector's name is created by default, though this value can be overridden with the `consumer.group.id` parameter. Note that we must allow overriding this parameter through the worker configuration.

In the connector configuration, we can define via `exactly.once.support` whether the Kafka Connect cluster must support exactly-once semantics (`required`) or if it's optional (`requested`). The latter is the default setting. If exactly-once support is required but not supported by the Kafka Connect cluster (`exactly.once.source.support` setting of the worker), the connector will fail, meaning it won't proceed with execution.

Kafka Connect also allows single message transformations of our messages before writing them from external systems to Kafka or from Kafka to external systems. We describe a few examples in section 11.7.

11.6.2 Error handling in Kafka Connect

Of course, like any other application, Kafka Connect can encounter problems during data processing. The most common causes of problems with Kafka Connect are temporary network problems or unprocessable data. To prevent the entire connector from stopping after every minor error, we have various options for configuring error handling based on the use case.

By default, Kafka Connect causes any connector to fail upon encountering an error, meaning the fault tolerance is zero. This behavior can be controlled via the `errors.tolerance` parameter. If we set this value to `all` instead of `none`, faulty records are simply skipped. When configuring this, it's recommended to also set `errors.log.enable` to `true`, as otherwise only nontolerated errors are logged. This way, we can still review the problematic operation afterward and possibly correct the error manually.

> **TIP** Depending on the use case, we recommend setting up alerts and notifications if fault tolerance is configured to skip problematic entries.

The `errors.log.include.messages` parameter allows us to specify whether the faulty record itself should be included in the error log along with the error message. By default, this isn't enabled because such a log entry may contain sensitive data, depending on the application, which shouldn't simply be written to an error log. If using a sink connector, the message can also be sent to a *dead-letter queue topic*, which must first be configured via `errors.deadletterqueue.topic.name`.

In addition to unprocessable records, a connector also fails if there are problems accessing third-party systems. In this case, retries can be configured with the parameters `errors.retry.timeout` and `errors.retry.delay.max.ms`. The `errors.retry.timeout` parameter sets the maximum time in milliseconds for retrying failed operations. By default, this parameter is set to `0`, meaning no retries are made. The `errors.retry.delay.max.ms` parameter determines the maximum time between retry attempts.

Specific connectors may also have their own parameters for retries, such as the JDBC sink and source connectors. For example, we can configure retries for database connections using the `connection.attempts` setting, which specifies the number of attempts, and `connection.backoff.ms`, which defines the interval between attempts. By default, the JDBC connector tries three times with 10-second intervals before failing to connect to the database.

> **TIP** We recommend giving JDBC connectors more time to restore database connections in the event of a failure, as the default 30-second configuration may not be sufficient during database maintenance or temporary network outages. Extending this time can prevent longer downtime, as once a connector fails, a (manual) restart of the connector and its tasks is required before it resumes operation.

Additionally, the JDBC sink connector allows us to retry operations in case of failure. The number of retries before the connector fails or skips the record, and the interval between attempts, can be configured with the `max.retries` and `retry.backoff.ms` parameters. This is useful, for instance, if corresponding database entries are temporarily locked. By default, the JDBC sink connector attempts to write an entry to the database 10 times, with 3-second intervals between attempts.

11.7 Single message transformations

Just moving data from A to B is often not enough, especially when the data needs to be transformed, manipulated, and routed to different destinations. While Kafka Connect isn't a fully fledged ETL (extract, transform, load) tool, it allows us to do some simpler transformations using the single message transformations (SMTs). SMTs work both for sink and source connectors and can be used to rename fields, to mask data, or even to move data from the value of a message to the key.

Figure 11.5 visualizes where the transformation happens. In a source connector, the actual connector is fetching data from the external data source and transforming it into Kafka Connect's internal data structure. Now, we can apply arbitrarily many transformations before the data is converted to the target data format as JSON and then written to the Kafka topic. For a sink connector, the process happens in the other direction: First, the data is converted from the topic data format to Kafka Connect's internal data structure. Next, Kafka Connect applies the SMTs, and finally the data is written to the sink system using the actual connector class.

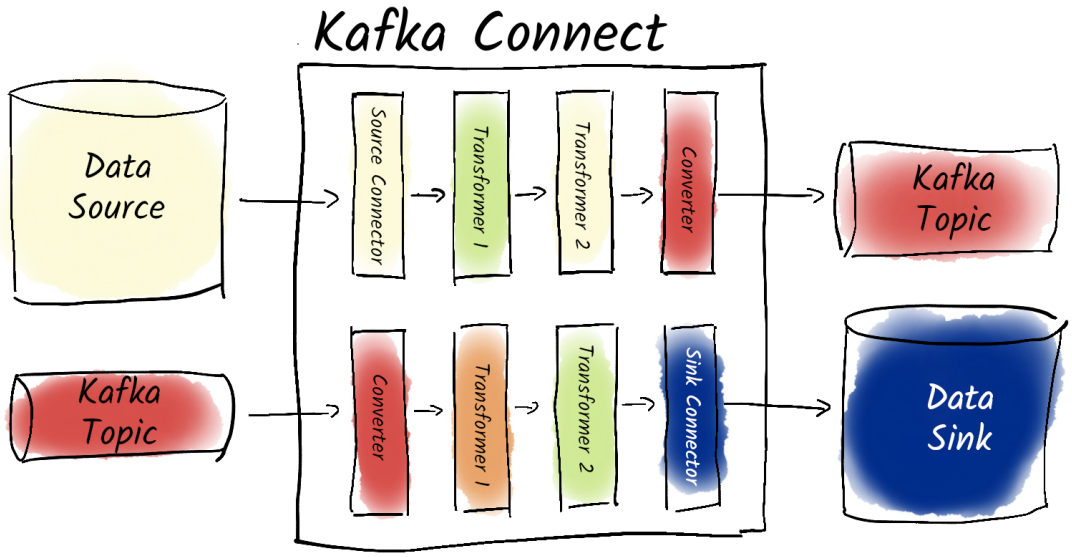


Figure 11.5 Single message transformations can be used for both source and sink connectors.

> **WARNING** We shouldn't overuse SMTs. This isn't a fully fledged ETL tool. If you want to do complex transformations, use Kafka Streams or another stream processing framework.

Let's see a few SMTs in action. To use an SMT, you need to configure the transformation steps in the connect configuration, that is, the JSON you're sending to Kafka Connect's REST API:

```
{
  "config": {
    […]
    "transforms": "mySMT1,mySMT2,…",
    "transforms.mySMT1.type": "java.class.name",
    "transforms.mySMT1.…": "some-config",
    "transforms.mySMT2.type": "java.class.name2",
    "transforms.mySMT2.…": "some-config2",
    […]
  }
}
```

First, we need to set the list of transformations that need to be executed (`transforms`). We can choose the names freely (`mySMT1`, `mySMT2`). Then, for every transformation, we need to set at least the Java class name (`type: java.class.name`) of the SMT and often additional configurations. The transformations will be executed in the order of the comma-separated list.

Let's look at a few different SMTs available. We can use the `ReplaceField` SMT to rename or to drop fields. For example, the following transformation drops the field `dropme` and renames the field `foo` to `bar`:

```
[…]
"transforms": "rename",
"transforms.rename.type":
    "org.apache.kafka.connect.transforms.ReplaceField$Value",
"transforms.rename.exclude": "dropme",
"transforms.rename.renames": "foo:bar",
[…]
```

As many source connectors only set the value of the message, but not the key, we can use the `ValueToKey` SMT to extract some data from the value to the key. The problem is that the key often becomes a JSON object containing only one key-value pair. For example if we have a `product_id` field and apply the `ValueToKey` SMT to use that field as the key, then the resulting key would be `{product_id: <ID of the product>}`. Using the `ExtractField` SMT, it's possible to extract a certain field to the key, meaning that our resulting key would be a simple string containing the ID of the product:

```
[…]
"transforms": "moveKey,extractKey",
"transforms.moveKey.type":
    "org.apache.kafka.connect.transforms.ValueToKey",
"transforms.moveKey.fields": "product_id",
"transforms.extractKey.type":
    "org.apache.kafka.connect.transforms.ExtractField$Key",
"transforms.extractKey.field": "product_id",
[…]
```

The `MaskField` SMT is also useful to set certain fields to null to avoid sending sensitive information to a system. Kafka Connect provides more SMTs out of the box, and some connectors (e.g., Debezium) add even more SMTs on top of that. See the Apache Kafka documentation for more SMTs: https://mng.bz/rK7B.

11.8 Kafka Connect example: JDBC Source Connector

After discussing general configuration options for connectors and specific ones for JDBC connectors in the previous section, we'll now deepen our Kafka Connect knowledge by practically applying it using a JDBC Source Connector as an example. For this, we'll once again refer to our customer example. Let's imagine that, until now, all customers have been stored in a database. In the future, however, we also want to make them available in Kafka. As visualized in figure 11.6, we approach this by using the JDBC source connector to move data from the `customers` table to the `sqlite_customers` topic.

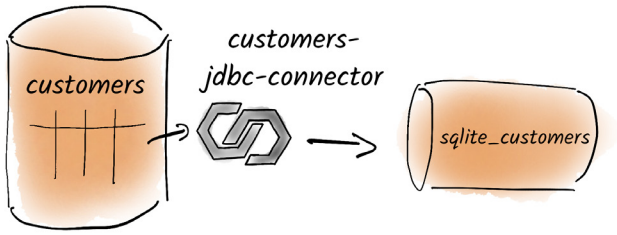


Figure 11.6 We want to move customer data from a relational database to Kafka.

11.8.1 Preparing the JDBC Source Connector

First, we need a simple SQLite database as a prerequisite. We can either download the SQLite binary directly from the website or install it using a package manager.

To begin, we initialize an SQLite database called `customers.db` and create a table within the database:

```
$ sqlite3 customers.db                          Inside SQLite
CREATE TABLE customers(
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  firstname VARCHAR,
  lastname VARCHAR,
  birthdate DATE,
  lastupdated TIMESTAMP DEFAULT current_timestamp
);
```

The `customers` table stores all data related to our customers. In addition to the primary key, we have columns for the customer's first name, last name, and birth date. Now, let's insert some records into our database so that we have data to produce into Kafka later:

```
INSERT INTO customers(id, firstname, lastname, birthdate) VALUES
  (1, 'Steve', 'Jobs', '1955-02-24'),
  (2, 'Bill', 'Gates', '1955-10-28');          Inside SQLite
```

Next, we need to install our JDBC connector, which we can simply download from the Confluent Hub. We must ensure that it's placed in the `plugin.path` defined in our `worker.properties`. Afterward, we restart our Kafka Connect cluster.

11.8.2 Configuring the JDBC Source Connector

Now, let's create the configuration for our connector and save it as `customers-jdbc-source-connector.json`:

```
{
  "name": "customers-jdbc-source-connector",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
```

```
    "connection.url": "jdbc:sqlite:/path/to/customers.db",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": false,
    "table.types": "TABLE",
    "table.whitelist": "customers",
    "topic.prefix": "sqlite_",
    "mode": "incrementing",
    "incrementing.column.name": "id",
    "batch.size": 1000
  }
}
```

In addition to familiar fields such as `name`, `connector.class`, or `batch.size`, there are some new parameters specific to our JDBC Source Connector. The `connection.url` serves several purposes. It specifies the connection type (`jdbc`), the database type (`sqlite`), and the database location (`/path/to/customers.db`), which in our case is simply the path to the SQLite file.

We also override the `value.converter` from `worker.properties` and use the `org.apache.kafka.connect.json.JsonConverter`. Furthermore, we disable schema usage because we want to work with simple JSON.

The `table.types` parameter is used to specify the type of database objects to query. Besides `TABLE`, another useful option is `VIEW`.

> **TIP** Although `TABLE` is the default value for `table.types`, it's often better to set such options explicitly to avoid unpleasant surprises after version updates.

The `table.whitelist` parameter allows us to specify which tables should be read. Alternatively, we could use `table.blacklist` to specify which tables shouldn't be read. In our example, we want to produce only the `customers` table to Kafka.

The name of the Kafka topic into which we produce is composed of the value of `topic.prefix` and the name of the database table. In our case, the prefix `sqlite_` and the database table `customers` form the Kafka topic `sqlite_customers`.

This parameter serves two important functions. First, it allows us to easily assign topics to a connector, and second, it prevents us from accidentally producing into an existing topic that happens to have the same name as the database table. This safeguard is particularly useful when using `table.blacklist` to exclude certain tables, as it helps prevent mistakes when we're not entirely sure which tables are being processed. The remaining parameters, `mode` and `incrementing.column.name`, will be discussed in more detail shortly.

11.8.3 Testing the JDBC Source Connector

Before creating the connector, we can open another window to start `kafka-console-consumer.sh` and consume from the target topic `sqlite_customers` of our connector:

```
$ kafka-console-consumer.sh --bootstrap-server localhost:9092 \
    --topic sqlite_customers --from-beginning
[..]sqlite_customers=UNKNOWN_TOPIC_OR_PARTITION[..]
```

As expected, the topic doesn't exist yet but is automatically created by default when we start consuming. Let's create the connector now:

```
$ curl -X POST -H "Content-Type: application/json" \
    --data @customers-jdbc-source-connector.json \
    http://localhost:8083/connectors
```

Once the connector is successfully configured, we'll see our `kafka-console-consumer.sh` begin consuming messages:

```
{"id":1,"firstname":"Steve","lastname":"Jobs", birthdate:"1955-02-24"}
{"id":2,"firstname":"Bill","lastname":"Gates", birthdate:"1955-10-28"}
```

As we can see, Kafka Connect automatically converted our database entries into JSON, with the column names serving as object keys. Now, let's see what happens when we insert a new row into our database table (with a purposely wrong birth date):

```
INSERT INTO customers(id, firstname, lastname, brithdate) VALUES
  (3, "Linus", "Torvalds", "1869-12-28");
```

Inside SQLite

As we can observe in our `kafka-console-consumer.sh`, the new database entry appears in our Kafka topic after a few seconds:

```
{"id":3,"firstname":"Linus","lastname":"Torvalds", birthdate:"1868-12-28"}
```

But what happens if we update an existing database entry? The code is shown here:

```
UPDATE customers SET birthdate="1969-12-28" where id=3;....
```

Inside SQLite

This time, we didn't consume any new messages due to the mode we selected for the connector (`mode=incrementing`) and the choice of the incrementing column (`incrementing.column.name=ID`). With this configuration, the connector only produces a new message to Kafka when there's a new ID, meaning a completely new entry in the database table.

Kafka Connect operates by querying the database table for new records based on the previously stored offset. For instance, if the last processed record had an ID of 3, the connector would execute a query like this:

```
SELECT * FROM customers WHERE id > 3;
```

Alternatively, we could have started the connector using `mode=timestamp` and set configuration `timestamp.column.name=lastupdated`. In this case, the connector would have picked up the change and written a message to Kafka with the new, correct timestamp. A combination of `incrementing` and `timestamp` mode is also possible

(`timestamp+incrementing`). It's important to ensure that the corresponding columns increase monotonically.

> **WARNING** The ID must be strictly incrementing because Kafka uses it as an increasing offset. If, for any reason, a new entry is assigned a smaller ID (e.g., if a record was deleted and the ID is reused), the connector won't recognize it as a new entry and, therefore, won't produce it to Kafka. In addition, the connector won't recognize if a record is deleted as this also doesn't produce a new ID.

However, this mode also has its limitations, as not every table has a separate timestamp column that we can use. For instance, if we have a database table just containing the addresses of our customers and they move to a new city, we wouldn't see this change in Kafka.

More problematic is the fact that even if we have a timestamp column, we can't guarantee that we'll produce all data to Kafka. For example, if two changes occur in the table at the same time (based on the entry in the timestamp column), Kafka Connect may read the first entry, but if another change is made to the table, Kafka Connect may miss the second one, assuming it has already read all changes up to that timestamp. Depending on how precise our timestamp is, the likelihood of this happening increases or decreases, but it's never zero.

> **TIP** The only way to guarantee capturing all changes with a JDBC Source Connector is to use the bulk mode. In this mode, the entire table is reloaded into Kafka every time. This is highly inefficient, and we strongly advise against it. This problem can be solved by using a change data capture (CDC) connector such as Debezium that we describe in the next section.

11.9 Kafka Connect example: Change data capture connector

In this section, we'll take a practical look at how to configure a *change data capture* (CDC) *connector*, specifically the Debezium connector for PostgreSQL. CDC is a concept where all changes to a data source, such as a database table, are monitored instead of the actual data itself. The technical implementation options depend on the specific third-party system and range from event triggers to logs. For relational databases, the transaction log can be monitored, into which all changes are recorded.

11.9.1 Preparing the Debezium connector for PostgreSQL

Before we deal with the connector itself, we need to prepare our PostgreSQL database. Download links and installation instructions for PostgreSQL servers can be found for all common operating systems on the official PostgreSQL website. After installing our PostgreSQL server, we must set the `wal_level` in our `postgresql.conf` to `logical`; otherwise, we won't be able to monitor the necessary changes:

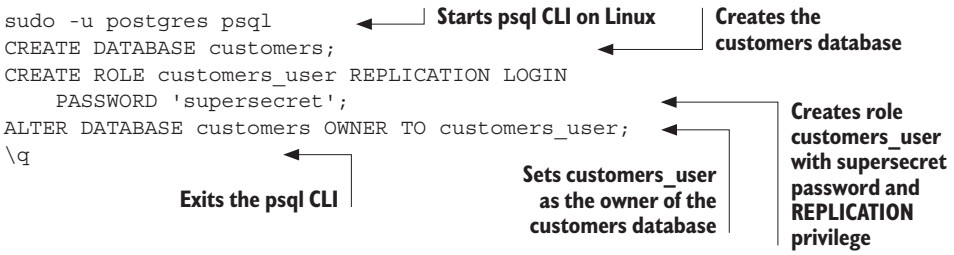
```
# Linux file path: /etc/postgresql/<version>/main/postgresql.conf
# Windows file path: C://Programs/PostgreSQL/<version>/data/postgresql.conf
wal_level = logical
```

Next, we need to restart our PostgreSQL server or service:

```
$ sudo service postgresql restart
```

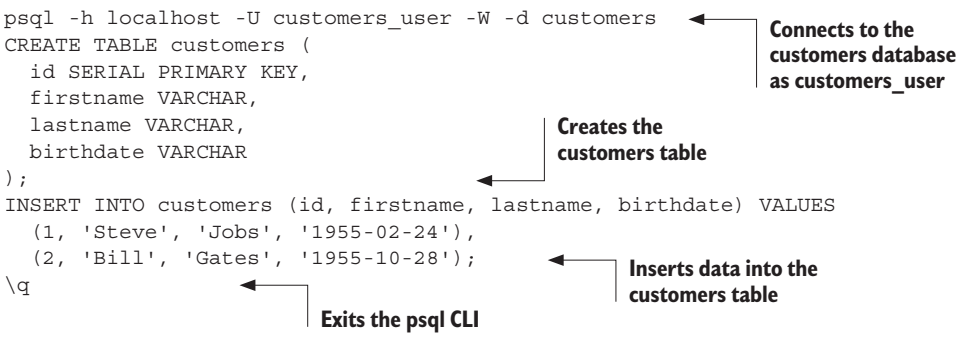
> **NOTE** Public cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud also offer support for these features.

Once our PostgreSQL server is setup and running, we create a database named `customer` and a database user named `customers_user`, to whom we'll assign ownership of the `customer` database. First, we start the interactive `psql` command-line interface (CLI):



It's important to grant the user `customers_user` the `REPLICATION` privilege, so they will have the necessary rights for our CDC connector.

Next, we connect to the PostgreSQL server again, this time with the newly created user `customers_user`. We also connect directly to the `customers` database and create a table named `customers`, filling it with appropriate data:



However, the most important part is still missing: the connector itself. Instructions for downloading and setting it up can be found in the Debezium documentation under the Deployment section. The process is similar to the installation of the JDBC connector.

11.9.2 Configuring the Debezium connector for PostgreSQL

Once we're ready, we create our connector configuration `customers_debezium_connector.json`:

```
{
  "name": "customers_debezium_connector",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "localhost",
    "database.port": "5432",
    "database.user": "customers_user",
    "database.password": "supersecret",
    "database.dbname" : "customers",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": false,
    "plugin.name": "pgoutput",
    "publication.autocreate.mode": "filtered",
    "topic.prefix": "debezium",
    "table.include.list": "public.customers"
  }
}
```

Most parameters are already familiar to us from our JDBC Source Connector, at least under similar names. All configurations for the database connection are found under the various `database` parameters. We determine the Kafka topic prefix just like with the JDBC Source Connector. However, the parameter name for the table whitelist in the Debezium connector for PostgreSQL is called `table.include.list`. The settings for `plugin.name` and `publication.autocreate.mode` are completely new. Both parameters determine how exactly the data is extracted from our PostgreSQL database.

11.9.3 Testing the Debezium connector for PostgreSQL

Let's first create our connector:

```
$ curl -X POST -H "Content-Type: application/json" \
    --data @customers_debezium_connector.json \
    http://localhost:8083/connectors
```

Next, we start our `kafka-console-consumer.sh`.

```
$ kafka-console-consumer.sh --bootstrap-server localhost:9092 \
    --topic debezium.public.customers --from-beginning
{
    "before":null,
    "after":{"id":1,"firstname":"Steve","lastname":"Jobs","birthdate":
        "1955-02-24"},
    "source":{"version":"2.1.2.Final","connector":"postgresql","name":
    "debezium","ts_ms":1738493972052,"snapshot":"first","db":"customers",
    "sequence":"[null,\"24287336\"]","schema":"public","table":"customers",
    "txId":739,"lsn":24287336,"xmin":null},
    "op":"r",
    "ts_ms":1738493972146,
    "transaction":null
}
```

> **NOTE** The resulting topic name `debezium.public.customers` is the combination of the `topic.prefix` property (`debezium`), a dot, and the table name in `table.include.list` (`public.customers`) in our connector configuration.

Oops! The output contains much more data than expected. Debezium adds additional information to each message that can help us understand where the data is coming from, so let's look at the different fields. The most important field is the `after` field, which contains the actual data in the table row that was changed in the operation. The `before` field can contain the previous data of the row if we configure Debezium and the database to include it. This is often not necessary. `source` explains where the data came from and provides additional technical information. Finally, `op` is the operation that triggered this message: `c` stands for create, `u` for an update, and `d` for delete. But here, the `op` is `r`, which means that this message wasn't triggered by any database operation, but was imported during the initial snapshot of the table by Debezium. Finally, `ts_ms` describes the timestamp of the operation, and `transaction` is set to `true` if the operation was inside a database transaction. Now let's add another entry:

```
INSERT INTO customers (id, firstname, lastname, birthdate) VALUES
  (3, 'Linus', 'Torvalds', '1869-12-28');
```

Not surprisingly, this entry appears in our `kafka-console-consumer.sh`, just like with the JDBC Source Connector:

```
{
    "before":null,
    "after":{"id":3,"firstname":"Linus","lastname":"Torvalds",
        "birthdate":"1869-12-28"},
    "source":{…},
    "op":"c",
    "ts_ms":1738494107344,
    "transaction":null
}
```

As you can see, the `op` is now `c`, meaning the message was triggered by the creation of the row. This time, we also need to correct our table entry:

```
UPDATE customers SET birthdate='1968-12-28' where id=3;
```

In contrast to the JDBC Source Connector, we see the change in Kafka with the Debezium connector for PostgreSQL. You can also see that the change was triggered by an update statement if you look at the `op=u`:

```
{
    "before":null,
    "after":{"id":3,"firstname":"Linus","lastname":"Torvalds",
        "birthdate":"1969-12-28"},
    "source":{…},
```

```
    "op":"u","ts_ms":1738494350114,
    "transaction":null
}
```

TIP In most cases, we probably don't want the entire change entry in Kafka. Using the `ExtractNewRecordState` SMT, Kafka Connect will write only the `after` value to Kafka.

Summary

Kafka Connect is a framework designed for scalable and reliable data integration between Kafka and other systems. It enables the movement of large amounts of data into and out of Kafka effortlessly using connectors.

- Connectors can be classified as source connectors (importing data into Kafka) or sink connectors (exporting data from Kafka).
- Each connector can be configured with various parameters to meet specific requirements.
- Kafka Connect uses a REST API for managing connectors and worker configurations, and it can also be configured using a properties file at startup.
- The REST API can be used to create, update, and delete connectors, as well as to check their status.
- Worker configurations play a vital role in determining how Kafka Connect operates, including aspects such as resource allocation, task distribution, and the overall resilience of the integration process.
- Error handling in Kafka Connect can be configured, including options for ignoring problematic entries via the `errors.tolerance` parameter.
- Logging errors is essential for troubleshooting, achieved through the `errors.log.enable` parameter.
- For sink connectors, data can be directed to a dead-letter queue for handling unprocessable records.
- Retry mechanisms can be configured for transient errors using parameters such as `errors.retry.timeout` and `errors.retry.delay.max.ms`.
- Single Message Transformations (SMTs) enable simple data modifications, including renaming fields, masking data, or moving values to keys during data transfer.
- SMTs work for both source and sink connectors but aren't suited for complex ETL tasks.
- SMTs are set up via JSON in the Kafka Connect API, specifying transformation order, Java class types, and configurations. Common SMTs include `ReplaceField`, `ValueToKey`, `ExtractField`, and `MaskField`.

- JDBC connectors have specific configurations for managing `database` connection retries and backoff strategies.
- The operational mode of connectors can be set to either incrementing, timestamp-based, or both, depending on change tracking requirements.
- Change data capture (CDC) can be implemented using connectors such as Debezium to monitor database changes.
- PostgreSQL requires specific configurations (e.g., setting `wal_level` to `logical`) for effective change data capture.
- Kafka Connect can automatically create Kafka topics based on database tables, ensuring seamless integration.
- The `topic.prefix` parameter organizes topics generated by connectors and prevents naming conflicts.
- Connector configurations can specify database connection details and table filtering using parameters such as `table.include.list`.
- The framework's flexibility allows for tailored solutions for various data integration scenarios, enhancing data flow management.

12 Stream processing

This chapter covers

- An overview of stream processing frameworks
- Partitioning and parallelization mechanisms in Kafka Streams
- Implementing SQL-like queries in stream processing
- Demonstrating use cases for Kafka Streams

We now know different ways to populate Kafka with data. We can use producers to send data directly to Kafka, which makes the most sense when we're near the data source. For example, our machines in the factory are equipped with Kafka producers to send measurement data and events directly to Kafka, or we use Kafka to collect log data from servers or website visits. On the other hand, if we want to collect data from databases, files, or cloud services with Kafka, it's worth taking a look at Kafka Connect.

Similarly, we're familiar with various methods to read data from Kafka and make it available to third-party systems. We often use Kafka consumers to display data

directly or trigger actions in third-party systems. Conversely, when we want to write data from Kafka to other systems, we advise our customers to consider Kafka Connect, as it's often a more suitable approach than implementing custom consumers.

With these tools, we have numerous ways to implement highly performant and useful systems. We can exchange data between different systems in near real-time or create modern integration pipelines. Originally, Kafka was used to provide massive data from various big data systems such as Hadoop to be batch-processed later.

However, if we already have the data in Kafka, and Kafka provides a platform to exchange data in near real-time between systems and also retain data for longer periods, can't we also perform data analyses in real time instead of batch processing?

Welcome to the world of stream processing! With *stream processing*, we can process continuous data streams, such as data in Kafka, in real time. This allows us to provide analyses instantly, which previously required waiting a day, and enables us to respond immediately to changes in our business operations.

Because we don't view this book as a Kafka development guide, we introduce the basic concepts of stream processing in this chapter, while avoiding implementation details and too many code examples, as there's already a lot of good literature on this topic.

12.1 Stream processing overview

In chapter 2, we introduced the use of consumers and producers with the example topic `products.prices.changelog`, which tracks changes to product prices on our online shopping platform. This topic records events such as price updates, discounts, promotions, and adjustments. In chapter 11, we worked with the `customers` topic, which contains basic customer information such as first name, last name, and birth date. However, in a more complex, real-world scenario, there would likely be additional topics related to customers, such as addresses, order history, and payment methods.

When we examine these two topics, we notice a difference in the type of data: in the `products.prices.changelog` topic, we're dealing with transactional data (price changes), while in the `customers` topic, we have basic master data. Kafka is suitable for handling both types of data; we just need to manage them appropriately.

In an online shopping platform, many challenges can arise, including discrepancies in product pricing that need to be addressed promptly. For example, a product's price might be incorrectly updated due to errors in upstream systems or human input, leading to either an unreasonably high price that deters customers or a low price that results in revenue loss. In this chapter, we'll use this scenario to introduce stream processing and show how such a problem can be detected and resolved in near real-time.

The platform must decide whether to correct or flag a product's price. This decision is time-sensitive and must strike a balance between accuracy and speed. Overcorrecting prices unnecessarily could lead to a loss of trust and sales, while delays in detecting incorrect prices might frustrate customers or result in financial harm. Through this

example, we'll demonstrate how stream processing enables timely and efficient handling of such dynamic and critical scenarios.

We use the data from the `products.prices.changelog` topic to determine whether a product's price should be corrected or flagged. Some events are independent of specific products. For example, if a system-wide pricing error is detected, we may need to pause certain promotions or make platform-wide adjustments. However, other situations depend on specific customer segments or product categories. For instance, a price adjustment might be normal for one customer group but be considered problematic for another based on their purchasing behavior or region. In a more complex setup, we'd also consider additional data from other customer-related topics, such as order history or geographic location.

12.1.1 Stream-processing libraries

Before we dive into how to implement such a system, let's first take a look at the different stream processing libraries available. As of the time of writing in winter 2024/2025, *Kafka Streams* is one of the most popular solutions. Kafka Streams stands out as it's a Kafka-native Java library that is both fault-tolerant and performant. It's also supported by popular Java frameworks such as Spring Boot. Kafka itself handles the coordination of different Kafka Streams instances, making Kafka Streams often easier to manage in production compared to other solutions.

Another increasingly popular option is *Apache Flink*. Like Kafka Streams, Flink is based on Java, but it also has a Python library. Flink offers a significantly broader range of features compared to Kafka Streams. However, one downside of Flink is that its operation is more complex, as Flink can't rely on Kafka for coordination and requires its own coordination layer.

One exciting advantage of Flink is *Flink SQL*, which allows us to implement stream-processing logic using SQL. This means we can often avoid writing extensive Java code, making it an appealing option for teams that prefer not to work with Java.

For Python teams, the *Faust Streaming* library is a good choice, and we've had positive experiences with it. For teams working with Go, there is *Goka*, although we haven't used it in practice yet.

> **TIP** We strongly advise against trying to implement stream processing on your own using consumers and producers. Instead, we recommend using one of the established stream-processing libraries.

12.1.2 Processing data

Figure 12.1 explores some approaches to processing data in Kafka. In the analytics world, the traditional method for handling large volumes of data is *batch processing*. For instance, at 4 a.m. every night, a cron job might run to analyze the previous day's transactions and generate reports. Many integration solutions also use batch processing to synchronize data between systems.

Batch processing isn't only used in conventional integration solutions; we can also use Kafka as a data source for batch systems, as these systems still offer certain advantages. Almost every large organization has been using such systems for years, if not decades, and they remain useful. In this scenario, Kafka is merely used as a data exchange layer.

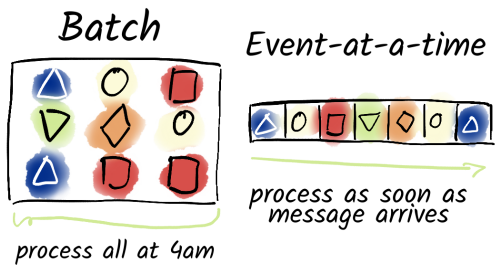


Figure 12.1 In batch processing, a large amount of data is processed at-once at specific times and the results are definite. When processing one event at a time, data is processed almost immediately, but it's difficult to provide definite results, as the system is eventually consistent.

Of course, this doesn't take full advantage of Kafka's real-time capabilities. However, in the batch world, the data is often assumed to be final, and batch processing can deliver a definitive result. This is useful for regularly recurring reports tied to specific deadlines, such as monthly revenue reports or inventory updates. Additionally, if batch systems are run in the cloud, cheaper spot instances can be used at night, helping reduce costs. The downside of batch processing is obvious: it's not real time. We must wait for results—sometimes for hours.

In contrast, stream processing follows an *event-at-a-time* approach, meaning that as soon as an event occurs, it's processed almost immediately. In the context of our online shopping platform, this might mean that when a price change happens, it's reflected right away. In addition, when a product goes out of stock, it's updated as soon as possible. Just like us, our customers are not inclined to wait for results; we all want everything to happen in real time. The world operates in real time, and businesses need to make decisions in real time, too—not hours later when it may be too late.

Stream processing, however, requires a shift in thinking. Instead of dealing with final results, we must always assume that what we're seeing is just an intermediate result that could change. Typically, each event is processed individually and in isolation. For example, if we receive the event `price drop detected for product X`, the immediate reactions might include adjusting the website display, triggering a promotional notification to customers, or sending alerts to the sales team to capitalize on the price change.

However, things get more complex when the response depends on other factors, such as customer segments or historical pricing data. What if the price change is only beneficial for loyalty program members? In such cases, we need to access data from other sources, such as the `customers` or `products.prices.changelog` topics. Stream-processing libraries assist us with this. Still, the challenge remains: we're always dealing with intermediate states, so we can only produce intermediate results.

WARNING Stream processing operations are eventually consistent. The clear advantage, however, is that we no longer have to wait hours for results.

12.2 Stream processors

Stream processors are components in stream processing frameworks, such as Kafka Streams or Apache Flink, that apply specific logic or transformations to messages in a data stream. These processors are the building blocks for analyzing, filtering, or transforming data in real time as it flows through the system. Regardless of which library we use, most stream processing frameworks operate based on similar principles: receiving input data, processing it through defined logic, and producing transformed or aggregated output.

The question is how best to express the operations that stream processors perform. With traditional producers and consumers, we typically rely on imperative programming, where we explicitly define step-by-step instructions—essentially, an *if this, then that* approach.

In *functional programming*, as well as in Kafka Streams, Apache Flink, and other stream processors, we define reusable functions and connect them to execute transformations on the data in a processing pipeline. This approach simplifies the processing of continuous data streams by abstracting much of the lower-level logic.

The third approach is *declarative programming*, such as SQL, where we specify what data we want, and the database or stream processing library executes that query for us. We'll discuss the capabilities of SQL in stream processing in more detail in section 12.3. Here, we focus on the functional programming approach and the core components known as processors in Kafka Streams.

12.2.1 Processor types

First, we need to read data from a Kafka topic. In Kafka Streams, we use the *source processor* `stream(topicName)`, which is provided by the `StreamsBuilder`. For instance, to read data from the topic `products.prices.changelog`, we could use the following code:

```
builder.stream("products.prices.changelog")
```

In Flink, the process is somewhat more complex because it supports not only Kafka but also other systems, necessitating the initial configuration of data sources and sinks.

One of the simplest processors in the stream processing world is `filter((Key, Value) -> Bool)`. The `filter()` function takes a predicate, which is a function that is executed on each message in the stream, and returns a stream containing only the messages for which the predicate returns `True`. For example, if we have a stream of price changes and are only interested in messages where the new price is greater than $100, we could use the following filter:

```
filter((key, value) -> value.price > 100)
```

However, `filter()` doesn't allow us to modify data. For that, we use the `map((Key, Value) -> (newKey, newValue))` processor. The `map()` function takes a function that

accepts a message with a key and value as input and returns a new message with a new key and value. If we want to change only the value, we should use the processor `mapValues((Key, Value) -> newValue)`, which doesn't alter the key. We use `mapValues()` whenever we need to convert, mask, or transform data. For instance, if we want to convert prices from dollars to cents, we could use the following function:

```
mapValues((key, value) -> value.price * 100)
```

The output stream will then contain the price in cents.

In contrast, we often use `map()` to process raw data. For example, if we receive messages without a key but with a value `{id: "Apple", price: 1.50}`, we can use the following function to format the data:

```
map((key, value) -> (value.id, value.price))
```

The result will be a message with the key `Apple` and the value `1.50`. Figure 12.2 provides a graphical representation of these processors.

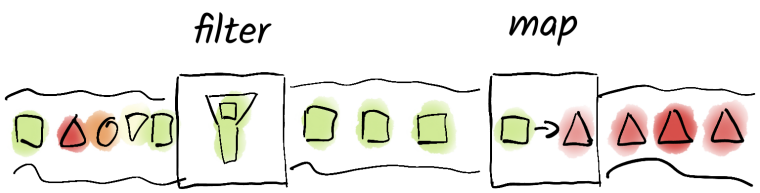


Figure 12.2 The filter and map processors

WARNING Kafka Streams provides a variant for many functions. For example, for the `map()` function, a variant is `mapValues()`. Because `mapValues()` doesn't change the key, it's always preferred. Changing the key often leads to performance losses because a repartitioning must occur afterward.

To write data back to Kafka, the frameworks offer various functions. In Kafka Streams, this is simplest: we call the processor `to(topicName)`. To write the processed prices into the prices topic, we can use the following code:

```
map(…).to("prices")
```

In Flink, as with reading from a topic, we first need to create a Flink Kafka connector and connect it to a topic.

Processors don't have to take exactly one data stream as input and output. With `merge(otherStream)`, we can combine two data streams into a single stream. The elements from both streams are inserted into the new stream, similar to the `UNION` operator in SQL. `merge()` is useful when we want to combine uniform data from different

sources. For example, we could have a central Kafka cluster where the sales data topics from different branches are merged, and we can use the following function to combine a stream of sales data from the northern and southern branches for further processing:

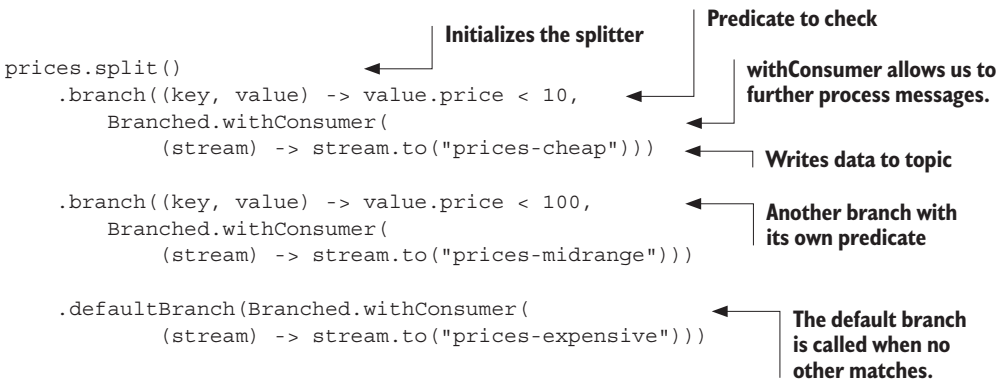
```
salesDataNorth.merge(salesDataSouth)
```

Stream processing frameworks, of course, provide not only a `merge()` processor for combining data but also support *joins*, similar to database operations of the same name, which we'll discuss later in the section 12.4.3.

If we want to split our data into multiple topics instead of merging it, we can use the `split()` processor. While it's possible to work with multiple `filter()` calls, `split()` can be an attractive alternative for certain use cases. For example, if we want to process sales data for products differently based on their prices, we can proceed as follows:

```
cheap = prices.filter((key, value) -> value.price < 10)
midrange = prices.filter((key, value) -> value.price >= 10 &&
    value.price < 100)
expensive = prices.filter((key, value) -> value.price >= 100)
```

Alternatively, we could implement this using `split()`. To process the stream of a branch further, we can use the `Branched.withConsumer()` method. There, we can process the branched stream even more, for example, by writing it out to another topic:



In `split()`, the different branches are evaluated from top to bottom, and the branch where the predicate is `True` is executed. This approach helps avoid errors and duplicates that may occur with `filter()`, for example, by forgetting one of the conditions for `midrange`. See figure 12.3 for a graphical representation.

All the processors we've considered so far examine each message individually and then perform actions based on the message. These processors are *stateless*, meaning they don't require internal storage or state to process messages. Often, we want to analyze messages in the context of previous messages. We'll discuss this in greater detail in section 12.4.2.

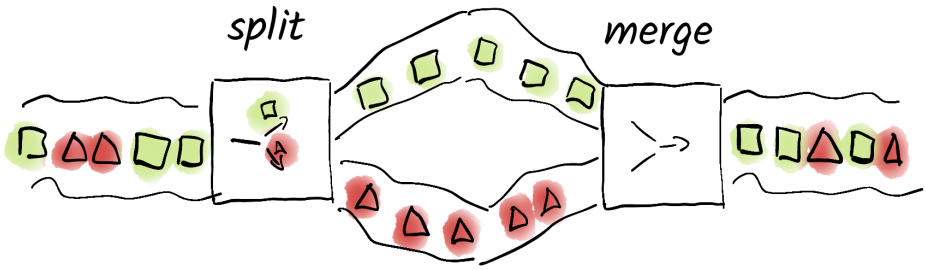


Figure 12.3 The split and merge processors

Let's now briefly look at how we can count the number of price change events for each product. Kafka stream processors operate on a partition basis, meaning that different instances of the stream's application are responsible for different partitions. This allows for high parallelism without instances interfering with one another.

However, this also means that it's not straightforward to determine, for example, the total number of messages in a Kafka topic. Instead, we count the occurrences per key. If we use the product ID as the key in the `products.prices.changelog` topic, we can easily determine the number of price changes, that is, the number of events, per product:

```
prices.groupByKey()
          .count()
```

First, we group the data in the topic by key (product) and then count the messages per key. The `count()` processor is *stateful*, as it must keep track of how many events have been read so far. Even if the state here is small, it still needs to be stored somewhere. We'll cover this in more detail later. Additionally, we must implement more complex aggregations, such as calculating a sum or an average, ourselves, but we'll also address this later. Nevertheless, you can find a graphical representation of these processors in figure 12.4

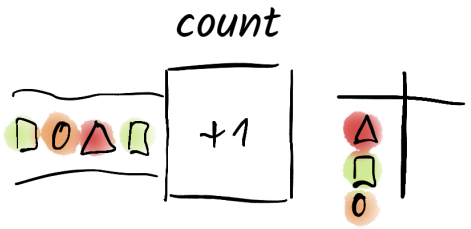


Figure 12.4 The count processor

12.2.2 Processor topologies

Now that we've learned about different processors, it's time to connect them to create more complex applications. Let's start with a simple example: we want to examine price changes that exceed a critical value (e.g., 10%) and write these into a topic called `prices.significantlyIncreased`. We can use the following code for this:

```
builder.stream("prices")
    .filter((key, value) -> value.price / value.oldPrice > 1.1)
    .to("prices.significantlyIncreased")
```

This type of program code is referred to as a *topology* in stream processing, as shown in figure 12.5. This graphical representation aids in debugging and enhances our understanding of the code without having to read it directly. Of course, this is just a small and simple example, making the graph trivial. However, it can become arbitrarily complex.

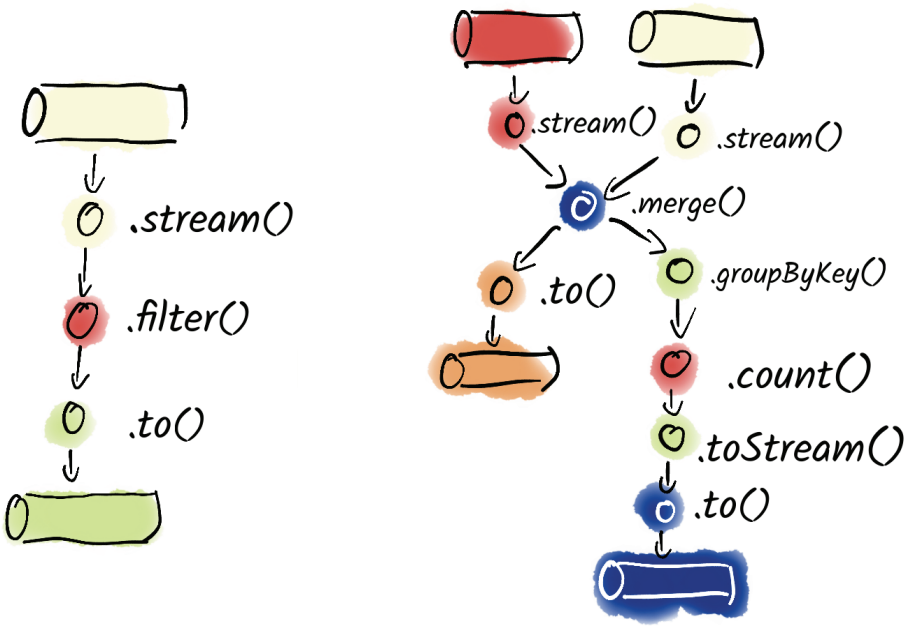


Figure 12.5 On the left, we have a simple topology that filters data from a topic and writes it back to another topic. On the right, we merge two topics and then write the result to a topic and in parallel count the number of messages per key. The result is then also written to a topic.

For instance, if we now want to write the sales data from the northern and southern branches into a central topic called `salesData.aggregated`, and then count the number of purchase events per customer to write into the topic `salesData.count`, we could use the following code:

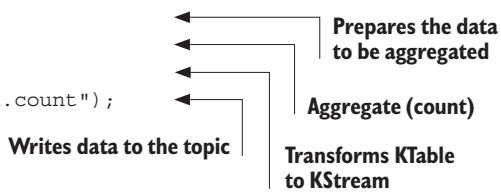
```
salesDataNorth = builder.stream("salesData.north");
salesDataSouth = builder.stream("salesData.south");

salesData_aggregated = salesDataNorth.merge(
    salesDataSouth);
salesData_aggregated.to("salesData.aggregated");
```

Fetches data from topics

Merges both topics

Writes data to the topic

```
salesData_aggregated.groupByKey()
                    .count()
                    .toStream()
                    .to("salesData.count");
```



First, we read the two topics using `builder.stream()`. Then, we merge the data with `merge()` to write it into the target topic using `to()`. After that, we group the data using `groupByKey()` and count the purchase events using `count()`. Because the result of a `count()` operation isn't a data stream but rather a table (more on this later), we must first convert it into a stream using `toStream()` before writing the data to the target topic with `to()`.

We can also represent this code graphically as shown earlier in figure 12.5 and see that the graph becomes more complex. We now have two input processors, and the data is processed in different ways: once by writing the merged data into a topic and once by performing the count.

12.3 Stream processing using SQL

In this book, we've frequently drawn comparisons between databases and Kafka. We won't make an exception in the context of stream processing, either. In relational databases, we have SQL as a powerful programming language that allows us to express even complex statements, which are then (hopefully) executed efficiently by the database. Often, SQL queries can express in a few lines what would take many pages of imperative or functional programming code.

Thus, it makes sense to transfer SQL to stream processing as well. However, just as databases can't always be directly compared to Kafka, there are also significant differences between database SQL and streaming SQL. In databases, we usually execute a SQL query when we need the result and then receive the result back more or less quickly. Of course, there are databases that allow query subscriptions, but these are the exceptions.

In Kafka, we face the challenge of storing events instead of states. This means that a streaming SQL query typically doesn't return a final answer, but rather produces a stream of data that can potentially be infinite. Therefore, a streaming SQL query usually doesn't deliver a single response but generates a new data stream of changes. Because most stream processing frameworks also support tables for storing states, it's often possible to write such queries that return a complete result at a specific point in time.

In this section, we don't intend to describe SQL or delve into the syntax of Flink SQL. Instead, we aim to provide an idea of what programs previously implemented in imperative (Java) code could look like in SQL. The syntax varies from framework to framework. Flink SQL strives to adhere to the SQL standard wherever possible, so it's compatible with many SQL-compliant libraries.

Before we start executing SQL queries, we need to create tables or streams, just like in a database. Unlike a database, we don't create empty tables, but rather connect them to Kafka topics. Now, let's create the corresponding counterparts in Flink SQL for our `products.prices.changelog` topic:

```
CREATE TABLE productprices (
  product_id INT,
  product_name STRING,
  price FLOAT,
  oldPrice FLOAT
) WITH (
  'connector' = 'kafka',
  'topic' = 'products.prices.changelog',
  'properties.bootstrap.servers' = 'localhost:9092',
  'format' = 'json',
  'key.format' = 'json',
  'key.fields' = 'product_id',
  'value.fields-include' = 'EXCEPT_KEY'
);
```

Because Flink SQL is SQL standard-compliant, it doesn't distinguish between streams and tables; therefore, we can create only tables. The fact that the product ID is the key isn't immediately apparent; we can only see this in the line `key.fields`. After that, we specify our fields as usual. Now we need to configure the source for this table. Because Flink supports more than just Kafka, we must indicate that the Kafka connector should be used. For this, the connector requires information about the bootstrap server and the topic. Finally, we need to specify that the format is JSON.

TIP Even though the SQL query says that a new table is to be created, the data in the table isn't stored on the Flink cluster. However, when somebody wants to read data from that table, the request is forwarded to the data source, that is, in this case Kafka.

Now we can execute our first queries. For example, if we want to see price changes for products where the price is greater than 100, the Flink SQL query would look like this:

```
SELECT *
FROM productprices
WHERE price > 100;
```

This query resembles a normal SQL query; however, instead of returning a single result and terminating, it continuously outputs new data. If we now want to count the number of price changes per product, we can use the following query:

```
SELECT product_id, COUNT(*) AS num_price_changes
FROM productprices
GROUP BY product_id;
```

This query also looks like a standard SQL query and would function similarly in traditional databases. At this point, we won't delve further into SQL; instead, we'll consider different use cases of stream processing in the next section, showcasing example SQL queries.

Typically, we don't manually execute SQL queries in production or use them in our program code. Instead, we can use the *headless mode* in Flink SQL, where we provide one or more SQL queries for continuous execution. The results of these queries can then be written back to Kafka, allowing consumers or Kafka Connect to view or forward the results. Because Flink supports other systems beyond Kafka, we can also write the results directly into another database system.

> **TIP** While it may seem straightforward to write streaming SQL queries and implement them, the performance of these queries is often not immediately apparent. Before executing queries in production, we recommend reviewing them and analyzing potential performance problems.

12.4 Stream states

So far, we've considered topics in Kafka as infinite data streams: there may be a start (the message with the smallest offset), but no end. These streams are ideal for representing continuously changing data, such as sensor measurements, bank transactions, or system updates. In these cases, we assume that new messages can arrive at any moment.

However, while streams are excellent for capturing ongoing changes, they can become cumbersome when we need to determine the current state of a system. For example, calculating a machine's most recent status or a customer's current account balance requires processing all messages from the start to the latest entry. This approach is computationally expensive and impractical for many applications.

In this section, we explore how the concept of stream states addresses this challenge. We start by examining how states are managed in streams and the relationship between streams and tables. Then, we delve into essential techniques such as aggregations and joins, which enable us to derive meaningful insights from data. Finally, we illustrate these concepts with an advanced example: a notification system. This example shows how stream processing solves complex problems efficiently.

12.4.1 Streams and tables

We've already pointed out several times in this book that this is where databases demonstrate their absolute strength: database tables are superb for storing and querying the current state of a system. And, it gets even better: tables and data streams are closely related. In chapter 11, we introduced *Debezium*, a solution for writing changes in databases to Kafka. This allows us to convert a static table into a dynamic data stream (in Kafka). Conversely, if we're observing a data stream of changes, we can capture the current (intermediate) state of our system from that data stream. You can see this interchangeability in figure 12.6.

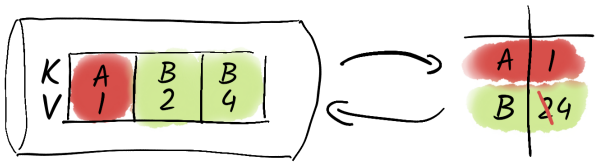


Figure 12.6 We can convert streams into tables, and the transformation works in the opposite direction as well.

This can also be applied in stream processing. Often, we need both: we want to be able to respond to changes in real time while also querying the current state of objects and systems. These two functionalities should be inherently supported within the stream processing environment so that we don't have to query external databases when we're interested in the current state of, say, our products.

Stream processing frameworks provide a distinction between data streams and tables. Kafka Streams refers to its streams as *KStreams* and Flink as *DataStreams*. Tables are referred to as *KTables* in Kafka Streams and *Tables* in Flink. Both data streams and tables are usually partitioned, meaning that each instance of a stream processing application only knows a part of the data. Furthermore, stream processing tables aren't tables in the traditional relational database sense; they only have two columns: the key and the value, making them basically a key-value store.

We can create both data streams and tables from a topic. For instance, in Kafka Streams, we create a KStream with `builder.stream("topic")` and a KTable with `builder.table("topic")`. For topics containing master data, it often makes sense to use topic compaction. When we create a table, Kafka Streams (or Flink) reads the topic from the beginning, updates the current intermediate state for each key in the table, and stores it. Most stream processing libraries use *RocksDB*, a very fast key-value store optimized for exactly such use cases.

Whenever we see tables in our stream processing application, we know that a state is being stored there, and this requires local storage. To prevent data loss after a system crash, Kafka Streams stores changes in a *changelog topic*. These topics are usually compacted. Flink also maintains these states in Kafka or other systems based on the configuration.

We've already encountered a useful application case for such tables: when we use the `count()` processor, the intermediate count states are stored in a table, and every time a new entry is added, the value in the table is updated, and the change is written to the changelog topic. The values in the tables can then be queried by other parts of the application, or we can use them to perform join operations or similar tasks.

The good news is we can execute most of the processors we've learned about on tables, as shown in figure 12.7. We can filter and map tables just like we do with data streams. To obtain a data stream from a table, we use the `toStream()` processor. In the opposite direction, there are more options: `toTable()` is the simplest processor that reads the data and fills a row in the table with the most current value for each key. However, we can also use `count()` or other aggregations. We'll cover more on that in the next section.

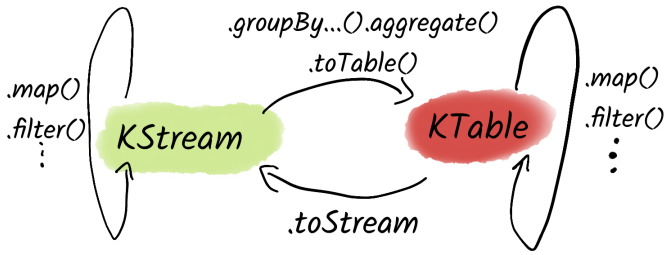


Figure 12.7 Processors such as `map`, `filter`, and `flatMap` work on both KStreams and KTables. We use aggregations to create KTables from KStreams and `toStream` to produce a KStream from a KTable.

12.4.2 Aggregations

To obtain the current state from a stream of data, we need to *aggregate* the data stream. For example, to obtain the current balance of a bank account, we can process all transactions, and for each bank account ID, we can sum up the transaction amounts. To develop an aggregation, we need two components: the initial state and an *adder*.

The initial state is needed each time we want to aggregate a message for which there is no previous state. We can view the initial state as the "empty" state for this specific aggregation. For `count()`, `sum()`, and so on, the initial state is the number 0. If we were multiplying values, the initial state would be 1. In mathematical terms, you can call it a neutral element because it doesn't influence the calculation. To get a better understanding of aggregations, we've visualized the count aggregation in figure 12.8.

When we have a previous state and read a new message, we then use the adder to derive a new state. In the case of `count()`, we ignore the content of the message and simply set the old state to `+1`.

For `sum()`, we add the value of the message to the old state to get the new state. Note that in stream processors, we typically aggregate based on keys, meaning we don't sum over a single overall state to get the total of all messages; instead, we sum based on keys, resulting in a distinct value for each key. These states are then stored in a table and can be further used from there.

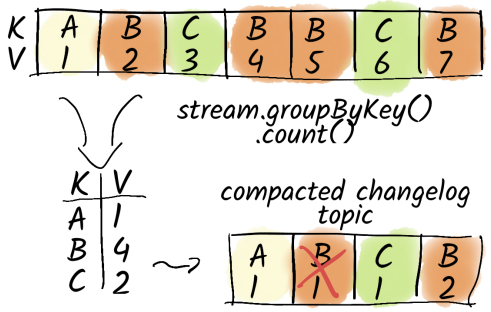


Figure 12.8 To count the number of messages per key, a state store with the current count per key is needed. The changes to the state store are stored in a compacted changelog topic.

To ensure that the values in the table survive a crash of the stream processor and that we don't have to start over with calculations, most frameworks use persistent storage to save these (intermediate) states. Kafka Streams uses the mentioned changelog topics, while Flink, as previously discussed, can use various technologies.

In table 12.1, we've gathered several additional aggregations with example implementations. Of course, creativity knows no bounds here. Aggregations are always useful when we want to analyze data in Kafka to recognize trends and patterns, for instance. They are also extensively used for billing and usage analyses. With the aggregation `average()`, it's often beneficial to run `mapValues()` over the state so that we have the results in the table rather than the internal states, including the intermediate sum and counter.

Table 12.1 Example aggregations and how they can be implemented in pseudocode

Aggregation	Data Type of Aggregate	Initial State	Adder	Example

`count()`	Number	`0`	`(agg, value) -> agg + 1`	`(K1, 2) -> K1: 1`
				`(K2, 1) -> K2: 1`
				`(K2, 2) -> K2: 2`

`sum()`	Number	`0`	`(agg, value) -> agg + value`	`(K1, 2) -> K1: 2`
				`(K2, 1) -> K2: 1`
				`(K2, 2) -> K2: 3`

`toTable()`	Data type of value	`Null`	`(agg, value) -> value`	`(K1, 2) -> K1: 2`
				`(K2, 1) -> K2: 1`
				`(K2, 2) -> K2: 2`

`average()`	Number	`(sum, count, avg)`	`(agg, value) -> (agg.sum + value, agg.count + 1, newSum/newCount)`	`(K1, 2) -> K1: (2,1,2)`
				`(K2, 1) -> K2: (1,1,1)`
				`(K2, 2) -> K2: (3,2,1.5)`

At this point, we need to address a peculiarity in Kafka Streams: we can't apply an aggregation over a KStream; instead, we must first group the stream. We typically do this with the function `kstream.groupByKey()`. If we want to group the stream by some new key other than the original key, we can use the function `kstream.groupBy((key, value) -> newKey)`.

This will internally change the key, create a new *repartitioning* topic, and then group by the key. In Flink, aggregations can be applied over the entire data stream or can be grouped by a value.

12.4.3 Streaming joins

Another very useful and often-used functionality of databases is the *join*. Here, we want to connect data from two or more tables to know, for example, which users bought which products. A simple implementation of a join is the creation of a cross table, where each entry in one table is linked with every entry from the other table. We can then run a filter on this cross table to extract the records that make sense to us.

But how can we use joins in stream processing? We can't apply cross tables over infinite data streams, as we don't know the entire data stream at any time; we always expect new entries.

The basic idea of a join in stream processing is that of a *lookup*: in stream processing, we usually process only one message at a time and need a state, for example, in the form of a table, where we can find the corresponding entry for our current message.

One of the most common use cases for joins in stream processing is data enrichment. For example, we have a data stream of sales data for our products and want to enrich this sales data with product master data so that in our monitoring, we can display not only the name of the product but also its manufacturer. The master data is stored in a compacted topic, and we change it relatively rarely. Therefore, it makes sense to consider the master data as a table. The key is the product ID, and the value is the current master data of the associated product. We see the sales data as a data stream keyed by the product ID. Each time we receive new sales data, we look up the corresponding master data for the product in the master data table using the ID and can then perform our enrichment operation. In code, we can express this as follows:

```
KStream salesData = builder.stream("salesData");
KTable masterData = builder.table("products");
ValueJoiner enricher = (salesData, masterData) -> {
    salesData.manufacturer = masterData.manufacturer;
    return salesData;
};
KStream salesData_enriched = salesData.join(
    masterData, enricher);
```

Left side of the join: KStream

Right side: KTable

Takes left and right value and returns a result

Executes the join

See figure 12.9 for a visual representation of this topology. First, we create our stream (`builder.stream()`) and the table (`builder.table()`). The enrichment function (`Joiner`) takes the key of the current message and its value, as well as the value found in the table for that key. We perform the join with `.join()`. The left side of the join is the `salesData` stream, the right side is the product `masterData` table, and we use the enrichment function as the joiner. The result of this join is again a stream.

But when is this join executed? Stream processors always process only one message at a time, meaning it's not possible (unless we use states) to hold messages and wait for subsequent messages. This means we have to approach it differently: when the join function receives a new message from the `sales data` stream, it looks for the corresponding entry in the `products` table and executes the enrichment function.

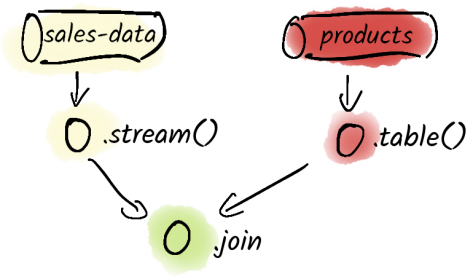


Figure 12.9 The topology for a join

However, if the join function is alerted that an entry in the `products` table has changed or a new one has been added, it can't look up the other entry in the `sales data` stream. This means that when the table changes, no join is executed. Only when there is new data in the stream will this join be executed.

This leads us to the question: Are there other types of joins besides stream-table joins? For example, we have two master data topics, `customers` and `loyalty_cards`, and we want to enrich the data of the customer card with the master data of the customers. Before creating the topics, we should consider the keys. To perform a join, the data streams or tables must be co-partitioned, meaning the number of partitions must be the same, and the keys must also be the same.

Let's assume that this is the case in our example; otherwise, we would need to change the key of one of the topics to be able to perform the join. Now we have two master data topics and want to connect the data. For this, we represent both data sets as tables and perform a (primary) key join. If there is a change in the left table, it's possible to look up this key in the right table and then perform the join operation. If there is a change in the right table, we can also execute a lookup here. This means such a table-table join is triggered by changes on both sides. We can describe this in code as follows:

```
KTable loyaltyCards = builder.table("loyalty_cards");
KTable customers = builder.table("customers");
ValueJoiner joiner = (loyaltyCard, customer) -> {
    loyaltyCard.customer = customer;
    return loyaltyCard;
};
KTable loyaltyCardWithCustomer = loyaltyCards.join(customers, joiner);
```

The left and right side are both tables.

ValueJoiner stays the same.

We start here again by initializing the input data (`builder.table()`). Then, we define how the join should be performed and finally execute the actual join. Here, we use an inner join because it doesn't make sense to have a loyalty card without being a customer.

Can we also join two streams together? We've stated that we need a way to look up a key on the other side of the join. However, this isn't possible with infinite data streams. This means a general stream-stream join isn't possible.

However, we can limit the infinity by constraining the joins to a time window. For example, we can keep the data for the past 5 minutes and find our values in the other stream there. This is useful if we want to correlate data over time. For example, we can join the price changes (`products.prices.changelog`) of our products with the sales data (`salesData`), assuming that the relevant data doesn't lie more than a minute apart.

Internally, the stream processor holds the data from both streams for a period of 1 minute, allowing us to obtain the relevant data and later filter or analyze it. As before, the topics must also be co-partitioned. It would be sensible to use the product ID as the key here:

```
KStream priceChanges = builder.stream(
    "products.prices.changelog");
KStream salesData = builder.stream("salesData");
JoinWindows joinWindow = JoinWindows.ofTimeDifferenceWithNoGrace(
    Duration.ofMinutes(1));
ValueJoiner joiner = (priceChange, saleData) ->
    new PriceSaleInfo(priceChange, saleData);
KStream correlated = priceChanges.join(salesData, joiner, joinWindow);
```

The left and right side are KStreams.

We need a time window now.

As usual, we start by reading the data from the topics. Then, we define a time window of 1 minute within which we want to correlate the data. (In the next section, we'll take a closer look at the different possible time windows.) The result is again a stream that reacts to changes in both streams. Unlike previous joins, we get a message in the output stream for every change, joining each message with every other in the other data stream with the same key and in the same time window.

In summary, it can be said that joins in stream processors are related to joins in databases but function differently, as we have to deal with infinite data streams here. Table 12.2 summarizes the most common types of joins and provides an overview of the data changes to which side of the join triggers the execution of the joiner function.

Table 12.2 Streaming join types

Left Side	Right Side	Changes to Which Side of the Join Triggers Execution of the Join
Table	Table	Both right and left
Data stream	Table	Data stream
Data stream	Data stream	Both right and left

12.4.4 Use case: Notifications

Let's take a look at how we can use aggregations and joins to perform actual work. Sometimes, it's necessary to send urgent notifications to users. For instance, we may have an app installed on multiple devices, and we want to send a notification to all devices.

On one side, we have a `notifications` topic where all notifications are written to. The key of this topic is the user ID. In a second topic, the `user-devices` topic, we find all devices associated with each user. It's common to have multiple messages per user, each indicating different devices.

To send a notification to all devices, we first want to aggregate all device information for a user and then join this information with the data from the `notifications` topic before sending it to the `device-notifications` topic. Figure 12.10 shows us a graphical representation of the topology.

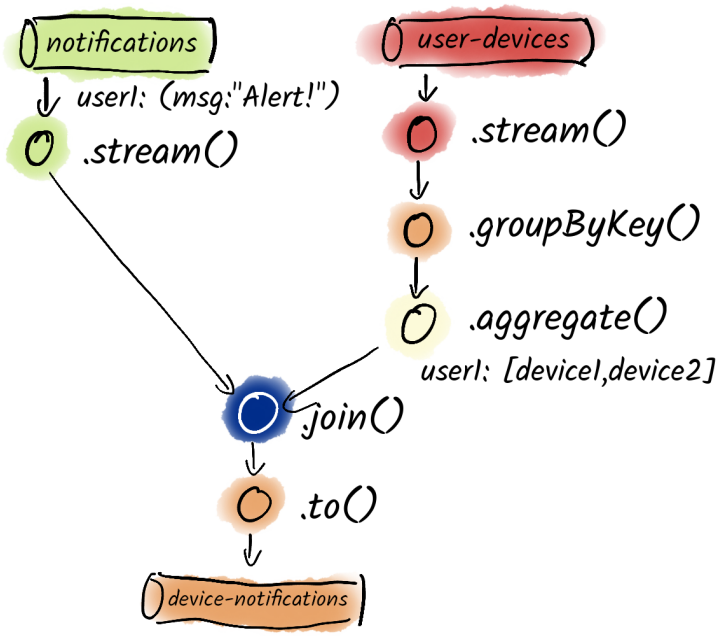


Figure 12.10
A streaming topology consisting of aggregation and join

In Kafka Streams, we can implement it as follows:

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, Notification> notifications =
builder.stream("notifications");
KStream<String, DeviceInfo> userDevicesStream = builder.stream(
    "user-devices");

Initializer initializer = List.of();

Aggregator adder = (userId, newDevice, aggDevices) -> {
        if (newDevice.isActive) {
            aggDevices.put(newDevice.deviceId, newDevice);
        } else {
            aggDevices.remove(newDevice.deviceId);
        }
        return aggDevices;
    };
KTable<String, List<DeviceInfo>> userDevicesTable = userDevicesStream
    .groupByKey()
    .aggregate(initializer, adder);

KStream<String, NotificationWithDeviceList> notificationsWithDevices =
    notifications.join(userDevicesTable,
        NotificationWithDeviceList::new);

deviceNotifications.toStream().to(
    "device-notifications");
```

Creates KStreams from topics

The initial element should be an empty list.

Adder: old state + current element = new state

Groups first . . .

then aggregates

Joins notifications with devices

Writes the result to the topic

Again, we start by building two KStreams from the Kafka topics: one KStream for the `notifications` and one for the `user-devices` topic. Next, we want to aggregate the data of the `user-devices` topic such that we have a list of devices for every user. As discussed previously, we need two functions for an aggregation.

First, the `initializer` is responsible for creating an initial state for keys that we haven't seen before. For our use case, it creates an empty list because we want to collect a list of devices. Then, the `adder` is responsible for taking the previous state and the currently read value and returns a new state.

For us, this means we want to append active devices to the list and remove inactive devices. We pass both functions to the `aggregate()` function and get a KTable containing all active devices for each user. Now, we can join the `notifications-topic` with this KTable and then finally write the result to a Kafka topic.

In Flink SQL, we split the query into parts. The first query creates a view `active_user_devices`. For each `user_id`, it contains the list of active devices:

```
CREATE VIEW active_user_devices AS
SELECT
    user_id,
    LISTAGG(
    CASE
        WHEN is_active = true THEN
            JSON_OBJECT(
                'deviceId' VALUE device_id,
                'deviceInfo' VALUE device_info
            )
    END
    ) as device_list
FROM user_devices
GROUP BY user_id;
```

Then, in another query, we perform the actual join of the `notifications` topic with `active_user_devices`:

```
CREATE VIEW device_notifications AS
SELECT
    n.user_id,
    n.notification_data,
    d.device_list
FROM notifications n
JOIN active_user_devices d
ON n.user_id = d.user_id;
```

12.5 Streaming and time

In the previous section, we already hinted at it: time can play a significant role in processing data streams. For example, when searching for fraud attempts in credit card transactions, the temporal component is often very important. If, for instance, there are three or more authorization attempts with a credit card within 5 minutes, it might

be suspicious, and we should investigate it. But how exactly can we proceed with Kafka in this regard?

12.5.1 Time is relative

First, we need to clarify Kafka's ordering guarantee: within a partition, the order of messages is guaranteed in the order they arrive at the leader broker. However, this doesn't mean that the timestamps of the messages are ascending. In fact, the meaning of a message's timestamp can vary.

In Kafka's default configuration, the timestamp of a message is set by the producer at the time we call the `send()` method. This means that the broker isn't responsible for the timestamp; instead, it's the producers. This is because the configuration `log.message.timestamp.type` is set to `CreateTime` by default. However, `CreateTime` isn't the time at which the event occurred. There could be seconds or more in between. Therefore, it's also possible for the producer to set the timestamp manually. However, we generally advise against manually changing the technical timestamp in Kafka. Instead, it's more sensible to add a new field in the data that represents the time at which the described event occurred. We typically call this time the *event time*.

We can also configure our brokers (via `log.message.timestamp.type`) and our topics (via `message.timestamp.type`) so that the message timestamp is set to the time when the message is received by the broker. The advantage of this is that the messages in a partition are also temporally sorted, at least if the time isn't adjusted. We call this time `LogAppendTime`.

Finally, there's another time that is often the easiest to work with: the *stream time*. This is the time at which a message is processed in our stream processor. Table 12.3 summarizes the different types of time.

Table 12.3 Streaming times

Time	Responsibility	Meaning
Event time	Our code	The time at which an event occurred
Create time	Producer	The time at which the message was created by the producer
Log append time	Broker	The time at which the message was written to the log by the broker
Stream time	Stream processor	The time at which a message is processed

It's important to be aware of these different times when we want to perform time-based operations. In stream processors, we work with time windows. For example, we may want to aggregate data over the past 2 hours or correlate data from two different topics within a certain time frame. Stream processing frameworks provide the option to choose between these different times. These methods are mostly called `TimestampExtractor`.

However, because Kafka doesn't guarantee that timestamps are ascending, messages may arrive "too late" in a partition. Without proper configuration, these messages are simply ignored. However, we can also set a *grace period* (or *lateness*) in the frameworks to allow for a certain amount of time to wait for late-arriving messages to be processed correctly.

12.5.2 Time windows

We've already hinted at time windows but haven't discussed what this means and what different types exist. In figure 12.11, we've summarized possible time windows.

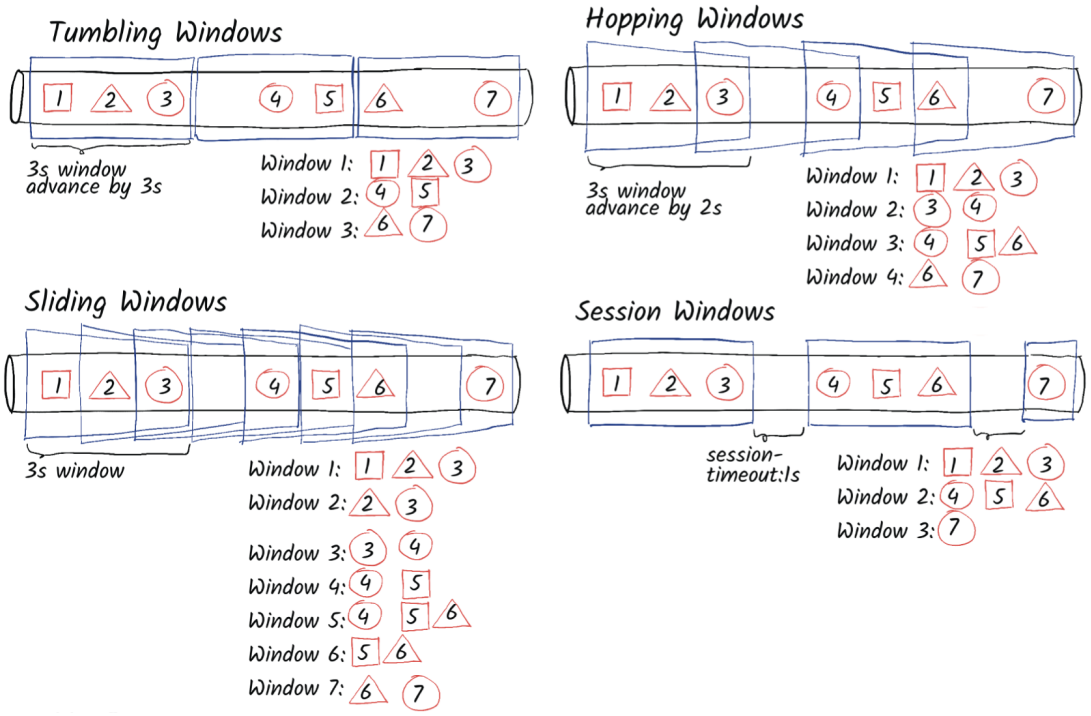


Figure 12.11 Stream processing frameworks support different types of time windows: Tumbling, Sliding, Hopping, and Session Windows.

The simplest type of time window is the *Tumbling Window*. We define a length for the window, and as soon as one time window ends, the next one begins. These time windows are nonoverlapping. We use Tumbling Windows whenever we want to process each message once within a time window. This is useful for billing or to measure the energy consumption of a device and only have the current power readings at certain times.

Tumbling Windows aren't suitable to answer questions such as "How many credit card transactions occurred in the last 5 minutes?" This is because we're not interested in the number of transactions from minute 1 to minute 2, but rather want to calculate the sum continuously. *Sliding Windows* are excellent for these tasks. Again, we only need the length of the time window as a parameter, allowing us to efficiently perform calculations by using the following equation: calculate X over the last Y minutes.

However, if we want a daily report on, for example, the average revenue over the past seven days, Sliding Windows are less well-suited. For this purpose, Kafka Streams offers *Hopping Windows*. Hopping Windows are configured using two parameters: the length of the time window (seven days, in our example) and the advancement time after which a new window should be opened (one day, in our example). Regardless of whether there are new messages, a window is opened, and calculations are performed. Hopping Windows should therefore be used whenever fixed-interval results over overlapping time periods are needed.

> **NOTE** For closely spaced intervals, Sliding Windows are usually the better choice as they are more performant and avoid frequent recalculations of results. Hopping Windows, on the other hand, are useful when periodically outputting results.

Finally, with *Session Windows*, we close a window when there has been no message for a certain period. This means there's a gap without messages between two windows. As the name suggests, this is useful whenever we want to analyze user sessions.

12.5.3 Use case: Fraud detection

One of the classic examples of streaming real-time data is detecting and preventing fraud attempts in real time. For instance, if we have an application where we know from experience that the risk of fraudulent authentication attempts is very high, we can set up a streaming pipeline that monitors authentication attempts in near-real-time. If a user makes too many attempts within a short period, we can require an additional factor to confirm their identity, such as a push notification or even a phone call.

We could implement this use case as follows (see figure 12.12 for a graphical representation): All authentication attempts are written to the `auth_requests` topic. Next, we have the Fraud Detection Service running, which checks if there has been at least three authentication attempts within 1 minute. If this is the case, we write a message to the `possible_frauds` topic. With Kafka Streams, we can implement the topology in the following code:

Prepares aggregation by grouping

```
KStream<String, String> authRequests = builder.stream("auth_requests");
KTable<Windowed<String>, Long> windowedCounts = authRequests
    .groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(
        Duration.ofMinutes(1)))
```

This time, we do a windowed aggregation.

```
    .count();
KStream<String, Long> possibleFrauds = windowedCounts
    .toStream()
    .filter((windowedKey, count) -> count > 3)
    .map((windowedKey, count) ->
        KeyValue.pair(windowedKey.key(), count));
possibleFrauds.to("possible_frauds");
```

The actual aggregation

Unpack windowed Key

Filters the fraud attempts

Writes the fraud attempts to the output topic

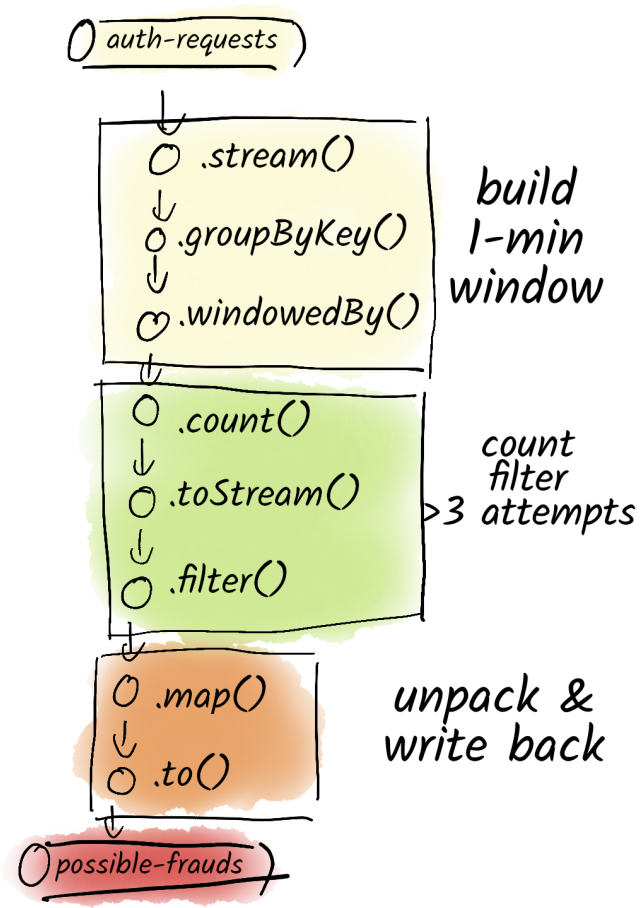


Figure 12.12 Kafka Streams topology to detect fraud attempts in real time

First, we create a KStream called `authRequests` to read data from the topic. Then, we create a time window of 1 minute and count the number of attempts within that window. Finally, we filter these counts so that we have at least three attempts and write them to the `possible_frauds` topic. In Flink SQL, this code can be implemented as follows, assuming that the input and output tables have already been created:

```
INSERT INTO possible_frauds
SELECT
    user_id,
    COUNT(*) AS request_count,
    TUMBLE_START(request_time, INTERVAL '1' MINUTE) AS window_start,
    TUMBLE_END(request_time, INTERVAL '1' MINUTE) AS window_end
FROM auth_requests
GROUP BY
    user_id,
    TUMBLE(request_time, INTERVAL '1' MINUTE)
HAVING COUNT(*) > 3;
```

12.6 Scaling Kafka Streams

Kafka's performance is largely based on parallelization, achieved through partitions. To parallelize our stream processing to improve performance, we need to break our work into smaller pieces so they can be processed independently. Because different stream processing frameworks approach this differently, this section focuses on Kafka Streams.

Apache Flink is typically set up as a separate cluster and manages its jobs autonomously and somewhat differently from Kafka Streams. There is also a Flink operator that can be used to simplify deploying and managing Flink clusters in Kubernetes environments. While we won't go into detail on Flink, the fundamental ideas are similar, and understanding these concepts is beneficial for applying them in other contexts.

Kafka Streams follows Kafka's approach to parallelization by assuming that partitions can be processed independently. It splits KStreams and KTables into partitions so that they can be processed by different tasks. *Tasks* are the smallest processing unit in Kafka Streams. Partitions are firmly assigned to tasks, and the number of tasks in an application solely depends on the number of partitions to be processed.

> **NOTE** If you remember Kafka Connect or consumer groups, you're right on track: the underlying protocol is (almost) the same.

For example, if we want to filter a topic with 12 partitions, Kafka Streams creates 12 tasks, exactly 1 for each partition. These tasks are then distributed across Kafka Streams instances. If we only have one instance, it processes all 12 tasks, and thus, in this example, all 12 partitions. If we have six instances, each instance typically processes only 2 tasks. To determine which instance is responsible for which tasks, Kafka Streams uses the *Kafka Rebalance Protocol*. If an instance fails, the tasks are redistributed to other instances, which then take over the work.

Kafka Streams not only supports splitting tasks across multiple instances but also within each instance across multiple threads. We configure the number of threads per instance through the Kafka Streams parameter `num.threads`. This way, we can distribute tasks not just across instances but also within an instance into threads.

Let's imagine we have a Kafka Streams application with nine tasks reading from a topic. We start two instances: one with two threads and the other with three. Then, as

shown in figure 12.13, we illustrate a possible division of labor: each thread has its own internal consumer and producer. All threads except Thread 3 on the second instance process two tasks, each executing its topology and reading messages from the Kafka topic. Each task has its own state store.

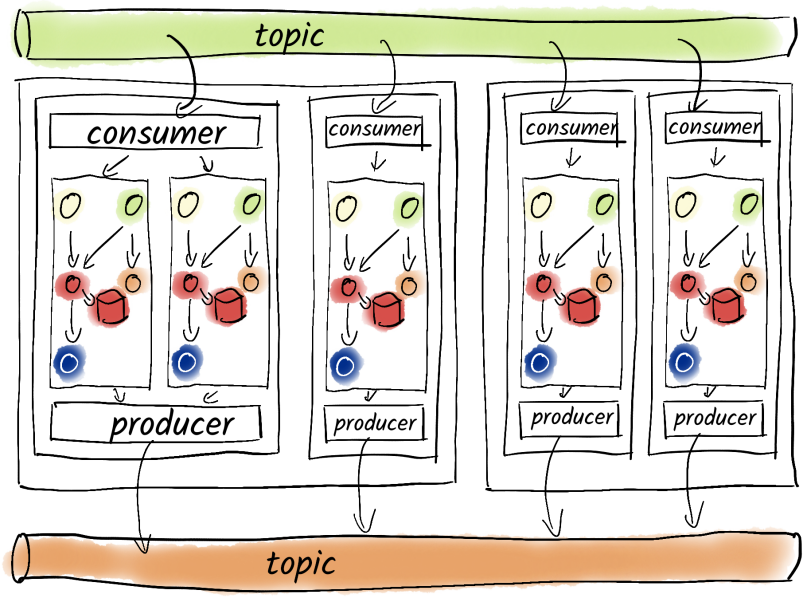


Figure 12.13 Kafka Streams applications are divided into tasks that are distributed across different threads on different instances. If an instance fails, another instance takes over the tasks.

But what happens to the *state stores*? If we need state stores, Kafka Streams creates one state store per Kafka Stream processor per task. If an instance fails and another takes over, the state store must be rebuilt from the internal changelog topic for that state store. The internal topic is read, and the new state store is populated accordingly.

> **TIP** By setting `num.standby.replicas=1` or higher, we can avoid rebuilding state stores, as another instance creates a copy of a state store. If the active instance fails, the other instance can take over the work without needing to rebuild the state store from scratch.

Keys allow us to keep related messages within a partition. In Kafka Streams, we've already encountered operations such as `selectKey()` or `map()` that can change the key. When we use such an operation, the stream (or table) is marked for *repartitioning*. If we execute stateless operations such as `filter()` or `mapValues()` on such a stream, we don't need to worry, as it makes no difference on which instance the operation is executed.

However, if we perform a join or aggregation, Kafka Streams must ensure that all messages are processed by the task corresponding to the key. To guarantee this, Kafka Streams creates an internal topic ending with `-repartition`. Before the join or aggregation, data is first written to this topic and then read from it immediately. While drastically increasing network traffic and storage requirements, it ensures that joins and aggregations are executed correctly.

But Kafka Streams doesn't have a concept of writing data to a topic and reading back from it inside of a task. Thus, we need to split up the topology, that is, the processing logic, into two parts called *subtopologies*. The subtopology is responsible for the first part of the processing and finishes writing data to the `-repartition` topic. The second subtopology is then responsible for reading data from this topic and for further processing.

In figure 12.14, we have a topology that first calls `selectKey()` and then performs a `count()`. As `selectKey()` changes the key, a repartitioning needs to happen, so the topology consists of two subtopologies: one for the `selectKey()` part and one for the aggregation.

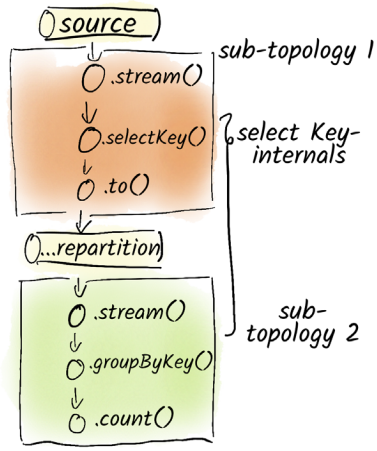


Figure 12.14 If we first execute a `selectKey()` and then an aggregation such as `count()`, the topology needs to be split up into multiple subtopologies.

Summary

- Kafka enables near real-time data exchange, eliminating delays from traditional batch processing.
- Kafka Streams, Apache Flink, Faust, and Goka are common stream-processing libraries, each with different strengths.
- Stream processing processes events individually in real time but requires managing eventual consistency.
- Stream processors such as those in Kafka Streams and Apache Flink transform, filter, and analyze real-time data streams using functional or declarative programming approaches.
- Processor types include `filter()` for selecting data, `map()` for transforming data, `merge()` for combining streams, and `split()` for branching data based on conditions.
- Stateful processing enables operations such as counting (`count()`) and aggregating data, which requires storing intermediate states.
- Processor topologies connect multiple processors to build complex workflows, such as merging streams, filtering significant changes, and aggregating data for insights.

- Streaming SQL allows real-time data processing similar to traditional SQL but continuously outputs results instead of a final response.
- Flink SQL enables SQL-based stream processing by defining tables linked to Kafka topics rather than storing data within the Flink cluster.
- Queries such as filtering (`WHERE`), aggregating (`GROUP BY`), and counting (`COUNT(*)`) work similarly to database SQL but operate on continuous data streams.
- Production usage typically relies on headless mode, where predefined SQL queries run continuously, with results written back to Kafka or other databases.
- Streams capture continuous data, while tables store the latest state for easier querying. Kafka Streams and Flink support both, using key-value stores and changelog topics to prevent data loss.
- Aggregations are used to determine the current state, such as summing transactions for an account. Results are stored in tables for further analysis.
- Joins enrich data by combining streams with tables or other streams. Stream-table joins trigger on new stream data, while table-table joins trigger on changes from both sides. Stream-stream joins are possible with time windows.
- Time in Kafka involves event time (event occurrence), create time (producer creation), log append time (broker write time), and stream time (processing time).
- Time-based operations in Kafka rely on time windows to aggregate or correlate data.
- Common window types include Tumbling Windows (nonoverlapping), Hopping Windows (overlapping), Sliding Windows (based on new data), and Session Windows (based on inactivity).
- Kafka Streams parallelization relies on splitting tasks based on partitions. Tasks are the smallest processing units and are distributed across Kafka Streams instances.
- State stores are created per task, and if an instance fails, the state store is rebuilt from the changelog topic. Configuring `num.standby.replicas` helps avoid rebuilding by using backup state stores.
- Repartitioning occurs when operations such as `selectKey()` are performed. Kafka Streams creates an internal repartition topic for joins and aggregations to ensure correct processing based on keys, often splitting processing into subtopologies.

13 Governance

This chapter covers

- Strategies for ensuring data integrity and compatibility
- Security measures to protect Kafka environments
- Controlling resource allocation
- Preventing cluster overload

Effective governance is crucial for any data-centric architecture, and Apache Kafka is no exception. As organizations use Kafka to manage their real-time data streams, the need for structured oversight becomes increasingly important.

What happens when proper governance is absent? Without structured schema management, data inconsistency can become a serious problem. For example, a change in a data producer's schema, such as adding a new field, can break downstream consumers that aren't prepared for it, leading to application failures and costly downtimes.

Similarly, the absence of robust security measures exposes the system to unauthorized access, which could result in data breaches, tampering, or even the loss

of sensitive customer information. A lack of resource management can also lead to resource hogging, where a single client overwhelms the Kafka cluster, degrading performance for other users and applications.

Consider our e-commerce platform that uses Kafka to process real-time transactions. Without schema governance, inconsistent product data from different sources could lead to errors in inventory systems or incorrect pricing.

If security is neglected, malicious actors might intercept sensitive payment data, causing financial losses and reputational damage. These scenarios highlight how proper governance isn't just about technical best practices; it's about ensuring reliability, trust, and seamless operation across the business.

We'll begin this chapter by exploring schema management, which serves as the backbone for data consistency and interoperability in Kafka. Proper schema governance ensures that data producers and consumers can communicate effectively, minimizing disruptions caused by schema evolution.

Next, we'll delve into the security landscape of Kafka. This includes critical practices for authentication and authorization, which protect sensitive data and ensure that only authorized users can access or modify it. We'll also touch upon the significance of encryption, both in transit and at rest, to safeguard data integrity and confidentiality.

Finally, we'll examine the implementation of quotas as a resource management strategy. Quotas help maintain the overall health of the Kafka cluster by preventing resource contention and ensuring that no single client can overwhelm the system.

By understanding these aspects of governance, organizations can foster a more controlled and secure Kafka environment, enabling them to harness the full potential of their data streams while maintaining compliance and reliability. This chapter aims to provide practical insights and best practices that will empower teams to establish effective governance frameworks within their Kafka implementations.

13.1 Schema management

Data always has a schema, whether someone has documented and agreed upon it or not. For example, with the emergence of NoSQL databases such as MongoDB, it was argued that development could be much faster there because less thought needed to be put into schemas compared to relational databases.

However, MongoDB documents also have a schema. At worst, it might not be described anywhere or enforced anywhere, but when an application reads data from a MongoDB document, it expects the data to be in a specific format. If the data is in a different and incompatible schema than assumed, this application will encounter a problem. In the best case, it crashes; in the worst case, it silently accepts the data and computes completely incorrect results.

This is made worse by the fact that schemas change over time as new requirements emerge or old assumptions no longer apply. In chapter 3, we've already examined the types of messages we see in Kafka practice and the data formats that can be used. In this chapter, we want to take a look at schemas through the lens of governance.

13.1.1 Why do we need schemas?

In figure 13.1, we start with a simple record from our supermarket example. A product is a JSON object that has an ID (e.g., `123`), a name (e.g., `coffee pad`), a price (e.g., `10`), and the quantity we have in stock (e.g., `1010`). First, we need to determine where this is defined: Is there a directory somewhere with all the schemas found in Kafka and documentation for them, or do stakeholders have to look at the data in the topics and put the schema together themselves?

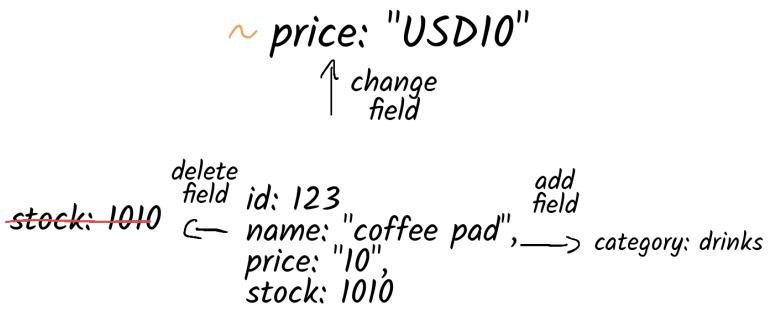


Figure 13.1 We start with a simple JSON object for a product. Based on this, we can add, delete, or modify fields. From the beginning, we should consider which schema changes should be supported and what the process will look like.

Regardless of whether a schema registry (more on that later) is used, we recommend having a central place for Kafka where schemas for individual topics are discoverable and documented. This can happen in the corporate wiki, in a central Git repository, or on a platform specifically set up for this purpose. For us, the "how" isn't crucial, we just need to know that it exists.

Over time, there will be new requirements for this data product. Perhaps a category (e.g., `drinks`) needs to be added. Existing consumers should hopefully be able to handle this change by ignoring this field. But what if a new consumer is added and needs to read historical data? This will likely cause problems. Perhaps we don't want to store the stock level or store it elsewhere and delete this field. New consumers should also be able to understand old data without any problems, but existing consumers will now cause problems.

Or, perhaps our supermarket is expanding internationally and needs to support different currencies. We could extend the price with the currency. However, this leads to problems both with old consumers that can't handle the new data and with new consumers that can't handle the old data.

As we can see, it's worthwhile to think about possible changes to our data schemas from the beginning. We need to discuss compatibility because the data schema is the contract between the producer (usually the data owner) and the consumer (usually the

data customer). It should also define which changes can be made without agreement and which can't.

Many are already familiar with this from relational databases, especially when multiple teams have to share a database. Every significant schema change may need to be discussed and coordinated with all teams under certain circumstances. However, the advantage of relational databases is that these schema migrations can often be performed atomically. Old data is transferred to the new schema, and all clients are immediately up-to-date. Of course, this is also very complicated with large amounts of data in detail.

In Kafka, however, we often don't have this luxury. Data can remain in a topic for a long time, and consumers may need to handle multiple versions of the same data in the worst case. It may even be the case that we have multiple producers producing simultaneously with different schemas. Unfortunately, it's not trivial to migrate old data to a new schema in this case.

> **TIP** Sometimes, changes must be made that break compatibility with previous versions. For this, it's often necessary to create a new topic, migrate the old data, and then migrate producers and consumers. The procedure here is the same as described for changing the number of partitions as we discussed in chapter 6.

13.1.2 Compatibility levels

We can view compatibility either from the consumer's or the producer's perspective, and this often leads to confusion about what exactly is meant. Because data is usually provided by the producer team, which is generally up-to-date, let's focus on the other side and consider data compatibility from the consumer's perspective.

To illustrate the different levels of compatibility, let's use a small example from the supermarket world. We'll start with the following product schema:

```
{
  "id": "123",
  "name": "Coffee Pad",
  "price": "10",
  "stock": 1010,
  "origin": null
}
```

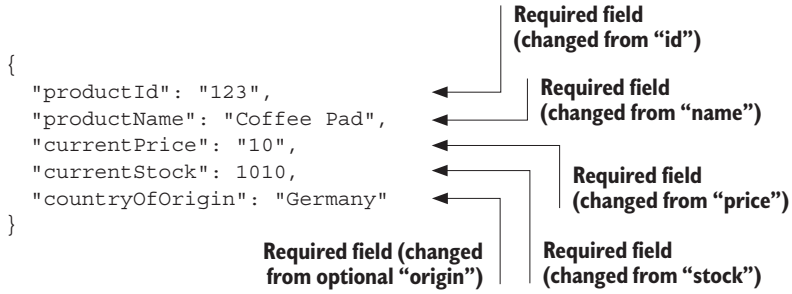
Required fields

Optional field

NO COMPATIBILITY

From the producer's perspective, the simplest case is when no consideration is given to compatibility, and consumers are forced to always be up-to-date. In this scenario, fields can be added or removed at will, but consumers will have difficulty reading historical data, and problems may arise if consumers aren't updated in sync with producers. This approach works well if the producer and consumer are managed by a single team,

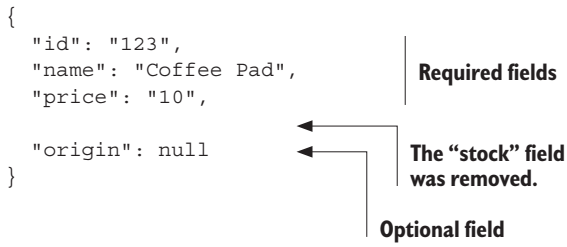
and the data in Kafka isn't stored for long periods. A classic example of a change that breaks all compatibility levels is renaming the fields:



A consumer expecting the previous version won't be able to work with the new data, and similarly, a consumer expecting the new data won't be able to handle the old data.

BACKWARD COMPATIBILITY

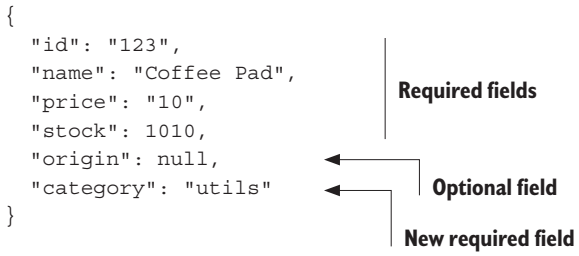
In this case, data is structured so that consumers using the current version can also read older, historical versions of the data. This is particularly useful, for instance, when all old data needs to be read during the launch of a new service. The challenge with this approach is that schema changes require consumers to be updated first, followed by producers. Because the producer usually controls the schema, this rarely aligns well with company processes. An example of a schema change that would be supported by backward-compatible consumers is removing fields:



A consumer familiar with this schema should have no problems with older schemas, as they can simply ignore the `stock` field. However, an older consumer who expects the `stock` field will face problems with the new schema, as this category is suddenly missing.

FORWARD COMPATIBILITY

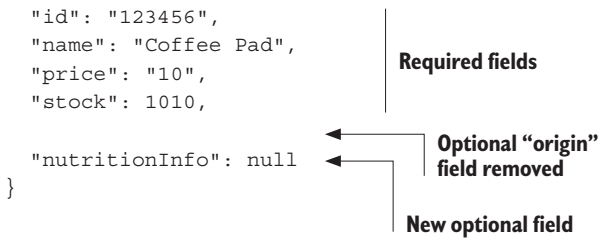
It's procedurally simpler when the schema is first updated by the producer, and consumers can catch up later. The downside here is that it's not guaranteed that consumers can read historical data. If data is only available in Kafka for a short period, and producers and consumers are developed by different teams, this is usually the easiest compatibility level to implement. In practice, this compatibility level captures the most common schema change, namely the addition of fields, as in the following example:



We've added a new field to the product, and an older consumer can read the new data without problems by simply ignoring the new field. The problem here is that if a consumer expecting the new schema wants to read historical data, they will miss the required `category` field. A solution is to add the `category` field as an optional field or as a field with a default value (e.g., `null`).

FULL COMPATIBILITY

Of course, it would be ideal if a consumer could read both old and new data. This limits the possibilities for evolving the schema and often forces us to break compatibility, but it's understandably desired in many environments. With full compatibility, it doesn't matter whether the producer or consumer updates the schema first. An example of a change that is compatible in both directions is deleting an optional field and adding new optional fields:



With this new schema, old consumers can still process it, as they will ignore `nutritionInfo` and already know that `origin` was optional. New consumers can also handle the original schema because the missing `nutritionInfo` is optional, and the `origin` field is ignored.

OVERVIEW OF COMPATIBILITY LEVELS

Figure 13.2 displays an overview of the different schema compatibility levels. In addition to the levels of compatibility, we can also differentiate whether the compatibility should apply transitively or not. In this context, *forward-transitive* means, for example, that a consumer must be able to understand all future schema versions, not just the next one. The holy grail is full transitive compatibility, where a consumer can read all historical and future data.

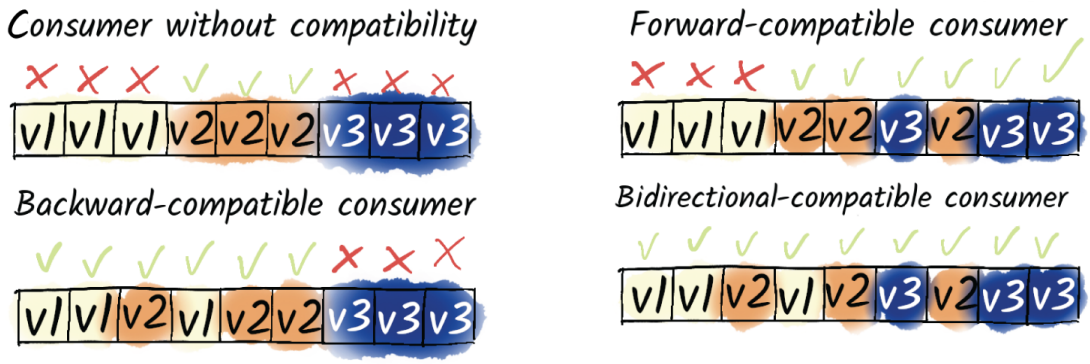


Figure 13.2 We consider schema compatibility from the consumer's perspective and which schema versions they can understand. If we can't or don't want to guarantee any compatibility, a consumer can only understand their own schema version. A backward-compatible consumer can read historical data but can't handle newer data. A forward-compatible consumer can read new data, but not historical data. The optimum is a consumer that can read both historical and new data, which is fully compatible in both directions.

To define which type of compatibility is needed for our schema, we need to understand how changes affect compatibility, which we've outlined in table 13.1. If our schema versions don't need to be compatible with each other, we can add or delete fields without much consideration. Optional fields can always be added or removed because this has no effect on the compatibility level. To maintain backward compatibility, we can delete required fields but not add them. For forward compatibility, it's the other way around: we can add required fields but not delete them. If the schema needs to be compatible in both directions, we can only add or delete optional fields.

Table 13.1 Field operations for different levels of compatibility

Field Operation	No Compatibility	Backward Compatibility	Forward Compatibility	Full Compatibility
Delete required field	Yes	Yes	No	No
Delete optional field	Yes	Yes	Yes	Yes
Add required field	Yes	No	Yes	No
Remove required field	Yes	Yes	Yes	Yes

TIP We recommend a central location for documenting schemas in individual topics. This documentation should also specify the compatibility level for the schema, and because the names can be confusing, it should clarify what this means for client update behavior. Kafka itself doesn't provide any mechanism to enforce these rules. For that, we need an additional component.

13.1.3 Schema registries

This is where schema registries come into play. They serve as a central authority for managing and enforcing schemas in a Kafka ecosystem. A schema registry is responsible for storing and versioning schemas, checking compatibility when schema changes occur, and providing this information to both producers and consumers.

Schema registries were initially developed to efficiently store Avro data in Kafka. However, their functionality has since expanded to support JSON and Protobuf schemas as well. They offer several critical functions: they act as a central management instance and serve as the *single source of truth* for all schemas in a Kafka cluster. In addition, they allow for managing different versions of a schema over time and can automatically verify if new schema versions are compatible with older ones. Lastly, they seamlessly integrate with Kafka producers and consumers to ensure compliance with schemas.

Despite these advantages, there are some challenges and criticisms regarding the use of schema registries. Unfortunately, the existing solutions are primarily technically driven and often don't provide sufficient options for documenting or visualizing schemas.

Additionally, the introduction of a schema registry adds another component to the system, increasing the overall complexity. If this component becomes unavailable, it may potentially affect all producers and consumers. Especially in multi-cluster environments, using schema registries becomes complex and requires careful planning and configuration.

Another problem involves the schema IDs used. These IDs have no semantic meaning and are merely numbered sequentially, which makes recovering a schema registry quite difficult if the data stored in Kafka is corrupted or incomplete.

> **WARNING** If schemas are manually exported from one schema registry and imported into another, the IDs may change, making clean migration impossible.

Because Kafka operates only with byte arrays, schemas are checked only on the client side. This theoretically opens the possibility for a malicious or faulty producer to write data to a topic that doesn't conform to the defined schema. To mitigate this risk, there are various approaches.

One option is the *Single Producer Pattern*, where only the topic owner's team is granted write access to a topic. While this approach increases security, it significantly limits Kafka's use. Another approach is to validate schemas on the broker. But this isn't implemented in Kafka. In Confluent environments, the Confluent Server and Confluent Cloud natively offer broker-side schema validation, which is highly effective but available only to Confluent customers.

An alternative is to use a Kafka proxy that can perform schema validation. Two noteworthy options in this space are Conduktor, a commercial provider and pioneer in Kafka proxies with schema validation features, and Kroxylicious, an open source

solution backed by Red Hat. Though still new, Kroxylicious also offers at-rest encryption and could develop into a promising alternative.

Another important aspect is schema version management. In production environments, we advise against allowing producers to automatically create new schema versions. Instead, we recommend managing schemas through the deployment pipeline in the chosen schema registry. This approach ensures better control and traceability of schema development and reduces the risk of unintended schema changes that could lead to compatibility problems.

Despite these challenges and the necessary precautions, there are currently no compelling alternatives to existing schema registry solutions. We're aware of three different implementations that can be considered depending on the use case and infrastructure.

> **TIP** Some may argue that a shared library containing the schema model used by producers and consumers can provide compile-time type safety, but it creates tight coupling between teams and requires coordinated deployments across services when schemas change.

The most well-known and widely used registry is the *Confluent Schema Registry*, which we recommend to customers already using the Confluent platform. For users of open source Kafka or other distributions, we advise against it due to licensing reasons.

An alternative is the *Karapace Schema Registry*, an alternative developed by Aiven that is API-compatible with the Confluent Schema Registry. Although Karapace has limited features, it proves sufficient for most use cases.

A third option is the *Apicurio Schema Registry*. This solution not only serves as a schema registry for Kafka but also supports OpenAPI and AsyncAPI. Apicurio additionally offers an API design studio and features for schema documentation. Although our experience with this solution is limited, we recommend that you explore it further, as it offers promising features for comprehensive API management.

13.1.4 Avro

Let's take a closer look at Avro. Although this is a data format that isn't directly related to Kafka, it was the motivation for introducing the schema registry in the first place and has become very popular in the Kafka world. Like the widely-used Protobuf, Avro is a binary data format. What makes Avro particularly interesting is that every Avro message consists of two parts: the schema and the actual payload of the message. This means that an Avro message is self-describing.

This is especially relevant for Kafka because data can be stored in Kafka for a long time, and it's common for multiple schema versions to coexist in a Kafka topic. The self-describing nature makes Avro an ideal format when schemas need to evolve and remain traceable over time.

Avro schemas are usually described in JSON. For example, we can describe our previous example using the following Avro schema:

```
{
  "type": "record",
  "name": "Product",
  "namespace": "com.example.supermarket",
  "fields": [
    {"name": "id", "type": "string"},
    {"name": "name", "type": "string"},
    {"name": "price", "type": "float"},
    {"name": "stock", "type": "int"},
    {"name": "origin", "type": ["null", "string"], "default": null}
  ]
}
```

Now, we can encode the following message in Avro:

```
{
  "id": "123",                    | Required
  "name": "Coffee Pad",           |
  "price": "10",                  |
  "stock": 1010,                  |
  "origin": null,                 <-- Optional
  "category": "utils"             <-- Required (new)
}
```

When encoded in binary, the payload is 35 bytes in size, but the schema, stored as JSON in Avro, is 292 bytes. This means the message is over 10 times larger than it needs to be.

The solution is to not send the schema with every message. Instead, we store the schema in the schema registry and include only the schema ID in the Kafka message. This ensures that our Kafka messages remain small, while also allowing us to track the schema for each individual message. This provides a solid foundation for evolving our schemas without everything descending into chaos.

13.2 Security

Securing IT systems is a fundamental task for any IT operation. Before discussing how to effectively secure Kafka, we need to clarify which threats we're protecting the system against. First, we need to ensure that only authorized users, whether human or technical, can access the system. This is achieved through *authentication*. Kafka performs authentication once during connection setup. For authenticated users, we then need to ensure that they can only perform permitted actions, which is managed through *authorization*.

In the past, internal network traffic was considered secure, but that is rarely the case today. In cloud environments, we must assume that network traffic can be intercepted or manipulated. Therefore, it's crucial to encrypt and sign network traffic to ensure data integrity.

Because Kafka persists data, it's advisable to encrypt data stored on disks. However, *encrypting data at rest* can be questioned, as the storage systems are usually active. Disk encryption primarily protects against physical attacks by making it harder to access data

on removed disks. But because encrypting disks is relatively easy and cost-effective, we should take advantage of this option.

Measures such as network encryption, signing, and disk encryption don't protect against administrators with server or Kafka admin rights. Therefore, data in Kafka is typically classified as internal and confidential. For highly sensitive data transmission via Kafka, *end-to-end encryption* is necessary.

Figure 13.3 illustrates these key security measures relevant to Kafka, including authentication to ensure only authorized users connect to the system, authorization to restrict their actions, network encryption and signing to protect data in transit, disk encryption to secure data at rest, and end-to-end encryption for highly sensitive data. In the next sections, we'll take a look at how we can secure Kafka in practice.

WARNING For older Kafka clusters, it's also important to properly secure ZooKeeper. In particular, enabling transport encryption is key, although this wasn't easily achievable for older Kafka clusters.

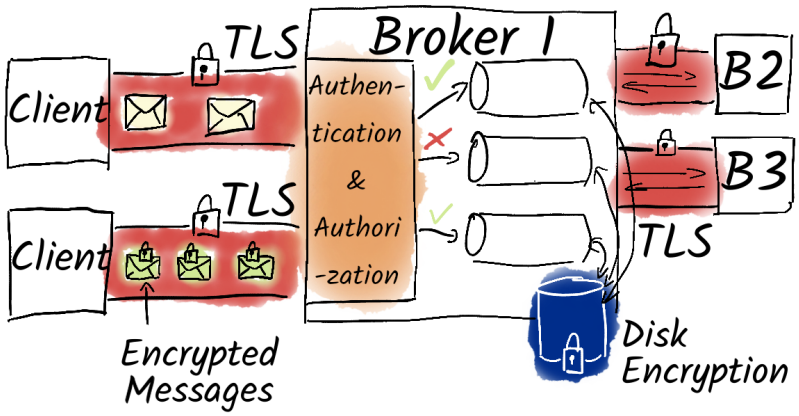


Figure 13.3 Securing a Kafka cluster involves transport encryption for communication, user authentication and authorization, and encryption of data at rest.

13.2.1 Transport encryption

No matter which protocol we use for authentication and how we proceed further, transport encryption is now mandatory for most IT systems. In practice, we often see Kafka environments that were set up without security measures, where there's a desire to add them later.

Fortunately, Kafka has a solution: it's possible to configure multiple listeners, allowing both encrypted and unencrypted communication during a transition period, for example, while reconfiguring the system. We'll explore this in detail later in section 13.2.7.

Kafka uses Transport Layer Security (TLS) for transport encryption, as is standard. However, TLS doesn't come without cost. Beyond the nontrivial configuration effort, Kafka sacrifices some performance characteristics with TLS. The general rule is if Kafka is already pushing our infrastructure to its limits without TLS, then thorough performance testing should be carried out during the migration.

In most of the applications we know, where CPU usage is manageable, enabling TLS results in minimal performance losses. In most environments, the security requirements are predefined, and the infrastructure must meet these standards.

In theory, configuring TLS isn't difficult, but in practice, it often comes with stumbling blocks and challenges. First, we need to decide how to set up TLS. Typically, both client-broker communication and broker-broker communication should be encrypted.

TLS is based on *asymmetric cryptography*, meaning *public-private key cryptography*. We generate a key pair (public and private) for each broker and sign each broker's key pair with a central *Certificate Authority* (CA). Next, we configure a *keystore* for each broker, which holds the associated private key. To ensure that clients can verify they are connected to the correct broker, we configure a *truststore* containing the public key of the CA and distribute it to the clients. This also needs to be done on the brokers so that they can trust each other.

At this point, we could fill several pages with *keytool* commands to show how to set up TLS correctly. However, because this would be a long digression with little didactic value, we'll skip it and instead refer to many detailed discussions that are available online. Here, we'll only show an excerpt from the broker configuration that illustrates a possible setup:

```
listeners=PLAINTEXT://$IP:9092,SSL://$IP:9093
listener.security.protocol.map=\
    PLAINTEXT:PLAINTEXT,SSL:SSL
ssl.keystore.location=/home/user/certs/broker.ks.p12
ssl.keystore.type=PKCS12
ssl.keystore.password=keystore-pw
ssl.key.password=keystore-pw
ssl.truststore.location=\
    /home/user/certs/ca.truststore.p12
ssl.truststore.password=truststore-pw
ssl.truststore.type=PKCS12
```

Configures two listeners on which this broker will accept requests

Maps the listener names to the corresponding encryption protocols

Configures the keystore

Configures the truststore

In addition to the unencrypted plaintext listener on port 9092, we configure a second listener using TLS on port 9093. We can choose custom names for these listeners—for instance, you might name one EXTERNAL for handling external traffic, PLAINTEXT for nonencrypted traffic, or SSL for encrypted traffic. For each listener, we need to specify its corresponding security protocol. Finally, we set up the keystore and truststore configurations.

13.2.2 Authentication

Now that we've established an encrypted connection between clients and brokers, we should consider how to ensure that clients truly are who they claim to be. Kafka provides two main options for this. Either we extend our TLS configuration so that clients authenticate themselves to brokers using TLS certificates (*mutual TLS*), or we use one of the many options offered by Simple Authentication and Security Layer (SASL).

It's important to emphasize that authentication occurs only during the connection setup and remains in place for the entire session. This means that if a malicious actor takes control of a client, simply deleting the user in Kafka isn't enough; all permissions for that user must also be revoked. We'll talk more about this in section 13.2.3.

If a user isn't authenticated, we can still assign (or revoke) permissions for them. Later, we'll learn that authorization is based on a *principal name*, and for unauthenticated users, this name is `User:ANONYMOUS`.

A popular authentication option is using mutual TLS. Typically, TLS is used for transport encryption and to authenticate brokers to clients, so enabling mutual TLS for client authentication requires a manageable additional effort. With mutual TLS, clients also authenticate themselves to brokers. This requires generating a TLS key pair for each client, which must be signed by the CA. This ensures that both clients can trust that brokers are who they claim to be, and brokers can trust that clients are legitimate.

Later, during authorization, we need a principal name to grant or revoke permissions. By default, in a mutual TLS setup, this principal name is the entire *Distinguished Name* (DN) string, for example:

```
CN=alice,OU=myOrgUnit,O=SampleCorp,L=SampleCity,ST=SampleState,C=SampleCountry
```

In most cases, the *Common Name* (CN) is sufficient to correctly address a user. Kafka provides a configuration option, `listener.name.<listener_name>.ssl.principal.mapping.rules`, which allows us to use regular expressions to convert the DN into a principal name. This setting allows multiple rules to be specified, but in this case, we can use the following rule:

```
listener.name.ssl.ssl.principal.mapping.rules= \
    RULE:^CN=([^,]*).*$/$1/L ,\
    DEFAULT
```

Here, we extract the CN value from the DN so that we can work with it later. If we choose not to use mutual TLS, then SASL, an authentication framework commonly used in the Java world, offers a variety of authentication options. While we won't cover all of them here, we'll provide an overview. The two most popular methods among our clients are *SASL-GSSAPI* (authentication via *Kerberos*) and *SASL-OAUTHBEARER* (authentication via *OpenID Connect*).

In both approaches, user data is managed in external systems. For even more flexibility, Kafka offers the option to use *SASL-PLAIN*, where a username and password are

accepted and a callback is invoked for authentication. However, this callback must be custom developed. A simpler alternative is *SASL-SCRAM*, a challenge-response mechanism often based on username-password combinations. In this case, usernames and passwords are stored in ZooKeeper or in KRaft mode, eliminating the need for an additional system to manage authentication data.

13.2.3 Authorization

We now have an authenticated principal and know who the client is that is talking to the broker. The next step is to define what it's allowed to do. We recommend following the *least-privilege principle* when setting up access control lists (ACLs).

Let's look at our online shop example: The checkout service needs write permissions to the `checkout.changelog` topic to record completed purchases. The payment-processing service needs read permissions for this topic as well as read permission for its consumer group `payment-processor` to process the payments. The checkout-analytics service, implemented with Kafka Streams, requires all permissions on topics starting with `checkout.analytics.*` because Kafka Streams manages its own internal topics plus read permissions for the `checkout.changelog` topic to analyze shopping patterns.

Kafka provides an extensible authorization system that determines whether a given *principal* (essentially the name of an authenticated user) is allowed to perform a specific action. By default, Kafka includes only the *AclAuthorizer* module, but some providers offer modules for *role-based access control* (RBAC).

While this can be useful for some scenarios, we believe it's not a "must-have" because only technical users typically access Kafka, and ACLs are generally sufficient for most cases. Naturally, this process needs to be automated, so we'll focus on the ACL module here.

Table 13.2 shows the structure of Kafka ACLs. Each ACL can be read as a statement: "A *principal* is *allowed* or *denied* permission to perform an *operation* on a *resource* identified by its *name* or *prefix*." Additionally, the ACL can be restricted to a specific *host*. For example, we can define that the principal `User:PriceAnalyticsService` has the `Allow` right to perform `Read` operations on a resource of type `Topic` with the name `products.prices.changelog` (Pattern: `Literal`).

For a given principal (and host), the ACL module first checks whether this principal is a *super-admin*. If so, the action is allowed. Otherwise, the module checks for a `Deny` rule. If a `Deny` rule exists, the action is rejected. Finally, the module checks if there is an `Allow` rule for that principal.

Different Kafka components require specific permissions to function properly. Brokers need `ClusterAction` permissions on the cluster resource to carry out their duties. For producers, `Topic:Write` permissions are the minimum requirement, with additional `Cluster:IdempotentWrite` permission needed for idempotent producers. When using transactions, producers must also have `Write` permissions on a specific `TransactionalId`. Consumers operate with simpler requirements, needing only `Topic:Read` permission for their consumed topics and `Group:Read` permission for

Table 13.2 Kafka ACLs structure

Component	Options	Examples
Principal	`<type:name>`	`User:PriceAnalyticsService` `Group:AnalyticsServices`
Resource type	`Cluster, Topic, Group, TransactionalId, DelegationToken`	`Topic`
Pattern type	`Literal, Prefixed`	`Literal`

Resource name		`Literal:`
		`products.prices.changelog`
		`Prefixed:`
		`products.prices`

Operation	`Describe, Create, Delete, Alter, Read, Write, DescribeConfigs, Alter-Configs,ClusterAction, IdempotentWrite`	`Read`
Permission type	`Allow, Deny`	`Allow`
Host	`<IP Address>, *`	`*`

their group ID. Kafka Streams applications, which manage their own topics, should be granted `Topic:Create` rights on the *application ID* prefix.

Setting up an automated CI/CD pipeline for user and topic management typically requires a broader set of permissions: `Create`, `Delete`, `Alter`, `AlterConfigs`, `Describe`, and `DescribeConfigs` rights on topics. The pipeline also needs `Alter` and `Describe` permissions on the cluster for ACL management, along with potential `AlterConfigs` and `DescribeConfigs` permissions on the cluster for quota management.

13.2.4 Encryption at rest

Transport encryption, authentication, and authorization are mandatory in IT systems. In many environments, especially regulated ones, it's also essential that data is encrypted at rest, that is, on disk.

The bad news is that Kafka doesn't offer native support for this. Instead, Kafka relies on the disk encryption capabilities provided by the operating systems, storage systems, or cloud providers. We believe that this doesn't provide significant improvements of the security, as anyone with admin access to the underlying system can access the data in Kafka too.

One alternative is using a Kafka proxy such as Kroxylicious, which handles encryption for producers and decryption for consumers. While not providing true end-to-end encryption, this approach prevents data center administrators or cloud providers from accessing unencrypted data, as long as the proxy remains under our control, as shown

in figure 13.4. The key advantage of this proxy-based encryption is that it enhances security without requiring complex end-to-end encryption implementations on the client side.

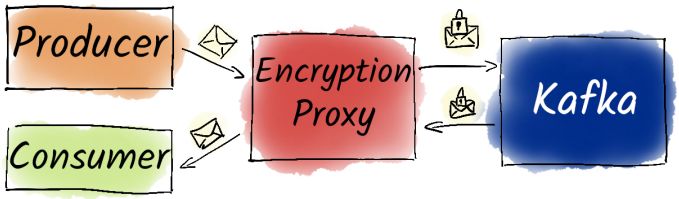


Figure 13.4 An encryption proxy can reduce the risk of using an untrusted Kafka provider.

13.2.5 End-to-end encryption

If data confidentiality is critical, a more reliable method is end-to-end encryption. Again, the bad news is that Kafka doesn't offer native support for this either—it must be implemented by the clients, that is, the producers and consumers.

It's common for teams responsible for messaging systems such as Kafka to guarantee only an internal level of confidentiality. For higher levels of confidentiality, development teams must take responsibility. Unfortunately, we're not aware of any widely recommended or usable library that guarantees end-to-end encryption.

For Kafka Connect, there is a plugin called *Kryptonite for Kafka Connect* that can encrypt and decrypt entire messages or individual fields. Another approach is to use a Kafka proxy such as the Conduktor Proxy or Kroxylicious, which encrypts incoming messages and decrypts outgoing ones.

> **WARNING** We caution against calling this method end-to-end encryption because the encryption only occurs between proxies. The path from the producer to the proxy, and from the proxy to the consumer, remains unencrypted, meaning the proxy can read all the data.

Ultimately, it's up to us to handle encryption within the producers and consumers. Our general recommendation is to implement this as a serializer or deserializer that first calls the regular serializer (e.g., a JSON serializer) and then encrypts the resulting serialized byte array. The reverse would be done with the deserializer. However, this approach doesn't allow for encryption or decryption of individual fields.

A big question that remains is how to manage encryption keys. This requires a dedicated key management process, which could be handled by tools such as Vault.

> **TIP** It's important to note that once data has been encrypted, it can no longer be effectively compressed, as well-encrypted data appears as random data, which isn't compressible.

13.2.6 ZooKeeper security

This section is relevant only for ZooKeeper-based clusters. ZooKeeper stores essential information about Kafka, making it of utmost importance to secure it properly. By default, all data is transmitted over the network in an unencrypted format. This includes information about topics, partition counts, and even ACLs. In the case of SASL-SCRAM, usernames and (*hashed* and *salted*) passwords are also stored in ZooKeeper and are thus transmitted over an unencrypted connection. Clearly, this isn't a good practice.

Fortunately, Kafka has supported a ZooKeeper version with TLS support since version 2.5. The configuration is similar to that of TLS transport encryption for Kafka brokers, so we won't delve into it in detail here. We particularly recommend using mutual TLS for authentication between brokers and ZooKeeper, although ZooKeeper also supports SASL for authentication.

Because a ZooKeeper ensemble typically manages a single Kafka cluster and shouldn't be shared with other systems, it's unnecessary to set up additional ACLs. Clients should, in any case, never access ZooKeeper directly. ZooKeeper is highly sensitive to latency, so when implementing encryption, it's crucial to ensure that latency doesn't become excessive, as this could affect the availability of the Kafka cluster.

13.2.7 Securing an unsecured Kafka cluster

We often encounter situations where a Kafka environment is set up without much thought given to security or other best practices. A common example is a proof-of-concept implementation that eventually ends up in production.

Fortunately, Kafka allows, as shown in figure 13.5, for migration from an unsecured to a secured cluster without affecting availability. The first step is to configure an additional listener with TLS and the desired authentication mechanism on all brokers. To make this configuration active, it's best to restart the brokers one by one. Once this is enabled, you can migrate the inter-broker communication to the new listener, which will require another restart of the brokers.

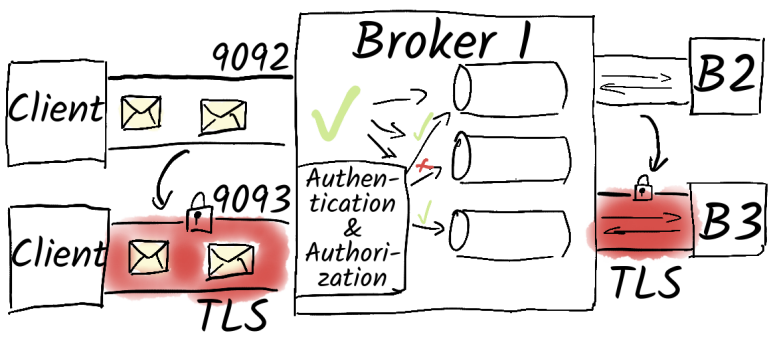


Figure 13.5 Required steps to secure an unsecured Kafka cluster

Next, it's time to update the clients. They need to be reconfigured to connect via the new listener, with TLS and authentication settings in place. Depending on the method, new key pairs may need to be generated, or credentials configured within the client. Usually, this reconfiguration will require restarting the clients. While updating, it's helpful to create a list of all users, which will be necessary in the next step.

Once all clients have been migrated, it's time to disable the old listener. It's best to do this during a day when most teams are working, as experience shows that a team or two might have been overlooked. These teams will likely notice and complain quickly, allowing for their clients to be migrated, or you may need to re-enable the old listener until the next release.

Now that all clients are authenticated, you can start creating ACLs for your users. There are two ways to proceed. One option is to set `allow.everyone.if.no.acl.found=true` in the broker configuration and gradually enable ACLs for users. However, this approach risks forgetting to disable the setting later or delaying migration as teams may defer because things continue to work.

In our opinion, it's more reliable to create ACLs for all users at once and enforce them in one go. The downside is that this might cause problems for many services simultaneously. Therefore, it's crucial to review the ACLs thoroughly and test this process on a test system beforehand.

Once this migration to a secure cluster is complete, teams can focus on their next tasks. As we can see, a seemingly straightforward migration requires extensive coordination between teams, and there are several points where things can go wrong.

> **TIP** Our recommendation is to think through these problems before setting up the cluster and ideally bring in expertise to benefit from experiences in other environments.

13.3 Quotas in Kafka: Protecting the cluster from overload

As in other systems, quotas can be used in Kafka to protect the cluster from excessive load. The purpose of quotas is often twofold: ensuring fair distribution of available resources and protecting the system from faulty or malicious clients that could potentially prevent other clients from using the system.

Kafka's ability to handle large volumes of data can complicate things further. For example, misconfigured producers could rapidly generate vast amounts of data, potentially filling the entire storage and bringing Kafka to a halt. Similarly, uncontrolled consumers could overwhelm brokers with requests, monopolizing resources and blocking other clients from interacting with Kafka. The risk increases with the size of the cluster and the number of teams and clients accessing it.

To better understand quotas, we must clarify which client can be throttled, which metrics Kafka brokers monitor to determine quota violations, and what actions the broker takes when a quota is exceeded.

Kafka offers several options for throttling clients. Default quotas can be set and later overridden as needed. If the Kafka cluster doesn't support authentication, quotas can be applied at the client ID level. However, this isn't very practical because client IDs can be arbitrarily set by clients, making it difficult to enforce quotas. It's more effective to set quotas at the user level by configuring them with the principal of a user.

> **TIP** We recommend using a conservative default quota that is sufficient for most clients, while also creating specific quotas for clients that require more resources. However, be cautious. Without proper monitoring, quotas can lead to significant problems.

Kafka allows quotas to be set in different ways. On one hand, you can define how many produce or consume requests a client can send in bytes per second. On the other hand, you can specify how much CPU time a broker can allocate to a client. Both types of quotas can only be enforced on a per-broker basis due to technical limitations. This means that a client could potentially produce a multiple of its assigned quota by connecting to more brokers. While this makes it difficult to treat quotas as guarantees, their primary purpose is to protect against overload, not for billing purposes.

It's also important to note that CPU usage is calculated per thread, meaning the total available capacity is the sum of the network and I/O threads times:

```
((num.io.threads + num.network.threads) * 100)%
```

In our online shop environment, we'll examine three key quota scenarios to understand how Kafka's quota system works. Each scenario demonstrates a different way to apply quotas to control resource usage.

The first scenario establishes a baseline by setting a default quota that applies to all clients unless they have specific quotas assigned. This default quota sets reasonable boundaries: producers are limited to sending 1 MB per second, while consumers can receive up to 2 MB per second. Additionally, these clients are allocated their fair share of request processing capacity, represented by a 100 % request rate limit:

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --alter \
    --entity-type clients \
    --entity-default \
    --add-config 'producer_byte_rate=1048576,\
    consumer_byte_rate=2097152,request_percentage=100'
```

The second scenario focuses on a specific producer—our newsletter service. Given its periodic nature and to prevent it from overwhelming the system, this service receives a more restrictive quota. It can only produce messages at half the default rate (512 KB per second) and is limited to using just 50% of its fair share of request processing capacity:

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --alter \
    --entity-type clients \
    --entity-name newsletter \
    --add-config 'producer_byte_rate=524288,request_percentage=50'
```

The third scenario addresses our warehouse management system, which needs to process larger amounts of data quickly. As a critical consumer of our system, it receives a more generous quota. It can consume up to 5 MB per second—significantly more than the default consumer quota—and is allowed to use double its fair share of request processing capacity with a 200% request rate limit.

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --alter \
    --entity-type users \
    --entity-name warehouse \
    --add-config 'consumer_byte_rate=5242880,request_percentage=200'
```

In all cases, we use the `kafka-configs.sh` command with the `--alter` option to modify configurations. The `--add-config` parameter specifies the quota values, while `--entity-type` and `--entity-name` (or `--entity-default` for the default quota) define the target of the configuration.

What happens when a client violates a quota? First, the broker calculates how long the client must pause to comply with the quota, using small time windows of 1 second. The broker waits for this duration before sending a response to the ongoing request. Because well-behaved clients have `max.inflight.requests` set, balance is quickly restored. Simultaneously, the broker mutes the connection to the client to prevent further requests during this time, protecting against faulty or malicious clients.

Beware of side effects when enforcing quotas. If quotas are set too tightly, they can have very negative effects on production environments. Aggressive limits might cause clients to malfunction, potentially leading to cascading problems. For example, a producer quota could cause the producer's `send()` function to block after filling up an internal buffer, eventually leading to timeouts as messages can't be sent fast enough.

Thus, it's essential to closely monitor applications when enforcing quotas. Kafka brokers offer *throttle-time* metrics in three forms: *Produce*, *Fetch*, and *Request*. The first two monitor whether bandwidth quotas are exceeded, and the Request metric monitors whether clients use too much CPU time. Ideally, these metrics should be zero. If they are greater than zero, brokers are currently throttling one or more clients. These metrics also indicate which client or user is being throttled. As throttling usually points to a misconfigured client or anomaly rather than a malicious attacker, so the problem should be quickly resolved with the client team.

> **TIP** Quotas are primarily used to protect the cluster from misconfigurations and prevent a single client from blocking the entire cluster. They aren't intended to artificially limit clients. We recommend setting quotas generously and monitoring the clients closely.

In summary, we recommend using quotas as more teams start using Kafka, giving rise to bottlenecks. We start with default quotas that are sufficient for most clients but can also handle peaks.

If specific teams need more resources, they should determine a reasonable maximum load for their application, and quotas can be set accordingly. However, setting quotas correctly isn't easy and requires a delicate touch. The goal should be to protect services in the cluster from poorly behaved clients without causing services to suffer from overly restrictive quotas.

> **WARNING** Before implementing quotas, we need to establish robust monitoring to get an overview of the cluster and applications. If monitoring is insufficient, then we shouldn't introduce quotas. Otherwise, we risk causing more harm than necessary, potentially hindering Kafka adoption within the organization.

Summary

- Proper schema management in Kafka helps ensure data consistency and compatibility.
- Using schema registries allows for the centralized management of data schemas, enabling easier updates.
- Schemas facilitate versioning, allowing producers and consumers to evolve independently without breaking changes.
- Compatibility checks can be enforced to prevent incompatible schema updates that might lead to data problems.
- Schemas can be serialized in different formats, such as Avro or JSON, enhancing interoperability across systems.
- Transport Layer Security (TLS) using encryption, authentication, and authorization is mandatory for safeguarding data in Kafka.
- Authentication in Kafka can be implemented using various mechanisms, such as Simple Authentication and Security Layer (SASL), to verify the identity of clients and ensure secure connections.
- Authorization is managed through access control lists (ACLs) to regulate which users or applications can access specific Kafka resources, ensuring only authorized actions are permitted.
- Data at rest must be encrypted using underlying storage systems, as Kafka lacks native support for this.
- End-to-end encryption is recommended to secure sensitive data, implemented at the client level.
- The key management process for encryption requires careful handling, potentially using tools such as Vault.

- ZooKeeper must be secured as it stores critical Kafka metadata and transmits data unencrypted by default. Kafka 2.5 and later supports TLS connections for secure communication with ZooKeeper, and mutual TLS is advised for broker-to-ZooKeeper communication.
- By sequentially configuring TLS listeners and restarting brokers, we can migrate from an unsecured Kafka cluster to a secured one without effecting availability.
- Quotas are essential for protecting Kafka clusters from excessive load caused by misconfigured or malicious clients.
- Quotas can limit the number of produce or consume requests and restrict CPU time on brokers.
- Quotas can be set at the client ID level or user level. User-level quotas provide better reliability.
- Kafka brokers monitor throttle-time metrics to detect and manage quota violations effectively.
- Care must be taken when setting quotas, as overly aggressive limits can disrupt service and lead to cascading failures.

14 Kafka reference architecture

This chapter covers

- Kafka tools, including GUIs, GitOps concepts, and Cruise Control
- Kafka's deployment models
- Key hardware requirements for Kafka brokers

Now that we've gained a good overview of what Kafka is, how we use it, and how it integrates architecturally into our existing enterprise IT landscapes, the big question arises: What do we need to operate Kafka successfully? As discussed in the previous chapters, Kafka alone is often insufficient to achieve our enterprise goals. A typical Kafka setup consists of much more than just the brokers and coordination cluster, as depicted in figure 14.1.

We need additional components for a successful Kafka environment. First and foremost, the applications that use Kafka are very important, such as simple producers and consumers that write data to or read from Kafka. We can use Java producers and consumers without restrictions. As mentioned in chapter 8, for non-Java producers, it's crucial to ensure that they use the same partitioning algorithm as Java producers to avoid potential problems.

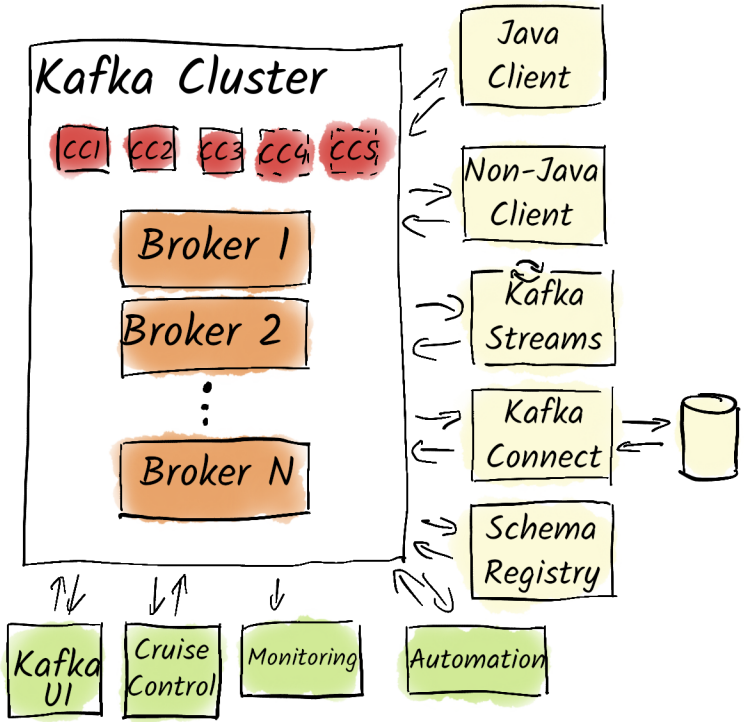


Figure 14.1 A typical structure of a Kafka installation with additional tools

In many cases, we don't even need custom producers and consumers to send data from third-party systems to Kafka or from Kafka to other systems. Many of these use cases can be elegantly solved with Kafka Connect without writing custom code. We extensively discussed the pros and cons of this solution in chapter 11.

Once data is in Kafka, it makes sense to process that data using Kafka tools. For this, we can use one of the numerous stream processing frameworks such as Kafka Streams or Apache Flink, as discussed in chapter 12.

Data always has a schema, whether we want to document and agree on it or not. We always advise our clients to explicitly consider schema management. Whether a schema registry is ultimately used or not is then secondary for us. We discussed this topic in chapter 13.

14.1 Useful components and tools

In the realm of Kafka, using the right components and tools can significantly enhance the efficiency and effectiveness of your data streaming and management processes. This section explores various utilities and interfaces that facilitate data manipulation, monitoring, and resource management within Kafka environments. From

command-line tools to GUIs, these components provide essential functionalities for both development and production scenarios.

We'll also discuss best practices for managing Kafka resources to ensure a smooth operation, particularly in automated and declarative settings. By using these tools, organizations can optimize their Kafka implementations and better meet their business objectives.

14.1.1 kcat

At the beginning of the book, we introduced the scripts that come with Kafka for writing and reading data to and from Kafka. An alternative to the provided console consumer and console producer is *kcat* (formerly known as *kafkacat*).

The kcat tool is primarily characterized by the fact that it's not based on Java but rather based on `librdkafka`, which is based on C. This has the advantage of negligible script startup time, making it better suited for use in custom scripts. kcat also includes some functions that the provided scripts don't support, such as producing or consuming headers. Additionally, the commands and parameters are often shorter and simpler to use. Let's look at a small example to produce some data into our old standby topic, `products.prices.changelog`:

```
$ kcat -b localhost:9092 -t products.prices.changelog -P
> energy drink 2
> energy drink 3
> cola 2
> cola 5
> energy drink 1
> cola 2
```

We end the command with Ctrl-D. kcat's parameters are much shorter. `-b` is for the bootstrap servers, `-t` is for the topic, and `-P` starts the producer mode. Let's consume the data again:

```
$ kcat -b localhost:9092 -t products.prices.changelog -C
energy drink 2
energy drink 3
cola 2
cola 5
energy drink 1
cola 2
% Reached end of topic products.prices.changelog [0] at offset 6
```

We end the command with Ctrl-C. The only difference is the change of mode. `-C` stands for consumer, and we could even have omitted the parameter because kcat attempts to guess the mode on its own. After all messages are read, kcat informs us about it. Of course, kcat also supports many additional parameters and options, which you can find in the project documentation at https://github.com/edenhill/kcat/.

WARNING We strongly advise against using these scripts in production, especially for writing data to Kafka. That should be the domain of Kafka producers.

14.1.2 Graphical user interfaces

In many cases, it's quite uncomfortable to navigate through Kafka topics using command-line tools. A graphical user interface (GUI) can help alleviate this. Note that a GUI should only be used for viewing in production. Data should never be produced or configurations altered through the interface in a production environment.

WARNING We recommend giving the user of the GUI only read permissions. For development environments, a GUI is, of course, very practical for writing test data and experimenting with the behavior of the connected systems.

We see the most important use case for a GUI in data exploration, that is, to see which topics exist in Kafka and to read data from topics. Often, GUIs also support options for searching or filtering data. When using Kafka Connect or the Schema Registry, it's also useful to view configurations and the current state graphically. Here again, we strongly advise against making changes in production.

Although there are many different GUIs for Kafka, a few favorites emerge in practical use. One of the simplest solutions is the *Kafbat Kafka UI*, formerly known as *Kafka UI* from Provectus. It's probably the most powerful open source UI for Kafka and supports not only topic inspection but also Kafka Connect, the Schema Registry, access control lists (ACLs), and monitoring consumer groups. Although its functionality may not match that of commercially available GUIs in some areas, we believe it's more than sufficient for most use cases.

Other known GUIs are usually available in a basic version for free but later become paid. Some of the most well-known are *Kpow*, *Kadeck*, or *Conduktor Console*. Of course, the providers Confluent and Aiven, as well as other cloud services, also offer GUIs for Kafka.

WARNING A GUI should never replace a good monitoring setup, as even the most powerful GUIs can only cover a portion of the necessary monitoring.

14.1.3 Managing Kafka resources

As mentioned several times throughout this book, from the preproduction phase onward, resources in Kafka must be managed in an automated and declarative manner. Under no circumstances should topics be created manually in production. We particularly advise against using GUIs for such operations. Instead, an automation solution is required.

Our preferred approach is the *GitOps* method. In GitOps, a Git repository is considered the source of truth, and changes are rolled out via a continuous deployment pipeline. We recommend this GitOps approach not only for Kafka but also for IT infrastructure in general. GitOps is a specific form of *Infrastructure as Code* (IaC).

In the case of Kafka, all desired topics, users, and ACLs are created in this Git repository, for example, through pull/merge requests, and after a manual or automated review, they are automatically created, modified, or even deleted in Kafka. Unfortunately, there's no one solution we can recommend for all cases. Based on our experience, most clients build such a solution independently. An additional challenge for a general solution is that the processes for resource approval differ from company to company.

However, the *kcctl* tool is available for managing Kafka Connect connectors. The source of inspiration for kcctl is the Kubernetes command-line tool *kubectl*. With kcctl, it's much easier to create and modify connectors than directly using the REST API. Connect configurations are stored in JSON files and can be created or modified using `kcctl apply -f file.json`.

14.1.4 Cruise Control for Apache Kafka

One of the primary tasks in Kafka operations is to keep the load of the brokers balanced. This means that all brokers should ideally have a similar number of partitions and leaders. Even more importantly, the resource consumption of the brokers should be similar. Therefore, CPU load, memory usage, and network load shouldn't differ significantly.

The approach to achieving this is to move partitions with the goal of distributing the load more evenly. In the early days, attempts were made to accomplish this manually. We've heard of very large Microsoft Excel spreadsheets being used to solve this problem. However, this isn't practical, especially for larger installations.

Fortunately, there is a better solution: *Cruise Control for Apache Kafka* from LinkedIn. Cruise Control for Apache Kafka consists of several components, as shown in figure 14.2. First, Cruise Control must monitor the individual brokers to create analyses from that data. This is the task of the *metrics* component, which must be integrated as a Java Archive (JAR) on each broker. This component writes the currently measured load values for the broker it's installed on into a specific Kafka topic.

Cruise Control contains an *analyzer* to which we can give different targets. The analyzer analyzes the metrics with respect to the targets and creates a proposal on how the targets can be met. This proposal is made available by Cruise Control for review and can then be executed by the *executor*.

As Kafka admins, we need to perform all of this manually via the REST

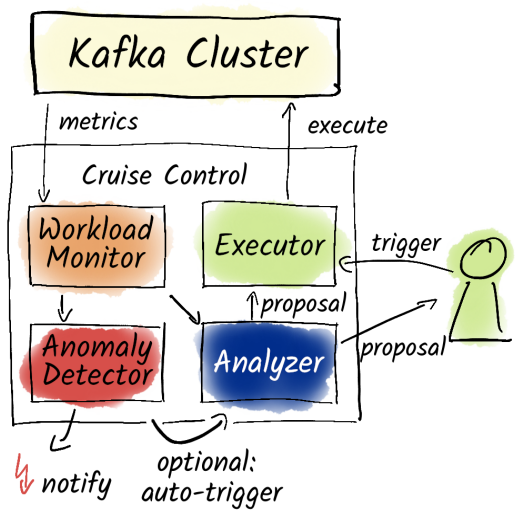


Figure 14.2 Cruise Control for Apache Kafka monitors and optimizes the load of a Kafka cluster.

API or the Cruise Control UI. We also recommend handling it this way at the beginning. If our monitoring detects that the load is unevenly distributed, we can run Cruise Control's analyzer, review the proposal, and then execute it.

Cruise Control also offers an automatic error detection mode. If it detects that targets aren't being met, Cruise Control automatically performs an analysis and executes it.

> **WARNING** Moving partitions across brokers can lead to very high load. Therefore, it's crucial to ensure that these operations don't occur during times when Kafka is operating at its limits.

In Cruise Control, we can set two essential goals for each of the CPU, RAM usage, disk, and network load resources: capacity goals and resource distribution goals. *Capacity* means we can set a fixed size, and Cruise Control tries to keep, for example, the disk utilization below this value. *Resource distribution* means that the load for this resource should be distributed as evenly as possible. Of course, there are many other configurable goals, which are described in the documentation.

Another important feature of Cruise Control is the ability to automatically distribute load across new brokers or prepare brokers for shutdown. By default, nothing further happens when a new broker is added. Cruise Control can solve this specific problem by calculating a better partition and leader distribution and then executing the proposed plan.

Conversely, when we want to downscale the cluster, meaning we want to remove a broker, we typically need to manually move partitions away from that broker before shutting it down. This is also quite tedious, and Cruise Control solves this problem as well.

14.2 Deployment environments

Kafka can be operated in a variety of ways. In our experience, we've seen everything from self-operated Kafka clusters to Kafka in Kubernetes, as well as fully managed Kafka as a Service (KaaS) solutions, where we don't even know how many brokers are available to us.

It fundamentally doesn't matter which infrastructure Kafka runs on. What matters is that the infrastructure is robust and there's a team that is knowledgeable about Kafka and can operate it. If a reliable infrastructure is unavailable or a team lacks the capacity to manage Kafka, our experience shows that it's often better to purchase a managed Kafka service in these cases.

However, it's also important to understand that Kafka doesn't operate itself. Even for a managed service, a team knowledgeable about Kafka is required to handle automations and the processes surrounding Kafka, and—most importantly—provide expertise to other teams. These tasks are rarely successfully outsourced to external service providers.

14.2.1 Kafka on a company's own hardware

The classic way to operate software is in one's own data center on dedicated hardware. We recommend automating all processes related to Kafka as much as possible, even in

one's own data center. This presents challenges in this use case but is achievable. We'll address specific hardware recommendations in the following sections, but note that careful planning of the Kafka deployment is essential, particularly in one's own data center.

How do we organize replacements for hardware failures? What do we do if there are additional resource requirements? Remember, in addition to production clusters, we should also set up Kafka in development and testing environments. The production cluster should never be used for testing or during development.

A suitable use case for this approach is a web analytics company with a skilled operations team and predictable workloads. With minimal reliance on Kubernetes and a data center infrastructure that includes limited virtualization, deploying Kafka on dedicated hardware is an excellent choice.

Predictable load patterns simplify capacity planning, and the lack of extensive virtualization ensures optimal performance. The operational team is also well-equipped to handle hardware failures and scaling needs, allowing for reliable and efficient Kafka deployments.

Another important point when operating in one's own data center is paying attention to the placement of the brokers within the data center. Kafka's performance partly relies on not persisting data to disk immediately. If multiple brokers fail simultaneously, this can lead to data loss. So, we should ensure that brokers aren't located in the same rack, or ideally, use different fire compartments to ensure better reliability along with rack awareness. An alternative is to distribute brokers across multiple data centers, but it's essential to pay attention to the latency between the data centers, which shouldn't exceed 30 ms.

14.2.2 Kafka in virtualized environments

In virtualized environments within one's own data centers, most of the same guidelines apply as when operating Kafka on physical hardware. It's important to ensure that the Kafka brokers are distributed as evenly as possible across virtual machine (VM) hosts. Additionally, network storage systems such as storage area network (SAN) systems are suitable for the use of Kafka in very few cases. If our SAN system is reliable and very performant, it may work well, but if there are any doubts, local SSDs should always be preferred.

Particularly for storage systems, it's crucial that they can handle the load that Kafka generates. We've seen cases where multiple brokers began compacting their data simultaneously, overwhelming the storage system.

This approach works well for organizations with established virtualization environments in their own data centers. For example, companies with predictable workloads and a strong operational team may find it convenient to run Kafka where other systems are already managed. By using existing expertise and infrastructure, these organizations can integrate Kafka into their virtualized environments while ensuring reliable performance for their data processing needs.

14.2.3 Kafka in Kubernetes: Strimzi

With the rise of Docker, and later, Kubernetes, we've seen more and more applications moving to Kubernetes clusters. When we first explored running Kafka on Kubernetes, many were skeptical due to the challenges of managing stateful applications at the time. However, today Kafka runs excellently in Kubernetes environments—though you should use recent Kubernetes versions and carefully consider your storage solution's performance. Based on our experience, we strongly advise against using GlusterFS or Network File System (NFS).

For organizations already using Kubernetes with a strong operational team and reliable infrastructure, running Kafka on Kubernetes is a highly effective option. In such cases, we recommend using *Strimzi*, a Kubernetes operator that automates many aspects of Kafka management and significantly reduces operational complexity. For enterprise support, Red Hat offers AMQ Streams, a version of Strimzi.

Strimzi essentially represents our reference architecture. It manages the setup of our Kafka cluster, including the coordination cluster (KRaft or ZooKeeper), as well as the configuration of other components such as Kafka Connect, monitoring, or automation with Cruise Control. Additionally, Strimzi provides excellent resource management, such as the administration of users and topics along with ACLs and various authorization options.

The core of Strimzi is the *Cluster Operator*. This operator manages the deployment of individual components such as brokers, coordination nodes, or other components such as Kafka Connect or the other two operators. Strimzi also provides two additional operators that can be used independently of Strimzi: the User Operator and the Topic Operator, collectively referred to as the Entity Operator. This allows us to create new topics or users with ACLs and generate authentication secrets with minimal effort. It even supports the creation and management of Transport Layer Security (TLS) key pairs. The components of Strimzi are shown in figure 14.3.

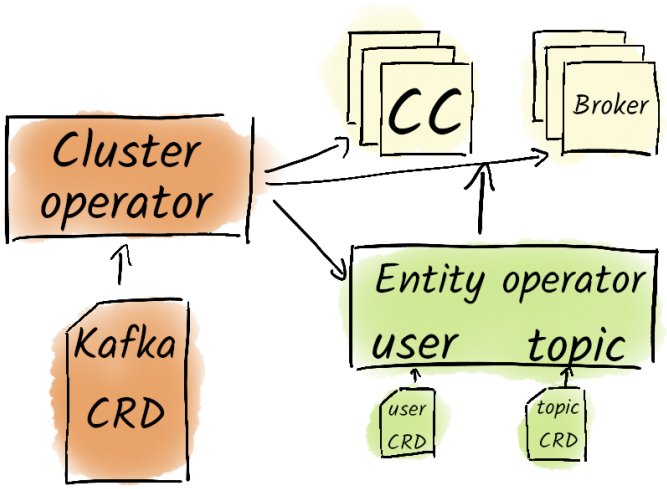


Figure 14.3 The architecture of Strimzi

If you have a reliable and performant Kubernetes environment, Strimzi is an excellent choice for managing Kafka yourself. Its architecture not only simplifies operations but also provides a robust framework for running Kafka at scale. If you can't use Strimzi directly, we recommend studying its design and incorporating its key patterns into your deployment.

> **TIP** A critical consideration when using Strimzi is ensuring that two brokers aren't scheduled on the same VM host or Kubernetes node. Strimzi allows you to enforce this separation through the use of anti-affinity rules, ensuring high availability and fault tolerance.

We won't go into further detail about Strimzi here, but you can find excellent documentation of this project at https://strimzi.io/documentation/.

14.2.4 Running Kafka in the public cloud

Although some cloud providers offer Kafka as a managed service, there are various reasons why it may be worthwhile to operate Kafka on your own in the cloud. Operating Kafka independently on one of the three major cloud platforms—Amazon Web Services (AWS), Microsoft Azure, or Google Cloud—generally works quite well based on our experience. We can at least be assured that the underlying infrastructure is reasonably reliable.

Although we initially mentioned that Kafka doesn't always perform well on SAN systems, we can give the all-clear with these offerings. The storage systems of these providers are both reliable and sufficiently performant for operating Kafka in most use cases.

> **WARNING** Choosing the cloud provider resources used to run Kafka can have a significant effect on data throughput, latency, and overall performance.

While cloud infrastructure simplifies the physical layer, Kafka's architecture in the cloud still requires careful planning. For organizations already using Kubernetes in the cloud, running Kafka with Strimzi can significantly streamline management and reduce operational overhead by integrating Kafka into the existing Kubernetes ecosystem.

It's even easier not to operate Kafka yourself but to use one of the numerous offerings for managed Kafka operations. Both *Amazon Managed Streaming for Apache Kafka* (MSK) and *Azure HDInsight* provide managed Kafka services. Compared to a self-managed Kafka, these services come with limitations and can become significantly more expensive than a self-operated cluster once a certain volume of data is reached. It's worthwhile to understand one's own requirements and align them with these offerings.

Customers who choose managed Kafka solutions often lean toward the more specialized services from Confluent Cloud or Aiven. We've had good experiences with both providers and can recommend them based on requirements and contract terms. Both, as well as other providers, run Kafka in the three cloud environments of AWS, Azure, and Google Cloud, and they also support VNet peering and Private Link. It's also

possible, upon request, to have Aiven manage Kafka in one's own cloud account, which is particularly useful for regulated environments.

Confluent's unique selling point is that clients don't need to worry about cluster sizes and can instead transparently book desired capacities. Both Aiven and Confluent Cloud also support the operation of Kafka Connect, Apache Flink, and other services useful in managing Kafka. Additionally, both providers are generally more adept at providing support for problems and questions related to Kafka.

> **WARNING** Managed Kafka services can become expensive as data volumes and usage increase. It's crucial to evaluate your needs and understand the pricing structure to avoid unexpected costs. Conduct a thorough cost-benefit analysis before committing to ensure the services fit your budget.

In summary, if your organization is already operating in the cloud, Kafka can easily be integrated into the environment, whether through self-management with Kubernetes or by using a managed service. The decision will depend on your team's expertise, your control requirements, and the specific needs of your organization regarding performance, scalability, and cost. Consider evaluating the limitations of managed services, such as retention periods or data throughput, and weigh them against the tradeoffs of operating Kafka yourself to make the best decision.

14.3 Hardware requirements

The performance and reliability of Kafka significantly depend on the underlying hardware used for its deployment. Proper hardware configuration is essential to ensure efficient data handling and processing.

This section outlines the key hardware requirements for Kafka brokers and coordination clusters, emphasizing the importance of resource allocation, storage solutions, and network performance. By adhering to these guidelines, organizations can optimize their Kafka environments to handle varying workloads while maintaining high availability and reliability.

14.3.1 Brokers

The brokers are the heart of Kafka as they store the actual data. We recommend starting with at least three brokers to ensure increased reliability. Depending on the size of the brokers and the load generated, additional brokers can be added gradually.

The necessary resources depend heavily on the requirements. Don't be discouraged by the official recommendations. We've successfully seen much smaller clusters in operation.

The focus for brokers should be on the performance and reliability of the storage and memory used. It's important to choose fast and dependable mass storage. In many cases, using multiple smaller SSDs is preferable to relying on a few larger ones, as this can improve both performance and resilience.

> **TIP** It's possible to start with less storage initially and extend it as needed. This approach allows for scaling resources in line with actual usage and demands, ensuring that additional storage is only invested in when necessary.

For example, the recommendation for large Kafka clusters is to work with 12 SSDs that are each 1 TB in size. Redundant Array of Independent Disks (RAID; except RAID 0, which splits data across drives to go faster) isn't needed because Kafka handles reliability through replication. Specifically, avoid RAID 5 and 6—they're too slow to be useful.

> **TIP** Since Kafka 3.9, Kafka supports tiered storage. This allows older data to be offloaded to cheaper storage such as object storage in the cloud or other storage systems. This makes it attractive and comparatively cost-effective to retain data in Kafka indefinitely.

When it comes to RAM, we must distinguish between the memory for the JVM heap and the memory available for the operating system's page cache. In the smallest configurations, it's possible to operate with less than 1 GB of RAM per broker, but we recommend providing at least 1 GB heap for the JVM. Even for very large clusters, Kafka usually doesn't require more than a 6 GB heap. The remaining available RAM can be used by the operating system as page cache.

We can calculate how much page cache we need. To do this, we need to consider how long consumers can lag behind without accessing the (slow) disk for data retrieval. Let's assume that our Kafka cluster with three brokers produces 30 GB of data per hour, and consumers can lag up to 2 hours without performance loss. We first need to estimate how much data is produced per broker, which here is 10 GB of data per hour, assuming an even distribution.

Because our consumers can lag up to 2 hours, we need at least 20 GB of page cache for that. This results in a total requirement of about 26 GB of RAM (6 GB for the heap, 20 GB for the page cache). With 32 GB of RAM per broker, the cluster is well-suited for this use case.

Another important point is a fast and reliable network. We recommend using at least 1 Gbit/s Ethernet. There are no upper limits. If network storage will be used, then we need a dedicated storage network; otherwise, performance losses can quickly occur.

The CPU is less important and usually depends on the RAM used. It's generally recommended to use smaller CPU cores rather than fewer large ones, as many operations in Kafka can be parallelized. Typically, memory-optimized instances are used for Kafka brokers. The required RAM is determined first, and the CPU is derived from that.

An important question is whether it's more sensible to have many small brokers or a few large brokers. We usually recommend finding a middle ground. If only a few very large brokers are used, the failure of a single broker carries significant weight. On the other hand, having too many brokers can make the overhead of the JVM non-negligible and management more challenging. We usually recommend scaling the brokers based on RAM, aiming to stay in the range of 32–64 GB. Smaller amounts of RAM make little

sense, and for larger amounts, we should add a few more brokers before increasing the RAM.

As previously mentioned, the physical separation of Kafka brokers is essential to ensure fault tolerance and data safety. Brokers shouldn't be hosted on the same VM or placed in the same rack, as this would increase the risk of data loss in the event of a power failure affecting the host or rack. By using Kafka's rack-awareness functionality, it's recommended to distribute brokers across different availability zones whenever possible.

However, distributing a single Kafka cluster across multiple cloud regions is generally not advisable. The latency between brokers should remain below 30 ms because brokers must communicate frequently to synchronize metadata, replicate partitions, and maintain cluster health. Higher latency between brokers can significantly degrade performance, leading to increased end-to-end latency and reduced reliability.

For global operations, it's typically more effective to deploy independent Kafka clusters in different regions rather than attempting to create one worldwide cluster. These regional clusters can be connected if necessary, using tools such as MirrorMaker to replicate data between them. This approach allows organizations to maintain high performance while minimizing the challenges associated with global synchronization. Replication, however, should only be employed when required, as it introduces additional complexity and overhead.

The decision to use separate clusters and replicate data across regions depends on specific requirements, such as regulatory compliance or cross-region data sharing. In most cases, a well-designed architecture with independent regional clusters provides better scalability, fault isolation, and performance while avoiding the pitfalls of high-latency synchronization.

14.3.2 Coordination cluster

Neither ZooKeeper nodes nor KRaft nodes require a large number of hardware resources. The most important thing is to ensure a good network connection, as ZooKeeper, in particular, is very sensitive to latency. The machines themselves don't need to be particularly large for coordination; 4–8 GB of RAM and an appropriate CPU are sufficient.

For the coordination cluster, we need an odd number of nodes. Usually, a cluster of three nodes is used for smaller environments, and five are used for larger ones. With three nodes, one node can fail without affecting availability. Five nodes can tolerate the failure of two instances.

Summary

- Kafka management can benefit from automated and declarative approaches, although it's not a strict requirement.
- Manual creation of topics in production is strongly discouraged, as is the use of GUIs for such tasks.
- The GitOps approach is recommended for managing Kafka resources by treating a Git repository as the source of truth.
- Kafka resources such as topics, users, and ACLs should be created and modified through pull/merge requests in the Git repository.
- Automated solutions, such as kcctl, simplify the management of Kafka Connect connectors compared to using the REST API directly.
- Cruise Control for Apache Kafka is essential for balancing broker loads by redistributing partitions based on resource consumption.
- Kafka administrators should monitor broker load and manually trigger Cruise Control's analyzer for load balancing.
- Automatic failure detection is available in Cruise Control, enabling proactive load balancing when targets aren't met.
- Kafka can be deployed on various infrastructures, including self-managed clusters, Kubernetes, and managed services.
- The performance and reliability of Kafka are contingent on a well-planned infrastructure and a knowledgeable team.
- It's recommended to start with at least three brokers to ensure reliability and allow for scalability based on workload.
- Storage solutions should prioritize reliable and fast disks, preferably multiple smaller SSDs over a few large ones.
- Kafka supports tiered storage, allowing older data to be offloaded to more cost-effective storage solutions.
- Sufficient RAM is critical, with a recommended minimum of 1 GB for JVM heaps per broker and an appropriate page cache based on data production rates.
- Network performance is vital for Kafka, with a minimum recommendation of 1 GBit/s Ethernet and the use of independent storage networks to avoid performance drops.
- The physical separation of brokers across different VM hosts or racks is essential to prevent data loss during hardware failures.

15 Kafka monitoring and alerting

This chapter covers

- Ensuring Kafka's performance and reliability
- Key metrics to track for Kafka
- Strategies for effective alerting
- Monitoring challenges in various Kafka deployment environments

Although Kafka is designed to be fault-tolerant and is therefore very robust against errors, it is, of course, not completely invulnerable. After all, Kafka runs on physical hardware, which can fail. While Kafka can easily compensate for the failure of individual brokers or, depending on the size of the cluster, even several brokers, we still need to ensure that such failures aren't prolonged and are resolved as quickly as possible. Otherwise, we risk the complete failure of our cluster because, naturally, with each broker that goes down, the remaining fault tolerance decreases.

So, what exactly do we mean when we talk about an error or failure? This refers to any impairment of full functionality, which in the worst case could mean the complete unavailability of the system. To respond appropriately, we need to understand where exactly the problem lies and what is causing it. Particularly in today's complex

IT systems, answering the latter question can often be difficult and may require lengthy detective work, as the root cause is often unexpected side effects from other IT systems that are otherwise functioning properly.

For the reliable operation of Kafka, or IT systems in general, good monitoring is essential. Monitoring helps us not only detect errors more quickly but also identify the root cause, enabling us to resolve the errors as swiftly as possible. In this chapter, we'll thoroughly cover how to monitor Kafka. This includes brokers, clients, and frameworks such as Kafka Connect and Kafka Streams, as well as which metrics we should pay particular attention to.

Appendix B provides a detailed guide on setting up an example monitoring system using Grafana and Prometheus, designed for hands-on testing of the metrics introduced in this chapter.

15.1 Infrastructure metrics

Even when discussing the monitoring of Kafka, we must not forget the hardware on which Kafka operates and the hardware resources consumed by Kafka. Our brokers or clients will quickly face performance or reliability problems without sufficient CPU, memory, storage, or networking capacity. Without comprehensive infrastructure monitoring, we may waste valuable minutes or hours troubleshooting because we mistakenly look for errors within Kafka itself, whether in the broker or the client.

Disk usage shouldn't be too high. During a rebalancing of partitions, we may need additional storage space, making it important to have enough buffer. Alerts are recommended for usage above 60% to ensure there's enough time to order new disks.

> **NOTE** The urgency of expanding storage also depends on how quickly it can be done; if it can happen within minutes, then the threshold may not need to be 60%.

Network utilization must also remain below 60%, as otherwise, we'll encounter problems moving data or partitions to new brokers when adding new resources or during the general rebalancing of partitions. Moreover, the failure of individual brokers could push network capacities to their limits.

We should also keep an eye on memory to ensure we have capacity for unexpected loads. Kafka actively uses the page cache and can operate well with less RAM, but performance will suffer as a result.

CPU load can also significantly affect our cluster. While short CPU spikes may only result in reduced performance, prolonged high utilization may prevent requests from being processed.

Here, we also see that a mere snapshot of a metric reveals very little about the state of our system. During partition rebalancing, the hardware resources consumed by brokers will naturally increase temporarily, which in most cases isn't a cause for concern. However, if CPU, memory, network, or storage remain high for an extended period, it's likely indicative of natural growth in our cluster or services, and we need to scale our Kafka cluster accordingly.

15.2 Broker metrics

Let's take a look at Kafka-specific metrics. We won't explain every single Kafka metric here, but rather focus on the most important ones. A comprehensive list of available metrics (without explanations) can be found in the official Kafka documentation.

Before we dive in, let's briefly examine the naming convention for Kafka metrics, which may vary depending on the monitoring system. Kafka follows the *Java MBeans* naming format, which uses a hierarchical structure to organize metrics. The names of our metrics consist of a domain, indicating where the metric originates, followed by a list of more specific attributes. For example, our brokers use the domains `kafka_server`, `kafka_log`, `kafka_network`, and `kafka_controller`.

Under these domains, there are various subtypes, ultimately leading to the precise name of the metric. Although this verboseness results in relatively long and cumbersome names, it also provides direct insight into the meaning of a metric without requiring extensive documentation.

To improve readability and clarity, we won't use the exact metric names in the following sections. Instead, we'll focus on describing the key metrics conceptually, highlighting particularly important ones as callouts throughout the text. Additionally, a table summarizing the most important metrics will be provided at the end of each section.

15.2.1 Kafka server metrics

One of the most crucial metrics is known as the `under-replicated partitions` metric, which indicates how many partitions have out-of-sync replicas. This value should always be zero. If it's consistently greater than zero, it suggests that a broker is offline. If this value fluctuates constantly, it indicates potential resource problems within the cluster.

Often, the load may simply be unevenly distributed, and we should address this as soon as possible. Indicators of load imbalance include the number of partitions (`partition count`) and the number of leader replicas (`leader count`) on a broker. These should normally be balanced across all brokers; if this isn't the case, a reassignment of the partitions to the brokers is recommended. In reality, some topics are significantly more active than others, meaning that production or consumption is above average. We can see how many producers are connected to a broker using the `producer ID count` metric.

The situation becomes critical when we have partitions where the minimum number of in-sync replicas (ISRs) is just met (`at minimum ISR partition count`) or even falls below it (`under minimum ISR partition count`).

> **NOTE** If the replication factor for a topic is set to one, it's expected that all partitions will have the minimum ISR count.

If this occurs, producers with `acks=all` are unable to produce. In the worst case, the leader of a partition is offline, making it impossible to write or read from it (`offline replica count`). Normally, in such failure cases, new leaders are appointed relatively quickly, which leads to an increase in the `reassigning partitions` metric on the

respective broker. Ideally, both of these metrics should be zero; if they are greater than zero for an extended period, it indicates serious problems within the cluster.

All the aforementioned metrics belong to the type `replica manager`, meaning the complete name for the `partition count` metric, for example, is `kafka_server_replicamanager_partitioncount`.

Under the type `replica fetcher manager`, we find the `max lag` metric, which displays the maximum lag between a broker or leader replica and all of its followers. While this metric is independent of follower, topic, or partition, the `consumer lag` metric of the type `fetcher lag` shows the lag for each follower, each topic, and each partition. Ideally, both values should be minimal and not exceed the maximum batch size during message production.

NOTE While a brief delay is normal, the lag shouldn't progressively increase.

Another significant group of metrics is of the type `broker topic metrics`. Here, we find numerous metrics related to individual topics. For example, there are various metrics for the number of different requests per topic (`produce`, `fetch`) and regarding the network traffic caused per topic (`messages in`, `bytes in/out`, `replication bytes in/out`, `reassignment bytes in/out`).

The names of these metrics directly indicate that the actual production of messages causes only a portion of the network load. Particularly, the reassignment of partitions can temporarily generate significant network traffic, as this affects many partitions in case of failure. All of these metrics are very helpful in identifying the cause of generally high load or even overload on our brokers, especially in ensuring that the load is distributed equally across them.

TIP When determining if a broker is overloaded, most Kafka metrics serve only as indicators. Therefore, we recommend also checking general metrics such as CPU or network usage.

Our brokers also collect various statistics regarding rejected messages and the reasons for rejection. For example, metrics are collected for records missing keys in compacted topics, records with incorrect Cyclic Redundancy Checks (CRCs), and records with out-of-sequence offsets.

TIP Producers do receive corresponding error messages for rejected messages, but depending on the quality of application monitoring, these can easily be overlooked. Developers would likely appreciate a heads-up.

Additionally, there are metrics of the type `delayed operation purgatory`. For example, the `purgatory size` metric indicates how many requests are currently waiting to be answered. These can include produce requests if the leader is still waiting for acknowledgments from follower replicas before confirming a message or fetch requests from consumers because `fetch.wait.max.ms` or `fetch.min.bytes` haven't yet been met.

15.2.2 Kafka log metrics

`Kafka log` metrics encompass all statistics related to our logs, which are the files stored on our brokers. This includes metrics for `log flush stats`, specifically a histogram regarding the time required to flush our messages and the number of flushes. High values in this context indicate a problem with our persistent storage.

The metric `offline log directory count` of the type `log manager` should always be `0`, as it indicates how many directories are unreachable. This may occur, for example, if a disk fails.

Additional metrics regarding the size, the number of segments, and the current offsets of partitions can be found under the type `log` (`size`, `number of log segments`, `log start offset`, `log end offset`). While these metrics may not be critical for setting up alerts, they provide valuable insights into how the load on the system develops over time. Metrics related to cleaning the cluster can be found under the types `log cleaner` and `log cleaner manager`.

15.2.3 Kafka network metrics

A very important `kafka network` metric is the `network processor average idle percent` metric of the type `socket server`. This metric indicates how many of our network processors are currently idle. If this value is too low, it can lead to delays in the network.

> **TIP** A consistently low value (`<0.3`) for the `network processor average idle percent` metric suggests that the cluster is on the verge of being overloaded or, in the worst-case scenario, is already overloaded. In this case, we need to scale our resources as quickly as possible to prevent potential performance problems or even outages.

Metrics of the type `request metrics` include general statistics (independent of the topic) for various requests, such as request, errors, request sizes, and various request times.

> **TIP** Monitoring these metrics is crucial for ensuring equal load distribution among brokers and preventing overloading.

15.2.4 Kafka controller metrics

`Kafka controller` metrics are used to monitor the current status of our controller or coordination cluster. Under the type `kafka controller`, various metrics provide insight into the state of our Kafka cluster. The metric `active controller` indicates which broker is currently acting as our controller, and its value should therefore only be `1` for one of our brokers (and `0` for the rest).

The metrics `active broker count` and `fenced broker count` are also very important, as they indicate how many brokers are currently active or inactive. Additionally, there are statistics regarding the total number of topics and partitions, as well as offline

partitions. The metric `preferred replica imbalance` count shows how many partitions per broker currently don't have their preferred broker as the leader replica.

> **TIP** Kafka regularly performs a rebalancing of the partitions when there are nonpreferred leaders for the partitions. The threshold for this can be set using the `leader.imbalance.per.broker.percentage`, which defaults to 10%.

Table 15.1 shows the most important broker metrics, and figure 15.1 shows a preconfigured Kafka dashboard from the Strimzi team with comprehensive information about the state of the Kafka cluster.

Table 15.1 The most important broker metrics

Java MBean Name	Description	Implications
`kafka.server:type=ReplicaManager, name=UnderReplicatedPartitions`	Number of partitions that have out-of-sync replicas	This metric should be `0`. If it's not `0` for a longer time, it indicates that a broker is offline.
`kafka.server:type=ReplicaManager, name=AtMinIsrPartitionCount`	Number of partitions where the minimum number of ISRs is just met	This isn't yet critical but could indicate some problems with replication. If the replication factor for a topic is set to `1`, it's expected that all partitions will have the minimum ISR count.
`kafka.server:type=ReplicaManager, name=UnderMinIsrPartitionCount`	Number of partitions where the minimum number of ISRs isn't met	This is critical as it prevents producers with `acks=all` from producing. Additionally, consuming is delayed.
`kafka.server:type=Fetcher-LagMetrics,name=ConsumerLag, clientId=([-.\w]+),topic= ([-.\w]+),partition=([0-9]+)`	The lag of the follower replicas per topic and partition	A brief delay is normal, but the lag shouldn't progressively increase. A constant lag indicates performance problems within the cluster and can affect the ISR, which can lead to producers not being able to produce new messages.
`kafka.network:type=Socket-Server,name=NetworkProcessorAvg-IdlePercent`	The percentage of currently idle network processors	The value should ideally be higher than `0.3`. Lower values indicate a potential overloading of the cluster, which can lead to performance problems.

15.3 Client metrics

While a Kafka cluster with its brokers undoubtedly forms the central core of a data-centric solution based on Kafka, we must not underestimate the crucial role of the clients. Even the most performant and reliable Kafka cluster is of no use without clients that continuously produce, process, or consume new data.

In this section, we'll delve into monitoring the diverse Kafka clients to achieve those last few percentage points of reliability. We'll not only examine producer and consumer

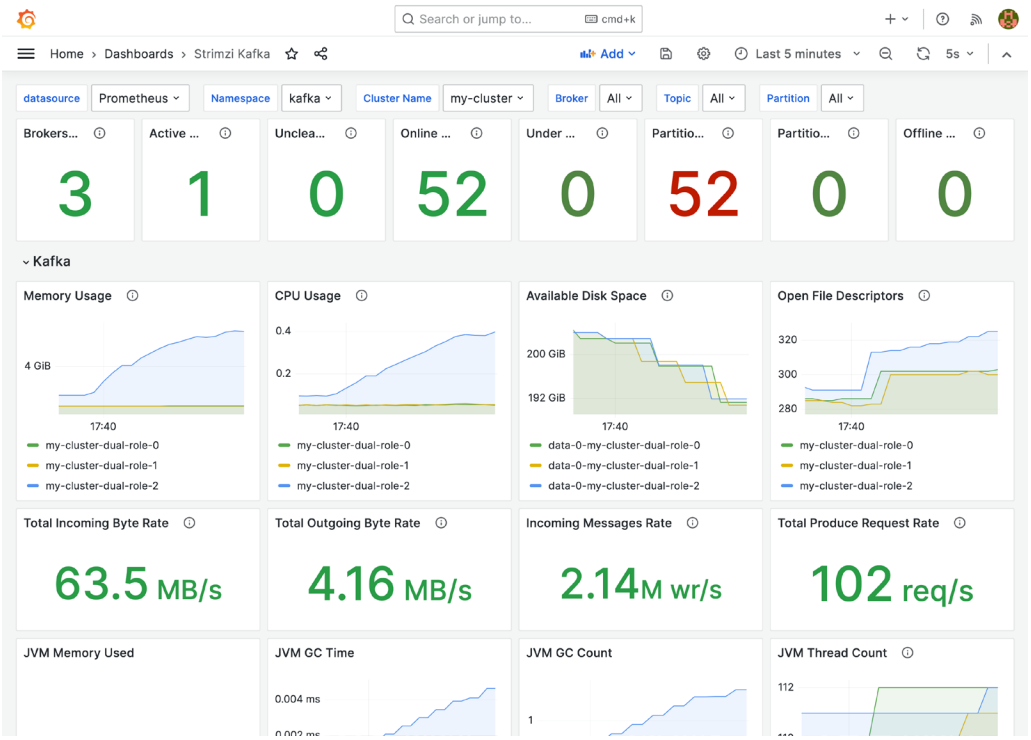


Figure 15.1 A Kafka dashboard example based on the Strimzi dashboards

metrics but also take a closer look at specific metrics for Kafka frameworks such as Kafka Connect and Kafka Streams.

15.3.1 General client metrics

Similar to monitoring our brokers, comprehensive monitoring of the infrastructure of our clients is crucial. Without sufficient hardware resources on the underlying machines, our clients will struggle to perform at their optimal level. A lack of CPU or network resources can lead to inefficient data processing, resulting in an inability to process all messages quickly enough.

This can cause producers, whether simple producers, Kafka Connect, or Kafka Streams, to fall behind, leading to outdated data in our Kafka clusters, which in turn affects all consumers of that data. When performance problems occur with individual consumers, usually only the respective application is affected. If there's insufficient memory, this can at best cause our clients to operate more slowly, but at worst can lead to repeated out-of-memory errors or crashes.

WARNING While problems with individual consumers may initially seem isolated, they can ultimately affect dependent applications as well.

In addition to infrastructure monitoring, some Kafka-specific metrics play a role that is independent of the type of client. Similar to our brokers, client metrics are assigned domains. In this context, the domains simply represent the type of client, whether it be `kafka producer`, `kafka consumer`, `kafka connect`, or `kafka streams`. The respective types are always `producer metrics`, `consumer metrics`, `connect metrics`, or `stream metrics`, although Prometheus omits the type. All client metrics can be associated with their respective clients using the corresponding `client ID` label.

There are also non-client-specific metrics that relate to aspects not specific to any particular client. These include metrics for the number of connections to the Kafka cluster or the success or failure of authentication.

The number of connections a client has depends on the number of partitions and brokers it communicates with and should generally remain relatively stable, ideally evenly distributed among the individual consumers in a consumer group. For example, if the number of connections from consumers frequently changes, it indicates frequent rebalancing within the respective consumer groups. This, in turn, could signal problems with individual consumers. It's important to note that during rebalancing, no data can be consumed, which causes the data in the applications to become outdated.

Failed authentications can indicate problems within our authentication system, aside from incorrect credentials, and should therefore be promptly investigated and resolved. These metrics are crucial for ensuring the security and integrity of the Kafka cluster and for detecting potential attacks or configuration problems early.

Additionally, there are—similar to brokers—numerous metrics related to requests and network traffic. However, it's challenging to make general statements about what should be considered too high or too low, as these values heavily depend on the specific requirements and usage patterns of the applications.

Nevertheless, they are excellent for anomaly detection. Particularly temporarily high values in these metrics can cause short-term delays in data processing. The aforementioned metrics are also available at the broker level and carry the prefix `node` before the metric name. This allows monitoring the performance and traffic of individual brokers, enabling the early detection and addressing of potential bottlenecks or overloads. Table 15.2 shows the most important general client metrics.

> **WARNING** Early detection of anomalies allows us to identify irregularities in our applications before they lead to serious problems or failures.

15.3.2 Producer metrics

Let's take a closer look at our producers. Many of the `producer metrics` have equivalent or similarly significant metrics on the broker side. For example, the metric `record error rate` allows us to determine if our brokers have rejected a produce request. Such errors can easily prevent new messages from being produced because we're blocked at this point. Therefore, it's extremely important to quickly identify and resolve the cause.

Table 15.2 The most important general client metrics

Java MBean Name	Description	Implications
`kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+),name=connection-count`	The number of connections a client has to the Kafka cluster	A frequent change of this value may indicate potential performance problems, especially for consumers as this can be an indicator for a high number of rebalances, which can further increase potential performance problems.
`kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+),name=failed-authentication-total`	The number of failed authentications a client has to the Kafka cluster	This could either be a configuration problem on the client (wrong credentials/connection configurations) or a problem with the authentication system of the Kafka cluster.
`kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+),name=request-rate`	The rate of requests (e.g., produce or fetch) a client sends to the Kafka cluster	This metric depends strongly on the application/client. Monitoring that metric can help detect anomalies in client behaviors, which are very often caused by malfunctioning applications.
`kafka.[producer|consumer|connect]:type=[producer|consumer|connect]-metrics,client-id=([-.\w]+),name=outgoing-byte-rate`	The outgoing (there's also one for incoming) byte rate of the client	This metric depends strongly on the application/client. Monitoring that metric can help detect anomalies in client behaviors, which are very often caused by malfunctioning applications.

A common reason for such errors is that the maximum batch size of our producer doesn't match the maximum configured message size of our brokers. Other problems may include temporary network disruptions, overloads, or even that the relevant broker is completely offline. The exact cause of such errors is usually found in the error log of our producer. It's crucial to proactively monitor and address these types of problems to ensure that everything runs smoothly and data can be reliably sent to the Kafka cluster.

> **NOTE** If a broker is offline, the leader for the partition will be reassigned automatically, and the producer will connect to the new leader.

Through the metric `requests in flight`, we gain insights into the number of pending requests that haven't yet been answered by our brokers. Particularly when using `acks=all`, there can be temporary delays in acknowledgment.

Occasionally, requests from our brokers may be delayed due to overload or quotas. These delays can be identified using the metric `produce throttle time max`.

In addition, we have statistics available that provide information on the size, number, latency, and compression rate of our messages. These metrics can be used for anomaly

detection. Some of these metrics are also topic-specific and carry the prefix `topic`. Table 15.3 shows the most important producer metrics.

TIP The metrics `record send rate` and `request latency avg` are especially important for monitoring the performance of our producer.

Table 15.3 The most important producer metrics

Java MBean Name	Description	Implications
`kafka.producer:type=producer-metrics,client-id="{client-id}",name=record-error-rate`	The error rate in producing records for the producer	Normally, there shouldn't be any errors during production. This metric very likely indicates serious problems that could either be caused by our brokers or the producer itself (e.g., no key for compacted topics or message sizes increasing the configured maximum).
`kafka.producer:type=producer-metrics,client-id="{client-id}",name=produce-throttle-time-avg`	The delay in message production introduced due to throttling	This metric can help detect performance problems. A common reason is also that the production rate is throttled because of quotas.
`kafka.producer:type=producer-metrics,client-id="{client-id}",name=request-latency-avg`	The average request latency. There are also similar metrics for subtypes.	This metric can help detect performance problems. A sudden increase of this metric can indicate performance problems in our cluster.

15.3.3 Consumer metrics

After reviewing the key producer metrics, we now turn our attention to our consumers. The metrics `time between poll max` and `last poll seconds ago` provide insights into when our consumer last queried new messages from our brokers.

The frequency of these queries depends on the corresponding settings in our consumers, such as `poll.interval.ms`, as well as the load on our consumers. It becomes critical when the load causes our queries to slow down, potentially reaching the limit set by `max.poll.interval.ms`. This results in the consumer being considered inactive from the perspective of the consumer group, triggering a rebalance of the entire group.

TIP Monitoring the metrics `time between poll max` and `last poll seconds ago` can help identify potential performance problems early, before they lead to disruptions within the consumer group or application. In the short term, options such as adjusting `max.poll.interval.ms` or increasing available hardware resources can be considered. In the worst-case scenario, it may even be necessary to rethink and adjust the architecture of our application to achieve better scalability. Thus, it's crucial to recognize such problems in a timely manner and take appropriate action.

Under the type `coordinator metrics`, we find various statistics related to the management of consumer groups. Notably, the metrics regarding rebalancing are of particular interest. The number of rebalances can be observed under `rebalances total`. It becomes critical when rebalances fail (`failed rebalances total`), and the time required for rebalancing can also be significant (`rebalance latency total` and `rebalance latency max`). While a certain number of rebalances may be normal depending on the setup, a high value can potentially indicate problems within our consumer group. It's important to note that depending on the partition assignment strategy, consumer groups can't consume new messages during rebalancing.

TIP If we set the strategy to `CooperativeStickyAssignor`, other consumers in our consumer group can continue to consume messages seamlessly.

Depending on the application and the topics to be consumed, the time required for rebalancing can vary significantly.

TIP In more complex applications, such as Kafka Streams, a rebalance can sometimes take several minutes. Therefore, it's essential to carefully monitor the rebalance metrics to identify and address potential problems in the application promptly.

Another valuable metric is `assigned partitions`. Ideally, the partitions within a consumer group should be evenly distributed. It's particularly concerning if a consumer doesn't receive any partitions for an extended period, as this may indicate a possible undetected error. Additionally, metrics are available that measure the number of commits (`commit total`) and their latency (`commit latency`). However, these metrics are primarily intended for anomaly detection.

The actual metrics related to consuming messages are found under the type `fetch manager metrics`. One of the most important metrics for consumers is the consumer lag.

Lag indicates how far a consumer is behind the current state per topic and partition. We can express lag in terms of the number of messages (`records lag`) or the time between the last message read by the consumer and the newest message in the partition. There's also the metric `records lead`, which indicates the lead of a consumer regarding the oldest message within a partition. Furthermore, there are various topic-specific metrics for consumed records or bytes. Table 15.4 shows the most important consumer metrics.

WARNING While a consumer lag (`records lag`) merely indicates that a consumer can't consume data quickly enough, a low `records lead` poses the risk that messages from the Kafka cluster may be deleted due to log compaction or log retention before they can be consumed. Because the normal values for `records lead` strongly depend on the size of our topics, it's difficult to set a universal threshold here. We recommend monitoring `records lead` in combination with `records lag` or setting individual thresholds per topic.

Table 15.4 The most important consumer metrics

Java MBean Name	Description	Implications
`kafka.consumer:type=consumer-metrics,client-id=([-.\w]+),name=time-between-poll`	The time between polls	Consumers normally (depending on the implementation) fetch new messages as soon as the processing of the previous fetched messages is finished. An increasing value can indicate a starting scalability problem (e.g., database becomes slower as it grows). Consumers are counted as inactive if they can't poll new messages in at least the maximum configured time between polls, which then causes rebalances in the consumer group and increases the performance problems. Monitoring this metric can help detect potential performance bottlenecks before they occur.
`kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+),name=rebalance-rate-per-hour`	The rate of rebalances	While a certain number of rebalances is no reason to worry (especially in Kubernetes environments), a constant high value indicates problems with the consumer group.
`kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+),name=commit-latency-max`	The average request latency. There are also similar metrics for subtypes.	This metric can help detect performance problems. A sudden increase of this metric can indicate performance problems in our cluster.
`kafka.consumer:type=consumer-fetch-manager-metrics,partition="{partition}",topic="{topic}",client-id="{client-id}",name=record-lag`	The consumer lag for a topic/partition	This is the most important consumer metric as this directly indicates performance problems. This metric should ideally be `0` or near `0` if there's a high production rate. A permanently high value indicates a performance bottleneck of the consuming application and can have a significantly negative effect on the application because the consumer isn't up-to-date with the latest messages.
`kafka.consumer:type=consumer-fetch-manager-metrics,partition="{partition}",topic="{topic}",client-id="{client-id}",name=records-lead`	The number of messages a consumer is ahead of the oldest messages in a topic/partition	This metric is only useful in combination with the consumer lag as the value highly depends on the number of records in a topic/partition. If the value is very low and the lag is high, then there is a potential risk of losing data (depending on the topic cleanup policy and configuration), as records could be cleaned up before the consumer was able to read them.

15.3.4 Kafka Connect and Kafka Streams metrics

After learning about the essential monitoring of our standard Kafka clients, let's take a brief look at the specific Kafka Connect metrics. Kafka Connect provides various types of metrics. Within the `connect worker metrics`, we find general information about the state of our Connect cluster, such as the number of connectors and tasks per worker. Additionally, we have connector-specific metrics that provide information about the number of tasks per connector and their status.

In the `connector metrics`, we find general information about each individual task, while the `connector task metrics`, `source connector task metrics`, and `sink connector task metrics` offer detailed information, similar to our consumer and producer metrics.

It's also noteworthy that there are `task error metrics` containing information about errors in our connectors or tasks such as the number of errors encountered or how often records were written to the dead-letter queue.

> **WARNING** We recommend carefully monitoring at least the status of the connectors. However, it should be considered that connectors may encounter temporary problems due to dependencies on third-party systems. Although the configuration of the connectors allows for control over the retry behavior, which is undoubtedly beneficial for managing temporary problems through automatic retries, it's crucial to remember that increasing the retry intervals also extends the time before a connector or task transitions to the failed status.

Finally, let's examine the monitoring of Kafka Streams. These metrics provide insights into the performance and behavior of your Kafka Streams application, enabling early detection of potential problems and optimization of the application. In addition to general metrics such as the number of running and failed threads, there are various subtypes of metrics. In the thread-specific metrics, found under `thread`, we receive general information about each thread, such as the number of tasks created or closed.

In the normal operation of a Kafka Streams application, apart from starting, stopping, or topology changes, the termination and recreation of tasks usually indicates errors or rebalancing. If this occurs too frequently, it's advisable to analyze the application more closely. Moreover, we can find information about the number of processed records and their processing duration, as well as the number of consumed messages and the latency required for consumption.

The `task` metrics offer detailed information about the individual tasks within the respective threads. Many of the available metrics resemble the thread metrics but are collected at the task level. Additionally, they contain information about unprocessed messages (`dropped records total`), which is critical to monitor.

Furthermore, there are metrics for the individual processor nodes (`processor node metrics`), which provide detailed information about the abstract Kafka Streams processors within a task. These metrics include, for example, data produced or consumed, the number of processed messages, and the average time taken to process messages.

Finally, under `state`, there are numerous metrics related to the individual state stores in Kafka Streams. For instance, `RocksDB` provides information about the number or size of entries in the memory tables. Table 15.5 shows the most important Kafka Connect and Kafka Streams metrics.

Table 15.5 The most important Kafka Connect and Kafka Streams metrics

Java MBean Name	Description	Implications
`kafka.connect:type=connect-worker-metrics,connector="{connector}",name=connector-failed-task-count`	The number of failed connector tasks	This metric indicates serious problems as a stopped task means that no messages are processed by the connector. The problem could be temporary (e.g., database maintenance), meaning that the task just needs to be restarted or could be caused by a processing error.
`kafka.streams:type=stream-metrics,client-id=([-.\w]+),name=failed-stream-threads`	The number of failed stream threads	This metric indicates serious problems as a failed stream thread means that the processing of messages is stopped.
`kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+),name=process-rate`	The rate of processed records	This metric depends strongly on the application/client. Monitoring that metric can help detect anomalies in client behaviors, which are very often caused by malfunctioning applications.
`kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),name=dropped-records-ratel`	The rate of dropped records	This metric indicates critical problems as records couldn't be correctly processed.

15.4 Alerting

After thoroughly learning which aspects and metrics to monitor in Kafka, we'll provide some general tips on alerting and handling notifications that don't apply only to Kafka. An *alert* is an automated notification based on predefined thresholds of metrics or aggregations of metrics.

Alerting plays a crucial role in the reliable operation of IT systems as it enables automatic monitoring of a wide range of metrics and informs responsible parties about anomalies or failures. In some cases, appropriate actions can even be taken automatically to minimize service disruptions. Ideally, through an alert, we can detect potential problems early and thus prevent a complete failure.

15.4.1 From metrics to alerts

Fortunately, we don't need to create an alert for every metric. This would be extremely labor-intensive as well as likely counterproductive. While most metrics help identify the root causes of errors, not all metrics are suitable for alerts.

Although it's relatively easy to create alerts for metrics that have only two states (good or bad), it's often more complex for other metrics. Visualizing these metrics can significantly help identify relevant metrics or aggregations and establish meaningful thresholds for alerts.

For example, if we find that a metric exhibits strong second-to-second fluctuations, it may make sense to smooth it over a larger time frame. One example is CPU utilization,

which can briefly reach 100%. However, as long as this state doesn't persist over a longer period (several minutes or more), there's likely no cause for concern and certainly no need to notify someone in the middle of the night.

Once we've identified suitable aggregations for our metrics, we need to set meaningful thresholds at which alerts should be triggered. It's advisable to take a conservative approach at first and adjust the thresholds in the event of frequent false alarms. Many IT systems now have self-healing capabilities. For instance, Kubernetes can automatically restart a container in case of a failure. Therefore, it's wise to allow the systems some time to self-repair before we take emergency measures. Of course, we should later review the error and not simply ignore it.

If a metric appears unsuitable for an alert after visualization, it doesn't necessarily mean that the already created dashboard or panel must be discarded. Often, these metrics can still help identify the root cause of other alerts. For instance, consider the `produced records` metric. Alone, it's usually not a cause for concern. However, if an alert is triggered that the consumer lag of a consumer group is too large, it can help determine whether there's a general problem with the consumers or if we simply have an unusually high number of records to process.

> **TIP** We've primarily outlined suitable metrics for alerting in the preceding callouts and tables throughout this chapter.

Some systems even automatically generate alerts for metrics by establishing thresholds based on the metric's past behavior. The problem with this is that such automatic alerts are often prone to false alarms. This can lead to these alerts being completely ignored in the long term, resulting in real problems being overlooked. Fortunately, most of these systems allow fine-tuning of alerts, meaning the thresholds can be adjusted. It's important to make these adjustments and regularly review and update them as needed.

> **WARNING** If we configure our alert systems to be excessively sensitive, leading to numerous false positives, it can negatively affect our responses to future alerts, potentially causing us to ignore them. This phenomenon is known as *alert fatigue*.

15.4.2 From alerts to problem solving

We should avoid going into panic mode when an alert is triggered and a service or system isn't functioning as expected, especially when waking up in the middle of the night. In such situations, it's crucial to have clear processes in place to efficiently resolve the problem while keeping a cool head.

A best practice is to enrich our alerts with additional information. In addition to the basic information about the relevant metric, we should attempt to contextualize the alert. This might include a brief description of the alert's significance, as the expert for the service may not always be immediately available.

Often, team members who don't regularly deal with the service may need to troubleshoot errors. Links to relevant dashboards can help quickly identify the

underlying cause, and alert playbooks can even include step-by-step troubleshooting guides, although this isn't always feasible. Such playbooks are living documents and should be regularly updated, especially after incidents, with new information. In some cases, it's possible to implement automatic alert handlers that trigger actions such as automatic restarts or scaling of our services.

15.5 Kafka deployment environments and their monitoring challenges

The environment in which Kafka is deployed plays a significant role in shaping the operational challenges and monitoring requirements. Whether Kafka is running on bare metal in your data center, on virtual machines (VMs), in the cloud, or as a managed service, each setup introduces unique risks that must be addressed to ensure stable performance and reliability.

15.5.1 Kafka on a company's own hardware

In traditional on-premise deployments, Kafka is hosted on physical hardware within a company's own data center. This setup offers full control over the hardware and network infrastructure but also comes with specific challenges. Hardware failures, such as disk crashes, power outages, or network problems, can have a direct effect on Kafka's operation.

The responsibility for hardware maintenance, infrastructure resilience, and recovery procedures falls entirely on the organization. In these cases, monitoring must cover not only Kafka but also the underlying hardware components. Disk health, CPU utilization, memory consumption, and network bandwidth are critical metrics to track to prevent bottlenecks or failures. Redundancy, backups, and disaster recovery plans are vital to mitigate the risk of data loss or prolonged downtime.

15.5.2 Kafka on virtual machines

When Kafka is deployed on virtual machines, often within on-premise or private cloud environments, there's an additional layer of abstraction due to virtualization. VMs provide flexibility and scalability but also introduce complexities such as resource contention between different VMs hosted on the same physical server. The hypervisor allocates CPU, memory, and I/O resources to multiple VMs, and any resource contention can lead to performance degradation in Kafka.

Monitoring VM deployments requires attention to both the application level (Kafka brokers and clients) and the virtualization layer (hypervisor performance, resource allocation, and potential contention). Tracking metrics such as CPU steal time, disk I/O latency, and memory swapping becomes crucial to ensure that Kafka's performance isn't affected by other workloads running on the same physical host.

15.5.3 Kafka in the public cloud

Cloud deployments offer elasticity and flexibility, allowing Kafka to scale more easily with fluctuating workloads. However, cloud environments come with their own set

of operational challenges. While physical hardware management is abstracted away, Kafka operators must deal with the complexities of distributed systems in a shared, multitenant infrastructure. Cloud providers may experience network latencies, instance performance variability, or regional outages that could affect Kafka's performance.

In the cloud, monitoring should focus on network latency, throughput, and autoscaling behaviors to ensure that Kafka can handle sudden spikes in load. Additionally, cloud-specific metrics such as instance health, storage performance, and inter-region traffic are important to track. The ephemeral nature of cloud instances also necessitates reliable mechanisms for data replication, backup, and recovery in case of node failures.

15.5.4 Kafka in Kubernetes

When deploying Kafka in Kubernetes, effective monitoring is essential for ensuring cluster performance and reliability. Key metrics should be collected via Java Management Extensions (JMX), including broker health indicators, producer and consumer metrics, and topic-specific data. Using tools such as Prometheus for scraping metrics and Grafana for visualization allows operators to gain comprehensive insights into Kafka's operation.

Additionally, monitoring Kubernetes resources such as pod health, node resource usage, and overall cluster performance is crucial to prevent problems related to resource contention. Implementing alerting mechanisms based on critical thresholds enables proactive management, ensuring timely responses to potential disruptions.

15.5.5 Kafka as a managed services

Using a managed Kafka service, such as Confluent Cloud or Amazon Managed Streaming for Apache Kafka (Amazon MSK), offloads much of the operational burden to the service provider. Infrastructure maintenance, scaling, and upgrades are handled by the provider, reducing the need for hands-on hardware or system management.

However, while the service provider manages the underlying infrastructure, operators still bear responsibility for monitoring Kafka's performance and ensuring application-level reliability. Key metrics such as partition distribution, message throughput, consumer lag, and latency remain critical.

Additionally, there may be service-specific limits and quotas (e.g., instance size, network bandwidth) that need to be monitored to avoid throttling or disruptions. Understanding the service-level agreements (SLAs) and how they align with business requirements is essential to ensuring that the managed service meets the necessary uptime and performance criteria.

15.5.6 Security considerations across environments

Regardless of the deployment environment, security is a critical consideration. Kafka clusters are often exposed to internal and external threats, and securing them requires diligent monitoring of access controls, authentication, and encryption protocols.

In on-premise environments, Kafka operators need to enforce security policies at the network and hardware level. In cloud and managed service environments, while

infrastructure security is handled by the provider, it remains essential to monitor the security of Kafka brokers, topics, and data in transit. Metrics related to failed authentication attempts, unauthorized access, and encryption overhead should be monitored to detect potential security breaches or vulnerabilities.

Summary

- Monitoring Kafka is vital for optimal performance and reliability.
- Infrastructure monitoring is crucial as performance problems can stem from insufficient CPU, memory, storage, or network capacity.
- Disk and network utilization should stay below 60% for effective partition rebalancing and to prevent data movement problems during broker failures.
- Contextualizing metrics is important; prolonged high resource usage may indicate a need for scaling rather than just temporary spikes.
- Kafka broker metrics, categorized under domains such as `kafka_server` and `kafka_log`, clarify their origin and significance.
- Key metrics such as under-replicated partitions and under-minimum ISRs should ideally be `0` for broker health and cluster reliability.
- Metrics related to request types and rejected messages help identify bottlenecks, emphasizing balanced load distribution among brokers.
- Monitoring network processor metrics is essential; a low average idle percentage can indicate broker overload, requiring immediate resource scaling.
- Client metrics are critical because even a strong Kafka cluster relies on clients for data production and consumption.
- General client metrics focus on infrastructure health, as CPU and network limitations can lead to processing delays affecting data reliability.
- Producer metrics such as `record error total` and `request latency avg` are vital for identifying message production problems.
- Consumer metrics, including `consumer lag` and `rebalances`, assess consumption efficiency and detect potential slowdowns.
- Kafka Connect metrics provide insights into connector health, highlighting the need to monitor dependencies on external systems.
- Kafka Streams metrics, such as processed records and processing duration, offer insights into application performance for early problem detection.
- Alerts are automated notifications based on predefined thresholds to detect anomalies early and reduce service disruptions.
- Avoid alert fatigue by setting meaningful thresholds and smoothing metrics to ensure alerts reflect real problems.
- Enhancing alerts with contextual information aids in efficient problem-solving during high-pressure situations.

- Deploying Kafka on physical hardware provides control but requires monitoring of system components such as disk health and CPU utilization.
- Kafka on VMs introduces complexities from resource contention; monitoring should cover both Kafka and the virtualization layer.
- Cloud environments offer scalability but pose challenges such as network latencies and instance performance variability; monitoring should focus on network throughput and instance health.
- Monitoring Kafka in Kubernetes involves using tools such as Prometheus and Grafana while tracking resource usage, with alerting based on critical thresholds.
- In managed Kafka services, while infrastructure is handled by providers, monitoring application-level metrics such as consumer lag and message throughput is essential for meeting SLAs.

16 Disaster management

This chapter covers

- Failure mitigation strategies in Kafka
- Data loss risks and strategies for fault tolerance
- MirrorMaker's synchronization of topics and access control lists
- Architectures for improved disaster recovery

In this chapter, we delve into the crucial topic of disaster management in Kafka. When disaster strikes and systems fail, the consequences can be severe: financial losses from interrupted business operations, damaged customer relationships due to service outages, and potential regulatory compliance violations in case of data loss.

Organizations may also suffer lasting reputation damage if they can't quickly recover from failures. These business risks make effective disaster management essential. We'll explore how to mitigate various types of failures, whether they stem from network problems, compute failures, or persistent storage problems, to ensure both system reliability and data integrity.

Understanding the nature of potential disasters allows organizations to differentiate between critical and less critical scenarios, enabling them to prioritize their strategies effectively. We'll explore the various options available for disaster recovery, including the limitations of traditional backup strategies and the need for robust solutions that minimize data loss and inconsistencies.

Additionally, we'll discuss advanced architectures such as stretched clusters and MirrorMaker, which enhance fault tolerance and streamline recovery processes. By understanding these concepts, organizations can develop a comprehensive approach to disaster management that not only safeguards their Kafka environments but also supports overall operational resilience.

16.1 What could possibly go wrong?

Figure 16.1 displays the key components and potential failure points we need to consider when operating Kafka. In a distributed system such as Kafka, we must account for various types of infrastructure failures: network problems that affect message transmission between components, compute failures that affect our brokers, storage system problems that threaten our persistent data, and data-center-wide failures that can bring down entire facilities.

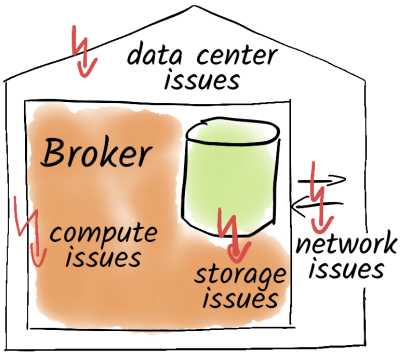


Figure 16.1 We need to consider many different failure cases when operating Kafka or other distributed systems.

Beyond infrastructure concerns, human factors introduce an entirely different dimension of operational risk. Misconfigurations, maintenance errors, and inadequate operational procedures can trigger or amplify technical failures, making it essential to implement robust operational safeguards and well-documented procedures to minimize both technical and human-induced failures.

16.1.1 Network failures

Let's first consider network failures. In a distributed system such as Kafka, it's almost guaranteed that network problems will occasionally arise. Two common scenarios deserve particular attention. First, network problems between clients and brokers can prevent applications from sending messages to Kafka. Second, inter-broker network problems can disrupt the communication between Kafka nodes themselves, potentially affecting replication and cluster stability.

When clients can't reach brokers, messages will start queuing up in the producer. Little can be done from Kafka's perspective except wait for network connectivity to be restored. The real challenge here lies in how producers and consumers should handle such situations. For the consumers, this is straightforward, they simply wait until the network problem is solved and then can consume messages as before. For producers, it's up

to the developers to decide whether to block, buffer, or even start dropping messages. Another valid option is to let the producer crash and signal the user that the system is currently unavailable. Although this isn't desirable, sometimes it's the simplest solution.

Inter-broker network problems are particularly challenging as they can affect Kafka's replication mechanism, leadership election, and even controller problems. If brokers can't communicate with each other, the cluster may initiate unnecessary leader elections or fail to maintain the desired replication factor. While Kafka will attempt to maintain cluster stability, the primary mitigation strategy is again waiting for network connectivity to be restored.

Generally, while network problems related to Kafka are disruptive, they are usually resolved relatively quickly. Sometimes they resolve themselves, as the link may have only been temporarily overloaded, or the network problem was outside our scope of responsibility. The good news is that due to Kafka's asynchronous architecture and persistence model, messages can simply be produced or consumed again once network problems are resolved, meaning we typically don't need to fear data loss or inconsistent states. Network problems become particularly problematic when dealing with real-time critical applications that can't tolerate even brief interruptions in message flow.

The key to handling network failures lies in proper monitoring and alerting to detect problems quickly, maintaining good relationships with network engineering teams, and having clear operational procedures for various network failure scenarios. However, from Kafka's perspective, the actual mitigation options during a network failure are limited, making proper network infrastructure and redundancy essential for production deployments.

16.1.2 Compute failures

Consider this common scenario. We're running a large-scale payment processing system handling millions of transactions per hour. Suddenly, one of your brokers becomes unresponsive during peak hours. While this situation might sound alarming, Kafka's architecture is designed to handle such failures gracefully, but only if configured correctly.

While Kafka can tolerate some broker failures, the actual tolerance threshold is determined by the topic's `min.insync.replicas` configuration relative to the replication factor. Keep in mind, that this setting only works together with the `acks=all` strategy of the producers.

To simplify the next discussion, let's assume that the replication factor is the same as the number of brokers. With a replication factor of three and `min.insync.replicas` set to two for all topics, our cluster can lose one broker without any effect on producers and consumers. As soon as another broker is lost, producers with `acks=all` can't produce to the cluster anymore, but consumers aren't affected as long as at least one replica (the leader) is available. Another critical problem is that now all the load from the clients is routed to the single remaining broker. Having more than three brokers will reduce the load on the individual brokers.

In some mission-critical scenarios, you might want to improve the reliability of Kafka even further. For this, you could increase the replication factor (and thus the number of brokers) to, for example, four and keep `min.insync.replicas=2`. With this configuration, the cluster can survive the failure of up to two brokers without any effect on producers and consumers. After this, one more broker can fail without affecting the consumers.

> **WARNING** We need to keep in mind that increasing the replication factor increases the cost of a Kafka cluster significantly and only protects against broker failures, not more likely failures such as human mistakes. If you need additional reliability, consider a multi-cluster setup.

The recovery from a broker failure depends on the circumstances that lead to the failure. In general, we just need to make sure that the broker comes back online. If it was a temporary failure, this often happens automatically, or we need to start the broker again manually.

If the broker fails permanently, for example, because of a hardware failure, we install new hardware, install Kafka, and then configure it in the same way as the failed broker. The critical setting is the broker ID. It needs to be the same as the ID of the failed broker. As soon as a broker is recognized by the controller, and it has a known ID but no data, the broker will start replicating the missing data from the leaders of the partitions and, after some time, will become in sync and the leader for which the failed broker was the preferred leader.

> **TIP** Kafka identifies its brokers only by the broker ID. When a broker with an empty disk but a previously used ID joins a cluster, it continues functioning as the original broker and will be the replica and the leader for the same partitions as the old broker.

16.1.3 Storage failures

When speaking about storage, we also need to distinguish between different types of failures that can occur. One seemingly simple failure that can have severe consequences is that the storage or volume that Kafka uses for storing the partitions becomes full. Linux is bad at handling full volumes; with Kafka, it's even worse, as it commits the data to disk immediately but lets the operating system handle this. In other words, if the storage runs out of space, and you have a lot of data in the page cache (i.e., memory), this data might get lost.

Critically, if all our brokers are configured in the same way, and this problem arises at the same time on all brokers, the data loss might be permanent, and there's little you can do about it. An old system administrator's wisdom is to have a spare 10 GB file on the volume; in the case of a full disk, this file can be deleted and you have some more time to fix the problem.

> **TIP** We need to make sure that Kafka never runs out of disk space. We believe that it's better to stop Kafka before this happens. Of course, this applies only if the data we store in Kafka is important to us.

In a virtualized environment or Kubernetes, this problem is often easily solvable by increasing the disk size. For bare-metal installations, this gets harder, as you need to replace the disks. The key to handling network failures lies in proper monitoring and alerting to detect problems quickly, as well as having clear operational procedures for this scenario.

Another scenario is the permanent failure of a disk. In old versions of Kafka, this always led to the failure of the broker. If you have multiple disks per broker and don't use a Redundant Array of Independent Disks (RAID), you can configure multiple log directories. If a log directory fails, the broker will notify the controller about the failure, but the other directories will continue working flawlessly.

It's then up to us to replace the failed disk with a new one. Then, the broker will detect an empty directory and will replicate all data that has been on that disk from the respective leaders and, after some time, will become in sync again.

If we use a central storage system and this fails, there is little that Kafka can do about this. We need to fix the error and probably also restart Kafka. It's likely that Kafka will lose some data, as we've discussed in the first scenario here.

16.1.4 Data center failures

Another infrastructure failure to consider is the failure of a whole data center. Usually, this means that not only Kafka fails but all surrounding systems too. A valid approach is to not consider this problem at all for a live system but to perform off-site backups of our mission-critical data. In addition, when a data center fails permanently, we'll need to recreate the infrastructure somewhere else. Yes, this means that we'll have significant downtimes, but for companies that aren't critical to our society, this might be a valid approach.

> **WARNING** Before implementing the solutions here, we should think twice whether we really need to cover this scenario. Probably Kafka won't be the largest problem we have when a data center fails. A stretched cluster, or even a mirrored cluster, increases the complexity and the cost of your system dramatically.

After this warning, let's look at how we can protect Kafka from failure scenarios like these. The simplest solution is to install the Kafka cluster across multiple data centers and operate a *stretched cluster,* as shown in figure 16.2. Whereas the examples in our book set up a local cluster with multiple brokers on a single machine, a stretched cluster does the opposite. That is, we attempt to distribute our brokers and thus our cluster across different data centers. This has the advantage that if one data center fails, we still have a functioning Kafka cluster because the failure of an entire data center is, for us, ultimately no different from the failure of a single broker. As long as multiple data centers don't fail simultaneously, we remain operational.

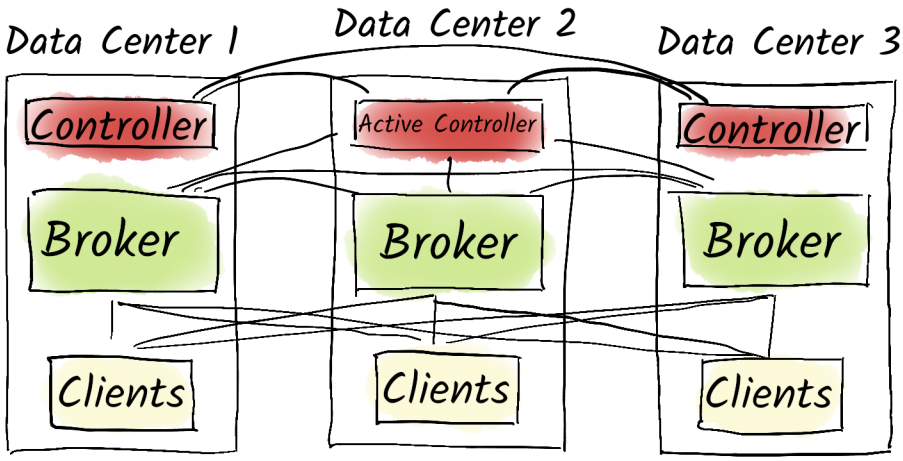


Figure 16.2 In a stretched cluster, our Kafka brokers are distributed among multiple data centers.

The probability of a serious disruption to our Kafka cluster is significantly reduced, as the likelihood of two, three, or even more data centers failing simultaneously is low. Conversely, if we operate our cluster in a single data center, the Kafka cluster would completely fail in the event of an problem.

Stretched clusters should be distributed over at least three data centers. Having only two data centers means the coordination cluster (either ZooKeeper or KRaft-based) will have a majority of its nodes in a single data center, and if this data center fails, we lose the coordination cluster and the whole Kafka cluster.

> **TIP** Stretched clusters should be distributed over at least three data centers. With two data centers, the coordination cluster won't be able to find a majority if two out of three nodes fail simultaneously.

There are other approaches, such as the 2.5 data center setup, in which a third "light" data center's only workload is a single coordination node; we advise against this if possible. Of course, nobody will build a third data center only for Kafka, so we understand that this is sometimes the only solution to the problem.

Now let's consider what we mean by data centers and how far apart they can be. Kafka is optimized for network latencies of up to 30 ms, so we recommend stretching Kafka across nearby data centers such as availability zones in the cloud but not across different regions. If you need a Kafka cluster to span multiple regions, it's usually better to use a mirroring approach, as discussed in section 16.3.

For this approach to make sense, we need to make sure that the replicas are evenly distributed across the different data centers by setting the `broker.rack` configuration on the brokers. Even though it's called the rack ID, it can also be the name of the data center or another key describing the location of the broker. If all brokers have a rack ID

set, then, when creating a new topic, Kafka will try to distribute the replicas as evenly as possible across the different rack IDs. For existing topics, there's no built-in solution to distribute them across the cluster, but we can use Cruise Control, discussed in chapter 14, to distribute the replicas.

Although the stretched cluster increases the fault tolerance, it will also dramatically increase the cost of operating Kafka in the cloud, as the traffic between availability zones is a significant cost driver. Even worse, consumers and producers always need to communicate with the leader, thus increasing the traffic cost even further. To mitigate this, KIP-392 allows consumers to consume from replicas in the same data center. For this, we need to set the `client.rack` configuration on the consumer side. This reduces the amount of traffic between the data centers significantly.

TIP If it's not possible to distribute our Kafka clusters across different data centers, we should at least try to distribute our brokers across different racks within the same data center to achieve the best possible fault tolerance within a data center. Ideally, these racks should be completely independent of each other, meaning they should have different network and power connections. The rack ID should also be set here.

16.2 Backing up Kafka

For most databases, good backup strategies are available. Sadly, this isn't true for Kafka. Although replication helps with mitigating the worst failures, it's not an alternative to backing up a system. But before we discuss the different approaches to backups, we need to clarify when a backup for Kafka is needed at all.

Many Kafka installations might not even need a backup strategy, as the data in Kafka can always be reimported from the original system. For example, if we use Kafka to move data from a core database to our microservices, we might not need to back up Kafka, as the data in Kafka is persisted in the original database. When Kafka fails, it might be enough to simply reproduce all the data from the original database.

When using Kafka as a messaging system where the data in Kafka is only needed for a few seconds, it might also be useless to back up Kafka. A customer of ours once said that if they ever needed to restore a backup of Kafka, the company would be broke anyway. In these cases, it doesn't make sense to back up Kafka. But more and more companies are using Kafka as the single source of truth for much of their most important business data, and thus they need a good strategy for when this system fails.

The most obvious approach to backing up Kafka is to simply do periodic filesystem snapshots and store this data off-site. The advantage is that system administrators are accustomed to this procedure, so there's probably a lot of knowledge about this in the organization. But this approach has a few disadvantages. First, as Kafka replicates the data itself, the backup will contain a lot of redundant data, which might or might not be a big problem for the organization.

More problematic is the timing of the backup. When you're able to do the snapshots for all brokers at approximately the same time and don't do any major operations such

as moving partitions between brokers, this might not be a problem at all. But imagine the case when you have Partition A on Broker 1, and we first do a backup of Broker 2. Next, we move Partition A from Broker 1 to Broker 2, and then we do a backup of Broker 1. In this case, we wouldn't have a backup of Partition A at all.

Restoring Kafka from such a backup should be relatively easy, especially if only some of the brokers failed. But as with any scheduled backups, we'll lose all the data that was produced after the snapshot was taken and the disruption happened.

For this, a continuous backup solution is a better approach. One of the authors tried to implement a solution based on Kafka Connect that would handle this procedure gracefully, but never brought it to production.

Some companies are trying to achieve this by using Kafka Connect to store the data in Kafka, for example, in Amazon Simple Storage Service (Amazon S3) or a similar service. This works very well, but the problem is that this isn't a complete backup, as most connectors don't store the offsets of the consumers. Thus, with this approach, we could restore all the data, but the consumers wouldn't be able to continue working at exactly the same position where they left off.

Another approach could be to rely on *tiered storage* in Kafka. But, at the time of writing, tiered storage supports neither compacted topics nor the consumer offsets topic. It's also hard to predict when a segment will be moved to the cold tier.

None of these approaches are perfect. A promising alternative solution is a relatively new commercial solution called *Kannika* that not only provides a backup and restore solution but also can clone Kafka clusters to, for example, clone a production environment to a test environment while masking data to be compliant with privacy regulations.

16.3 Mirroring Kafka clusters with MirrorMaker

Another way to protect ourselves from failures and data loss is by mirroring our entire cluster. Unlike conventional backups, we continuously copy our messages from one cluster to another. And unlike regular replication, we can mirror clusters over larger distances. For this purpose, as mentioned in earlier chapters, Kafka already has a tool called *MirrorMaker* that takes care of the entire process. MirrorMaker itself is based on Kafka Connect, which we've already encountered throughout this book.

The best part about MirrorMaker is that we can not only mirror our topics and messages but also the ACLs of the topics and the consumer offsets at the same time. MirrorMaker uses three different Kafka connectors for this purpose:

- *MirrorSourceConnector* is responsible for mirroring our actual topics (and ACLs).
- *MirrorCheckpointConnector* handles the offsets.
- *MirrorHeartbeatConnector* periodically checks the connection between the clusters.

NOTE *Confluent Replicator* offers a proprietary alternative to MirrorMaker for mirroring clusters.

We now know that there's a simple and efficient way to mirror our clusters using MirrorMaker. Let's look at the different architectures we can create in practice with MirrorMaker, and, most importantly, what this means for our disaster management.

16.3.1 Active-passive cluster

The simplest topology is the *active-passive pairing*, as shown in figure 16.3. In this setup, we have an active cluster where we produce and consume messages. We then mirror this cluster to a second cluster. Aside from this, we don't produce any messages in this cluster, so we refer to it as passive.

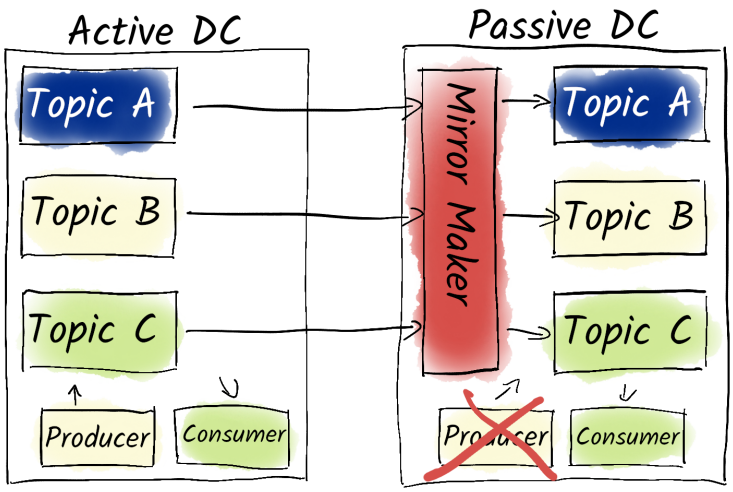


Figure 16.3 An active-passive cluster with MirrorMaker

If our active cluster fails, the passive cluster takes on the role of the active cluster. In addition, our producers start producing their messages to this cluster, while consumers now read messages from this cluster.

So far, this approach is very straightforward. But what do you do after you've switched the clusters? The previous active cluster is now out of sync with the passive cluster, and you don't have any fallback if the currently active cluster fails. The cleanest solution is to delete the previous active cluster and create a new, now passive cluster in its place.

The problem is that this must be done after every failover. We dislike this approach because of the complexity after the recovery step, but especially for companies that have only two distant data centers, this might be a viable option.

Another use case of this approach is to migrate data, for example, during data center migrations or when moving to or from the cloud. An active-passive cluster can be also used for scenarios where we want to move data from one location to another and, for example, analyze the data in a central headquarters. But this is very similar to the hub-and-spoke architecture we'll discuss later.

16.3.2 Active-active cluster

In the *active-active pairing* topology shown in figure 16.4, two completely equal clusters mirror each other. This means a MirrorMaker in each cluster mirrors the other cluster into its own. To enable this, MirrorMaker 2 introduced *remote topics*. Remote topics are identified by a special naming convention that adds the originating cluster name as a prefix to the topic names. The special feature of these topics is that they aren't mirrored by MirrorMaker, thus preventing endless loops of topic mirroring that would otherwise occur.

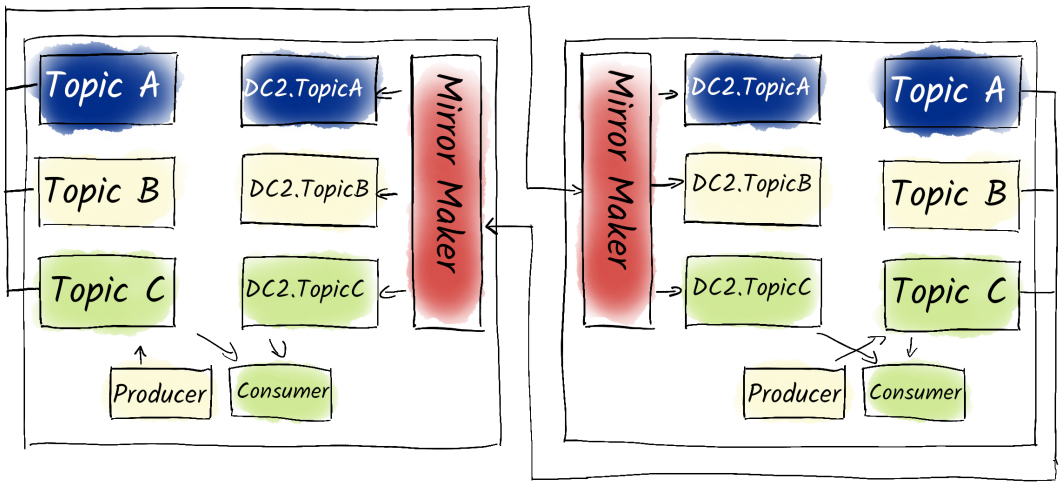


Figure 16.4 An active-active cluster with MirrorMaker 2

MirrorMaker also makes sure that the remote topics are read-only by setting the required ACLs. This ensures that producers don't produce data by accident into that topic, as that data will never be replicated to the other data center.

But what about our clients? In an active-active pairing, we can have producers and consumers in both data centers. Producers simply produce only into the local topics as usual. For consumers, it's a bit more complicated. Usually we want them to consume from both the local and the remote topics. For this, Kafka supports regular expression (REGEX) subscriptions, where the consumer specifies a REGEX and subscribes to all topics matching this expression.

So far, this sounds straightforward, but you always need to consider that there might be services that consume a topic and produce into another topic. Maybe we have a payment-initiating service that consumes from the orders topic and writes a confirmation to a payment-status topic. If we start this service in both data centers and let them both consume from the local topic and the remote topic, then each order will

probably be charged twice. So, when using active-active pairing, you need to take extra precautions against such problems.

> **TIP** To prevent duplicate processing in scenarios such as payment-initiating services, we can use idempotence and transaction IDs with deduplication. We can use idempotence, meaning our application is designed to handle repeated processing of the same message without adverse effects, such as duplicate payments. Additionally, by using transaction IDs for deduplication, our service can ensure that any transaction is processed only once, even in active-active setups across multiple data centers.

The big advantage of this approach is that if one data center fails, it shouldn't be noticeable, and there should be no, or very little, action taken on the surviving data center as the producers continue producing to the local topics, the consumers continue consuming messages from the local topics, and the unconsumed messages are consumed from the remote topics. As soon as the failed data center is fixed, it can be started, and operations can continue as before.

16.3.3 Hub-and-spoke topology

The last architecture we'll explain in this section is the *hub-and-spoke* topology, as shown in figure 16.5. The idea is to have a central Kafka cluster acting as a hub that aggregates the data from multiple spoke-clusters.

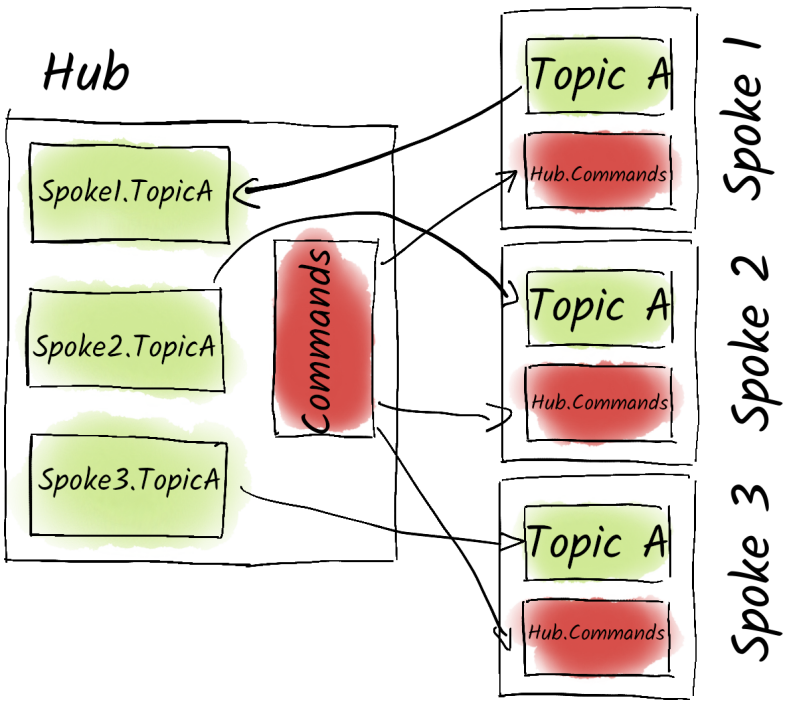


Figure 16.5 A hub-and-spoke topology that uses MirrorMaker 2 (Kafka Connect) to replicate topics from the spokes to the hub and to the commands topic from the hub to the spokes

This can be used, for example, if we have a company headquarters that wants to collect all the data from its subordinate companies and then analyze that data centrally. Then, we can install a MirrorMaker in the central hub, or even one per subordinate, that fetches all the required data from the spoke-clusters and writes it to the local cluster. Then, the data can be processed as usual by local consumers.

This topology also allows us to send data from the hub to the spokes. For example, if we have a supermarket chain consisting of a headquarters and multiple supermarkets, the headquarters could publish new prices in a central topic that is then replicated to the local supermarkets. On the other hand, the supermarkets publish sales data to their local Kafka cluster, and then, using MirrorMaker, the data is replicated to the headquarters, and the total revenue can be calculated.

If something bad happens and the connection can't be established, the local supermarkets can continue operating, but in the worst case, they would still have the old prices. The headquarters will be unhappy about the missing sales data, but it's not lost. As soon as the connection is reestablished, the data will be synchronized in both directions, and the operations can continue as normal.

To summarize, this topology is useful for cases where our organization is distributed across many locations that should be able to continue operating even when the connection to the central hub can't be established. Apart from supermarkets, cruise ships are using similar approaches to synchronize their data when in harbor and then operate autonomously at sea. Approaches like these are also crucial for government operations that must function even when an internet connection can't be established.

NOTE MirrorMaker works asyncronously. This means that the latest data could still be lost as MirrorMaker is catching up with the latest messages.

Summary

- Disaster management in Kafka focuses on strategies to handle failures and minimize the likelihood of disasters.
- The three types of failures in Kafka are network problems, broker problems, and persistent storage failures, often exacerbated by human error.
- Network failures are common in distributed systems. Individual client connection problems can arise, but these are typically resolved quickly.
- Broker problems can lead to data loss if messages aren't properly committed before a broker failure. Ensuring `acks=all` and a sufficient number of in-sync replicas is critical for data delivery assurance.
- Persistent storage failures are one of the most severe problems in Kafka. Ideally, these failures only affect a single broker, but they can lead to irreversible data corruption if not handled properly.
- Conventional backups aren't practical in Kafka due to continuous message production and consumption, which can lead to potential data loss and inconsistencies.

- Stretched clusters reduce the likelihood of total failure by operating a Kafka cluster across multiple data centers, mitigating risks from data center outages.
- Confluent Replicator provides a proprietary alternative to MirrorMaker for mirroring clusters.
- An active-passive pairing involves one active cluster mirroring another passive cluster, taking over in case of a failure, but not reverting back to active.
- Active-active pairing consists of two equally capable clusters mirroring each other, allowing continuous operation without needing to rebuild clusters during a failure.
- In active-active configurations, consumers can read from both clusters and need to aggregate data appropriately.
- Remote topics, introduced in MirrorMaker 2, prevent endless loops of topic mirroring while ensuring seamless data availability during failures.
- The hub-and-spoke topology features a central cluster that aggregates data from smaller local clusters, allowing independent operation even if the central cluster is down.
- Both active-active and hub-and-spoke topologies aim to maintain data consistency even during failures.
- It's recommended to avoid active-passive pairings and prefer active-active configurations for better resilience and performance.

17 Comparison with other technologies

This chapter covers

- Inside data versus outside data
- Kafka as a decoupled system
- Comparing Kafka to the REST framework
- Comparing Kafka to relational databases
- Kafka and event-driven architectures

We've described Kafka extensively in this book and touched on many topics relevant to its use in companies. However, what we haven't yet discussed is when the use of Kafka is appropriate and how we can compare Kafka with other technologies.

In our opinion, it's the responsibility of architects and engineers to find the best technology or architecture for a given use case and, in particular, for the given nonfunctional requirements. What makes sense as a technological solution in one company may not work in another company, even with the same use case.

Exciting new technologies such as Kafka can help us solve architectural and technical challenges and enable business models that wouldn't be easily feasible otherwise. But in the end, it's never about the technology itself, but how we implement that technology in the company and how we bring along the people affected by it.

There are definitely no cure-alls for all problems, and finding the one and only solution for a given problem is rare. In most companies, we don't start planning from scratch but must integrate into the existing IT landscape.

17.1 Data on the outside vs. data on the inside

With which technologies can we compare Kafka? To address this, we would first like to refer to a highly regarded and worthwhile paper by Pat Helland: "Data on the Outside vs. Data on the Inside" (www.cidrdb.org/cidr2005/papers/P12.pdf). In this paper, Helland describes the differences and challenges of data that resides outside of services and inside services, as shown in figure 17.1.

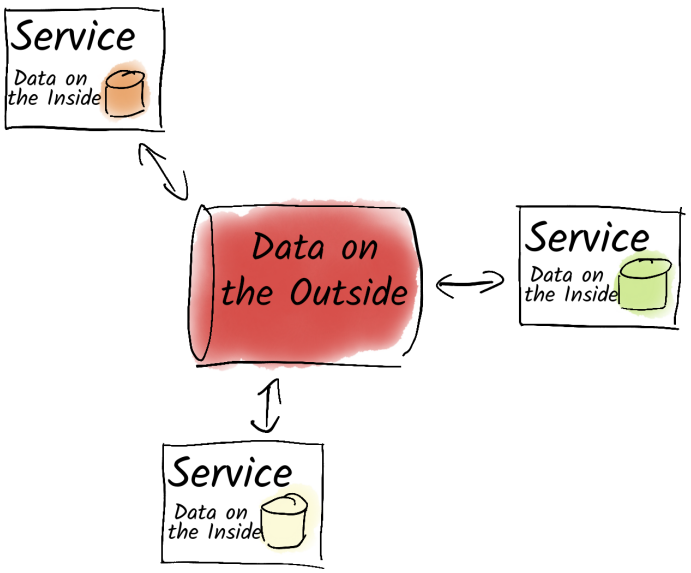


Figure 17.1 We differentiate between service-internal data, that is, data on the inside, and data exchanged between services, that is, data on the outside.

With *data on the inside*, Helland describes service-internal data, which should be stored in a way and format that fits the specific service managing that data. Other services shouldn't concern themselves with this data. Because users often interact directly with this data, we often desire synchronous write and read operations and high consistency guarantees. This ensures, for example, that written data can be read immediately, and users don't perform operations multiple times because they believe the service didn't respond.

One notable aspect of service-internal data is that a single service may need to combine and aggregate various subsets of its own internal data to provide meaningful functionality or insights to its users. This results in a tight coupling between the service and its data model, as any changes to the data model will likely necessitate corresponding changes to the service itself. While this strong dependency may seem restrictive,

it ensures that the service can fully optimize its internal data for performance and usability.

On the other hand, *data on the outside* refers to data that lies outside of service boundaries. This data is structured to enable data exchange between services, so the data must be service-agnostic. If a service changes its internal data model, other systems don't necessarily need to follow suit, as data on the outside is structured differently. Ideally, this data is immutable: once created, they shouldn't be altered or deleted. The reason for this is that once a service produces outside data, it must assume that other services have already consumed this data.

Services should always be loosely coupled to outside data. Ideally, service-internal and service-external data models can evolve independently of each other. This isn't always feasible, and in such cases, coordination between service teams is required. The system responsible for handling data on the outside must be resilient and performant. If this system fails, the entire data exchange within the company could be compromised.

Often, for data within services, we use relational databases or very specialized technologies best suited for the tasks at hand. For data outside of services, we use messaging systems, or we might not have an explicit storage system for such data but rather transmit it via synchronous protocols such as HTTP-REST. In many companies, however, we typically have database systems that are accessed by many different services, with all the advantages and disadvantages that come with this.

Kafka fits well into the data on the outside category, as it's a messaging system designed to facilitate the exchange of data between services. Kafka provides a highly resilient and performant solution for streaming data that is immutable once produced, ensuring that services are loosely coupled.

Kafka's publish-subscribe model and topic-based architecture make it ideal for handling service-external data, where services can produce and consume data asynchronously without being tightly bound to one another's internal data models. This makes Kafka particularly well-suited for managing data flows in microservice architectures, where agility and independence between services are crucial.

17.2 Classic messaging systems vs. Kafka

The comparison with classic messaging systems is also relevant because Kafka is often used as such. The goal of a messaging system is to move messages from producers to consumers, ideally performing this task efficiently and reliably. Vendors of these systems often offer numerous additional features, up to and including *enterprise service buses* (ESBs), which are designed to integrate all of a company's systems with each other.

There are many widely used products, including the SAP Process Integration (SAP PI), Seeburger BIS, Microsoft BizTalk Server, and many more. They are used for enterprise-level integration, providing tools for message routing, transformation, and communication across different platforms.

In figure 17.2, we illustrate several key features of ESBs. These include message routing, where the producer creates a message and the messaging system ensures the consumer receives it, regardless of the systems' locations, jurisdictions, or corporate divisions. ESBs handle the delivery, so the producer doesn't need to manage these complexities.

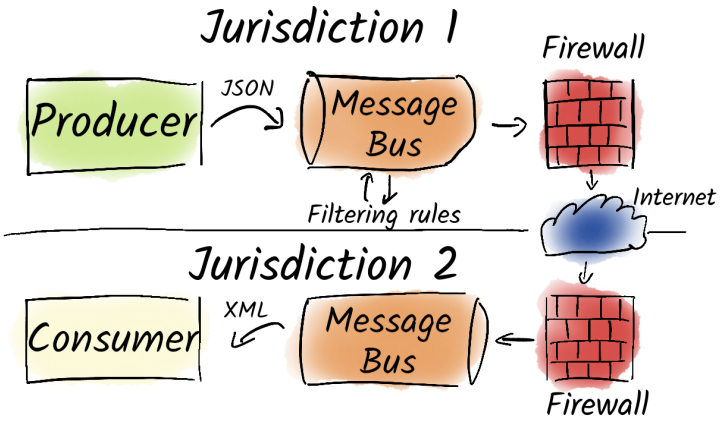


Figure 17.2 ESBs route messages across different locations, ensure data protection compliance, and support various message formats.

Additionally, ESBs can manage incompatible message formats between producers and consumers, converting messages automatically without requiring changes to the systems. Messaging systems can also prevent consumers from being overwhelmed by notifying the producer to slow down production when necessary.

Furthermore, ESBs often handle compliance with data protection regulations and support a wide range of formats, such as JSON, XML, and SAP intermediate documents (IDocs). However, managing these systems can overload teams of experts, leading to delays in system changes that may take weeks or even months.

17.2.1 Kafka is agnostic

At this point, we notice that Kafka follows a different philosophy. Unlike traditional messaging systems, it doesn't provide all the features just discussed out of the box. We have to implement them ourselves if they are important to us.

The philosophy of classic messaging systems, especially ESBs, is that services don't have to worry about communication; they only send data to the transport layer, which takes care of everything. In this context, we can say that, in terms of communication, the services themselves are dumb, and the transport layer, the pipe, is smart.

Kafka follows the opposite philosophy. Kafka itself is dumb in terms of communication. It's essentially a system that allows us to place data in logs, where it's persisted and can be retrieved by consumers at a later time (within the constraints of the cleanup policy). The services themselves are responsible for determining how they interact with

Kafka, which messages they read, and how they write data. This approach provides multiple levels of decoupling:

- *Platform agnostic*—Kafka can be used across different platforms without requiring services to be on the same infrastructure. Producers and consumers can be on completely different systems and still communicate effectively.
- *Data format agnostic*—Kafka doesn't impose a specific format on the data. This allows producers and consumers to exchange data in various formats (e.g., JSON, Avro, or Protobuf) without Kafka dictating how that data should be structured.
- *Programming language agnostic*—Kafka supports a wide range of programming languages. It provides client libraries for languages such as Java, Python, Go, and more, allowing services written in different languages to communicate seamlessly through Kafka.
- *Consumer agnostic*—The producer doesn't need to know who the consumer is or even whether the consumer exists. Kafka's model decouples producers and consumers entirely, meaning producers can send data without worrying about the state or existence of consumers. The consumer might not even be online, might be undergoing maintenance, or could be experiencing high load. Kafka's system allows the producer to continue producing data without being affected by the consumer's state.

17.2.2 Operational complexity in classic messaging systems

Technically, this is different with classic messaging systems. They aren't designed to retain data for long periods but must do so to survive failures. Classically, these systems delete data as soon as the consumers have read the messages. This means that we can't easily track what data has flowed through the system afterward.

If we now want to connect a new service to our messaging system that needs all historical data, the service can't read this data without effort. Instead, we need to either send the data through the messaging system again or find another way for the initial population. With Kafka's log-based approach, this use case is easier to implement because data can be stored for long periods, if desired.

The approach of deleting data immediately after consumption presents further challenges. The messaging system must be aware of the consumers. Typically, messaging systems also proactively push the data to the consumers. But what happens if consumers are undergoing maintenance or are overwhelmed?

To handle this, the messaging system must incorporate complex logic to mitigate these problems. Incorrect configuration or significant failures could lead to an overload of the messaging system and thus large-scale outages. One way to prevent this overload is through backpressure mitigation.

If the messaging system detects slow consumers, it can ask producers to slow down production. However, this couples the consumers back to the producers, making them dependent on each other, as shown in figure 17.3. Instead, if the messaging system

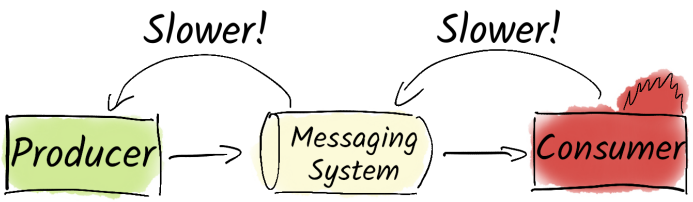


Figure 17.3 A slow consumer can slow down a producer in classic messaging systems.

assumes that data should be stored for extended periods, it can skip these steps entirely and leave the responsibility for retrieving messages to the consumers. If a consumer is slow, it simply processes fewer messages and (hopefully) catches up later. If a consumer fails, it will be restarted eventually and can independently read the old data. The messaging system doesn't need to be concerned with who has read what. The producers certainly don't need to worry about this. It's entirely the consumers' responsibility.

17.2.3 Governance of classic messaging systems

Another point, which is particularly relevant in larger companies, is determining who is responsible for the correct operation and management of the messaging technology. Usually, a central team is responsible for this. Our experience with ESB systems shows that this team manages all configurations, and complex processes usually need to be followed before changes can be made. While this structure helps to control sprawl, it also reduces agility because desired changes can take weeks or months to implement.

In Kafka, the need for extensive configuration is significantly reduced. Topics are managed with straightforward naming conventions, and configuration guidelines are established by a *center of excellence* that provides recommendations. Permission management is consistent across the system, simplifying the process of granting access. Additionally, every data system is expected to have a schema, raising the question of whether it's properly documented.

> **TIP** We recommend implementing processes for configuring Kafka topics, technical users, and access rights, as discussed in chapter 14.

One of the advantages of Kafka is that it eliminates the need for centralized management of transformation logic for mapping between different formats. This decentralization allows teams to work independently without relying on a central team for data format transformations.

> **NOTE** We want to emphasize again that messaging systems outside of Kafka also have many good use cases, and we often find Kafka used in conjunction with other messaging systems, for example, IBM MQ in larger enterprises or Message Queuing Telemetry Transport (MQTT) for Internet of Things (IoT) use cases.

In practice, we often see that classic messaging systems are primarily used to connect existing core business systems, for example, to exchange data between SAP ERP and production control in a manufacturing company. Kafka, on the other hand, is primarily used in the early stages to develop new products and adapt products to new customer requirements.

Some companies refer to this as using classic messaging systems for the Old World and Kafka for the New World, and using them to communicate between the old and new worlds. However, we now also see Kafka's areas of application expanding, and companies are beginning to use Kafka in areas where classic messaging systems were dominant until recently.

To highlight the differences and the benefits Kafka offers, table 17.1 compares the limitations of classic messaging systems with Kafka's advantages in various use cases.

Table 17.1 Comparing classic messaging systems and Kafka's benefits across key aspects

Characteristic	Classic Messaging Systems	Kafka's Benefits
Integration with legacy systems	Often used to connect legacy systems (e.g., SAP ERP)	Kafka is ideal for integrating modern, cloud-native apps and microservices.
Message processing model	Typically synchronous, with tight coupling between producers and consumers	Kafka supports asynchronous communication, enabling temporal decoupling (producers and consumers can be out of sync).
Data persistence	Limited persistence; messages may be deleted after delivery	Kafka persists messages, allowing for long-term storage and replayability of data.
Scalability	Often harder to scale, especially in large, distributed environments	Kafka is highly scalable, capable of handling massive volumes of data with horizontal scaling.
Flexibility	Rigid integration that often requires customization or middleware for format conversion	Kafka supports multiple data formats (JSON, Avro, etc.), making it more format-agnostic and adaptable to diverse use cases.
Latency	Higher latency due to synchronous processing	Kafka is designed for low-latency, high-throughput data streaming.
Consumer independence	Consumers tightly coupled to producers, limiting flexibility	Kafka allows loose coupling. Consumers can consume messages at their own pace, even if they are temporarily offline.
Use case flexibility	Primarily used for established, core business applications	Kafka is increasingly being used for real-time data streaming, event-driven architectures, and new application scenarios.

17.3 REST vs. Kafka

HTTP or other synchronous communication systems are often used and proposed as the preferred method for microservice-based architectures. REST-like protocols are ubiquitous in IT and provide a simple and efficient way to connect services.

One of the major advantages of HTTP-REST is that it's very easy to get started. Nearly every programming language supports this protocol, and once written, an API can be used for a variety of use cases. We can use the same API, for example, to exchange data between the frontend of a service and its backend, as well as between the backend and other services.

17.3.1 Challenges of synchronous communication

What seems intuitive at first glance carries the risk of introducing significant complexity later on. Microservice-based architectures are often justified by the fact that making changes in monolithic systems is very difficult because all components depend on each other, as shown in figure 17.4. Unfortunately, synchronous communication protocols rarely solve this problem, reproducing the same entanglements that existed previously in the monolith. This results in combining the disadvantages of monolithic systems with those of distributed systems.

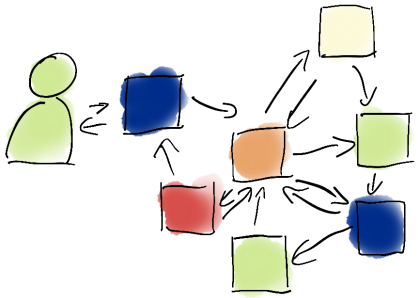


Figure 17.4 Synchronous communication protocols such as HTTP-REST often lead to strong dependencies between services. The failure of one service can cause domino effects, resulting in the failure of the entire system.

The previously discussed paper by Pat Helland explicitly stated that we should handle service-internal data differently from service-external data. With REST, service-internal data is usually exposed via an API. In many cases, it would be better to exchange external data through an explicit and appropriate communication structure.

One of the core problems with REST for data exchange is its synchrony. Synchronous communication leads to temporal and physical coupling. In practice, services require data from other services to respond to requests. If these other services also need data from additional services, it often leads to problems.

> **WARNING** The failure of a single service can quickly trigger a domino effect that affects all services.

One way to address this is to create local caches of the most recently read data. However, how do we keep this cache up to date if we don't receive notifications about new data? This is a very challenging problem without simple solutions.

17.3.2 Alternative communication strategies

An alternative to synchronous communication channels (e.g., HTTP) for data exchange between services is to communicate asynchronously. Asynchronous communication can mitigate many problems when a service fails. Instead of actively retrieving the latest data from another service, we rely on being notified of changes. This

approach enables us to precompute data when an event occurs, allowing us to respond to customer inquiries more quickly.

While synchronous communication is often necessary for certain use cases, such as real-time scenarios in video games or the trading industry, asynchronous systems such as Kafka shine in situations where high throughput, scalability, and decoupling between services are priorities. Kafka's architecture is particularly effective for handling large volumes of events and ensuring data flows reliably through a distributed system.

With REST-based synchronous communication, data is transmitted only when explicitly requested by a service. However, it's important to clarify that REST itself doesn't imply that data is at rest. REST is a protocol designed for resource-oriented communication, and it can certainly support near-real-time interactions, albeit with limitations when scaling to massive amounts of data or handling frequent state changes.

Kafka, by contrast, is fundamentally built around the concept of data in motion. When an event occurs, Kafka streams the data in near real-time, triggering workflows across the organization. This event-driven design enables the immediate evaluation and utilization of data, which can improve responsiveness and efficiency.

However, implementing Kafka does come with challenges. Managing a central messaging system such as Kafka requires additional infrastructure and operational expertise. Maintaining two APIs, one for service-internal use and another for service-external use, can add complexity. For smaller projects or teams, this effort might outweigh the benefits. In larger organizations with multiple teams and systems, though, this investment often proves worthwhile.

Another limitation of Kafka is the lack of a central queryable repository. Unlike relational databases where data can be analyzed ad hoc with SQL, Kafka's distributed log structure makes this process less intuitive. While tools such as Flink SQL offer solutions for querying streaming data, they aren't yet as straightforward or robust as traditional SQL engines. As a result, data from Kafka is often exported into data lakes, data warehouses, or other analytical systems, where it can be aggregated and evaluated for analytical use cases.

17.4 Relational databases vs. Kafka

Before we dive into comparing Kafka with relational databases, we want to emphasize that we're strong advocates of traditional relational databases. When we start private projects, PostgreSQL is usually at the center of our architecture. It's interesting to consider what core questions different systems answer. For relational databases, this can be summarized with the question "What is?", or more elaborately, "What is the current state of my world?" Clearly, this is very, very useful.

17.4.1 Strengths and weaknesses of relational databases

Databases are perfect for making the most of the data. With SQL, we have a powerful language to express and execute even the most complex queries efficiently. However, this leads to a strong coupling of our applications to the database, making databases

ideal for data on the inside. We believe that databases are perfect for storing service-specific data and provide guarantees that we don't want to miss.

WARNING However, if we use a database in such a way that many services access it and there are no clear separations between my data and your data, then this creates a shaky house of cards where even the slightest change can lead to a collapse, as shown in figure 17.5. At that point, one wouldn't touch the system at all.

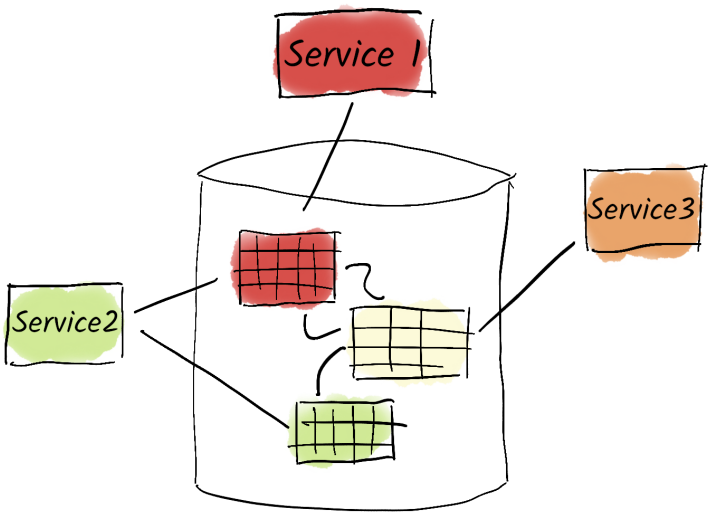


Figure 17.5 Relational databases excel with their consistency guarantees, but when a database is used by many services, this leads to very strong coupling, and even small changes must be coordinated with all teams.

Before we delve further into this, let's look at the main question Kafka answers: "What was?", or more elaborately, "How did I arrive at the current state?"—in other words, the history. Given the history, we can reconstruct the current state. When we have different services reading the same history, we can build the state as needed for our specific use case. This history of events is perfectly suited for exchanging data between different services, making it ideal for data on the outside.

17.4.2 Complementary roles of Kafka and relational databases in modern data architectures

While we definitely aren't advocating for replacing databases with Kafka, we do advocate for using databases for service-internal data and systems such as Kafka for data exchange. Again, this may not always be worthwhile for small use cases, but the more services interact with each other, the more difficult it becomes to implement changes in a central database and the quicker the investment in Kafka pays off.

TIP With the help of Kafka Connect, we can connect databases to Kafka with relatively little configuration effort.

By the way, Kafka and databases aren't so different after all. When we take a closer look at databases, we find that nearly every (relational) database is internally based on logs. The commit log keeps track of all changes to the database before the actual tables are updated.

These logs are used, among other things, to replicate data between different instances of a database and also for recovery in case of errors. If our database server crashes and is restarted, databases often use the commit log to restore the correct state. Based on the data in the commit log, we can not only create the tables but also keep indexes, materialized views, and other structures up- to-date, as shown in figure 17.6.

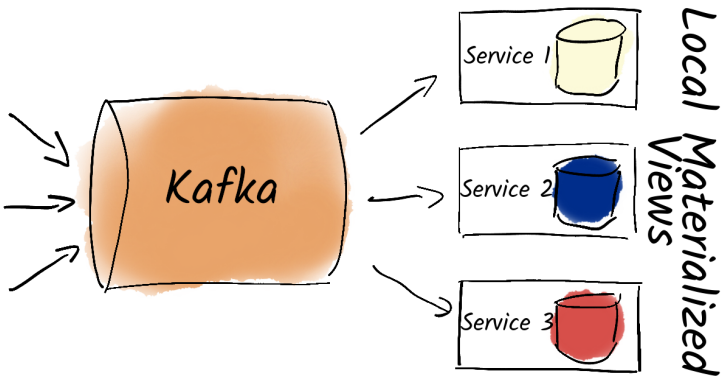


Figure 17.6 Local materialized views with Kafka

We can now view topics in Kafka as precisely such logs. But instead of trying to replicate the functions of a database in Kafka, we offload these functionalities to our services, just as Helland described with data inside and outside of services.

The log serves as the perfect data structure for exchanging data between services and decoupling them. Within the service itself, we build materialized views of the data in the log. The nice thing is that, unlike views in a database, these views in the service don't have to be homogeneous. We can use the technology for the view that best fits the service and our IT landscape for each service.

If we want to search for data in the log, why not simply use Elasticsearch or OpenSearch? If we want to keep the data warm for retrieval on our highly visited website by using a cache, why not store the data in an in-memory cache such as Redis? Perhaps we also want to perform complex ad hoc queries over the data? Then, we use a relational database such as PostgreSQL in the service.

However, we don't even need to go that far and deploy other products. In many cases, stream processing with Kafka Streams or Apache Flink, in combination with stateful operations, can replace the use of additional database systems. We discussed this in chapter 12.

Thus, with Kafka, we can create an architecture where we hold immutable data centrally for all interested parties, and each service can independently retrieve the data it needs and store it in exactly the format it requires using local materialized views. Martin Kleppmann referred to this idea in his conference presentations and blog articles as "turning the database inside out" (https://mng.bz/nR78). He describes precisely this idea: using the log as a central entity for our enterprise architecture. Based on this, we build features of databases, such as tables, views, materialized views, indexes, and so on, in a distributed manner. We also like to call this architectural pattern the central nervous system for data or streaming platforms.

17.5 Kafka is the core of a streaming platform

Some refer to Kafka as a messaging system, but from our perspective, this is only one possible application and not always a sufficient view. For some, it's enough that Kafka is a distributed log. However, we believe that Kafka is best described as the core of a streaming platform.

For us, the core idea of a streaming platform is that data is always in motion. Whenever something happens in our organization—and something is always happening—this event triggers an event in a producing system, which is then written to the central log. Other systems can then capture this event in near real-time or at a later time. Instead of waiting for up to a day or even longer, as in batch-based systems, to become aware of certain events and make decisions based on them, we can react promptly. But if we see Kafka as the core of a streaming platform, as shown in figure 17.7, what other systems do we need in addition?

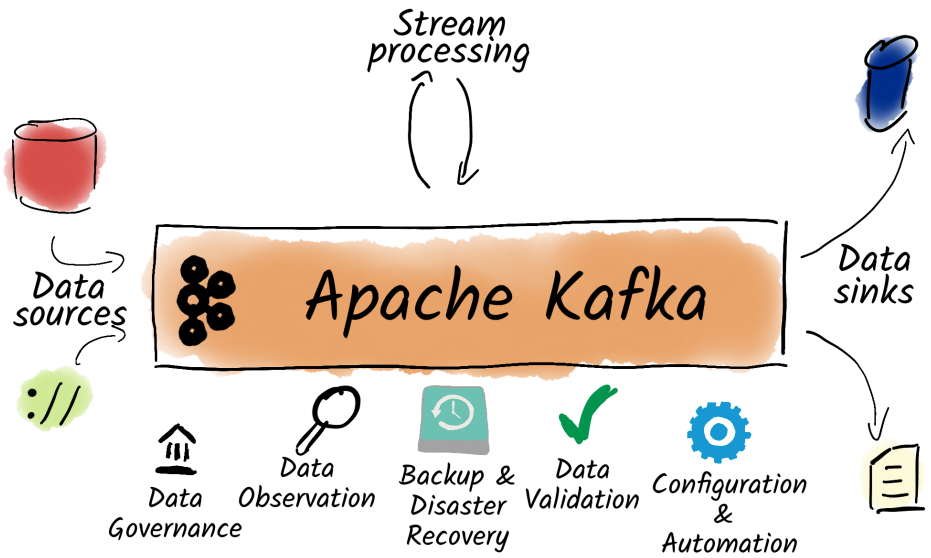


Figure 17.7 Kafka as the central nervous system for data in the organization

In most cases, Kafka isn't set up on a greenfield; that is, numerous other systems already exist. With Kafka Connect, we've learned about a tool to write data from third-party systems to Kafka or send data from Kafka to third-party systems such as databases, files, or even other messaging systems. We believe that Kafka Connect is an integral part of architectures that use Kafka for more than moving large amounts of data between two points.

Once the data is in Kafka, we usually don't just want to forward it to other destinations; we often want to perform additional processing steps. For this purpose, a stream processing tool such as Kafka Streams or Apache Flink is excellent. This allows us to process our data almost in real time and respond to changes. The processed data can then be used by our services or even by services from other teams across the organization.

When more than one team is expected to work with Kafka, a whole range of topics often gets ignored until it's almost too late—namely, the problem of Kafka management and automation. On one hand, this involves how we can set up Kafka as easily and automatically as possible if we want to operate it ourselves. But for everyday use, we find it equally important to automate the management of Kafka resources and processes as much as possible by answering the following questions:

- How are topics created?
- What naming conventions are used?
- How are users created, and what rights do these users have?
- How are the credentials distributed to our services?
- How do we automate the configuration of schemas and connectors?

These are all questions that can vary greatly from company to company, and there aren't always clear solutions. We favor a Git-based approach, where teams can submit pull or merge requests that are automatically checked on one side and also reviewed by people, at least for production environments. We've had good experiences with this, and companies find a good balance between maintainability and agility.

Once we've implemented all of this, we're on a good path to using Kafka as the central nervous system for our data. However, it's always true that the best practices are of no use if they don't fit the organization and the rest of the architecture isn't coherent. It takes more than just Kafka to transition our communication systems from data at rest to data flowing through an organization.

Summary

- IT architects must select the best technology for specific use cases, as solutions can vary between organizations.
- Data on the inside is tightly coupled with services and requires high consistency, while data on the outside is service-agnostic and ideally immutable.
- Integrating new technologies such as Kafka often involves adapting to existing IT landscapes rather than starting from scratch.

- Classic messaging systems aim to efficiently transfer messages but often require centralized management and can introduce operational complexities.
- Unlike traditional systems that delete data after consumption, Kafka retains messages, allowing consumers to access historical data independently.
- Kafka promotes decentralization, enabling teams to manage messaging autonomously, making it ideal for adapting to evolving product needs.
- HTTP-REST simplifies service communication but can create dependencies leading to systemic failures.
- Synchronous protocols such as REST may cause cascading failures when services rely on each other for data.
- Asynchronous communication via Kafka enables real-time data flow but adds complexity in managing separate APIs for inside and outside data.
- Relational databases capture the current state of data but can create strong coupling risks when accessed by multiple services.
- Kafka focuses on the historical aspect of data, allowing services to independently reconstruct their states.
- We recommend using relational databases for internal data and Kafka for data exchange, enabling a flexible and decoupled architecture.
- Kafka can serve as the core of a streaming platform, enabling real-time data flow and rapid responses to organizational events.
- Stream processing tools such as Kafka Streams and Apache Flink, along with Kafka Connect, facilitate near-real-time data processing and enable seamless integration with third-party systems.
- Effective management and automation, using a Git-based approach, enhance Kafka's maintainability and agility across teams.

18 Kafka's role in modern enterprise architectures

This chapter covers

- Integrating Kafka within a data mesh architecture
- Real-time data processing and event-driven architectures
- Common anti-patterns in Kafka implementation

In this chapter, we'll explore some use cases in which Kafka excels, showcasing its versatility in industrial applications, data integration, and analytics. From manufacturing to pharmaceuticals, Kafka's ability to process vast amounts of data in real time enables businesses to make informed decisions, optimize processes, and enhance customer experiences.

It's equally important to recognize the limitations of Kafka. While it's a powerful tool, it's not a universal solution for every data challenge. Misapplication can lead to inefficiencies and complications that undermine its intended benefits. Therefore, we'll also highlight scenarios where Kafka may not be the best choice, addressing common misconceptions and providing guidance on when to consider alternative solutions. By understanding both the strengths and limitations of Kafka,

organizations can strategically deploy this technology to create robust and efficient data systems that meet their unique needs.

18.1 Kafka as the core of a data mesh

In recent years, concepts for data management within companies have evolved. Just as the term *microservices* became popular in software development, the term *data mesh* has emerged in the data world. But what exactly does it mean?

18.1.1 The challenges of traditional data management

To understand the need for a data mesh, let's consider a common scenario in many organizations. Imagine a sales department that tracks customer interactions in a *customer relationship management* (CRM) *system* and a production team that manages operations in an *enterprise resource planning* (ERP) *platform*. A central data engineering team is tasked with integrating these disparate sources into a data warehouse to enable company-wide reporting.

The sales department regularly updates its CRM system, adding new fields or renaming old ones, without informing the data engineering team. Meanwhile, the production team frequently exports incomplete data due to system quirks. The data engineering team, working with limited resources and no domain expertise, struggles to keep up with these constant changes. ETL (extract, transform, load) pipelines break often, delaying reports that are critical for business decisions.

This leads to frustration on all sides: the sales and production teams see the data engineers as slow and unhelpful, while the data engineers feel overworked and misunderstood. The reports they produce are inconsistent and unreliable, leaving business leaders to question their validity. This scenario illustrates the core limitations of traditional data management approaches: they rely on centralized teams to handle all aspects of data integration and quality control, creating bottlenecks and inefficiencies that stifle agility and collaboration.

18.1.2 Principles of a data mesh

One of the core ideas of a data mesh is to reverse the traditional responsibilities in data management. Rather than a centralized data team being solely responsible for managing ETL pipelines and ensuring data quality across all departments, each department becomes responsible for making its own data available to other teams in a well-documented, high-quality format. The central data team shifts its focus to providing and maintaining the technical infrastructure of the data platform, as well as ensuring adherence to governance rules such as access control, security, and compliance.

The concept of the data mesh, as shown in figure 18.1, is built on four core principles introduced by Zhamak Dehghani in 2019 and first outlined in her article published on Martin Fowler's blog (see https://mng.bz/dXAN). These principles have since become widely recognized as the foundation for implementing a data mesh architecture.

The four core principles are described here:

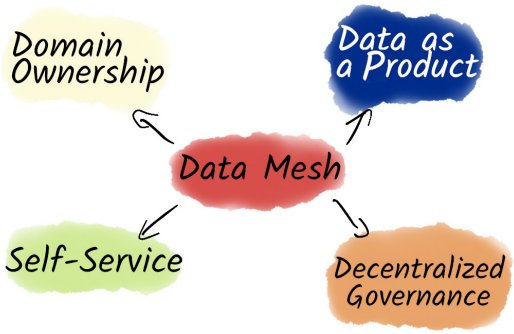


Figure 18.1 The core principles of a data mesh

- *Domain ownership*—The data team isn't responsible for gathering data from the departments. Instead, the departments are responsible for delivering high-quality, agreed-upon data. Business units no longer see data as waste but as a product.
- *Data as a product*—Data is viewed as a product just like any other (service) product that the department provides. The customers of these products are typically other departments but can also include external partners. The departments are responsible for satisfying their customers' needs.
- *Self-service data infrastructure*—Departments should be able to publish, consume, and modify data without lengthy processes or requests. The data team is responsible for providing a platform where, while adhering to processes and data protection rules, departments can meet their own needs. This platform is domain-agnostic.
- *Decentralized governance*—The goal of governance is to make data usage within the company as simple and compliant as possible, especially in terms of security. Data should be standardized, described, documented, and interoperable.

18.1.3 Data mesh vs. traditional approaches

Traditionally, a team of data engineers ensures that data from all services is centrally collected in a data lake. In its ideal state, a *data lake* serves as a centralized repository where structured and unstructured data is stored in its raw form, allowing for easy access, scalability, and flexible analysis. However, when data is ingested without proper governance, organization, or quality control, the data lake often degrades into what is referred to as a *data swamp*, as shown in figure 18.2.

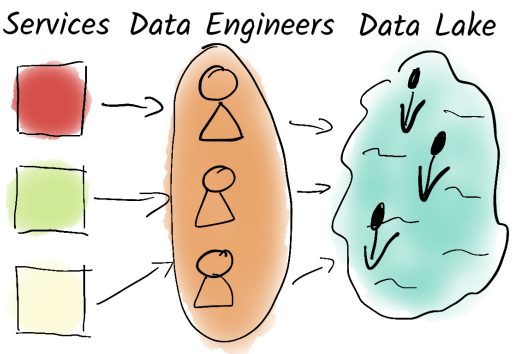


Figure 18.2 A team of data engineers ensures that all data from all services is centrally collected in a data lake. Because these teams can't accurately assess the quality of the data, the data lake often turns into a data swamp.

A data swamp is characterized by incomplete metadata, poor documentation, and inconsistent ingestion processes, which make it difficult to locate, interpret, or trust the data. This lack of usability often stems from the challenges of maintaining quality and organization in centralized systems.

Initially, the data mesh concept was introduced to better organize analytical data, but it's equally applicable to transactional data. To understand its value, let's revisit the challenges described in our earlier example.

In traditional approaches, each business unit maintains its own data storage while a central data team manages a data lake or warehouse. The data engineering team writes ETL pipelines to consolidate data from individual business units into the central repository.

However, as we saw in the example, business units often fail to inform the data team of structural changes, leading to broken pipelines and delays. While this model allows business units to continue their processes with minimal disruption, it creates significant inefficiencies and bottlenecks for the central data team.

The data mesh reverses these responsibilities. Instead of the central data team handling all aspects of data integration and quality, they now focus on providing a robust technical platform and ensuring adherence to governance rules. Business units take ownership of their data, treating it as a product that must meet the needs of internal or external consumers. These data products are published on the central platform and made accessible for both analytical purposes and other business processes.

As data evolves, the owners of these data products are responsible for updating and documenting their schemas. For example, if the sales department needs to modify its data schema to accommodate new CRM features, they ensure that the changes are clearly documented and communicated to other teams using their data. In cases of significant changes, the central data team steps in to assist with clean schema evolution and ensures compatibility across systems.

By shifting responsibilities, the data mesh enables the central data team to move from firefighting broken pipelines to becoming a center of excellence. They advise business units on best practices for data management, governance, and optimal structures. They also provide templates to streamline the implementation of producer and consumer functionalities, ensuring that all teams can efficiently participate in the data ecosystem.

This decentralized approach, guided by the data mesh core principles—domain ownership, data as a product, self-service infrastructure, and decentralized governance—creates a more agile and scalable data management system, addressing the limitations of traditional methods.

18.1.4 Kafka's role and responsibilities in implementing a data mesh

Kafka is a popular choice for implementing a data mesh, acting as a central platform for real-time data exchange across the enterprise. Its ability to handle large volumes of data with low-latency communication, as well as its cost-effective long-term storage with

tiered storage, make Kafka well-suited for supporting a data mesh architecture. Kafka helps facilitate the seamless flow of data products between producing and consuming departments in near real-time, making it an ideal technology for this approach. In a data mesh architecture, Kafka plays the following key roles, as illustrated in figure 18.3:

- *Data publishing and consumption*—Kafka provides a scalable, reliable, and high-performance platform for publishing and consuming data products in real time, supporting seamless data flow between departments.
- *Real-time data streaming*—Kafka enables near real-time data streaming, allowing departments to act quickly on changes and make faster, data-driven decisions.
- *Schema enforcement*—Kafka ensures data consistency by facilitating schema enforcement, helping both producers and consumers adhere to predefined data structures. This guarantees that data is correctly formatted and compatible across the system, reducing errors and ensuring reliability.
- *Governance integration*—Kafka integrates with governance tools, as discussed in chapter 13, to manage access control, ensure compliance, and enforce security policies. This includes mechanisms for managing data access, setting quotas, and ensuring regulatory compliance, all of which contribute to a secure and efficient data exchange environment.

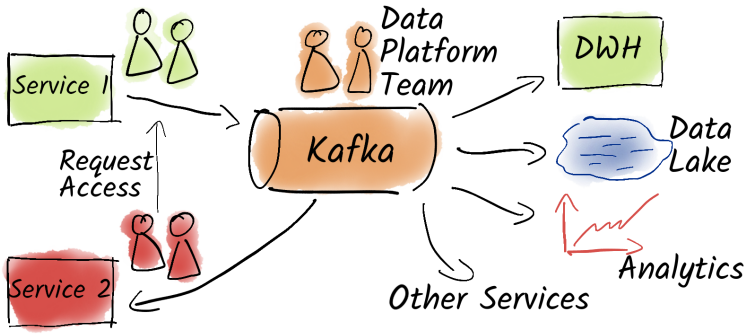


Figure 18.3 Kafka in a data mesh architecture

Let's take a closer look at departmental responsibilities in this setup:

- *Producing business units (data owners) are responsible for creating and publishing their data products to the central platform, which can be built on Kafka.* This includes managing the data life cycle, ensuring data quality, and adhering to the agreed-upon schema contracts with consumers. They also determine which departments or external partners are granted access to their data using access control mechanisms.
- *Consuming business units request access to specific data products via the central platform.* Once access is granted, they are responsible for ensuring their software complies with schema contracts, data protection rules, and other governance policies.

- *Data platform team provides the infrastructure and the processes such that the other teams can work seamlessly together and don't need to contact the data platform team for every change required.* This decentralized approach ensures significantly higher data quality and simplifies data exchange within the company.

By enabling a data mesh to operate in near real-time, technologies such as Kafka reduce the time required to move data across departments. This allows companies to process larger data volumes efficiently, extract more value from the data, and adapt to changes with greater agility.

18.2 Liberating data from core systems with Kafka

Core systems are widely used in larger companies, whether in banking, insurance, manufacturing, or larger service providers. These core systems are precisely what the term suggests: they are essential components for the central processes of the company. Often, they are large, monolithic applications based on a central database. They have supported critical business processes for decades and are indispensable.

The problem is that they are often cumbersome and very challenging to adapt to modern customer demands. For instance, banks don't want end customers to directly access the core banking system through a mobile app, as it would likely become quickly overwhelmed. Many online shops face similar challenges with an SAP inventory management system behind them, for example.

The core system, which manages essential processes, needs to be decoupled from new, agile services that meet new customer requirements. We need to determine how to exchange data between systems so that it arrives in the target system as close to real time as possible, without overloading the core system. Additionally, we must ensure that core processes continue to be handled by the core system because these systems won't disappear overnight.

A popular idea in the Kafka ecosystem is to liberate data from core systems and build applications based on this liberated data to address new requirements, as shown in figure 18.4. The challenge is to minimize interaction with the core system, particularly to avoid custom developments that are costly to maintain.

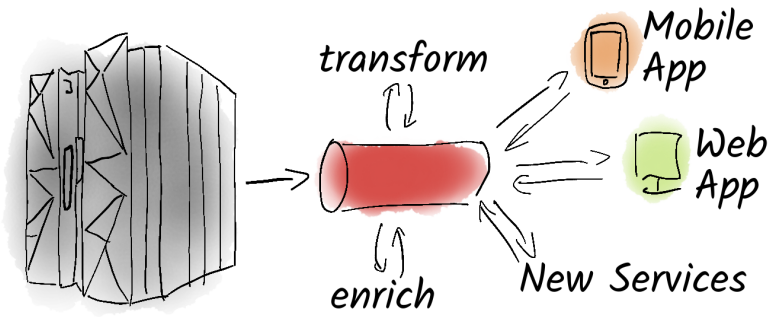


Figure 18.4 With Kafka, we can cost-effectively provide data from mainframes or core systems to other services.

One solution is to use Debezium to write data from the core system's database to Kafka using change data capture (CDC), as shown in figure 18.5. The challenge here is that we push the core system's data format directly to Kafka. Typically, only a few experts are familiar with these data formats, making it nearly impossible to transfer this knowledge to other teams in the necessary depth. Furthermore, the data is normalized and often spread across numerous tables, including various versions, and so on.

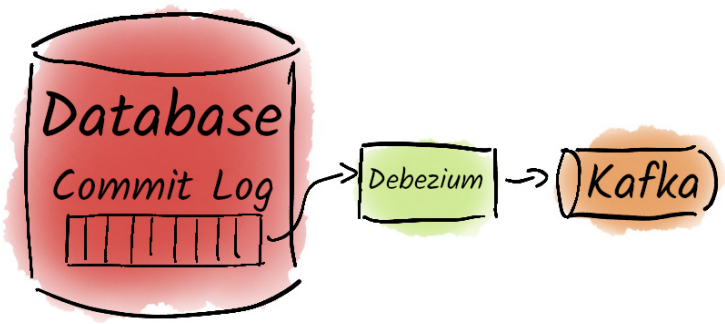


Figure 18.5 Debezium accesses the commit log of the database directly.

This means that the raw data shouldn't be reused by other departments. Instead, new data structures should be developed for other applications to use. Because the raw data is in Kafka, additional pipelines can be built as needed to accommodate further use cases.

There are different approaches for implementation. Ideally, the core system would natively provide an event stream, but this is rarely the case. Another approach is to create a Kafka Streams application that processes the raw data from the core system and makes it available in a consolidated data format to other applications. The challenge with this approach is that it requires a large number of joins and state stores, making the Kafka Streams application very complex and difficult to operate.

For many use cases, however, finding a better solution is challenging. An example of this complexity comes from a customer with the following question: "I have a thousand-line join across 200 tables. How do I implement this in Kafka Streams?" The best approach may be to avoid it altogether.

Another possibility is to implement these joins in a relational database anyway. The core system's database is usually not suitable for this, as it often operates at its load limit. Instead, data can be transferred to a second relational database via CDC. This could even involve a different database system. The complex (join) operations can then be more easily performed in SQL, and the data results can be written to outbox tables, which can subsequently be pushed to Kafka for further use via CDC.

While it's exciting to liberate data from core systems and make it available to other services, the process is far from simple. There are numerous challenges but also significant opportunities for growing companies in this space.

18.3 Kafka for big data

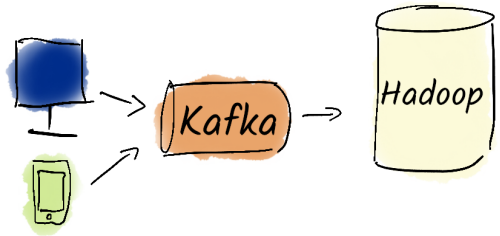


Figure 18.6 Kafka was originally developed at LinkedIn to quickly, securely, and efficiently move large amounts of data from sensors on the website and app into their Hadoop cluster.

Kafka was initially designed at LinkedIn to transfer large volumes of data from point A to point B. We summarize this in the context of the big data use case. The LinkedIn team was searching for a solution to send all data generated on the website to the central Hadoop cluster—every click, every comment, every like, essentially every interaction that users performed on LinkedIn, as shown in figure 18.6. This data needed to be measured and analyzed. The goal was to find or develop a system that could reliably and efficiently deliver data to Hadoop. Because the team couldn't find a solution that met LinkedIn's requirements at the time, they built the system we now know as Kafka.

The concept of the log was central to the system from the beginning and significantly contributed to its success. By using the log in Kafka, the necessary data throughput can be achieved because the required operations are relatively trivial. Even if Hadoop faced problems, the data wasn't lost as it was stored in the log. Additionally, even with significant fluctuations in LinkedIn's usage, the log allowed LinkedIn to buffer load spikes and reliably deliver data to the target system.

This use case remains highly relevant for the deployment of Kafka. It involves sending large amounts of highly valuable data from point A to point B, which is relatively straightforward. Some of our customers have reported that, given their data volumes, Kafka is even the system with the lowest operational overhead in their entire data processing pipeline.

Of course, there are exciting challenges at a certain scale. How can systems be optimized for efficiency to achieve the necessary data throughput with minimal effort and cost? How can peak performance be optimized? How do we operate a large Kafka cluster? A significant portion of the answers to these questions has been addressed in this book; however, at a certain size, we must delve deeper into the details. From an architectural perspective, this use case is quite manageable and almost a bit mundane—it's about connecting two (or more) systems.

18.4 Kafka for the Industrial Internet of Things

Kafka is also very intriguing in industrial applications. The amount of data produced continues to rise, whether in factories, vehicles, traditional power plants, or wind turbines. But what can companies do with all this data? Some of this data, such as that collected in warehouses and fulfillment centers, is interesting for end customers. Other data plays a critical role in quality management, especially in industries such as

pharmaceuticals, where precise oversight of production processes is essential to ensure the manufacturing of clean, high-quality medications.

Beyond these essential functions, other data is useful for understanding the overall performance of a factory, tracking production rates, and identifying inefficiencies. Some data points are key to optimizing machine performance and preventing potential failures, a practice commonly referred to as *predictive maintenance*.

As organizations look to harness the full value of their data, the ability to view and analyze it in real time becomes crucial. Many industrial processes require instant insights to drive decision-making and control systems.

The challenge, however, is how to centralize and manage the collection of data from a diverse range of machines and devices while making it available for different use cases. This is where Kafka shines, as it supports both real-time data streams and batch-based reports, making it a versatile solution for a wide range of Industrial Internet of Things (IIoT) applications.

18.4.1 Use cases for Kafka in the IIoT

To better understand the potential of Kafka in IIoT environments, here are some specific use cases where Kafka proves to be an invaluable tool:

- *Predictive maintenance*—Kafka enables the collection and real-time streaming of sensor data from industrial machines, allowing organizations to monitor the health of equipment continuously. This data feeds predictive maintenance models, helping to forecast failures before they occur and optimize maintenance schedules to minimize downtime and reduce operational costs.
- *Factory automation*—In smart factories, Kafka serves as the backbone for real-time data integration. It connects various IoT sensors across the production line, enabling seamless communication between devices and machines. This real-time data helps automate processes such as adjusting machine parameters or reallocating resources based on production demands, ensuring smoother operations and improved efficiency.
- *Supply chain optimization*—Kafka streamlines data flow across the supply chain by collecting data from various points, such as IoT-enabled vehicles, warehouses, and suppliers. This enables real-time tracking and dynamic adjustments to the supply chain, such as optimizing delivery routes, managing inventory levels, and predicting delays, which enhances overall operational efficiency.
- *Energy management*—Kafka is also crucial for energy management in industrial environments, such as power plants and wind farms. By collecting real-time data from energy-producing equipment such as turbines and generators, Kafka enables the monitoring and optimization of energy production. This data is essential for enhancing energy efficiency, reducing waste, and improving overall system reliability.
- *Real-time production analytics*—In manufacturing, Kafka supports real-time analytics by collecting data from various sensors on the production floor. This data

can be analyzed in real time to identify problems such as production bottlenecks, quality control problems, or operational inefficiencies. With this insight, factory managers can make quick decisions to optimize the production process and improve throughput.

18.4.2 Data storage and retention challenges

First, we must consider where and how we want to store the data. Kafka is well-suited for many use cases in this regard. We already know it as a system for exchanging data between systems in near real-time and for storing it over extended periods. In many applications, an intriguing question arises: How long should the data be retained in Kafka? It should certainly be long enough for the interested systems to retrieve the data.

Another consideration is whether historical data should also be stored in Kafka. This can be very costly in a traditional Kafka environment, but it has become very attractive with tiered storage. With tiered storage, organizations can efficiently manage data retention costs by keeping frequently accessed data in Kafka while moving less critical data to lower-cost storage solutions. This dual approach allows companies to balance performance and cost-effectiveness.

However, offloading historical data to another system introduces new challenges. For instance, organizations must ensure that they can access different data depending on its age, and this can complicate data retrieval processes. Therefore, tiered storage presents a practical solution by addressing many of these problems while allowing for a more streamlined data management process.

18.4.3 Data integration and access management

The second question is how to efficiently get the data from the machines into Kafka. In our opinion, the primary concern here is how reliable and powerful the machines and internet connection are. If the computational and internet resources are sufficient, it's very appealing to write directly to Kafka.

This simplifies development but may require more computing power than is available on a small microcontroller. Additionally, we need to consider what to do if a connection to Kafka is temporarily lost. We should also avoid granting access to Kafka from untrusted devices. Often, a more specialized protocol such as *Message Queuing Telemetry Transport* (MQTT) presents a more interesting solution. The end devices communicate with the MQTT broker, which then forwards the data to Kafka, as shown in figure 18.7.

Again, we encounter the question of where to direct the data once it's in Kafka. When it comes to integrating analytical software, we can usually use an existing connector to Kafka. If we want to write aggregated data into a database, we can also rely on a Java Database Connectivity (JDBC) connector. For custom software, we can either develop our own consumers or, in many cases, use Kafka Streams applications for further data processing.

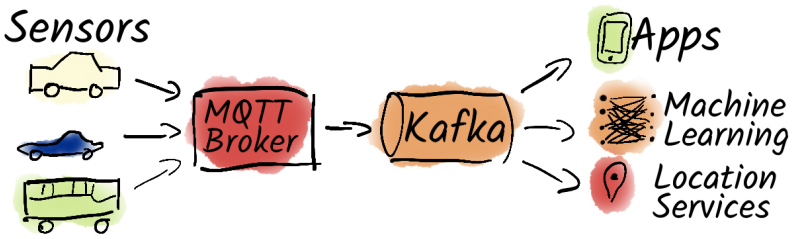


Figure 18.7 Kafka as an interface between MQTT and other applications

It's also important to remember that data often creates demand. Accessing data from Kafka is straightforward, so we encourage our clients to consider self-service tools from the start. This way, the Kafka team isn't overwhelmed with trivial requests, and teams can independently access the data within the established processes.

Often, historical data storage is initially avoided for cost reasons, even though it can be particularly valuable for nonpersonal data. Especially in industrial use cases, we see the risk that someone will eventually come along asking for historical data to train algorithms.

As previously mentioned, the topics of security and data protection in the Kafka realm come with challenges. Kafka can only set permissions at the topic level, meaning we're reluctant to grant end users read rights to entire topics. However, it's possible to give users write rights for data collection. When granular permissions are required, it may still be feasible to set up one topic per customer for a few clients, but this becomes impractical for consumer devices. Instead, application logic in the backend should handle rights management. The use of a *Kafka proxy*, such as *Kroxylicious*, is also an intriguing option.

18.4.4 When to use multiple Kafka clusters

This context often raises the question of how many Kafka clusters are necessary to manage the data effectively. In our experience, the answer largely depends on organizational and technical factors, such as who manages the clusters and the specific needs of different teams. Sometimes, it makes sense to have a raw data cluster, and then, when further processing steps are needed, an additional cluster is provided for the teams.

The downside of using multiple Kafka clusters is that it involves significant additional effort in terms of management and maintenance. Therefore, the recommendation is to aim for a single, centralized Kafka cluster unless there are specific technical or organizational constraints that justify the need for multiple clusters, such as the following:

- *Performance isolation*—Different teams may require varying levels of performance or dedicated resources for their Kafka clusters. For example, a high-throughput, low-latency Kafka cluster might be necessary for real-time processing in applications such as fraud detection, while another cluster could handle batch

processing or archival data with less stringent performance requirements, such as generating reports from historical data.

- *Data security*—Security concerns can also drive the decision to use separate clusters. For example, an HR department may need to isolate its Kafka cluster due to privacy regulations such as General Data Protection Regulation (GDPR), ensuring sensitive employee data is kept separate from other operational data. Meanwhile, the manufacturing department may use another cluster for real-time operational data related to production metrics, with different security policies.
- *Geographical distribution*—Organizations with distributed teams across multiple regions might prefer localized Kafka clusters to ensure faster data access. For example, a company with offices in North America and Europe may have separate Kafka clusters to comply with regional data sovereignty laws and reduce latency by storing data closer to where it's being processed.
- *Scalability and fault tolerance*—In very large enterprises, Kafka clusters might be split based on usage scale to ensure they are scalable and fault-tolerant. For example, a global retailer with vast data processing needs might separate their clusters for transactional data, inventory data, and customer analytics, ensuring that each cluster can grow independently without overloading a single Kafka cluster with too many diverse workloads.

18.5 What Kafka is not

While Kafka is well-suited for many use cases, it's not appropriate for all scenarios. One common question that arises is whether Kafka can be used as a database, or if it should be used for synchronous communication. In the following subsections, we'll address some situations where Kafka isn't the best solution. Of course, we can't cover every possible use case in this book, but it's important to evaluate the tradeoffs and determine the best approach based on your organization's specific needs.

18.5.1 Kafka isn't a relational database

The question of whether Kafka is a database or not is likely as old as Kafka itself. Kafka excels at distributing data between systems and, with the help of tiered storage, can also store data for long periods. By many definitions, it can thus be considered a database.

However, it's not a relational database! Relational databases are very good at representing the current state of a system and executing complex queries across multiple tables. Although we have the capability to perform joins using Kafka Streams, Kafka shouldn't be used for use cases that involve querying the current state.

There are several reasons for this. Kafka offers much weaker guarantees than relational databases. Relational databases provide the full range of *ACID guarantees* (*atomicity*, *consistency*, *isolation*, *durability*), while Kafka can guarantee atomic operations and good consistency but lacks transactional isolation and strong consistency across multiple partitions. The most notable aspect of this is that Kafka can't be used synchronously;

it always operates asynchronously. Furthermore, joins in Kafka are significantly more expensive than in traditional relational databases.

Kafka is very well-suited for asynchronous data exchange between services, and each message in Kafka should be self-contained and meaningful. This means that messages should be denormalized. In relational databases, data is typically highly normalized, stored across many tables.

Conversely, Kafka is poorly suited for denormalizing normalized data using large joins. Denormalization refers to the process of combining related data from multiple tables into a single, flattened structure, typically by joining them together. For instance, imagine we have a normalized database with a `customers` table and an `orders` table.

Denormalization involves merging customer and order information into a single message, so consumers don't have to perform the join themselves. However, performing such joins within Kafka or in a real-time processing pipeline can be inefficient and complex, especially with large amounts of data. This should ideally occur in the relational database beforehand and then, for example, be written to Kafka through an outbox table.

However, it's essential to note that this isn't always possible. Sometimes, data needs to be consumed in a normalized form and then be denormalized through joins. If feasible, this should be avoided, and we recommend not making normalized topics accessible to regular consumers.

We also find it challenging to write data one-to-one from Kafka in its denormalized state into a relational database. Such data should ideally be normalized in the consumer beforehand.

> **TIP** We use Kafka for data exchange between systems and relational databases for service-internal data. This way, we fully exploit the advantages of each system and avoid their disadvantages. Even in a company that fully embraces data mesh, many databases will still be found, as they are better suited for storing service-internal data.

18.5.2 Kafka isn't a synchronous communication interface

A common anti-pattern in using asynchronous communication interfaces is to simulate synchrony. For example, if a user clicks a button in a web interface, data is written to Kafka in the backend, and the system waits for the data to be processed in another system and get sent back via Kafka. This could cause the user to wait a long time for something to happen after clicking the button.

Instead, as displayed in figure 18.8, it's generally a better idea to write the action into a relational and synchronous database and inform the user that the system has received their input. The data can then be moved to Kafka using CDC and the Outbox Pattern and processed asynchronously in the third system. Once the third system has finished processing, it updates the process step via Kafka, allowing the backend to inform the user about the progress of the process.

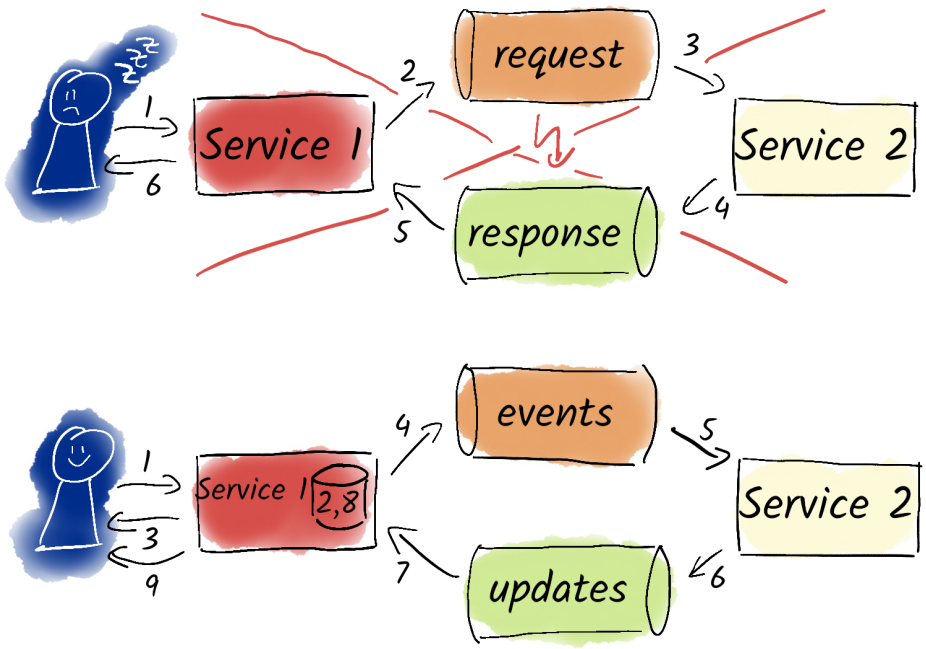


Figure 18.8 We shouldn't use Kafka for synchronous request-response communication. Instead, we notify users about asynchronous processes and confirm the process when it's done.

If the backend doesn't require a database, it's perfectly fine for the frontend to make a synchronous REST call, and then the backend produces the data to Kafka. Upon successful production, the backend confirms to the frontend that the action was triggered.

18.5.3 Kafka isn't a file exchange platform

We've often been asked how to best exchange PDFs via Kafka, as these PDFs are also messages. Kafka is optimized for exchanging many small messages (up to 1 MB by default). Generally, these messages should be machine-readable, so sending PDFs via Kafka is usually a very bad idea. They are too large and not suited for Kafka in terms of message type.

For this use case, it would be much better to send the data contained in the PDF in a machine-readable format (e.g., as a JSON object) over Kafka and then, after processing, send it to the PDF generation service, which would then create the PDF. Alternatively, as shown in figure 18.9, the PDF should be stored in an object storage solution, either in the cloud or on-premises, and Kafka should instead send a message indicating that the PDF can be found at a specific address. The consumer then receives the notification via Kafka and retrieves the PDF from the object storage.

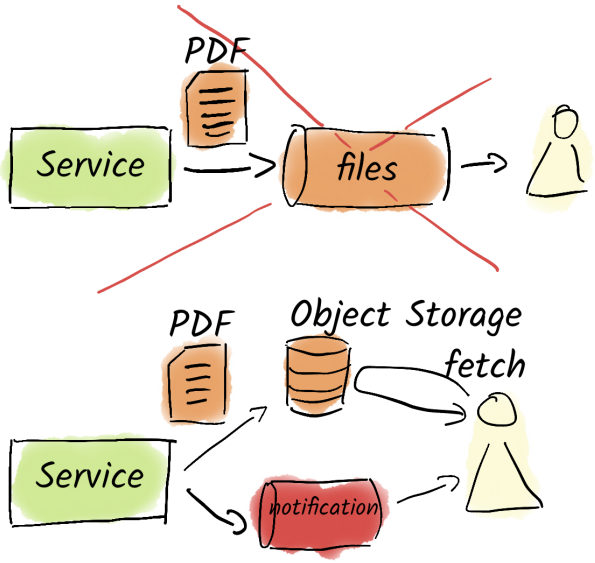


Figure 18.9 Don't use Kafka to exchange files between systems. Use, for example, an object storage solution instead, and then use Kafka to notify others about the file that was uploaded or changed.

Sometimes, there are also Simple Object Access Protocol (SOAP) messages that are significantly larger than 1 MB. In most cases, such a SOAP message consists of thousands of events, which means the message can often be semantically split, with one message sent to Kafka per event. This is the better approach from both a technical and architectural perspective. Kafka is designed so that each message describes a self-contained event, making subsequent consumer operations much easier.

18.5.4 Kafka for small applications is questionable

Kafka excels at exchanging data among many services and especially among teams. Kafka is optimized to reduce coupling between services and thus also among teams, allowing them to work more independently. Naturally, Kafka is also very well-suited for moving large amounts of data from A to B. However, if neither of these conditions is met, and Kafka is to be used by a single team to exchange (for Kafka) small amounts of data within a monolithic application, we generally recommend avoiding Kafka, even if the log itself is an exciting concept.

Here are more concrete numbers: if there are fewer than 1,000 messages per hour with no expectation that the data volume will increase significantly in the next year, and only one team uses the data, we generally advise against using Kafka.

> **NOTE** If Kafka is already part of the enterprise architecture, it often makes sense to use it nonetheless. However, introducing Kafka for such a small application without prospects for further use cases is questionable.

In such cases, one or more tables in a PostgreSQL database can be misused as a log. In PostgreSQL, we can also achieve fast lookups through indexes and use the power

of SQL much more easily, without having to operate additional complex systems. And when the time for scaling does come, a significant portion of the system will likely need to be rewritten anyway.

If a monolith technically and organizationally suffices for a given use case, the complexity of the system shouldn't be voluntarily increased without necessity. A monolith isn't inherently bad; we just need to ensure that, if scaling unexpectedly becomes necessary, the architecture is reconsidered. But perhaps a database index will suffice to improve performance.

Of course, this isn't trendy, but as a small company, there's no need for anything else. This results in far fewer problems, quicker development, and much greater flexibility.

18.5.5 Kafka isn't a substitute for good architecture

We've often seen organizations turn to Kafka after unsuccessfully trying multiple messaging platforms, hoping it will solve their problems. However, in our experience, the real problem often lies in a lack of expertise in both how to build the system properly and how to use Kafka effectively.

If Kafka simply replaces a previous system without addressing the underlying gaps in understanding the system's requirements, nonfunctional requirements, cost, time, and scalability, little will improve. Kafka isn't a magic solution, and without proper planning and expertise, organizations may continue to face the same challenges.

In addition, just because we're using Kafka currently doesn't mean it should be used for everything. Some of the good, but also problematic, aspects have been discussed in previous sections. The system that makes the most sense for a specific application should always be used, considering the unique characteristics of each company.

Of course, it's not always sensible to use the perfect product for every use case if it means introducing and managing an entirely new, complex system. Sometimes, it's acceptable to use Kafka where it's not 100% optimal. But as consultants (and trainers) like to say: "It depends."

Good architecture is a fundamental prerequisite for the use of any IT systems. Kafka adds the challenge that it provides users with many freedoms. If garbage is produced after Kafka, garbage will also come out at the consumer side. Kafka doesn't care about the messages it transports. This can lead to significant problems if thorough considerations regarding system architecture aren't made in advance.

We hope this book, and especially this chapter, provides architectural teams with the tools to better decide where it makes sense to use Kafka and where it doesn't. As always, think first, plan, and then execute.

Summary

- The data mesh decentralizes data management, empowering departments to take ownership of their data products while the data team focuses on technical support.
- Key principles of the data mesh include treating data as a product, domain ownership, self-service data infrastructure, and decentralized governance.

- Kafka acts as a central hub for real-time data exchange within a data mesh, enhancing data quality and accelerating data movement across the organization.
- Core systems are essential but rigid; liberating data from them with Kafka enables agile services while minimizing direct interaction and maintenance costs.
- Debezium facilitates real-time data exchange using change data capture (CDC), but new data structures must be created to avoid complexity and improve usability for other applications.
- Kafka was designed for efficiently transferring large volumes of data in big data environments by using a log-based architecture for reliable delivery and buffering against load spikes.
- Kafka efficiently handles the growing data volumes from industrial applications, enabling real-time monitoring, predictive maintenance, and centralized data collection for various use cases.
- Tiered storage in Kafka allows organizations to balance performance and cost by retaining frequently accessed data while offloading historical data to lower-cost storage, streamlining data management.
- Effective data integration and access management can be achieved through protocols such as Message Queuing Telemetry Transport (MQTT) for reliable data transmission, while centralized Kafka clusters simplify management and support self-service tools for teams.
- Kafka isn't a relational database, making it unsuitable for complex queries or maintaining the current state, as it lacks full ACID guarantees, particularly transactional isolation.
- Kafka isn't a synchronous communication interface; it operates asynchronously and requires relational databases for immediate feedback.
- Kafka isn't a file exchange platform, as it's not designed for large files such as PDFs; it's more effective to send machine-readable data or links to files stored externally.
- Kafka isn't ideal for small applications with low data volumes. In these cases, simpler solutions (e.g., a database) may be more efficient.

appendix A
Setting up a Kafka test environment

In this appendix, we'll provide a brief description of how to set up a Kafka test environment on a single machine, such as a personal laptop. This setup should never be used for a production environment, but it's great to use throughout the book. In contrast to many blog posts on the internet, we focus on setting up a multi-node environment to be able to see how Kafka survives partial failures.

In figure A.1 we display the test environment, which consists of three Kafka brokers. Broker 1 listens on port `9092`, Broker 2 on `9093`, and Broker 3 on `9094`. To simplify the setup, we'll run the brokers in dual mode, meaning that the three brokers will act both as controllers and normal brokers. The controllers need additional listeners, and so the controller of Broker 1 will listen on port `9192`, Broker 2's controller will listen on `9193`, and Broker 3's will listen on `9194`.

A.1 Operating systems

Apache Kafka runs on the Java Virtual Machine (JVM) and thus supports all major operating systems. We have tested this environment on a macOS 15.2 Sequoia on Apple Silicon and on an Ubuntu 24.04 LTS machine with an Intel CPU. We recommend using the Windows Subsystem for Linux (WSL) on Windows.

A.2 Downloading Kafka

You can always find the most up-to-date version of Kafka at https://kafka.apache.org/downloads. At the time of writing, the latest version is 3.9.0. As some parts of

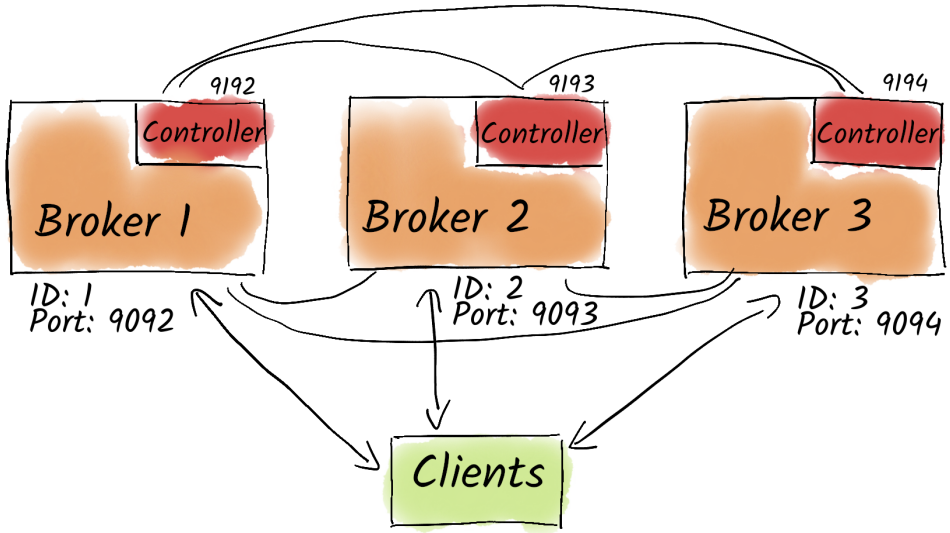


Figure A.1 Our test environment consists of three Kafka brokers in dual mode.

Kafka are still written in Scala, you'll find downloads of the form `kafka_2.13-3.9.0.tgz` where `2.13` is the Scala version and `3.9.0` is the Kafka version. We recommend always using the latest Scala version. Let's download Kafka and extract it to the directory `~/kafka`:

```
$ wget "https://downloads.apache.org/kafka/3.9.0/kafka_2.13-3.9.0.tgz"
$ tar xfz kafka_2.13-3.9.0.tgz
$ rm kafka_2.13-3.9.0.tgz
$ mv kafka_2.13-3.9.0/ ~/kafka
```

We'll find the following subdirectories in the `~/kafka` directory:

- `LICENSE`, `NOTICE`—Files containing license information
- `bin`—Command-line tools to interact with Kafka
- `config`—Configuration files
- `libs`—Java library files (`*.jar`)
- `site-docs`—Documentation

To be able to call the command-line tools without providing the full path, it's best to add the `bin` directory to the `PATH` variable:

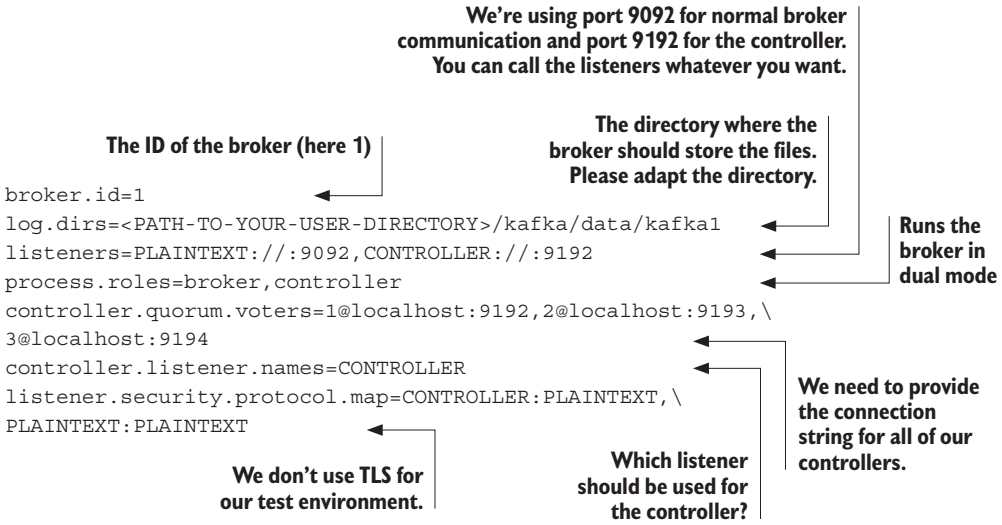
```
$ export PATH=~/kafka/bin:"$PATH"
```

Of course, you can also add this line to your `~/.bashrc`, `~/.zshrc` or the file for your shell.

NOTE Kafka 4.0.0 was released in March 2025. At the time of writing this book, all examples were tested using Kafka 3.9.0, but we do cover all the important new features of Kafka 4.0.0, such as KRaft, queues, and the new rebalance protocol.

A.3 Configuring Kafka

Now let's create the configuration files for the Kafka brokers in the directory `~/kafka/config/kafka1.properties`:




```
broker.id=1
log.dirs=<PATH-TO-YOUR-USER-DIRECTORY>/kafka/data/kafka1
listeners=PLAINTEXT://:9092,CONTROLLER://:9192
process.roles=broker,controller
controller.quorum.voters=1@localhost:9192,2@localhost:9193,\
3@localhost:9194
controller.listener.names=CONTROLLER
listener.security.protocol.map=CONTROLLER:PLAINTEXT,\
PLAINTEXT:PLAINTEXT
```

Let's also configure Broker 2 in `~/kafka/config/kafka2.properties`:

```
broker.id=2
log.dirs=<PATH-TO-YOUR-USER-DIRECTORY>/kafka/data/kafka2
listeners=PLAINTEXT://:9093,CONTROLLER://:9193
process.roles=broker,controller
controller.quorum.voters=1@localhost:9192,2@localhost:9193,3@localhost:9194
controller.listener.names=CONTROLLER
listener.security.protocol.map=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
```

And, finally, we'll configure Broker 3 in `~/kafka/config/kafka3.properties`:

```
broker.id=3
log.dirs=<PATH-TO-YOUR-USER-DIRECTORY>/kafka/data/kafka3
listeners=PLAINTEXT://:9094,CONTROLLER://:9194
process.roles=broker,controller
controller.quorum.voters=1@localhost:9192,2@localhost:9193,3@localhost:9194
controller.listener.names=CONTROLLER
listener.security.protocol.map=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
```

A.4 Preparing the data directories

First, we'll list the directories where the brokers should store the data. Then, we need to initialize (*format*) the directories so that we can use them in our cluster. For this, we need to create a cluster ID so that it's impossible to use a directory with another cluster, which limits the mistakes you can make.

```
$ mkdir -p ~/kafka/data/kafka1 ~/kafka/data/kafka2 \
    ~/kafka/data/kafka3
$ export KAFKA_CLUSTER_ID=\
    "$(kafka-storage.sh random-uuid)"
$ kafka-storage.sh format -t $KAFKA_CLUSTER_ID \
  -c ~/kafka/config/kafka1.properties
Formatting metadata directory <PATH-TO-YOUR-USER-DIRECTORY>
/kafka/data/kafka1 with metadata.version 3.9-IV0.
$ kafka-storage.sh format -t $KAFKA_CLUSTER_ID \
    -c ~/kafka/config/kafka2.properties
Formatting metadata directory <PATH-TO-YOUR-USER-DIRECTORY>
/kafka/data/kafka2 with metadata.version 3.9-IV0.
$ kafka-storage.sh format -t $KAFKA_CLUSTER_ID \
    -c ~/kafka/config/kafka3.properties
Formatting metadata directory <PATH-TO-YOUR-USER-DIRECTORY>
/kafka/data/kafka3 with metadata.version 3.9-IV0.
```

Creates the data directories

Generates a new Kafka cluster ID

Formats the directories

A.5 Starting Kafka

Now it's time to start our brokers. First, start the brokers in the foreground to see any possible errors; if everything works, start the brokers in the background as a daemon. Let's start the first broker:

```
~/kafka/bin/kafka-server-start.sh ~/kafka/config/kafka1.properties
```

If this command crashes immediately, you've misconfigured the broker. Check the configuration, and try again. Otherwise, the Kafka broker should be complaining about node 2 and 3 being unreachable:

```
WARN [RaftManager nodeId=1] Connection to node 2 (localhost/127.0.0.1:9193)
could not be established. Broker may not be available.
(org.apache.kafka.clients.NetworkClient)
```

This makes sense because we haven't started the other brokers yet. Let's open a new terminal window and start the second broker:

```
~/kafka/bin/kafka-server-start.sh ~/kafka/config/kafka2.properties
```

We should see a lot of log messages. The brokers will still be complaining about the missing node 3, but we might find the following line in between all log messages:

```
[KafkaRaftServer nodeId=2] Kafka Server started
```

So, let's finally start the remaining broker:

```
~/kafka/bin/kafka-server-start.sh ~/kafka/config/kafka3.properties
```

After that, we should see the following line in the logs of broker 3, and the brokers should have stopped complaining about not being able to connect to another node:

```
[KafkaRaftServer nodeId=3] Kafka Server started
```

Of course, we can leave it at that, but we can also stop the brokers one after another and replace them with brokers in daemon mode. This way, we won't be able to see the logs, but we also won't need to have three terminal windows open all the time:

```
# Stop Broker 1 with Ctrl-C
~/kafka/bin/kafka-server-start.sh \
    -daemon ~/kafka/config/kafka1.properties
# Stop Broker 2 with Ctrl-C
~/kafka/bin/kafka-server-start.sh \
    -daemon ~/kafka/config/kafka2.properties
# Stop Broker 3 with Ctrl-C
~/kafka/bin/kafka-server-start.sh
    -daemon ~/kafka/config/kafka3.properties
```

Starts the broker in the background

To check whether everything is still running, we can use the following command:

```
$ kafka-broker-api-versions.sh \
    --bootstrap-server localhost:9092,localhost:9093,localhost:9094
localhost:9092 (id: 1 rack: null) -> (
# A lot of text
)
localhost:9093 (id: 2 rack: null) -> (
# A lot of text
)
localhost:9094 (id: 3 rack: null) -> (
# A lot of text
)
```

On the one hand, it outputs the API versions each broker supports, but more importantly, it shows us easily which brokers are online.

A.6 Stopping Kafka

To stop all Kafka brokers, we can simply execute the following command:

```
~/kafka/bin/kafka-server-stop.sh
```

But this command stops all Kafka brokers. To stop a single broker, we can create a new script `~/kafka/bin/kafka-broker-stop.sh`:

```
#!/bin/bash
BROKER_ID="$1"

if [ -z "$BROKER_ID" ]; then
  echo "usage ./kafka-broker-stop.sh [BROKER-ID]"
  exit 1
fi

PIDS=$(ps ax | grep -i 'kafka\.Kafka' | grep java \
    | grep "kafka${BROKER_ID}.properties" | grep -v grep | awk '{print $1}')

if [ -z "$PIDS" ]; then
  echo "No kafka server to stop"
  exit 1
else
  kill -s TERM $PIDS
fi
```

Instead of stopping all brokers, it will stop only the one for which the broker ID matches. To stop broker 1, for example, we can use the following command:

```
$ chmod +x ~/kafka/bin/kafka-broker-stop.sh
$ ~/kafka/bin/kafka-broker-stop.sh 1
```

We can check with the `kafka-broker-api-versions.sh` script to find out whether this worked:

```
$ kafka-broker-api-versions.sh \
    --bootstrap-server localhost:9092,localhost:9093,localhost:9094
Connection to node -1 (localhost/127.0.0.1:9092) could not be established.
Broker may not be available.
localhost:9093 (id: 2 rack: null) -> (
# A lot of text
)
localhost:9094 (id: 3 rack: null) -> (
# A lot of text
)
```

Broker 1 is missing, and there's an error message instead.

We can start the broker with the command shown earlier again or stop the whole environment. We wish you happy streaming, and have fun with the rest of the book.

appendix B
Monitoring setup

In this appendix, we'll provide a brief overview of how to set up a basic monitoring system for our Kafka services using Prometheus (https://prometheus.io/) and Grafana (https://grafana.com/). While this isn't a dedicated monitoring guide, it will give you a foundational understanding of how to implement monitoring for your Kafka environment.

B.1 Prometheus

Prometheus is an open source system for monitoring and alerting. It periodically scrapes the metric endpoints of the targets to be monitored and stores, in addition to the metric name, a set of labels for identification in its time-series database.

Prometheus also supports service-discovery mechanisms to automatically find scrape targets. Additionally, there's an option to send metrics directly to Prometheus. Data can then be queried and aggregated using its own query language, *PromQL*. Alerts can also be defined based on PromQL. For visualization, Prometheus provides its own web interface.

To install Prometheus, we simply need to download the version compatible with our operating system from the official Prometheus website (https://prometheus.io/download/) and extract it. A general installation guide and an introduction to Prometheus are also available at https://mng.bz/YD5z.

Before running the Prometheus binary, however, we make a few changes to the default configuration. For this, we adjust the `prometheus.yml` file (see configuration documentation at https://mng.bz/GeKD):

```
global :
  scrape_interval: 15s
  evaluation_interval: 15s
alerting:
  alertmanagers:
  - static_configs:
    - targets: ["localhost:9095"]

rule_files:
- "rules.yml"
scrape_configs:
- job_name: "kafka"
  static_configs:
  - targets: ["localhost:9101","localhost:9102","localhost:9103"]
- job_name: "prometheus"
  static_configs:
  - targets: ["localhost:9090"]
```

The global configuration under `global` contains settings such as how often Prometheus should scrape its targets and how often alerts should be evaluated. The address for the *Prometheus Alertmanager*, which we'll explore in section 2.3, is specified under `alerting` (`localhost:9095`). Additionally, we define two jobs in the `scrape_config`. The `kafka` job scrapes the Java Management Extensions (JMX) metric endpoints of our Kafka brokers (`localhost:9101, localhost:9102, localhost:9103`), while the `prometheus` job collects metrics from Prometheus itself (`localhost:9090`). Under `rule_files`, we can specify a list of files that contain alerts for evaluation. In our minimal example, we define a simple alert in the `rules.yml` file:

```
groups:
- name: Kafka
  rules:
  - alert: Kafka Metrics Not Reachable
    expr: up{job="kafka"} == 0
    for: 1m
```

Alerts are defined within an alerting group. Within a group, we can define different evaluation intervals from the global default value. This can be useful, for example, if a specific query requires a lot of resources and should therefore be executed less frequently.

The actual list of alerts is defined under `rules`. Alerts always have a name, set by the `alert` field (e.g., `Kafka Metrics Not Reachable`). The expression to evaluate is found under `expr` (`up{job="kafka"} == 0`), and the duration for which the expression must be true before the alert is triggered is defined with `for` (`1m`). In our example, we've

created an alert that triggers when one of our Kafka metric endpoints is unreachable for 1 minute.

After adjusting our configuration, we can start Prometheus by running the Prometheus binary. Then, we can access the web interface in our browser at port `9090`. We should immediately see the first metrics. Prometheus also provides autocomplete for queries via PromQL. Two screenshots of the web interface are shown in figure B.1 and figure B.2.

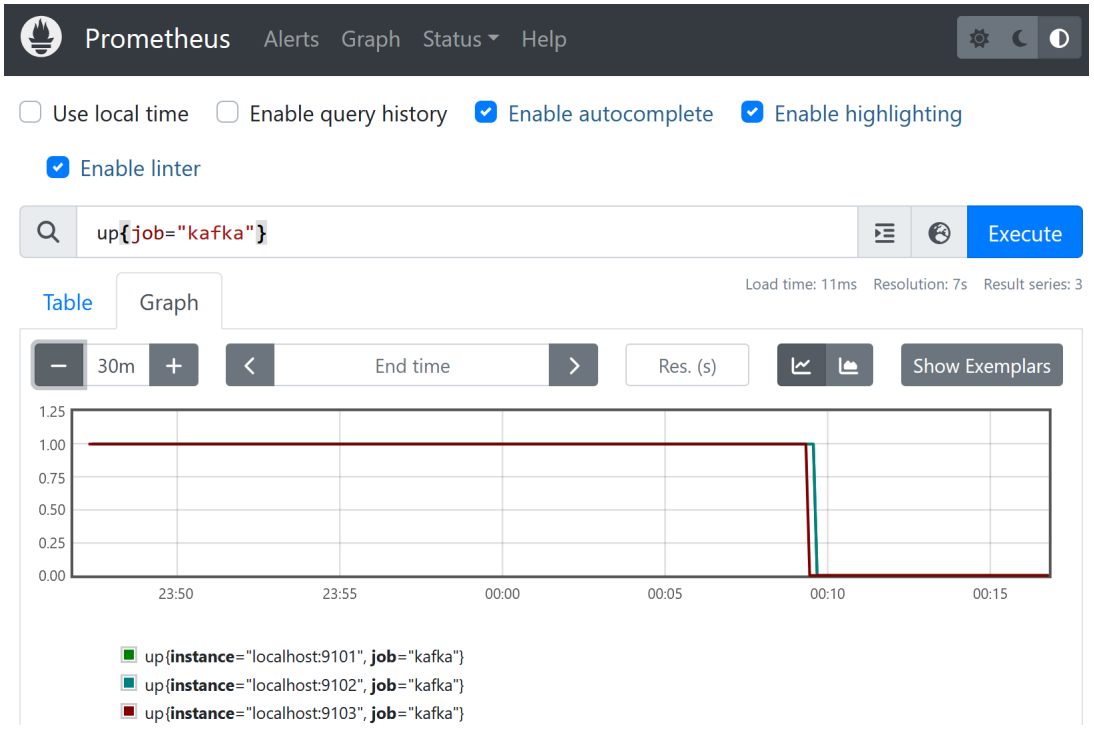


Figure B.1 Through the Prometheus web interface (accessible by default on port 9090), we can query and visualize Prometheus data using PromQL.

B.2 Prometheus Exporter

Not every service or server natively provides Prometheus metrics via HTTP. Fortunately, there are numerous official and unofficial Prometheus exporters that convert metrics from existing formats into Prometheus format and make them available via HTTP. A list of exporters is available at https://prometheus.io/docs/instrumenting/exporters/.

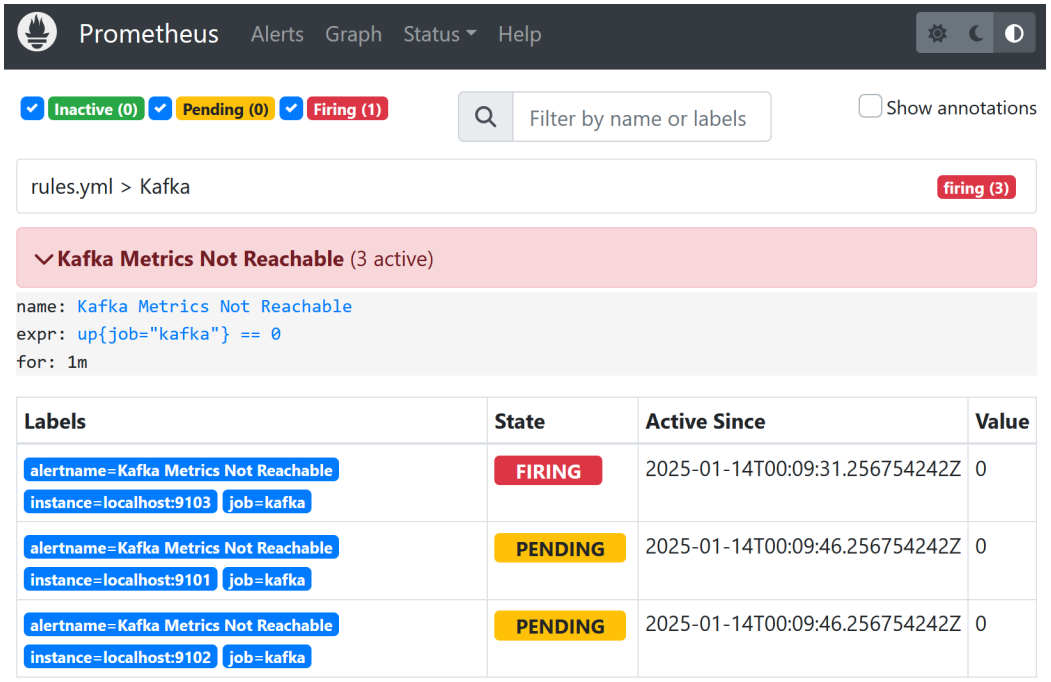


Figure B.2 Under Alerts, we find a list of all alerts. Alerts can be in one of three statuses: Inactive (expression is false), Pending (expression is true, but the minimum duration hasn't yet been reached), or Firing (expression is true and has been true for at least the specified duration).

For example, Kafka uses Java's *Java Management Extensions* (JMX) to forward various parameters to monitoring tools. We won't dive too deeply into this topic here, as there are better and more up-to-date resources available online or in our training sessions. Most monitoring tools we know support JMX, to varying degrees, out of the box.

For monitoring with Prometheus, we use the *Prometheus JMX Exporter* (https://prometheus.io/docs/instrumenting/exporters/) with the recommended configuration for Kafka. However, depending on the configuration, metric names may vary, and some metrics may not be provided at all.

Let's configure our Kafka test environment from appendix A to use the Prometheus Exporter so that Prometheus is able to scrape the metrics. The JMX Exporter runs as a Java agent. For that, we first download the JMX Exporter and also the configuration file:

Downloads the JMX Exporter

```
$ cd ~/kafka
$ wget https://github.com/prometheus/jmx_exporter/releases/download/1.1.0/\
  jmx_prometheus_javaagent-1.1.0.jar
$ wget https://raw.githubusercontent.com/prometheus/jmx_exporter/refs/\
  heads/main/examples/kafka-kraft-3_0_0.yml
```

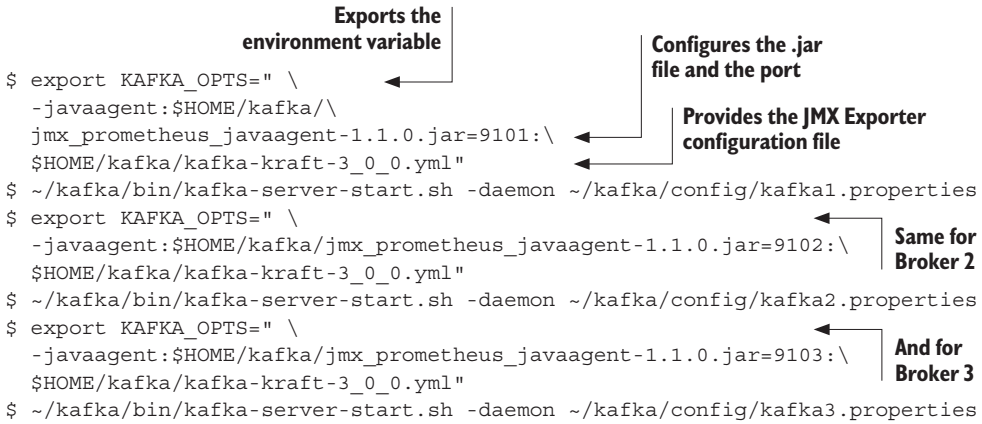
Downloads the configuration for Kafka

Next, we need to stop our Kafka brokers:

```
$ kafka-server-stop.sh
```

We add the Java agent by exporting the `KAFKA_OPTS` environment variable and then starting Kafka. In the variable, we're adding the JMX exporter `.jar` file as an agent and also provide the path to the JMX configuration file. Additionally we need to tell the JMX Exporter on which port to listen.

As configured in the Prometheus configuration file, the Broker 1 port is 9101, Broker 2 is 9102, and Broker 3 is 9103. After exporting the environment variable, we start the broker. Let' do this for all three brokers:




```
$ export KAFKA_OPTS=" \
  -javaagent:$HOME/kafka/\
  jmx_prometheus_javaagent-1.1.0.jar=9101:\
  $HOME/kafka/kafka-kraft-3_0_0.yml"
$ ~/kafka/bin/kafka-server-start.sh -daemon ~/kafka/config/kafka1.properties
$ export KAFKA_OPTS=" \
  -javaagent:$HOME/kafka/jmx_prometheus_javaagent-1.1.0.jar=9102:\
  $HOME/kafka/kafka-kraft-3_0_0.yml"
$ ~/kafka/bin/kafka-server-start.sh -daemon ~/kafka/config/kafka2.properties
$ export KAFKA_OPTS=" \
  -javaagent:$HOME/kafka/jmx_prometheus_javaagent-1.1.0.jar=9103:\
  $HOME/kafka/kafka-kraft-3_0_0.yml"
$ ~/kafka/bin/kafka-server-start.sh -daemon ~/kafka/config/kafka3.properties
```

To check whether this worked, we can query the metrics-endpoint with `curl`:

```
$ curl localhost:9101
# Long list of metrics
$ curl localhost:9102
# Long list of metrics
$ curl localhost:9103
# Long list of metrics
```

After a few seconds, we should also see the metrics arriving in Prometheus.

Another relevant exporter for monitoring Kafka is the *Prometheus Blackbox Exporter* (https://github.com/prometheus/blackbox_exporter). The Blackbox Exporter allows testing (probing) endpoints using various protocols, from simple TCP connection tests to more complex HTTPS requests. This enables monitoring the availability of services such as our Kafka brokers.

The Blackbox Exporter generates the `probe_success` metric, which indicates whether a probe was successful. Depending on the probe type, additional metrics, for example, HTTP request duration, including steps such as DNS queries or TCP connection establishment, are also generated. This allows for complex insights into the

end-to-end performance of a service, which isn't always explained solely by the service's CPU load, but often involves a complex interaction of multiple components.

Another useful tool is the *Prometheus Node Exporter* (https://github.com/prometheus/node_exporter), which collects metrics about the current host system. The Node Exporter uses a variety of collectors to read and transform the corresponding metrics. Primarily, it collects metrics related to CPU, memory, network, and filesystem usage. Specifically, the *Textfile Collector* can be used to construct any metrics based on text files.

B.3 Prometheus Alertmanager

The *Alertmanager* is also part of the Prometheus project and is responsible for managing alerts. It groups alerts and determines which recipients should receive notifications. Additionally, alerts or notifications can be temporarily silenced, which is particularly useful during scheduled maintenance. Inhibit rules can also be defined, allowing notifications for certain alerts to be suppressed based on other alerts.

> **TIP** When a basic problem occurs, such as a server or database being unreachable, many alerts can be triggered because related services are no longer fully functional. With well-defined inhibit rules, we can avoid having secondary alerts overwhelm the main problem by suppressing those secondary alerts. However, it's important to define inhibit rules carefully to prevent accidentally suppressing important alerts.

The Alertmanager can also be downloaded from the Prometheus download page. The configuration for the Alertmanager is defined in the `alertmanager.yml` file. For the Alertmanager to send notifications, we typically need to configure a valid recipient (and if we want to send emails, an SMTP configuration as well), but we'll skip that part here. Detailed documentation of the Alertmanager's configuration options can be found on the Prometheus website at https://mng.bz/9Y4j.

To start the Alertmanager, we execute the `alertmanager` binary. However, we need to overwrite the listening addresses because the default ports `9093` and `9094` are already in use by our Kafka brokers:

```
$ ./alertmanager --web.listen-address=:9095 \
  --cluster.listen-address=<address>
```

A screenshot of the web interface, accessible on port `9095` with active alerts, is shown in figure B.3.

B.4 Grafana

Grafana (https://grafana.com/), like Prometheus, is open source software. While the Prometheus UI itself supports only simple queries, Grafana allows the creation of complex dashboards from multiple data sources (e.g., Prometheus). This makes it possible to display several metrics within a single dashboard, providing a clearer view of complex relationships, which can be especially helpful when troubleshooting.

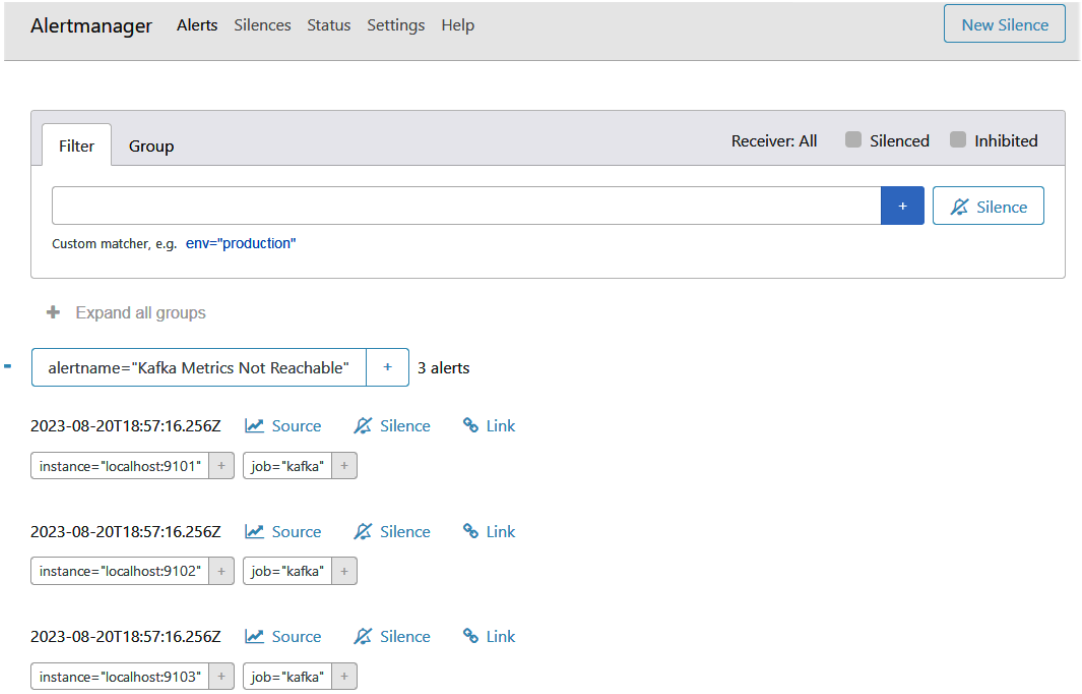


Figure B.3 The Alertmanager web interface is accessible on port 9095 (the default is 9093) and displays a list of active alerts. Additionally, through the web interface, we can temporarily silence alerts or notifications.

Additionally, dashboards can be equipped with dynamic variables to enable quick switching between different resources. Grafana also supports various authentication methods and provides granular permissions for dashboards and data sources.

Furthermore, Grafana allows for the creation and management of alerts, and a wide variety of plugins can extend its functionality as needed (https://grafana.com/grafana/plugins/).

Grafana can be downloaded from https://grafana.com/grafana/download/, either directly as a binary or via an installer or package manager. How you start Grafana depends on the installation method. The official Grafana documentation at https://mng.bz/jpza provides detailed information on how to configure authentication, plugins, and much more.

> **TIP** Good dashboards are an ideal foundation for creating meaningful alerts based on visualized metrics. They often serve as the first reference point when diagnosing problems. Therefore, it's advisable to invest resources in creating quality dashboards, as every second counts if there's a failure.

Figure B.4 shows a preconfigured Kafka dashboard from the Strimzi team with comprehensive information about the state of the Kafka cluster.

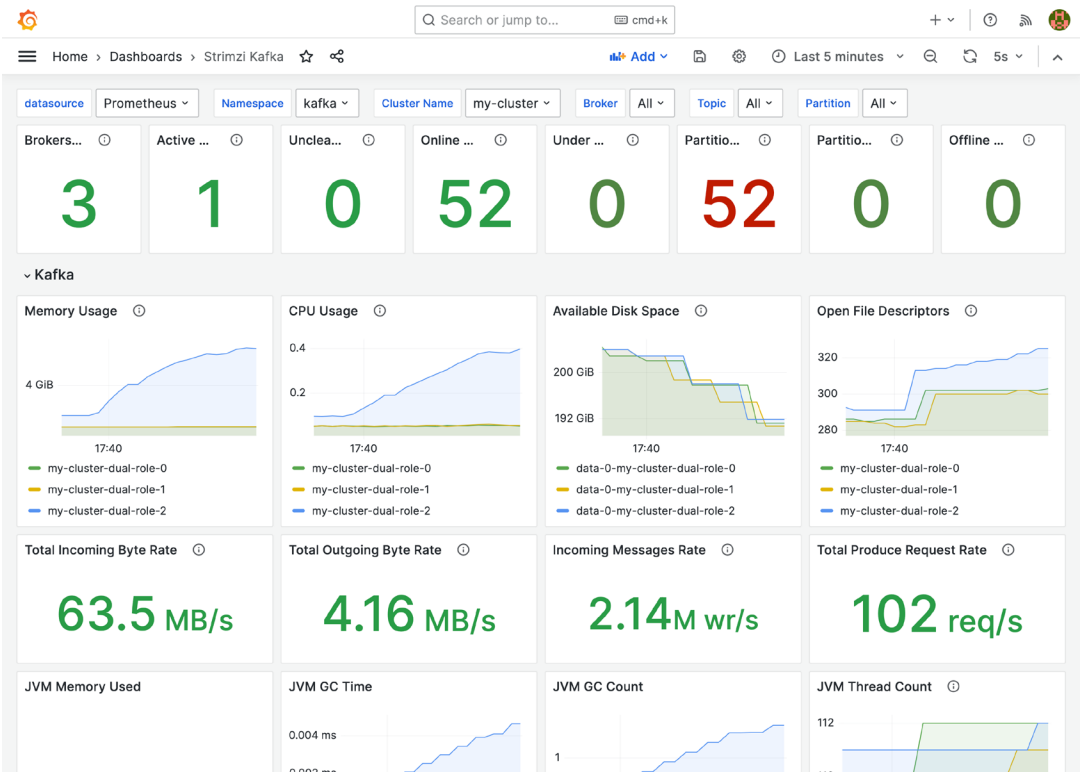


Figure B.4 A Kafka dashboard example based on the Strimzi dashboards

TIP The Strimzi community provides one of the best Kafka dashboards (https://mng.bz/W2Rg). However, they shouldn't be adopted blindly, as some may be not suitable for specific use cases. In any case, it's important to familiarize yourself with the dashboard so that it isn't being used for the first time in the event of a problem.

index

C

D

E

F

G

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Apache Kafka: Important Commands

PRODUCE AND CONSUME MESSAGES

```
kafka-console-producer.sh ↗Produce messages
      --bootstrap-server localhost:9092
      --topic mytopic ↗To which topic?
          OPTIONAL ↗Produce with keys (key:value)
          --property parse.key=true
          --property key.separator=:

kafka-console-consumer.sh ↗Consume messages
      --bootstrap-server localhost:9092
      --topic mytopic ↗From which topic?
          OPTIONAL
          --from-beginning ↗ Read from the beginning
          --property print.key=true ↗ Display keys
          --group my_group ↗ Assign to consumer group
```

TOPICS AND CONSUMER GROUPS

```
kafka-topics.sh ↗Topic operations
      --bootstrap-server localhost:9092
      --list ↗ List all topics
      --topic mytopic
      --create ↗ Create topic
        --replication-factor 3
        ↗How many replicas?
        --partitions 3
        ↗How many partitions
          OPTIONAL
          --config min.insync.replicas=2
          ↗Other settings
      --describe ↗ Describe topic
      --delete ↗Delete it

kafka-consumer-groups.sh
↗Manage consumer groups
      --bootstrap-server localhost:9092
      --list ↗ Show all consumer groups
      --describe ↗ More info about the groups
      --reset-offsets ↗ Reset offsets
        --execute ↗ Otherwise: dry run
        --to-datetime ↗ Reset to a time
        --to-earliest ↗ To the oldest message
        --to-latest ↗ To the latest message
        ↗ Read more: see help page

      --delete-offsets ↗ Delete only offsets?
      --delete ↗ Or the whole group?
      ↗ What does the command refer to? (combinable)
        --group mygroup ↗ To a group?
        --all-groups ↗ To all groups?
        --topic mytopic ↗ To a topic?
        --all-topics ↗ To all topics?
```

PERFORMANCE TESTS

```
kafka-producer-perf-test.sh
↗ Producer Performance Tests
      --topic mytopic
      --num-records 1000 ↗ Number of messages to
      produce
      --throughput -1 ↗ No throttling or MSG/s
      --record-size 1000 ↗ Size in bytes
          ALTERNATIVE
          --payload-file myfile.json
          ↗ One message per line
      --producer-props bootstrap.servers=...
       ↗ Settings
          OPTIONAL
          --producer.config producer.properties
          ↗ Read from file

kafka-consumer-perf-test.sh
↗ Consumer performance tests
      --topic mytopic
      --consumer.config consumer.properties
      ↗ Settings
      --messages 1000000 ↗ How many messages to read?
```

USEFUL TOOLS

```
kafka-streams-application-reset.sh
↗ Reset Kafka Streams applications
kafka-leader-election.sh ↗ Select new partition leader
kafka-reassign-partitions.sh ↗ Move partitions
between brokers
kafka-configs.sh ↗ Manage settings
kafka-dump-log.sh ↗ View segments
kafka-acls.sh ↗ Manage permissions

kcat ↗ Alternative to kafka-console-producer and consumer
      -b localhost:9092
      -t mytopic
      -C/-P ↗ Consumer or producer

kcctl ↗ Manage Kafka Connect
```

SOFTWARE DEVELOPMENT

Apache Kafka IN ACTION

Zelenin • Kropp • Foreword by Adam Bellemare

Apache Kafka is the gold standard streaming data platform for real-time analytics, event sourcing, and stream processing. Acting as a central hub for distributed data, it enables seamless flow between producers and consumers via a publish-subscribe model. Kafka easily handles millions of events per second, and its rock-solid design ensures high fault tolerance and smooth scalability.

Apache Kafka in Action is a practical guide for IT professionals who are integrating Kafka into data-intensive applications and infrastructures. The book covers everything from Kafka fundamentals to advanced operations, with interesting visuals and real-world examples. Readers will learn to set up Kafka clusters, produce and consume messages, handle real-time streaming, and integrate Kafka into enterprise systems. This easy-to-follow book emphasizes building reliable Kafka applications and taking advantage of its distributed architecture for scalability and resilience.

What's Inside

- Master Kafka's distributed streaming capabilities
- Implement real-time data solutions
- Integrate Kafka into enterprise environments
- Build and manage Kafka applications
- Achieve fault tolerance and scalability

For IT operators, software architects and developers. No experience with Kafka required.

Anatoly Zelenin is a Kafka expert known for workshops across Europe, especially in banking and manufacturing. **Alexander Kropp** specializes in Kafka and Kubernetes, contributing to cloud platform design and monitoring.

For print book owners, all digital formats are free:
https://www.manning.com/freebook

MANNING

"A great introduction. Even experienced users will go back to it again and again."
—Jakub Scholz, Red Hat

"Approachable, practical, well-illustrated, and easy to follow. A must-read."
—Olena Kutsenko, Confluent

"A zero to hero journey to understanding and using Kafka!"
—Anthony Nandaa, Microsoft

"Thoughtfully explores a wide range of topics. A wealth of valuable information seamlessly presented and easily accessible."
—Olena Babenko, Aiven Oy

ISBN-13: 978-1-63343-759-3