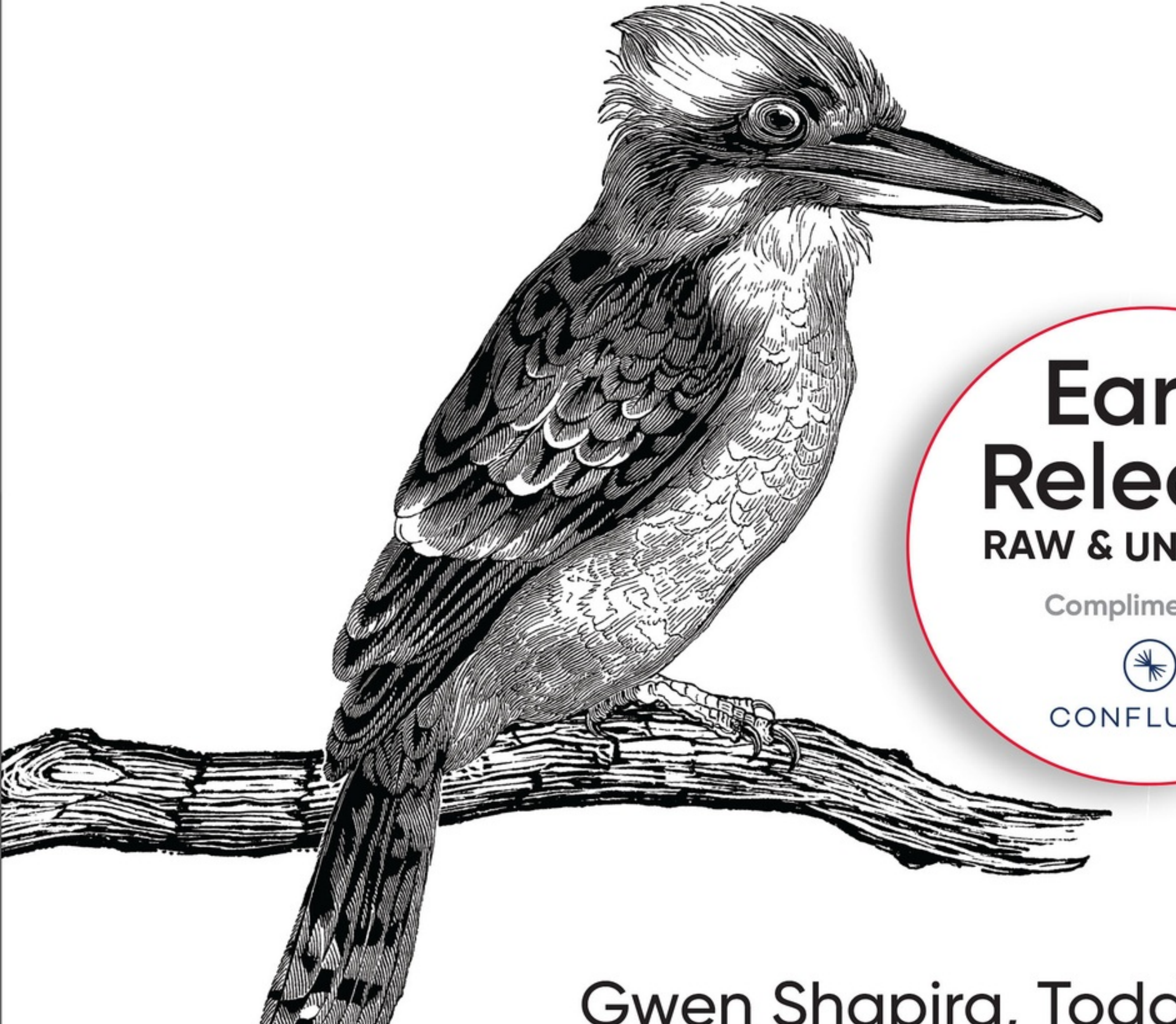
O'REILLY®

Second Edition

Kafka
The Definitive Guide

Real-Time Data and Stream Processing at Scale



Early Release
RAW & UNEDITED
Compliments of
CONFLUENT

Gwen Shapira, Todd Palino,
Rajini Sivaram & Neha Narkhede

Kafka: The Definitive Guide

SECOND EDITION

Real-Time Data and Stream Processing at Scale

Gwen Shapira, Todd Palino, Rajini Sivaram, and Neha Narkhede

Kafka: The Definitive Guide

by Gwen Shapira, Todd Palino, Rajini Sivaram, and Neha Narkhede

Printed in the United States of America.

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

Acquisitions Editor: Jonathan Hassell

Development Editor: Gary O'Brien

Production Editor: Kate Galloway

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

October 2021: Second Edition

Revision History for the Early Release

- 2020-05-22: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492043089 for release details.

978-1-492-04301-0

Chapter 1. Meet Kafka

Every enterprise is powered by data. We take information in, analyze it, manipulate it, and create more as output. Every application creates data, whether it is log messages, metrics, user activity, outgoing messages, or something else. Every byte of data has a story to tell, something of importance that will inform the next thing to be done. In order to know what that is, we need to get the data from where it is created to where it can be analyzed. We see this every day on websites like Amazon, where our clicks on items of interest to us are turned into recommendations that are shown to us a little later.

The faster we can do this, the more agile and responsive our organizations can be. The less effort we spend on moving data around, the more we can focus on the core business at hand. This is why the pipeline is a critical component in the data-driven enterprise. How we move the data becomes nearly as important as the data itself.

> *Any time scientists disagree, it's because we have insufficient data. Then we can agree on what kind of data to get; we get the data; and the data solves the problem. Either I'm right, or you're right, or we're both wrong. And we move on.*
>
> —Neil deGrasse Tyson

Publish/Subscribe Messaging

Before discussing the specifics of Apache Kafka, it is important for us to understand the concept of publish/subscribe messaging and why it is important. *Publish/subscribe messaging* is a pattern that is characterized by the sender (publisher) of a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of messages. Pub/sub systems often have a broker, a central point where messages are published, to facilitate this.

How It Starts

Many use cases for publish/subscribe start out the same way: with a simple message queue or interprocess communication channel. For example, you create an application that needs to send monitoring information somewhere, so you write in a direct connection from your application to an app that displays your metrics on a dashboard, and push metrics over that connection, as seen in Figure 1-1.

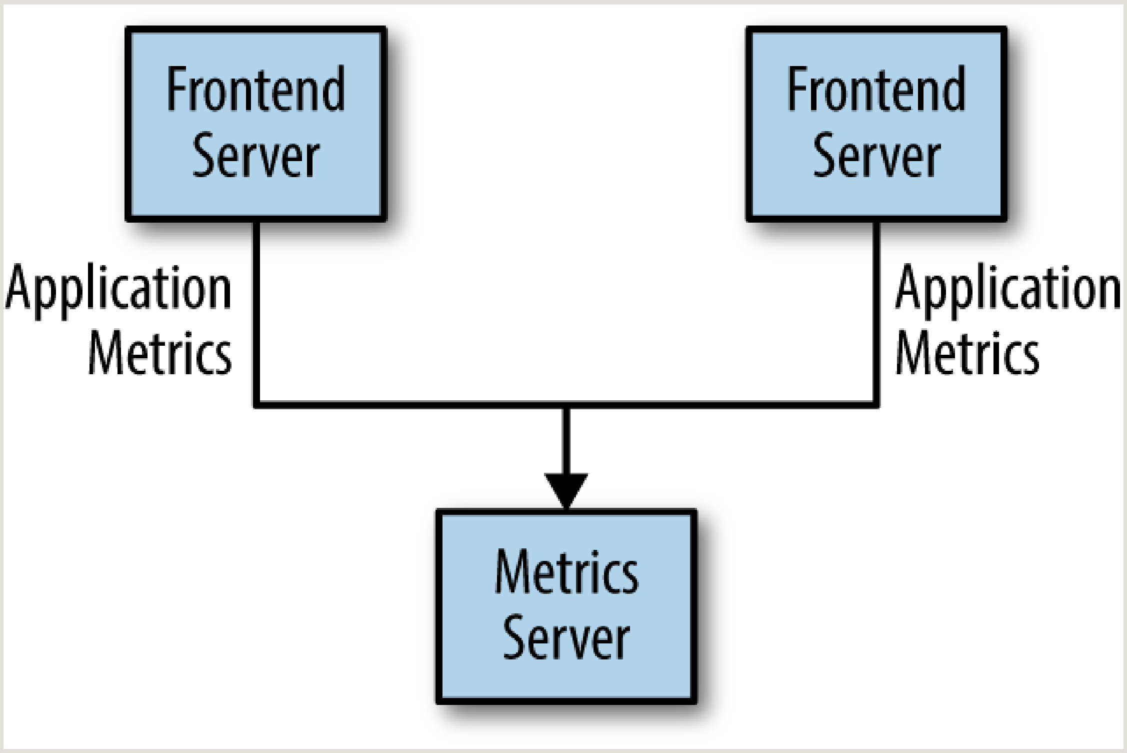


Figure 1-1. A single, direct metrics publisher

This is a simple solution to a simple problem that works when you are getting started with monitoring. Before long, you decide you would like to analyze your metrics over a longer term, and that doesn't work well in the dashboard. You start a new service that can receive metrics, store them, and analyze them. In order to support this, you modify your application to write metrics to both systems. By now you have three more applications that are generating metrics, and they all make the same connections to these two services. Your coworker thinks it would be a good idea to do active polling of the services for alerting as well, so you add a server on each of the applications to provide metrics on request. After a while, you have more applications that are using those servers to get individual metrics and use them for various purposes. This architecture can look much like Figure 1-2, with connections that are even harder to trace.

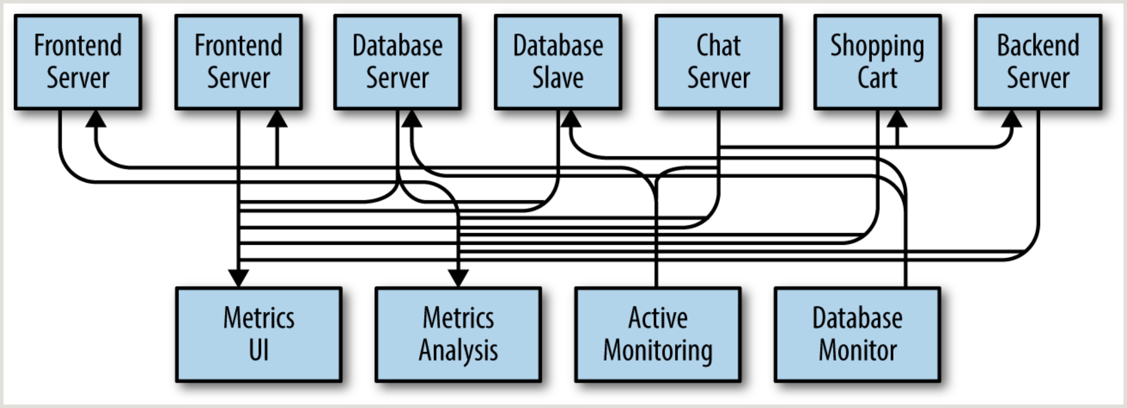


Figure 1-2. Many metrics publishers, using direct connections

The technical debt built up here is obvious, so you decide to pay some of it back. You set up a single application that receives metrics from all the applications out there, and provide a server to query those metrics for any system that needs them. This reduces the complexity of the architecture to something similar to Figure 1-3. Congratulations, you have built a publish-subscribe messaging system!

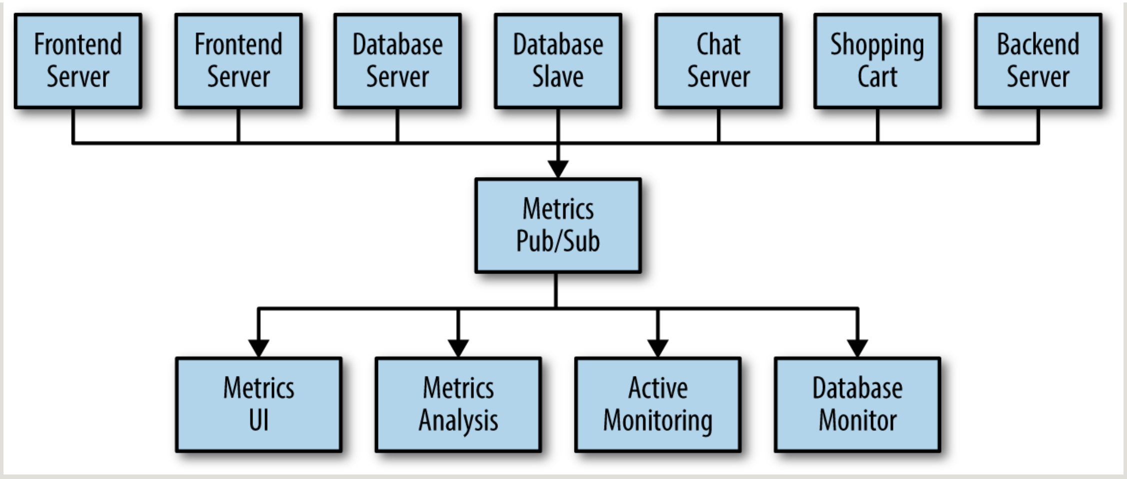


Figure 1-3. A metrics publish/subscribe system

Individual Queue Systems

At the same time that you have been waging this war with metrics, one of your coworkers has been doing similar work with log messages. Another has been working on tracking user behavior on the frontend website and providing that information to developers who are working on machine learning, as well as creating some reports for management. You have all followed a similar path of building out systems that decouple the publishers of the information from the subscribers to that information. Figure 1-4 shows such an infrastructure, with three separate pub/sub systems.

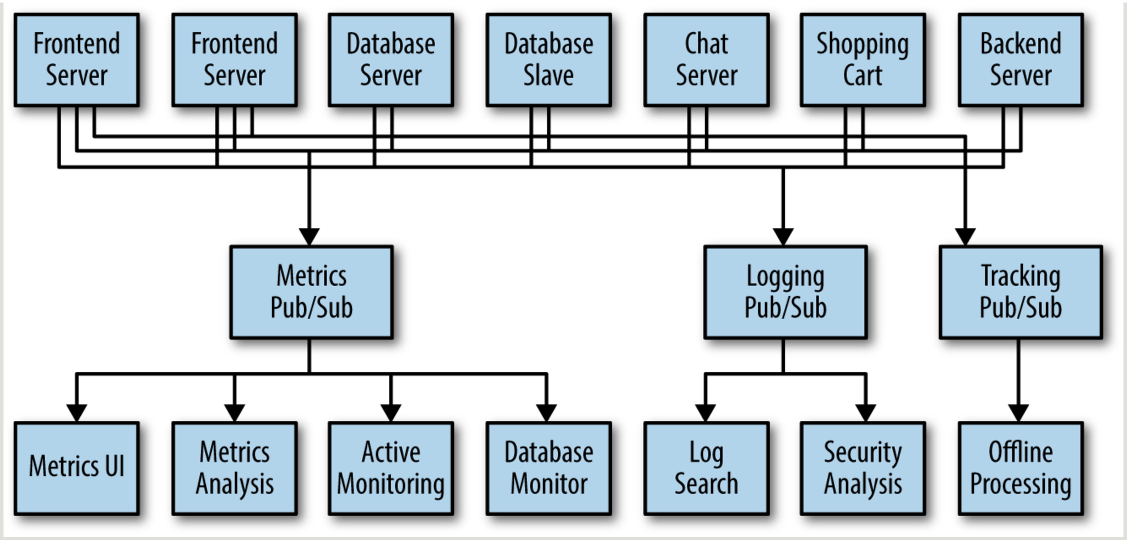


Figure 1-4. Multiple publish/subscribe systems

This is certainly a lot better than utilizing point-to-point connections (as in Figure 1-2), but there is a lot of duplication. Your company is maintaining multiple systems for queuing data, all of which have their own individual bugs and limitations. You also know that there will be more use cases for messaging coming soon. What you would like to have is a single centralized system that allows for publishing generic types of data, which will grow as your business grows.

Enter Kafka

Apache Kafka is a publish/subscribe messaging system designed to solve this problem. It is often described as a "distributed commit log" or more recently as a "distributing streaming platform." A filesystem or database commit log is designed to provide a durable record of all transactions so that they can be replayed to consistently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

Messages and Batches

The unit of data within Kafka is called a *message*. If you are approaching Kafka from a database background, you can think of this as similar to a *row* or a *record*. A message is simply an array of bytes as far as Kafka is concerned, so the data contained within it does not have a specific format or meaning to Kafka. A message can have an optional bit of metadata, which is referred to as a *key*. The key is also a byte array and, as with the message, has no specific meaning to Kafka. Keys are used when messages are to be written to partitions in a more controlled manner. The simplest such scheme is to generate a consistent hash of the key, and then select the partition number for that message by taking the result of the hash modulo, the total number of partitions in the topic. This assures that messages with the same key are always written to the same partition.

For efficiency, messages are written into Kafka in batches. A *batch* is just a collection of messages, all of which are being produced to the same topic and partition. An individual roundtrip across the network for each message would result in excessive overhead, and collecting messages together into a batch reduces this. Of course, this is a tradeoff between latency and throughput: the larger the batches, the more messages that can be handled per unit of time, but the longer it takes an individual message to propagate. Batches are also typically compressed, providing more efficient data transfer and storage at the cost of some processing power. Both keys and batches are discussed in more detail in Chapter 4.

Schemas

While messages are opaque byte arrays to Kafka itself, it is recommended that additional structure, or schema, be imposed on the message content so that it can be easily understood. There are many options available for message *schema*, depending on your application's individual needs. Simplistic systems, such as Javascript Object Notation (JSON) and Extensible Markup Language (XML), are easy to use and human-readable. However, they lack features such as robust type handling and compatibility between schema versions. Many Kafka developers favor the use of Apache Avro, which is a serialization framework originally developed for Hadoop. Avro provides a compact serialization format; schemas that are separate from the message payloads and that do not require code to be generated when they change; and strong data typing and schema evolution, with both backward and forward compatibility.

A consistent data format is important in Kafka, as it allows writing and reading messages to be decoupled. When these tasks are tightly coupled, applications that subscribe to messages must be updated to handle the new data format, in parallel with the old format. Only then can the applications that publish the messages be updated to utilize the new format. By using well-defined schemas and storing them in a common repository, the messages in Kafka can be understood without coordination. Schemas and serialization are covered in more detail in Chapter 4.

Topics and Partitions

Messages in Kafka are categorized into *topics*. The closest analogies for a topic are a database table or a folder in a filesystem. Topics are additionally broken down into a number of *partitions*. Going back to the “commit log” description, a partition is a single log. Messages are written to it in an append-only fashion, and are read in order from beginning to end. Note that as a topic typically has multiple partitions, there is no guarantee of message time-ordering across the entire topic, just within a single partition. Figure 1-5 shows a topic with four partitions, with writes being appended to the end of each one. Partitions are also the way that Kafka provides redundancy and scalability. Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers to provide performance far beyond the ability of a single server.

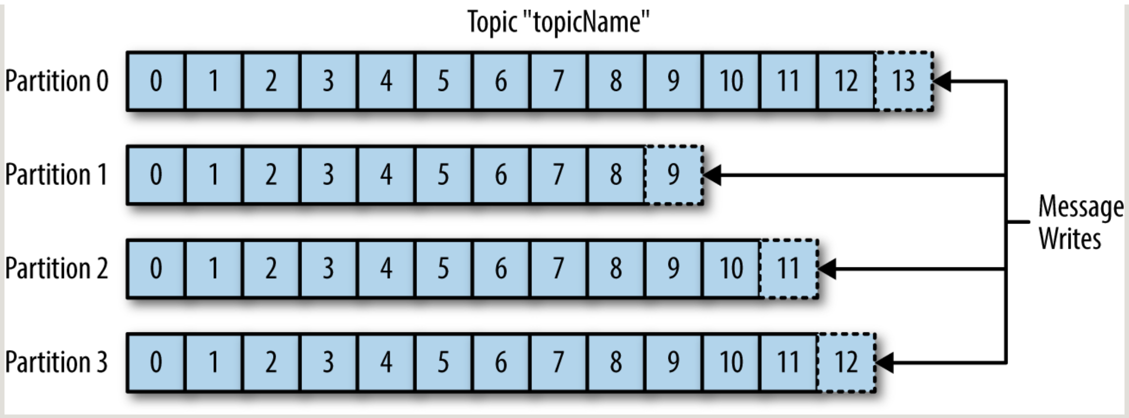


Figure 1-5. Representation of a topic with multiple partitions

The term *stream* is often used when discussing data within systems like Kafka. Most often, a stream is considered to be a single topic of data, regardless of the number of partitions. This represents a single stream of data moving from the producers to the consumers. This way of referring to messages is most common when discussing stream processing, which is when frameworks—some of which are Kafka Streams, Apache Samza, and Storm—operate on the messages in real time. This method of operation can be compared to the way offline frameworks, namely Hadoop, are designed to work on bulk data at a later time. An overview of stream processing is provided in Chapter 14.

Producers and Consumers

Kafka clients are users of the system, and there are two basic types: producers and consumers. There are also advanced client APIs—Kafka Connect API for data integration and Kafka Streams for stream processing. The advanced clients use producers and consumers as building blocks and provide higher-level functionality on top.

Producers create new messages. In other publish/subscribe systems, these may be called *publishers* or *writers*. In general, a message will be produced to a specific topic. By default, the producer does not care what partition a specific message is written to and will balance messages over all partitions of a topic evenly. In some cases, the producer will direct messages to specific partitions. This is typically done using the message key and a partitioner that will generate a hash of the key and map it to a specific partition. This assures that all messages produced with a given key will get written to the same partition. The producer could also use a custom partitioner that follows other business rules for mapping messages to partitions. Producers are covered in more detail in Chapter 4.

Consumers read messages. In other publish/subscribe systems, these clients may be called *subscribers* or *readers*. The consumer subscribes to one or more topics and reads the messages in the order in which they were produced. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages. The *offset* is another bit of metadata—an integer value that continually increases—that Kafka adds to each message as it is produced. Each message in a given partition has a unique offset. By storing the offset of the last consumed message for each partition, either in Zookeeper or in Kafka itself, a consumer can stop and restart without losing its place.

Consumers work as part of a *consumer group*, which is one or more consumers that work together to consume a topic. The group assures that each partition is only consumed by one member. In Figure 1-6, there are three consumers in a single group consuming a topic. Two of the consumers are working from one partition each, while the third consumer is working from two partitions. The mapping of a consumer to a partition is often called *ownership* of the partition by the consumer.

In this way, consumers can horizontally scale to consume topics with a large number of messages. Additionally, if a single consumer fails, the remaining members of the group will rebalance the partitions being consumed to take over for the missing member. Consumers and consumer groups are discussed in more detail in Chapter 6.

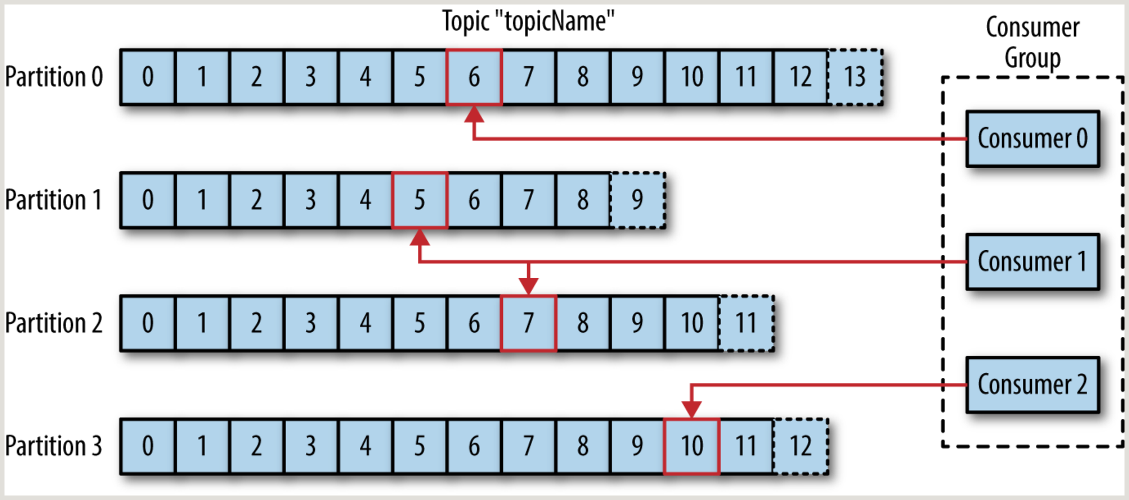


Figure 1-6. A consumer group reading from a topic

Brokers and Clusters

A single Kafka server is called a *broker*. The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been committed to disk. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second.

Kafka brokers are designed to operate as part of a *cluster*. Within a cluster of brokers, one broker will also function as the cluster *controller* (elected automatically from the live members of the cluster). The controller is responsible for administrative operations, including assigning partitions to brokers and monitoring for broker failures. A partition is owned by a single broker in the cluster, and that broker is called the *leader* of the partition. A partition may be assigned to multiple brokers, which will result in the partition being replicated (as seen in Figure 1-7). This provides redundancy of messages in the partition, such that another broker can take over leadership if there is a broker failure. However, all consumers and producers operating on that partition must connect to the leader. Cluster operations, including partition replication, are covered in detail in Chapter 8.

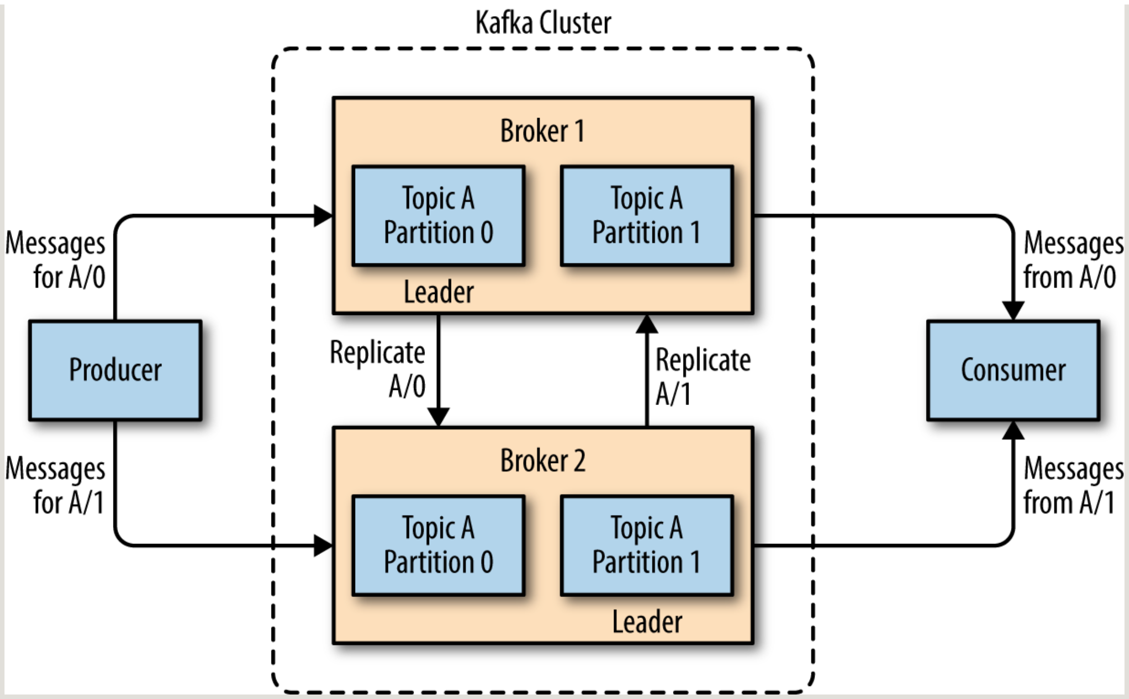


Figure 1-7. Replication of partitions in a cluster

A key feature of Apache Kafka is that of *retention*, which is the durable storage of messages for some period of time. Kafka brokers are configured with a default retention setting for topics, either retaining messages for some period of time (e.g., 7 days) or until the topic reaches a certain size in bytes (e.g., 1 GB). Once these limits are reached, messages are expired and deleted so that the retention configuration is a minimum amount of data available at any time. Individual topics can also be configured with their own retention settings so that messages are stored for only as long as they are useful. For example, a tracking topic might be retained for several days, whereas application metrics might be retained for only a few hours. Topics can also be configured as *log compacted*, which means that Kafka will retain only the last message produced with a specific key. This can be useful for changelog-type data, where only the last update is interesting.

Multiple Clusters

As Kafka deployments grow, it is often advantageous to have multiple clusters. There are several reasons why this can be useful:

- Segregation of types of data
- Isolation for security requirements
- Multiple datacenters (disaster recovery)

When working with multiple datacenters in particular, it is often required that messages be copied between them. In this way, online applications can have access to user activity at both sites. For example, if a user changes public information in their profile, that change will need to be visible regardless of the datacenter in which search results are displayed. Or, monitoring data can be collected from many sites into a single central location where the analysis and alerting systems are hosted. The replication mechanisms within the Kafka clusters are designed only to work within a single cluster, not between multiple clusters.

The Kafka project includes a tool called *MirrorMaker*, used for this purpose. At its core, MirrorMaker is simply a Kafka consumer and producer, linked together with a queue.

Messages are consumed from one Kafka cluster and produced for another. Figure 1-8 shows an example of an architecture that uses MirrorMaker, aggregating messages from two local clusters into an aggregate cluster, and then copying that cluster to other datacenters. The simple nature of the application belies its power in creating sophisticated data pipelines, which will be detailed further in Chapter 9.

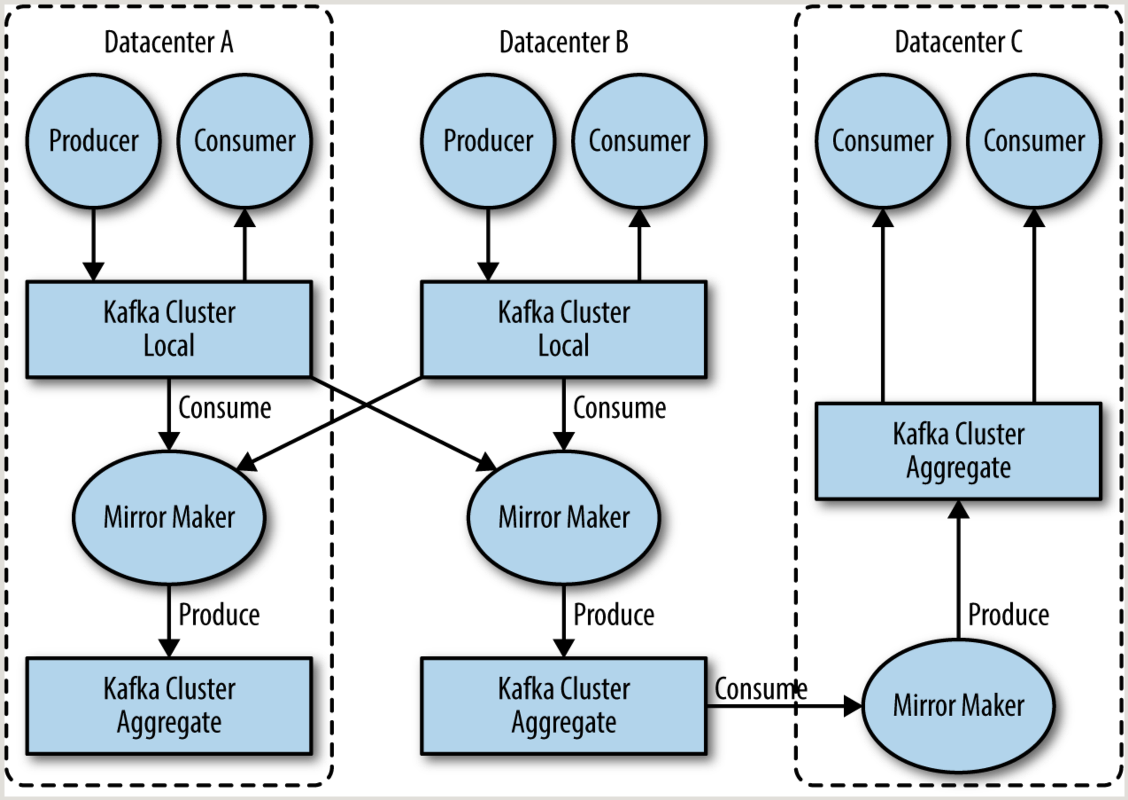


Figure 1-8. Multiple datacenter architecture

Why Kafka?

There are many choices for publish/subscribe messaging systems, so what makes Apache Kafka a good choice?

Multiple Producers

Kafka is able to seamlessly handle multiple producers, whether those clients are using many topics or the same topic. This makes the system ideal for aggregating data from many frontend systems and making it consistent. For example, a site that serves content to users via a number of microservices can have a single topic for page views that all services can write to using a common format. Consumer applications can then receive a single stream of page views for all applications on the site without having to coordinate consuming from multiple topics, one for each application.

Multiple Consumers

In addition to multiple producers, Kafka is designed for multiple consumers to read any single stream of messages without interfering with each other. This is in contrast to many queuing systems where once a message is consumed by one client, it is not available to any other. Multiple Kafka consumers can choose to operate as part of a group and share a stream, assuring that the entire group processes a given message only once.

Disk-Based Retention

Not only can Kafka handle multiple consumers, but durable message retention means that consumers do not always need to work in real time. Messages are committed to disk, and will be stored with configurable retention rules. These options can be selected on a per-topic basis, allowing for different streams of messages to have different amounts of retention depending on the consumer needs. Durable retention means that if a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data. It also means that maintenance can be performed on consumers, taking applications offline for a short period of time, with no concern about messages backing up on the producer or getting lost. Consumers can be stopped, and the messages will be retained in Kafka. This allows them to restart and pick up processing messages where they left off with no data loss.

Scalable

Kafka's flexible scalability makes it easy to handle any amount of data. Users can start with a single broker as a proof of concept, expand to a small development cluster of three brokers, and move into production with a larger cluster of tens or even hundreds of brokers that grows over time as the data scales up. Expansions can be performed while the cluster is online, with no impact on the availability of the system as a whole. This also means that a cluster of multiple brokers can handle the failure of an individual broker, and continue servicing clients. Clusters that need to tolerate more simultaneous failures can be configured with higher replication factors. Replication is discussed in more detail in Chapter 8.

High Performance

All of these features come together to make Apache Kafka a publish/subscribe messaging system with excellent performance under high load. Producers, consumers, and brokers can all be scaled out to handle very large message streams with ease. This can be done while still providing subsecond message latency from producing a message to availability to consumers.

The Data Ecosystem

Many applications participate in the environments we build for data processing. We have defined inputs in the form of applications that create data or otherwise introduce it to the system. We have defined outputs in the form of metrics, reports, and other data products. We create loops, with some components reading data from the system, transforming it using data from other sources, and then introducing it back into the data infrastructure to be used elsewhere. This is done for numerous types of data, with each having unique qualities of content, size, and usage.

Apache Kafka provides the circulatory system for the data ecosystem, as shown in Figure 1-9. It carries messages between the various members of the infrastructure, providing a consistent interface for all clients. When coupled with a system to provide message schemas, producers and consumers no longer require tight coupling or direct connections of any sort. Components can be added and removed as business cases are created and dissolved, and producers do not need to be concerned about who is using the data or the number of consuming applications.

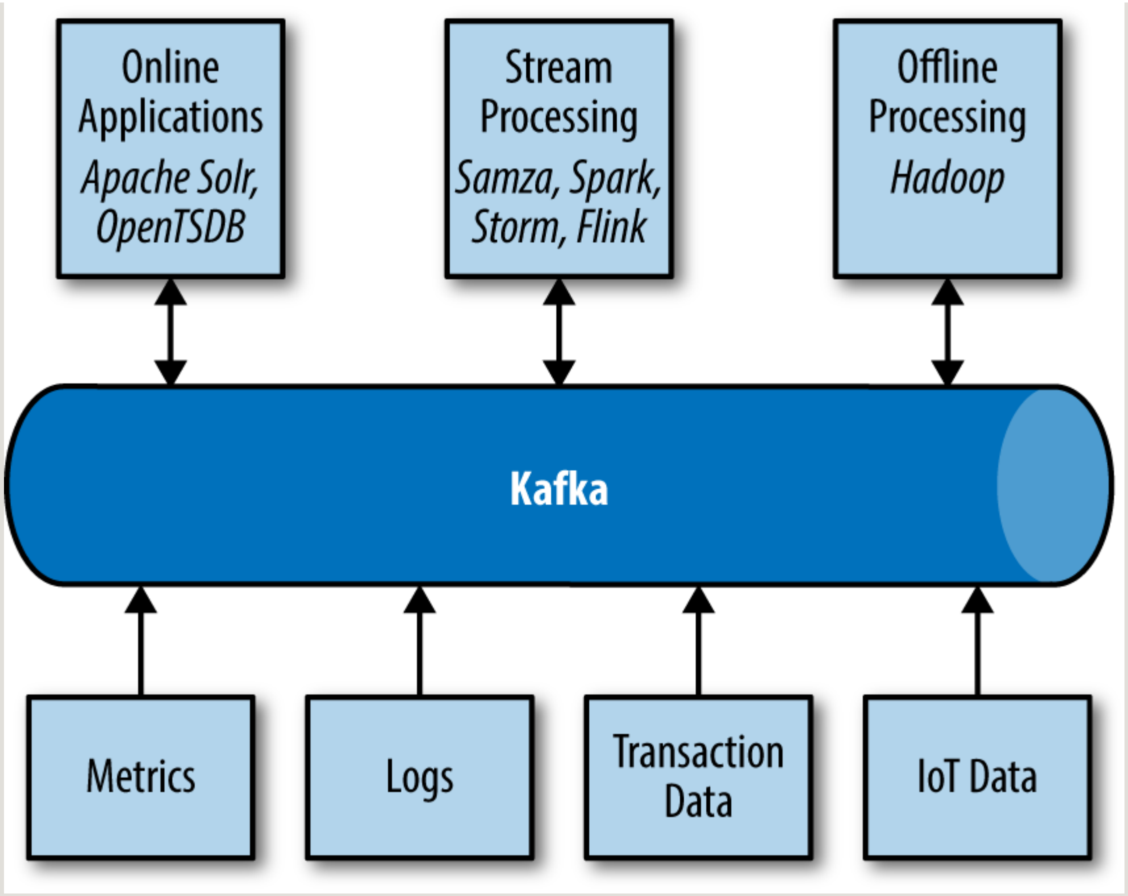


Figure 1-9. A big data ecosystem

Use Cases

ACTIVITY TRACKING

The original use case for Kafka, as it was designed at LinkedIn, is that of user activity tracking. A website's users interact with frontend applications, which generate messages regarding actions the user is taking. This can be passive information, such as page views and click tracking, or it can be more complex actions, such as information that a user adds to their profile. The messages are published to one or more topics, which are then consumed by applications on the backend. These applications may be generating reports, feeding machine learning systems, updating search results, or performing other operations that are necessary to provide a rich user experience.

MESSAGING

Kafka is also used for messaging, where applications need to send notifications (such as emails) to users. Those applications can produce messages without needing to be concerned about formatting or how the messages will actually be sent. A single application can then read all the messages to be sent and handle them consistently, including:

- Formatting the messages (also known as decorating) using a common look and feel
- Collecting multiple messages into a single notification to be sent
- Applying a user's preferences for how they want to receive messages

Using a single application for this avoids the need to duplicate functionality in multiple applications, as well as allows operations like aggregation which would not otherwise be possible.

METRICS AND LOGGING

Kafka is also ideal for collecting application and system metrics and logs. This is a use case in which the ability to have multiple applications producing the same type of message shines.

Applications publish metrics on a regular basis to a Kafka topic, and those metrics can be consumed by systems for monitoring and alerting. They can also be used in an offline system like Hadoop to perform longer-term analysis, such as growth projections. Log messages can be published in the same way, and can be routed to dedicated log search systems like Elasticsearch or security analysis applications. Another added benefit of Kafka is that when the destination system needs to change (e.g., it's time to update the log storage system), there is no need to alter the frontend applications or the means of aggregation.

COMMIT LOG

Since Kafka is based on the concept of a commit log, database changes can be published to Kafka and applications can easily monitor this stream to receive live updates as they happen. This changelog stream can also be used for replicating database updates to a remote system, or for consolidating changes from multiple applications into a single database view. Durable retention is useful here for providing a buffer for the changelog, meaning it can be replayed in the event of a failure of the consuming applications. Alternately, log-compacted topics can be used to provide longer retention by only retaining a single change per key.

STREAM PROCESSING

Another area that provides numerous types of applications is stream processing. While almost all usage of Kafka can be thought of as stream processing, the term is typically used to refer to applications that provide similar functionality to map/reduce processing in Hadoop. Hadoop usually relies on aggregation of data over a long time frame, either hours or days. Stream processing operates on data in real time, as quickly as messages are produced. Stream frameworks allow users to write small applications to operate on Kafka messages, performing tasks such as counting metrics, partitioning messages for efficient processing by other applications, or transforming messages using data from multiple sources. Stream processing is covered in Chapter 14.

Kafka's Origin

Kafka was created to address the data pipeline problem at LinkedIn. It was designed to provide a high-performance messaging system that can handle many types of data and provide clean, structured data about user activity and system metrics in real time.

Data really powers everything that we do.

—Jeff Weiner, CEO of LinkedIn

LinkedIn's Problem

Similar to the example described at the beginning of this chapter, LinkedIn had a system for collecting system and application metrics that used custom collectors and open source tools for storing and presenting data internally. In addition to traditional metrics, such as CPU usage and application performance, there was a sophisticated request-tracing feature that used the monitoring system and could provide introspection into how a single user request propagated through internal applications. The monitoring system had many faults, however. This included metrics collection based on polling, large intervals between metrics, and no ability for application owners to manage their own metrics. The system was high-touch, requiring human intervention for most simple tasks, and inconsistent, with differing metric names for the same measurement across different systems.

At the same time, there was a system created for tracking user activity information. This was an HTTP service that frontend servers would connect to periodically and publish a batch of messages (in XML format) to the HTTP service. These batches were then moved to offline processing, which is where the files were parsed and collated. This system had many faults. The XML formatting was inconsistent, and parsing it was computationally expensive. Changing the type of user activity that was tracked required a significant amount of coordinated work between frontends and offline processing. Even then, the system would break constantly due to changing schemas. Tracking was built on hourly batching, so it could not be used in real-time.

Monitoring and user-activity tracking could not use the same backend service. The monitoring service was too clunky, the data format was not oriented for activity tracking, and the polling model for monitoring was not compatible with the push model for tracking. At the same time, the tracking service was too fragile to use for metrics, and the batch-oriented processing was not the right model for real-time monitoring and alerting. However, the monitoring and tracking data shared many traits, and correlation of the information (such as how specific types of user activity affected application performance) was highly desirable. A drop in specific types of user activity could indicate problems with the application that serviced it, but hours of delay in processing activity batches meant a slow response to these types of issues.

At first, existing off-the-shelf open source solutions were thoroughly investigated to find a new system that would provide real-time access to the data and scale out to handle the amount of message traffic needed. Prototype systems were set up using ActiveMQ, but at the time it could not handle the scale. It was also a fragile solution for the way LinkedIn needed to use it, discovering many flaws in ActiveMQ that would cause the brokers to pause. This would back up connections to clients and interfere with the ability of the applications to serve requests to users. The decision was made to move forward with a custom infrastructure for the data pipeline.

The Birth of Kafka

The development team at LinkedIn was led by Jay Kreps, a principal software engineer who was previously responsible for the development and open source release of Voldemort, a distributed key-value storage system. The initial team also included Neha Narkhede and, later, Jun Rao. Together, they set out to create a messaging system that could meet the needs of both the monitoring and tracking systems, and scale for the future. The primary goals were to:

- Decouple producers and consumers by using a push-pull model
- Provide persistence for message data within the messaging system to allow multiple consumers
- Optimize for high throughput of messages
- Allow for horizontal scaling of the system to grow as the data streams grew

The result was a publish/subscribe messaging system that had an interface typical of messaging systems but a storage layer more like a log-aggregation system. Combined with the adoption of Apache Avro for message serialization, Kafka was effective for handling both metrics and user-activity tracking at a scale of billions of messages per day. The scalability of Kafka has helped LinkedIn's usage grow in excess of seven trillion messages produced (as of February 2020) and over five petabytes of data consumed daily.

Open Source

Kafka was released as an open source project on GitHub in late 2010. As it started to gain attention in the open source community, it was proposed and accepted as an Apache Software Foundation incubator project in July of 2011. Apache Kafka graduated from the incubator in October of 2012. Since then, it has continuously been worked on and has found a robust community of contributors and committers outside of LinkedIn. Kafka is now used in some of the largest data pipelines in the world, including those at Netflix, Uber, and many other companies.

Widespread adoption of Kafka has created a healthy ecosystem around the core project as well. There are active meetup groups in dozens of countries around the world, providing local discussion and support of stream processing. There are also numerous open source projects related to Apache Kafka. The largest concentrations of these are from Confluent (including KSQL, as well as their own schema registy and REST projects), and LinkedIn (including Cruise Control, Kafka Monitor, and Burrow).

Commercial Engagement

In the fall of 2014, Jay Kreps, Neha Narkhede, and Jun Rao left LinkedIn to found Confluent, a company centered around providing development, enterprise support, and training for Apache Kafka. They also joined other companies (such as Heroku) in providing cloud services for Kafka. Confluent, through a partnership with Google, provides managed Kafka clusters on Google Cloud Platform, as well as providing similar services on Amazon Web Services and Azure. One of the other major initiatives of Confluent is to organize the Kafka Summit conference series. Started in 2016, with conferences held annually in the United States and in London, Kafka Summit provides a place for the community to come together on a global scale and share knowlege about Apache Kafka and related projects.

The Name

People often ask how Kafka got its name and if it signifies anything specific about the application itself. Jay Kreps offered the following insight:

> *I thought that since Kafka was a system optimized for writing, using a writer's name would make sense. I had taken a lot of lit classes in college and liked Franz Kafka. Plus the name sounded cool for an open source project.*
>
> *So basically there is not much of a relationship.*

Getting Started with Kafka

Now that we know all about Kafka and its history, we can set it up and build our own data pipeline. In the next chapter, we will explore installing and configuring Kafka. We will also cover selecting the right hardware to run Kafka on, and some things to keep in mind when moving to production operations.

Chapter 2. Managing Apache Kafka Programmatically

There are many CLI and GUI tools for managing Kafka (we'll discuss them in chapter 9), but there are also times when you want to execute some administrative commands from within your client application. Creating new topics on demand based on user input or data is an especially common use-case: IOT apps often receive events from user devices, and write events to topics based on the device type. If the manufacturer produces a new type of device, you either have to remember, via some process, to also create a topic. Or alternatively, the application can dynamically create a new topic if it receives events with unrecognized device type. The second alternative has downsides but avoiding the dependency on additional process to generate topics is an attractive feature in the right scenarios.

Apache Kafka added the AdminClient in version 0.11 to provide a programmatic API for administrative functionality that was previously done in the command line: Listing, creating and deleting topics, describing the cluster, managing ACLs and modifying configuration.

Here's one example: Your application is going to produce events to a specific topic. This means that before producing the first event, the topic has to exist. Before Apache Kafka added the admin client, there were few options, and none of them particularly user-friendly: You could capture `UNKNOWN_TOPIC_OR_PARTITION` exception from the `producer.send()` method and let your user know that they need to create the topic, or you could hope that the Kafka cluster you are writing to enabled automatic topic creation, or you can try to rely on internal APIs and deal with the consequences of no compatibility guarantees. Now that Apache Kafka provides AdminClient, there is a much better solution: Use AdminClient to check whether the topic exists, and if it does not, create it on the spot.

In this chapter we'll give an overview of the AdminClient before we drill down into the details of how to use it in your applications. We'll focus on the most commonly used functionality - management of topics, consumer groups and entity configuration.

AdminClient Overview

As you start using Kafka AdminClient, it helps to be aware of its core design principles. When you understand how the AdminClient was designed and how it should be used, the specifics of each method will be much more intuitive.

Asynchronous and Eventually Consistent API

Perhaps the most important thing to understand about Kafka's AdminClient is that it is asynchronous. Each method returns immediately after delivering a request to the cluster Controller, and each method returns one or more `Future` objects. Future objects are the results of asynchronous operations and they have methods for checking the status of the asynchronous operation, cancelling it, waiting for it to complete and executing functions after its completion. Kafka's AdminClient wraps the Future objects into Result objects, which provide methods to wait for the operation to complete and helper methods for common follow-up operations. For example, `KafkaAdminClient.createTopics` returns `CreateTopicsResult` object which lets you wait until all topics are created, lets you check each topic status individually and also lets you retrieve the configuration of a specific topic after it was created.

Because Kafka's propagation of metadata from the Controller to the brokers is asynchronous, the Futures that AdminClient APIs return are considered complete when the Controller state has been fully updated. It is possible that at that point not every broker is aware of the new state, so a `listTopics` request may end up handled by a broker that is not up to date and will not contain a topic that was very recently created. This property is also called **eventual consistency** - eventually every broker will know about every topic, but we can’t guarantee exactly when this will happen.

Options

Every method in AdminClient takes as an argument an Options object that is specific to that method. For example, `listTopics` method takes `ListTopicsOptions` object as an argument and `describeCluster` takes `DescribeClusterOptions` as an argument. Those objects contain different settings for how the request will be handled by the broker. The one setting that all AdminClient methods have is `timeoutMs` - this controls how long the client will wait for a response from the cluster before throwing a `TimeoutException`. This limits the time in which your application may be blocked by AdminClient operation. Other options can be things like whether `listTopics` should also return internal topics and whether `describeCluster` should also return which operations the client is authorized to perform on the cluster.

Flat Hierarchy

All the admin operations that are supported by the Apache Kafka protocol are implemented in `KafkaAdminClient` directly. There is no object hierarchy or namespaces. This is a bit controversial as the interface can be quite large and perhaps a bit overwhelming, but the main benefit is that if you want to know how to programmatically perform any admin operation on Kafka, you have exactly one JavaDoc to search and your IDE’s autocomplete will be quite handy. You don’t have to wonder whether you are just missing the right place to look. If it isn’t in AdminClient, it was not implemented yet (but contributions are welcome!).

TIP

If you are interested in contributing to Apache Kafka, take a look at our How To Contribute guide. Start with smaller, non-controversial bug fixes and improvements, before tackling a more significant change to the architecture or the protocol. Non-code contributions such as bug reports, documentation improvements, responses to questions and blog posts are also encouraged.

Additional Notes

- All the operations that modify the cluster state - create, delete and alter, are handled by the Controller. Operations that read the cluster state - list and describe, can be handled by any broker and are directed to the least loaded broker (based on what the client knows). This shouldn’t impact you as a user of the API, but it can be good to know - in case you are seeing unexpected behavior, you notice that some operations succeed while others fail, or if you are trying to figure out why an operation is taking too long.

- At the time we are writing this chapter (Apache Kafka 2.5 is about to be released), most admin operations can be performed either through AdminClient or directly by modifying the cluster metadata in Zookeeper. We highly encourage you to never use Zookeeper directly, and if you absolutely have to, report this as a bug to Apache Kafka. The reason is that in the near future, the Apache Kafka community will remove the Zookeeper dependency, and every application that uses Zookeeper directly for admin operations will have to be modified. The AdminClient API on the other hand, will remain exactly the same, just with a different implementation inside the Kafka cluster.

AdminClient Lifecycle: Creating, Configuring and Closing

In order to use Kafka's AdminClient, the first thing you have to do is construct an instance of the AdminClient class. This is quite straight forward:

```
Properties props = new Properties();
props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "loca
AdminClient admin = AdminClient.create(props);
// TODO: Do something useful with AdminClient
admin.close(Duration.ofSeconds(30));
```

The static `create` method takes as an argument a `Properties` object with configuration. The only mandatory configuration is the URI for your cluster - a comma separated list of brokers to connect to. As usual, in production environments, you want to specify at least 3 brokers, just in case one is currently unavailable. We'll discuss how to configure a secure and authenticated connection separately in the Kafka Security chapter.

If you start an AdminClient, eventually you want to close it. It is important to remember that when you call `close`, there could still be some AdminClient operations in progress. Therefore `close` method accepts a timeout parameter. Once you call `close`, you can't call any other methods and send any more requests, but the client will wait for responses until the timeout expires. After the timeout expires, the client will abort all on-going operations with timeout exception and release all resources. Calling `close` without a timeout implies that you'll wait as long as it takes for all on-going operations to complete.

You probably recall from chapters 3 and 4 that the KafkaProducer and KafkaConsumer have quite a few important configuration parameters. The good news is that AdminClient is much simpler and there is not much to configure. You can read about all the configuration parameters in Configurations Kafka documentation. In our opinion, the important configuration parameters are:

client.dns.lookup

This configuration was introduced in Apache Kafka 2.1.0 release.

By default, Kafka validates, resolves and creates connections based on the hostname provided in bootstrap server configuration (and later in the names returned by the brokers as specified in `advertised.listeners` configuration). This simple model works most of the time, but fails to cover two important use-cases - use of DNS aliases, especially in bootstrap configuration, and use of a single DNS that maps to multiple IP addresses. These sound similar, but are slightly different. Lets look at each of these mutually-exclusive scenarios in a bit more detail.

USE OF DNS ALIAS

Suppose you have multiple brokers, with the following naming convention: `broker1.hostname.com`, `broker2.hostname.com`, etc. Rather than specifying all of them in bootstrap servers configuration, which can easily become challenging to maintain, you may want to create a single DNS alias that will map to all of them. You'll use `all-brokers.hostname.com` for bootstrapping, since you don't actually care which broker gets the initial connection from clients. This is all very convenient, except if you use SASL to authenticate. If you use SASL, the client will try to authenticate `all-brokers.hostname.com`, but the server principal will be `broker2.hostname.com`, if the names don't match, SASL will refuse to authenticate (the broker certificate could be a man-in-the-middle attack), and the connection will fail.

In this scenario, you’ll want to use `client.dns.lookup=resolve_canonical_bootstrap_servers_only`. With this configuration, the client will “expend” the DNS alias, and the result will be the same as if you included all the broker names the DNS alias connects to as brokers in the original bootstrap list.

DNS NAME WITH MULTIPLE IP ADDRESSES

With modern network architectures, it is common to put all the brokers behind a proxy or a load balancer. This is especially common if you use Kubernetes, where load-balancers are necessary to allow connections from outside the Kubernetes cluster. In these cases, you don’t want the load balancers to become a single point of failure. It is therefore very common to make `broker1.hostname.com` point at a list of IPs, all of which resolve to load balancers, and all of them route traffic to the same broker. These IPs are also likely to change over time. By default, KafkaClient will just try to connect to the first IP that the hostname resolves. This means that if that IP becomes unavailable, the client will fail to connect, even though the broker is fully available. It is therefore highly recommended to use `client.dns.lookup=use_all_dns_ips` to make sure the client doesn’t miss out on the benefits of a highly-available load balancing layer.

request.timeout.ms

This configuration limits the time that your application can spend waiting for AdminClient to respond. This includes the time spent on retrying if the client receives a retriable error.

The default value is 120 seconds, which is quite long - but some AdminClient operations, especially consumer group management commands, can take a while to respond. As we mentioned in the Overview section, each AdminClient method accepts an Options object, which can contain a timeout value that applies specifically to that call. If an AdminClient operation is on the critical path for your application, you may want to use a lower timeout value and handle lack of timely response from Kafka in a different way. A common example is that services try to validate existence of specific topics when they first start, but if Kafka takes longer than 30s to respond, you may want to continue starting the server and validate the existence of topics later (or skip this validation entirely).

Essential Topic Management

Now that we created and configured an AdminClient, it is time to see what we can do with it. The most common use case for Kafka's AdminClient is topic management. This includes listing topics, describing them, creating topics and deleting them.

Lets start by listing all topics in the cluster:

```
ListTopicsResult topics = admin.listTopics();
topics.names().get().forEach(System.out::println);
```

Note that `admin.listTopics()` returns `ListTopicsResult` object which is a thin wrapper over a collection of `Futures`. `topics.name()` returns a future set of name. When we call `get()` on this future, the executing thread will wait until the server responds with a set of topic names, or we get a timeout exception. Once we get the list, we iterate over it to print all the topic names.

Now lets try something a bit more ambitious: Check if a topic exists, and create it if it doesn't. One way to check if a specific topic exists is to get a list of all topics and check if the topic you need is in the list. But on a large cluster, this can be inefficient. In addition, sometimes you want to check for more than just whether the topic exists - you want to make sure the topic has the right number of partitions and replicas. For example, Kafka Connect and Confluent Schema Registry use a Kafka topic to store configuration. When they start up, they check if the configuration topic exists, that it has only one partition to guarantee that configuration changes will arrive in strict order, that it has three replicas to guarantee availability and that the topic is compacted so old configuration will be retained indefinitely.

```
DescribeTopicsResult demoTopic = admin.describeTopics(TOPIC
try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).g
    System.out.println("Description of demo topic:" + topic

    if (topicDescription.partitions().size() != NUM_PARTITI
      System.out.println("Topic has wrong number of partiti
      System.exit(-1);
    }
} catch (ExecutionException e) { ❹
    // exit early for almost all exceptions
    if (! (e.getCause() instanceof UnknownTopicOrPartitionE
        e.printStackTrace();
        throw e;
    }

    // if we are here, topic doesn't exist
    System.out.println("Topic " + TOPIC_NAME +
        " does not exist. Going to create it now");
    // Note that number of partitions and replicas are opti
    // not specified, the defaults configured on the Kafka
    CreateTopicsResult newTopic = admin.createTopics(Collec
            new NewTopic(TOPIC_NAME, NUM_PARTITIONS, REP_FA

    // Check that the topic was created correctly:
    if (newTopic.numPartitions(TOPIC_NAME).get() != NUM_PAR
        System.out.println("Topic has wrong number of parti
        System.exit(-1);
    }
}
```

❶ To check that the topic exists with the correct configuration, we call `describeTopics()` with a list of topic names that we want to validate. This returns `DescribeTopicResult` object, which wraps a map of topic names to future descriptions.

❷ We’ve already seen that if we wait for the future to complete, using `get()`, we can get the result we wanted, in this case a TopicDescription. But there is also a possibility that the server can’t complete the request correctly - if the topic does not exist, the server can’t respond with its description. In this case the server will send back and error, and the future will complete by throwing an `ExecutionException`. The actual error sent by the server will be the `cause` of the exception. Since we want to handle the case where the topic doesn’t exist, we handle these exceptions.

❸ If the topic does exist, the future completes by returning a `TopicDescription`, which contains a list of all the partitions of the topic and for each partition which broker is the leader, a list of replicas and a list of in-sync replicas. Note that this does not include the configuration of the topic. We’ll discuss configuration later in this chapter.

❹ Note that all AdminClient result objects throw `ExecutionException` when Kafka responds with an error. This is because AdminClient results are wrapped Future objects and those wrap exceptions. You always need to examine the cause of `ExecutionException` to get the error that Kafka returned.

❺ If the topic does not exist, we create a new topic. When creating a topic, you can specify just the name and use default values for all the details. You can also specify the number of partitions, number of replicas and configuration.

❻ Finally, you want to wait for topic creation to return, and perhaps validate the result. In this example, we are checking the number of partitions. Since we specified the number of partitions when we created the topic, we are fairly certain it is correct. Checking the result is more common if you relied on broker defaults when creating the topic. Note that since we are again calling `get()` to check the results of `CreateTopic`, this method could throw an exception. `TopicExistsException` is common in this scenario and you'll want to handle it (perhaps by describing the topic to check for correct configuration).

Now that we have a topic, lets delete it:

```
admin.deleteTopics(TOPIC_LIST).all().get();

// Check that it is gone. Note that due to the async nature
// it is possible that at this point the topic still exists
try {
    topicDescription = demoTopic.values().get(TOPIC_NAME).g
    System.out.println("Topic " + TOPIC_NAME + " is still a
} catch (ExecutionException e) {
    System.out.println("Topic " + TOPIC_NAME + " is gone");
}
```

At this point the code should be quite familiar. We call the method `deleteTopics` with a list of topic names to delete, and we use `get()` to wait for this to complete.

WARNING

Although the code is simple, please remember that in Kafka, deletion of topics is final - there is no “recyclebin” or “trashcan” to help you rescue the deleted topic and no checks to validate that the topic is empty and that you really meant to delete it. Deleting the wrong topic could mean un-recoverable loss of data - so handle this method with extra care.

All the examples so far have used the blocking `get()` call on the future returned by the different `AdminClient` methods. Most of the time, this is all you need - admin operations are rare and usually waiting until the operation succeeds or times out is acceptable. There is one exception - if you are writing a server that is expected to process large number of admin requests. In this case, you don’t want to block the server threads while waiting for Kafka to respond. You want to continue accepting requests from your users, sending them to Kafka and when Kafka responds, send the response to the client. In these scenarios, the versatility of `KafkaFuture` becomes quite useful. Here’s a simple example.

```
vertx.createHttpServer().requestHandler(request -> { ❶
    String topic = request.getParam("topic"); ❷
    String timeout = request.getParam("timeout");
    int timeoutMs = NumberUtils.toInt(timeout, 1000);

    DescribeTopicsResult demoTopic = admin.describeTopics(
            Collections.singletonList(topic),
            new DescribeTopicsOptions().timeoutMs(timeoutMs

    demoTopic.values().get(topic).whenComplete( ❹
            new KafkaFuture.BiConsumer<TopicDescription, Th
                @Override
                public void accept(final TopicDescription t
                                   final Throwable throwabl
                    if (throwable != null) { ❺
                      request.response().end("Error trying
                              + topic + " due to " + throwa
                    } else {
                        request.response().end(topicDescrip
                    }
                }
            });
}).listen(8080);
```

❶ We are using Vert.X to create a simple HTTP server. Whenever this server receives a request, it calls the `requestHandler` that we are defining here.

❷ The request includes topic name as a parameter, and we’ll respond with a description of this topic

❸ We call `AdminClient.describeTopics` as usual and get a wrapped Future in response

❹ But instead of using the blocking `get()` call, we instead construct a function that will be called when the Future completes.

❺ If the future completes with an exception, we send the error to the HTTP client

❻ If the future completes successfully, we respond to the client with the topic description.

The key here is that we are not waiting for response from Kafka. `DescribeTopicResult` will send the response to the HTTP client when a response arrives from Kafka. Meanwhile the HTTP server can continue processing other requests. You can check this behavior by using `SIGSTOP` to pause Kafka (don't try this in production!) and send two HTTP requests to Vert.X - one with long timeout value and one with short value. Even though you sent the second request after the first, it will respond earlier thanks to the lower timeout value, and not block behind the first request.

Configuration management

Configuration management is done by describing and updating collections of `ConfigResource`. Config resources can be brokers, broker loggers and topics. Checking and modifying broker and broker logging configuration is typically done via tools like `kafka-config.sh` or other Kafka management tools, but checking and updating topic configuration from the applications that use them is quite common.

For example, many applications rely on compacted topics for their correct operation. It makes sense that periodically (more frequently than the default retention period, just to be safe), those applications will check that the topic is indeed compacted and take action to correct the topic configuration if this is not the case.

Here's an example of how this is done:

```
ConfigResource configResource =
        new ConfigResource(ConfigResource.Type.TOPIC,TOPIC_
DescribeConfigsResult configsResult =
        admin.describeConfigs(Collections.singleton(configR
Config configs = configsResult.all().get().get(configResour

// print non-default configs
configs.entries().stream().filter(
        entry -> !entry.isDefault()).forEach(System.out::pr


// Check if topic is compacted
ConfigEntry compaction = new ConfigEntry(TopicConfig.CLEANU
        TopicConfig.CLEANUP_POLICY_COMPACT);
if (! configs.entries().contains(compaction)) {
    // if topic is not compacted, compact it
    Collection<AlterConfigOp> configOp = new ArrayList<Alte
    configOp.add(new AlterConfigOp(compaction, AlterConfigO
    Map<ConfigResource, Collection<AlterConfigOp>> alterCon
    alterConf.put(configResource, configOp);
    admin.incrementalAlterConfigs(alterConf).all().get();
} else {
    System.out.println("Topic " + TOPIC_NAME + " is compact
}
```

❶ As mentioned above, there are several types of `ConfigResource`, here we are checking the configuration for a specific topic. You can specify multiple different resources from different types in the same request.

❷ The result of `describeConfigs` is a map from each `ConfigResource` to a collection of configurations. Each configuration entry has `isDefault()` method that lets us know which configs were modified. A topic configuration is considered non-default if a user configured the topic to have a non-default value, or if a broker level configuration was modified and the topic that was created inherited this non-default value from the broker.

❸ In order to modify a configuration, you specify a map of the `ConfigResource` you want to modify and a collection of operations. Each configuration modifying operation consists of configuration entry (which is the name and value of the configuration, in this case `cleanup.policy` is the configuration name and `compacted` is the value) and the operation type. There are four types of operations that modify configuration in Kafka: `SET`, which sets the configuration value, `DELETE` which removes the value and resets to default, `APPEND` and `SUBSTRACT` - those apply only to configurations with `List` type and allows adding and removing values from the list without having to send the entire list to Kafka every time.

Describing configuration can be surprisingly handy in an emergency. I remember a time when during an upgrade, the configuration file for the brokers was accidentally replaced with a broken copy. This was discovered after restarting the first broker and noticing that it fails to start. The team did not have a way to recover the original, and we prepared for significant trial and error as we attempt to reconstruct the correct configuration and bring the broker back to life. A Site Reliability Engineer (SRE) saved the day by connecting to one of the remaining brokers and dumping their configuration using the AdminClient.

Consumer group management

We've mentioned before that unlike most message queues, Kafka allows you to re-process data in the exact order in which it was consumed and processed earlier. In Chapter 4, where we discussed consumer groups, we explained how to use the Consumer APIs to go back and re-read older messages from a topic. But using these APIs means that you programmed the ability to re-process data in advance into your application. Your application itself must expose the “re-process” functionality.

There are several scenarios in which you’ll want to cause an application to re-process messages, even if this capability was not built into the application in advance. Troubleshooting a malfunctioning application during an incident is one such scenario. Another is when preparing an application to start running on a new cluster during a disaster recovery failover scenario (we’ll discuss this in more detail in Chapter 9, when we discuss disaster recovery techniques).

In this section, we’ll look at how you can use the AdminClient to programmatically explore and modify consumer groups and the offsets that were committed by those groups. In Chapter 10 we’ll look at external tools available to perform the same operations.

Exploring Consumer Groups

If you want to explore and modify consumer groups, the first step would be to list them:

```
admin.listConsumerGroups().valid().get().forEach(System.out
```

Note that by using `valid()` method, the collection that `get()` will return will only contain the consumer groups that the cluster returned without errors, if any. Any errors will be completely ignored, rather than thrown as exceptions. The `errors()` method can be used to get all the exceptions. If you use `all()` as we did in other examples, only the first error the cluster returned will be thrown as an exception. Likely causes of such errors are authorization, where you don't have permission to view the group, or cases when the coordinator for some of the consumer groups is not available.

If we want more information about some of the groups, we can describe them:

```
ConsumerGroupDescription groupDescription = admin
        .describeConsumerGroups(CONSUMER_GRP_LIST)
        .describedGroups().get(CONSUMER_GROUP).get();
        System.out.println("Description of group " + CONSUM
                + ":" + groupDescription);
```

The description contains a wealth of information about the group. This includes the group members, their identifiers and hosts, the partitions assigned to them, the algorithm used for the assignment and the host of the group coordinator. This description is very useful when troubleshooting consumer groups. One of the most important pieces of information about a consumer group is missing from this description - inevitably, we'll want to know what was the last offset committed by the group for each partition that it is consuming, and how much it is lagging behind the latest messages in the log.

In the past, the only way to get this information was to parse the commit messages that the consumer groups wrote to an internal Kafka topic. While this method accomplished its intent, Kafka does not guarantee compatibility of the internal message formats and therefore the old method is not recommended. We'll take a look at how Kafka's AdminClient allows us to retrieve this information.

```
Map<TopicPartition, OffsetAndMetadata> offsets =
        admin.listConsumerGroupOffsets(CONSUMER_GROUP)
                .partitionsToOffsetAndMetadata().get(); ❶

Map<TopicPartition, OffsetSpec> requestLatestOffsets = new

for(TopicPartition tp: offsets.keySet()) {
    requestLatestOffsets.put(tp, OffsetSpec.latest()); ❷
}

Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo
        admin.listOffsets(requestLatestOffsets).all().get()

for (Map.Entry<TopicPartition, OffsetAndMetadata> e: offset
    String topic = e.getKey().topic();
    int partition =  e.getKey().partition();
    long committedOffset = e.getValue().offset();
    long latestOffset = latestOffsets.get(e.getKey()).offse

    System.out.println("Consumer group " + CONSUMER_GROUP
            + " has committed offset " + committedOffset
            + " to topic " + topic + " partition " + partit
            + ". The latest offset in the partition is "
            +  latestOffset + " so consumer group is "
            + (latestOffset - committedOffset) + " records
}
```

❶ We retrieve a map of all topics and partitions that the consumer group handles, and the latest committed offset for each. Note that unlike `describeConsumerGroups`, `listConsumerGroupOffsets` only accepts a single consumer group and not a collection.

❷ For each one of the topics and partitions in the results, we want to get the offset of the last message in the partition. `OffsetSpec` has three very convenient implementations - `earliest()`, `latest()` and `forTimestamp()`, those allow us to get the earlier and latest offsets in the partition, as well as the offset of the record written on or immediately after the time specified.

❸ Finally, we iterate over all the partitions and for each partition print the last committed offset, the latest offset in the partition and the lag between them.

Modifying consumer groups

Until now, we just explored available information. AdminClient also has methods for modifying consumer groups - deleting groups, removing members, deleting committed offsets and modifying offsets. These are commonly used by SREs who use them to build ad-hoc tooling to recover from an emergency.

From all those, modifying offsets is the most useful. Deleting offsets might seem like a simple way to get a consumer to “start from scratch”, but this really depends on the configuration of the consumer - if the consumer starts and no offsets are found, will it start from the beginning? Or jump to the latest message? Unless we have the code for the consumer, we can’t know. Explicitly modifying the committed offsets to the earliest available offsets will force the consumer to start processing from the beginning of the topic, and essentially cause the consumer to “reset”.

This is very useful for stateless consumers, but keep in mind that if the consumer application maintains state (and most stream processing applications maintain state), resetting the offsets and causing the consumer group to start processing from the beginning of the topic can have strange impact on the stored state. For example, suppose that you have a streams application that is continuously counting shoes sold in your store, and suppose that at 8:00 am you discover that there was an error in inputs and you want to completely re-calculate the count since 3:00 am. If you reset the offsets to 3:00 am without appropriately modifying the stored aggregate, you will count twice every shoe that was sold today (you will also process all the data between 3:00 am and 8:00 am, but lets assume that this is necessary to correct the error). You need to take care to update the stored state accordingly. In development environment we usually delete the state store completely before resetting the offsets to the start of the input topic.

Also keep in mind that consumer groups don’t receive updates when offsets change in the offset topic. They only read offsets when a consumer is assigned a new partition or on startup. To prevent you from making changes to offsets that the consumers will not know about (and will therefore override), Kafka will prevent you from modifying offsets while the consumer group is active.

With all these warnings in mind, lets look at an example:

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo

Map<TopicPartition, OffsetAndMetadata> resetOffsets = new H
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffset
        earliestOffsets.entrySet()) {
  resetOffsets.put(e.getKey(), new OffsetAndMetadata(e.getV

try {
  admin.alterConsumerGroupOffsets(CONSUMER_GROUP, resetOffs
} catch (ExecutionException e) {
  System.out.println("Failed to update the offsets committe
            + CONSUMER_GROUP + " with error " + e.getMessag
  if (e.getCause() instanceof UnknownMemberIdException)
      System.out.println("Check if consumer group is still
}
```

❶ In order to reset the consumer group so it will start processing from the earliest offset, we need to get the earliest offsets first.

❷

`alterConsumerGroupOffsets` takes as an argument a map with `OffsetAndMetadata` values. But `listOffsets` returns `ListOffsetsResultInfo`, we need to massage the results of the first method a bit, so we can use them as an argument.

❸ We are waiting on the future to complete so we can see if it completed successfully.

❹ One of the most common reasons that `alterConsumerGroupOffsets` will fail is when we didn’t stop the consumer group first. If the group is still active, our attempt to modify the offsets will appear to the consumer coordinator as if a client that is not a member in the group is committing an offset for that group. In this case, we’ll get `UnknownMemberIdException`.

Cluster Metadata

It is rare that an application has to explicitly discover anything at all about the cluster to which it connected. You can produce and consume messages without ever learning how many brokers exist and which one is the controller. Kafka clients abstract away this information - clients only need to be concerned with topics and partitions.

But just in case you are curious, this little snippet will satisfy your curiosity:

```
DescribeClusterResult cluster = admin.describeCluster();

System.out.println("Connected to cluster " + cluster.cluste
System.out.println("The brokers in the cluster are:");
cluster.nodes().get().forEach(node -> System.out.println("
System.out.println("The controller is: " + cluster.controll
```

❶ Cluster identifier is a GUID and therefore is not human readable. It is still useful to check whether your client connected to the correct cluster.

Advanced Admin Operations

In this subsection, we'll discuss few methods that are rarely used, and can be risky to use… but are incredibly useful when needed. Those are mostly important for SREs during incidents - but don't wait until you are in an incident to learn how to use them. Read and practice before it is too late. Note that the methods here have little to do with each other, except that they all fit into this category.

Adding partitions to a topic

Usually the number of partitions in a topic is set when a topic is created. And since each partition can have very high throughput, bumping against the capacity limits of a topic is rare. In addition, if messages in the topic have keys, then consumers can assume that all messages with the same key will always go to the same partition and will be processed in the same order by the same consumer.

For these reasons, adding partitions to a topic is rarely needed and can be risky - you’ll need to check that the operation will not break any application that consumes from the topic. At times, however, you really hit the ceiling of how much throughput you can process with the existing partitions and have no choice but to add some.

You can add partitions to a collection of topics using `createPartitions` method. Note that if you try to expand multiple topics at once, it is possible that some of the topics will be successfully expanded while others will fail.

```
Map<String, NewPartitions> newPartitions = new HashMap<>();
newPartitions.put(TOPIC_NAME, NewPartitions.increaseTo(NUM_
admin.createPartitions(newPartitions).all().get();
```

❶ When expanding topics, you need to specify the total number of partitions the topic will have after the partitions are added and not the number of new partitions.

TIP

Since `createPartition` method takes as a parameter the total number of partitions in the topic after new partitions are added, you may need to describe the topic it and find out how many partitions exist prior to expanding it.

Deleting records from a topic

Current privacy laws mandate specific retention policies for data. Unfortunately, while Kafka has retention policies for topics, they were not implemented in a way that guarantees legal compliance. A topic with retention policy of 30 days can have older data if all the data fits into a single segment in each partition.

`deleteRecords` method will delete all the records with offsets older than those specified when calling the method. Remember that `listOffsets` method can be used to get offsets for records that were written on or immediately after a specific time. Together, these methods can be used to delete records older than any specific point in time.

```
Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo
Map<TopicPartition, RecordsToDelete> recordsToDelete = new
for (Map.Entry<TopicPartition, ListOffsetsResult.ListOffset
        olderOffsets.entrySet())
    recordsToDelete.put(e.getKey(), RecordsToDelete.beforeO
 admin.deleteRecords(recordsToDelete).all().get();
```

Leader Election

This method allows you to trigger two different types of leader election:

- Preferred leader election: Each partition has a replica that is designated as the “preferred leader”. It is preferred because if all partitions use their preferred leader replica as leader, the number of leaders on each broker should be balanced. By default, Kafka will check every 5 minutes if the preferred leader replica is indeed the leader, and if it isn’t but it is eligible to become the leader, it will elect the preferred leader replica as leader. If this option is turned off, or if you want this to happen faster, `electLeader()` method can trigger this process.
- Unclean leader election: If the leader replica of a partition becomes unavailable, and the other replicas are not eligible to become leaders (usually because they are missing data), the partition will be without leader and therefore unavailable. One way to resolve this is to trigger “unclean” leader election - which means electing a replica that is otherwise ineligible to become a leader as the leader anyway. This will cause data loss - all the events that were written to the old leader and were not replicated to the new leader will be lost. `electLeader()` method can also be used to trigger unclean leader elections.

The method is asynchronous, which means that even after it returns successfully, it takes a while until all brokers become aware of the new state and calls to `describeTopics()` can return inconsistent results. If you trigger leader election for multiple partitions, it is possible that the operation will be successful for some partitions and will fail for others.

```
Set<TopicPartition> electableTopics = new HashSet<>();
electableTopics.add(new TopicPartition(TOPIC_NAME, 0));
try {
    admin.electLeaders(ElectionType.PREFERRED, electableTop
} catch (ExecutionException e) {
    if (e.getCause() instanceof ElectionNotNeededException)
        System.out.println("All leaders are preferred alrea
    }
}
```

❶ We are electing the preferred leader on a single partition of a specific topic. We can specify any number of partitions and topics. If you call the command with `null` instead of a collection of partitions, it will trigger the election type you chose for all partitions.

❷ If the cluster is in a healthy state, the command will do nothing - preferred leader election and unclean leader election only have effect when a replica other than the preferred leader is the current leader.

Reassigning Replicas

Sometimes, you don’t like the current location of some of the replicas. Maybe a broker is overloaded and you want to move some replicas away. Maybe you want to add more replicas. Maybe you want to move all replicas away from a broker so you can remove the machine. Or maybe few topics are so noisy that you need to isolate them away from the rest of the workload. In all these scenarios, `alterPartitionReassignments` gives you fine-grain control over the placement of every single replica for a partition. Keep in mind that when you reassign replicas from one broker to another, it may involve copying large amounts of data from one broker to another. Be mindful of the available network bandwidth and throttle replication using quotas if needed: quotas are broker configuration, so you can describe them and update them with `AdminClient`.

For this example, assume that we have a single broker with id 0. Our topic has several partitions, all with one replica on this broker. After adding a new broker, we want to use it to store some of the replicas of the topic. So we are going to assign each partition in the topic in a slightly different way:

```
Map<TopicPartition, Optional<NewPartitionReassignment>> rea
reassignment.put(new TopicPartition(TOPIC_NAME, 0),
        Optional.of(new NewPartitionReassignment(Arrays.asL
reassignment.put(new TopicPartition(TOPIC_NAME, 1),
        Optional.of(new NewPartitionReassignment(Arrays.asL
reassignment.put(new TopicPartition(TOPIC_NAME, 2),
        Optional.of(new NewPartitionReassignment(Arrays.asL
reassignment.put(new TopicPartition(TOPIC_NAME, 3), Optiona
try {
    admin.alterPartitionReassignments(reassignment).all().g
} catch (ExecutionException e) {
    if (e.getCause() instanceof NoReassignmentInProgressExc
        System.out.println("Cancelling a reassignment that
    }
}
System.out.println("currently reassigning: " +
        admin.listPartitionReassignments().reassignments().
demoTopic = admin.describeTopics(TOPIC_LIST);
topicDescription = demoTopic.values().get(TOPIC_NAME).get()
System.out.println("Description of demo topic:" + topicDesc
❻
```

❶ We've added another replica to partition 0, placed the new replica on the new broker, but left the leader on the existing broker

❷ We didn't add any replicas to partition 1, simply moved the one existing replica to the new broker. Since I have only one replica, it is also the leader.

❸ We've added another replica to partition 2 and made it the preferred leader. The next preferred leader election will switch leadership to the new replica on the new broker. The existing replica will then become a follower.

❹ There is no on-going reassignment for partition 3, but if there was, this would have cancelled it and returned the state to what it was before the reassignment operation started.

❺ We can list the on-going reassignments

❻ We can also try to print the new state, but remember that it can take a while until it shows consistent results

Testing

Apache Kafka provides a test class `MockAdminClient`, which you can initialize with any number of brokers and use to test that your applications behave correctly without having to run an actual Kafka cluster and really perform the admin operations on it. Some of the methods have very comprehensive mocking - you can create topics with `MockAdminClient` and a subsequent call to `listTopics()` will list the topics you "created".

However, not all methods are mocked - if you use `AdminClient` with version 2.5 or earlier and call `incrementalAlterConfigs()` of the `MockAdminClient`, you will get an `UnsupportedOperationException`, but you can handle this by injecting your own implementation.

In order to demonstrate how to test using `MockAdminClient`, lets start by implementing a class that is instantiated with an admin client and uses it to create topics:

```
public TopicCreator(AdminClient admin) {
    this.admin = admin;
}

// Example of a method that will create a topic if its name
public void maybeCreateTopic(String topicName)
        throws ExecutionException, InterruptedException {
    Collection<NewTopic> topics = new ArrayList<>();
    topics.add(new NewTopic(topicName, 1, (short) 1));
    if (topicName.toLowerCase().startsWith("test")) {
        admin.createTopics(topics);

        // alter configs just to demonstrate a point
        ConfigResource configResource =
                  new ConfigResource(ConfigResource.Type.TO
        ConfigEntry compaction =
                  new ConfigEntry(TopicConfig.CLEANUP_POLIC
                          TopicConfig.CLEANUP_POLICY_COMPAC
        Collection<AlterConfigOp> configOp = new ArrayList<
        configOp.add(new AlterConfigOp(compaction, AlterCon
        Map<ConfigResource, Collection<AlterConfigOp>> alte
        alterConf.put(configResource, configOp);
        admin.incrementalAlterConfigs(alterConf).all().get(
    }
}
```

The logic here isn't sophisticated: `maybeCreateTopic` will create the topic if the topic name starts with "test". We are also modifying the topic configuration, so we can show how to handle a case where the method we use isn't implemented in the mock client.

We'll start testing by instantiating our mock client:

NOTE

We are using the Mockito testing framework to verify that the `MockAdminClient` methods are called as expected and to fill in for the unimplemented methods. Mockito is a fairly simple mocking framework with nice APIs, which makes it a good fit for a small example of a unit test.

```
@Before
public void setUp() {
    Node broker = new Node(0,"localhost",9092);
    this.admin = spy(new MockAdminClient(Collections.single

    // without this, the tests will throw
    // `java.lang.UnsupportedOperationException: Not implem
    AlterConfigsResult emptyResult = mock(AlterConfigsResul
    doReturn(KafkaFuture.completedFuture(null)).when(emptyR
    doReturn(emptyResult).when(admin).incrementalAlterConfi
}
```

❶ `MockAdminClient` is instantiated with a list of brokers (here I'm using just one), and one broker that will be our controller. The brokers are just the broker id, hostname and port - all fake, of course. No brokers will run while executing these tests. We'll use Mockito's `spy` injection, so we can later check that `TopicCreator` executed correctly.

❷ Here we use Mockito's `doReturn` methods to make sure the mock admin client doesn't throw exceptions. Since the method we are testing expects `AlterConfigResult` that returns a `KafkaFuture` when calling its `all()` method, we made sure that the fake `incrementalAlterConfigs` returns exactly that.

Now that we have a properly fake AdminClient, we can use it to test whether `maybeCreateTopic()` method works properly:

```
@Test
public void testCreateTestTopic()
        throws ExecutionException, InterruptedException {
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("test.is.a.test.topic");
    verify(admin, times(1)).createTopics(any()); ❶
}

@Test
public void testNotTopic() throws ExecutionException, Inter
    TopicCreator tc = new TopicCreator(admin);
    tc.maybeCreateTopic("not.a.test");
    verify(admin, never()).createTopics(any()); ❷
}
```

❶ The topic name starts with “test”, so we expect `maybeCreateTopic()` to create a topic. We are checking that `createTopics()` was called once.

❷ When the topic name doesn’t start with “test”, we’re verifying that `createTopics()` was not called at all.

One last note: Apache Kafka published MockAdminClient in a test jar, so make sure your `pom.xml` includes a test dependency:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.5.0</version>
    <classifier>test</classifier>
    <scope>test</scope>
</dependency>
```

Summary

AdminClient is a useful tool to have in your Kafka development kit. It is useful for application developers who want to create topics on the fly and validate that the topics they are using are configured correctly for their application. It is also useful for operators and SREs who want to create tooling and automation around Kafka or need to recover from an incident. AdminClient has so many useful methods that SREs can think of it as a Swiss Army Knife for Kafka operations.

In this chapter we covered all the basics of using Kafka's AdminClient - topic management, configuration management and consumer group management. Plus few other useful methods that are good to have in your back pocket - you never know when you'll need them.

About the Authors

Gwen Shapira is a system architect at Confluent helping customers achieve success with their Apache Kafka implementation. She has 15 years of experience working with code and customers to build scalable data architectures, integrating relational and big data technologies. She currently specializes in building real-time reliable data processing pipelines using Apache Kafka. Gwen is an Oracle Ace Director, an author of “Hadoop Application Architectures”, and a frequent presenter at data driven conferences. Gwen is also a committer on the Apache Kafka and Apache Sqoop projects.

Todd Palino is a senior staff site reliability engineer at LinkedIn, tasked with keeping the largest deployment of Apache Kafka, Zookeeper, and Samza fed and watered. He is responsible for architecture, day-to-day operations, and tools development, including the creation of an advanced monitoring and notification system. Todd is the developer of the open source project Burrow, a Kafka consumer monitoring tool, and can be found sharing his experience with Apache Kafka at industry conferences and tech talks. Todd has spent more than 20 years in the technology industry running infrastructure services, most recently as a systems engineer at Verisign, developing service management automation for DNS, networking, and hardware management, as well as managing hardware and software standards across the company.

Rajini Sivaram is a Software Engineer at Confluent designing and developing security features for Kafka. She is an Apache Kafka Committer and member of the Apache Kafka Program Management Committee. Prior to joining Confluent, she was at Pivotal working on a high-performance reactive API for Kafka based on Project Reactor. Earlier, Rajini was a key developer on IBM Message Hub which provides Kafka-as-a-Service on the IBM Bluemix platform. Her experience ranges from parallel and distributed systems to Java virtual machines and messaging systems.

Neha Narkhede is cofounder and head of engineering at Confluent, a company backing the popular Apache Kafka messaging system. Prior to founding Confluent, Neha led streams infrastructure at LinkedIn, where she was responsible for LinkedIn's streaming infrastructure built on top of Apache Kafka and Apache Samza. She is one of the initial authors of Apache Kafka and a committer and PMC member on the project.

Colophon

The animal on the cover of *Kafka: The Definitive Guide* is a blue-winged kookaburra (*Dacelo leachii*). It is part of the Alcedinidae family and can be found in southern New Guinea and the less dry area of northern Australia. They are considered to be river kingfisher birds.

The male kookaburra has a colorful look. The lower wing and tail feathers are blue, hence its name, but tails of females are reddish-brown with black bars. Both sexes have cream colored undersides with streaks of brown, and white irises in their eyes. Adult kookaburras are smaller than other kingfishers at just 15 to 17 inches in length and, on average, weigh about 260 to 330 grams.

The diet of the blue-winged kookaburra is heavily carnivorous, with prey varying slightly given changing seasons. For example, in the summer months there is a larger abundance of lizards, insects, and frogs that this bird feeds on, but drier months introduce more crayfish, fish, rodents, and even smaller birds into their diet. They're not alone in eating other birds, however, as red goshawks and rufous owls have the blue-winged kookaburra on their menu when in season.

Breeding for the blue-winged kookaburra occurs in the months of September through December. Nests are hollows in the high parts of trees. Raising young is a community effort, as there is at least one helper bird to help mom and dad. Three to four eggs are laid and incubated for about 26 days. Chicks will fledge around 36 days after hatching—if they survive. Older siblings have been known to kill the younger ones in their aggressive and competitive first week of life. Those who aren't victims of fratricide or other causes of death will be trained by their parents to hunt for 6 to 10 weeks before heading off on their own.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *English Cyclopedia*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.