8 Kafka storage

This chapters covers

- How long to retain data
- Data movement into and out of Kafka
- Data architectures Kafka enables
- Storage for cloud instances and containers

So far we have thought of our data as moving into and out of Kafka for brief periods of time. Another decision to consider is where our data should live long term. When you use databases like MySQL or MongoDB®, you may not always think about if or how that data expires. Rather, you know that the data is (likely) going to exist for the majority of your application's entire lifetime. In comparison, Kafka's storage logically sits somewhere between the long-term storage solutions of a database and the transient storage of a message broker, especially if we think of message brokers holding onto messages until they are consumed by a client, as it often is in other message brokers. Let's look at a couple of options for storing and moving data in our Kafka environment.

8.1 How long to store data

Currently, the default retention limit for data in Kafka topics is seven days, but we can easily configure this by time or data size [1]. But can Kafka hold data itself for a period of years? One real-world example is how the *New York Times* uses Kafka. The content in their cluster is in a single partition that was less than 100 GB at the time of writing [2]. If you recall from our discussion in chapter 7 about partitions, you know that all of this data exists on a single broker drive (as do any replica copies on their own drives) as partitions are not split between brokers. Because storage is considered to be relatively cheap and the capacity of modern hard drives is way beyond hundreds of gigabytes, most companies would not have any size issues with keeping that data around. Is this a valid use of Kafka or an abuse of its intended purpose and design? As long as you have the space for your planned growth on a disk for future use, you might have found a good pattern for handling your specific workload.

How do we configure retention for brokers? The main considerations are the size of the logs and the length of time the data exists. Table 8.1 shows some of the broker configuration options that are helpful for retention [3].

Table 8.1 Broker retention configuration

Key	Purpose
`log.retention.bytes`	The largest size threshold in bytes for deleting a log.
`log.retention.ms`	The length in milliseconds a log will be maintained before being deleted.
`log.retention.minutes`	Length before deletion in minutes. `log.retention.ms` is used as well if both are set.
`log.retention.hours`	Length before deletion in hours. `log.retention.ms` and `log.retention.minutes` would be used before this value if either of those are set.

How do we disable log retention limits and allow them to stay forever? By setting both `log.retention.bytes` and `log.retention.ms` to `–1`, we can effectively turn off data deletion [4].

Another thing to consider is how we can get similar retention for the latest values by using keyed events with a compacted topic. Although we can still remove data during compaction cleaning, the most recent keyed messages will always be in the log. This is a good way to retain data in use cases where we do not need every event (or history) of how a key changed state from the current value.

What if we want our data to stick around for a while, but simply do not have the disk space to hold our data on brokers? Another option for long-term storage is to move the data outside of Kafka and not retain it internally to the Kafka brokers themselves. Before data is removed by retention from Kafka, we could store the data in a database, in a Hadoop Distributed File System (HDFS™), or upload our event messages into something like cloud storage. All of these paths are valid options and could provide more cost-effective means of holding onto our data after our consumers process it.

8.2 Data movement

Almost all companies seem to have a need for transforming the data that they receive. Sometimes, it is specific to an area within the company or due to third-party integrations. A popular term that many people use in this data transformation space is *ETL* (extract, transform, load). We can use tooling or code to take data in its original format, transform the data, and then place it into a different table or data store. Kafka can play a key role in these data pipelines.

8.2.1 Keeping the original event

One thing that we would like to note is our preference for event formats inside of Kafka. Although open to debate and your use case requirements, our preference is to store messages in the original format in a topic. Why keep the original message and not format it immediately before placing it into a topic? Having the original message makes it easier to go back and start over if you inadvertently messed up your transform logic. Instead of having to try to figure out how to fix your mistake on the altered data, you can always just go back to the original data and start again. We know that most of us usually have that experience when trying to format a date or the first time we run a regular expression. Sometimes you need a couple of shots at formatting the data the way you want.

Another plus for getting the entire original message is that data you don't use today might be used in the future. Let's say the year is 1995, and you are getting a field from a vendor called `mobile`. Your business will never need that field, right? Once you see the need to launch your first text marketing campaign, you'll be thanking your past self that you kept that original, "useless" data.

Although the `mobile` field might be a trivial example for some, it is interesting to think about usage for data analysis. What if your models start to see trends on data that you once thought wouldn't matter? By retaining all the data fields, you might be able to go back to that data and find insights you never expected.

8.2.2 Moving away from a batch mindset

Does the general topic of ETL or data pipelines bring terms to mind such as *batch*, *end of day*, *monthly*, or even *yearly*? One of the shifts from the data transformation processes of the past is the idea that you can continuously stream your data into various systems without delay. With Kafka, for example, you can keep the pipeline running in near-real time, and you can use its stream-processing platform to treat your data as an infinite series of events.

We mention this as a reminder that Kafka can help enable a shift in the way you think of your data altogether. You do not have to wait for a nightly job to run and update a database. You also do not have to wait for a nightly window with less traffic to do intensive ETL tasks; you can do these as they stream into your system and have pipelines that are constantly working for your applications in real time. Let's take a look at tools available that might help you use your pipelines in the future or make better use of your pipelines today.

8.3 Tools

Data movement is a key to many systems, Kafka included. Although you can stay inside the open source Kafka and Confluent offerings like Connect, which was discussed in chapter 3, there are other tools that might fit your infrastructure or are already available in your tool suite. Depending on your specific data source or sinks, the options mentioned in the following sections might help you achieve your goals. Note that although some tools in this section include sample configuration and commands, more setup (not shown) might be required before you can run these commands on your local machines. Hopefully, this section gives you enough information to pique your interest and allow you to start exploring on your own.

8.3.1 Apache Flume

If you were first introduced to Kafka through work in the big data space, it is a strong possibility that you might have used Flume in relation to your cluster. If you have ever heard the term *Flafka*, you have definitely used this Kafka and Flume integration. Flume can provide an easier path for getting data into a cluster and relies more on configuration than on custom code. For example, if you want to ingest data into your Hadoop cluster and already have support from a vendor on these various pieces, Flume is a solid option to get data into your Kafka cluster.

Figure 8.1 shows an example of how a Flume agent runs on a node as its own process. It watches the files local to that server and then uses the configuration for the agent that you provided to send data to a sink.

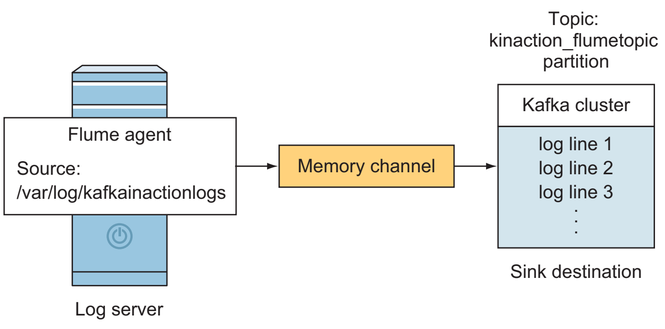

Figure 8.1 Flume agent

Let's take a look again at integrating log files (our source of data) using a Flume agent into a Kafka topic (our data sink). Listing 8.1 shows a sample configuration file that we could use to set up a local Flume agent to watch a directory for changes [5]. The changes are placed in a Kafka topic, titled `kinaction_flumetopic`. To imagine this example, here's a comparison: it is like using a `cat` command on a file in a directory to read the file and send the result to a specific Kafka topic.

Listing 8.1 Flume configuration for watching a directory

```
ag.sources = logdir
ag.sinks = kafkasink
ag.channels = c1

#Configure the source directory to watch
ag.sources.logdir.type = spooldir
ag.sources.logdir.spoolDir = /var/log/kafkainactionlogs
...
ag.sinks.kafkasink.channel = c1
ag.sinks.kafkasink.type = org.apache.flume.sink.kafka.KafkaSink
ag.sinks.kafkasink.kafka.topic = kinaction_flumetopic
...
# Bind both the sink and source to the same channel
ag.sources.logdir.channels = c1
ag.sinks.kafkasink.channel = c1
```

Defines custom names for the source, sink, and channel

Specific spooldir source lets Flume know which directory to watch for log entries.

This section defines our topic and Kafka cluster information where we want our data to end up.

Attaches the source to the sink by the defined channel

Listing 8.1 shows how we could configure a Flume agent running on a server. You should notice that the sink configuration looks a lot like the properties we have used before in our Java client producer code.

It is also interesting to note that Flume can use Kafka as not only a source or as a sink, but also as a channel. Because Kafka is seen as a more reliable channel for events, Flume can use Kafka to deliver messages between various sources and sinks.

If you are reviewing Flume configurations and see Kafka mentioned, be sure to notice where and how it is actually used. The following listing shows the Flume agent configuration we can use to provide a reliable channel between various sources and sinks that Flume supports [5].

Listing 8.2 Flume Kafka channel configuration

```
ag.channels.channel1.type =
➥ org.apache.flume.channel.kafka.KafkaChannel
ag.channels.channel1.kafka.bootstrap.servers =
➥ localhost:9092,localhost:9093,localhost:9094
ag.channels.channel1.kafka.topic = kinaction_channel1_ch
ag.channels.channel1.kafka.consumer.group.id =
➥ kinaction_flume
```

Flume uses the KafkaChannel class as the Kafka channel type.

Provides our servers to connect to

The topic that holds the data between source and sink

Provides a consumer group to avoid collisions with other consumers

8.3.2 Red Hat® Debezium™

Debezium (https://debezium.io) describes itself as a distributed platform that helps turn databases into event streams. In other words, updates to our database can be treated as events! If you have a database background (or not), you may have heard of the term *change data capture* (CDC). As the name implies, the data changes can be tracked and used to react to those changes. At the time of writing this chapter, Debezium supports MySQL, MongoDB, PostgreSQL®, Microsoft SQL Server™, Oracle, and IBM Db2. Cassandra™ and Vitess™ are in an incubating status as well [6]. Please see the current list of connectors at https://debezium.io/documentation/reference/connectors/.

Debezium uses connectors and Kafka Connect to record the events our application consumes from Kafka as a normal client. Figure 8.2 shows an example of Debezium when it is registered as a connector in regard to Kafka Connect.

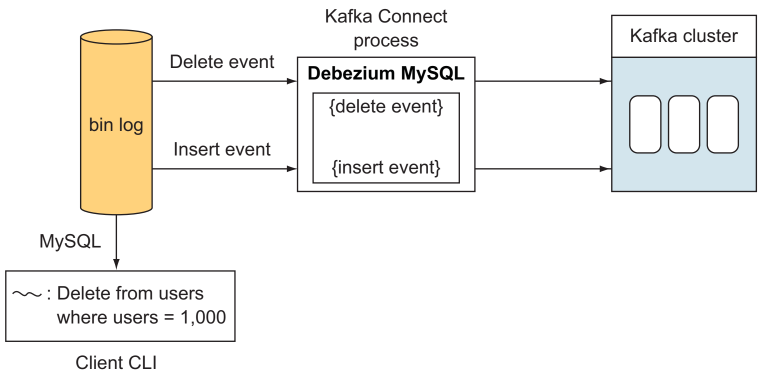

Figure 8.2 Kafka Connect and Debezium used with a MySQL database

In our scenario, a developer uses a command line interface (CLI) and deletes a user against the MySQL database instance that is being monitored for changes. Debezium captures the event written to the database's internal log, and that event goes through the connector service and feeds into Kafka. If a second event, such as a new user, is inserted into the database, a new event is captured.

As an additional note, although not Kafka-specific, there are other examples of using techniques like CDC to provide timely events or changes to your data that might help you draw a parallel to what Debezium is aiming for overall.

8.3.3 Secor

Secor (https://github.com/pinterest/secor) is an interesting project from Pinterest that has been around since 2014. It aims to help persist Kafka log data to a variety of storage options, including S3 and Google Cloud Storage™ [7]. The options for output are also various, including sequence, Apache ORC™, and Apache Parquet™ files as well as other formats. As always, one major benefit of projects having source code in a public repository is that we can see how other teams have implemented requirements that might be similar to ours.

Figure 8.3 shows how Secor would act as a consumer of a Kafka cluster, much like any other application. Having a consumer added to a cluster for data backup is not a big deal. It leverages the way Kafka has always handled multiple readers of the events.

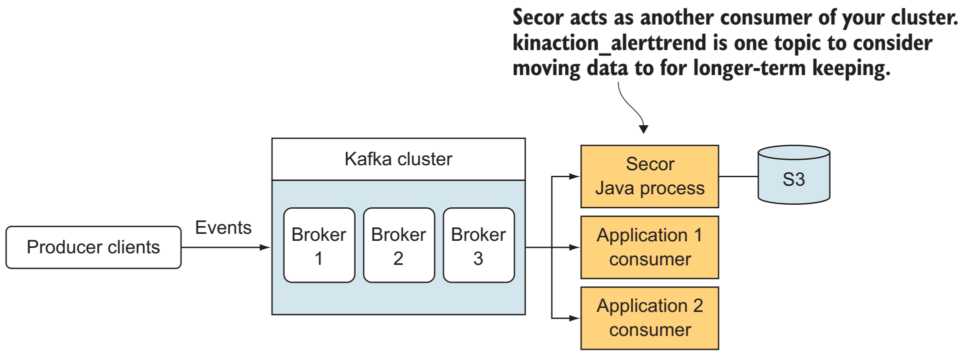
Secor acts as another consumer of your cluster. kinaction_alerttrend is one topic to consider moving data to for longer-term keeping.
Kafka cluster
Producer clients
Events
Broker 1
Broker 2
Broker 3
Secor Java process
Application 1 consumer
Application 2 consumer
S3

Figure 8.3 Secor acting as a consumer and placing data into storage.

Secor runs as a Java process and can be fed our specific configurations. In effect, it acts as another consumer of our existing topic(s) to gather data to end up in a specific destination like an S3 bucket. Secor does not get in the way of our other consumers, and it allows us to have a copy of our events so that they are not lost once Kafka retention removes data from its logs.

Invoking Secor should be familiar to those who are used to working with JARs in a Java environment. We can pass arguments with the standard `-D` parameters to the Secor application. In this instance, the most important file to update is the properties file with the configuration options. This file lets us fill in the details about our specific cloud storage bucket, for example.

8.3.4 Example use case for data storage

Let's look at an example of how moving data out of Kafka for storage could be used at a later time. First, to clarify, we will break down our usage of the same data between two different areas. One area is working with the data in an operational manner as it comes into Kafka.

Operational data is the events that are produced by our day-to-day operations. We can think of an event to order an item from a website as an example. A purchase event triggers our application into motion and does so in a low-latency way. The value of this data to our real-time applications might warrant keeping the data for a couple of days until the order is completed and mailed. After this timeframe, the event may become more important for our analytical systems.

Analytical data, while based on that same operational data, is usually used more to make business decisions. In traditional systems, this is where processes like a data warehouse, an online analytical processing system (OLAP), and Hadoop shine. That event data can be mined using different combinations of fields in our events in different scenarios to find insights into sales data, for instance. If we notice that sales of cleaning supplies always spike before a holiday, we might use that data to generate better sale options for our business in the future.

8.4 Bringing data back into Kafka

One of the most important things to note is that just because our data has left Kafka does not mean that it can't be put back in again. Figure 8.4 shows an example of data that lived out its normal lifespan in Kafka and was archived in cloud storage like S3. When a new application logic change required the older data be reprocessed, we did not have to create a client to read from both S3 and Kafka. Rather, using a tool like Kafka Connect, we can load that data from S3 back into Kafka! The interface stays the same from the point of view of our applications. Although it might not seem obvious at first glance why we would want to do such a thing, let's consider a situation in which we find value in moving our data back into Kafka after we have processed it and the retention period has passed.

Imagine a team working on trying to find patterns in data that they collected throughout years of handling events. In our example, there are terabytes of data. To serve operational real-time data collection, this data was moved from Kafka into HDFS after real-time consumers dealt with the messages. Does our application logic now have to pull from HDFS directly? Why not just pull it back into Kafka, and our application can process the data as it had before? Loading data into Kafka again is a valid way of reprocessing data that may have aged out of our system. Figure 8.4 shows another example of how we can move data back into Kafka.

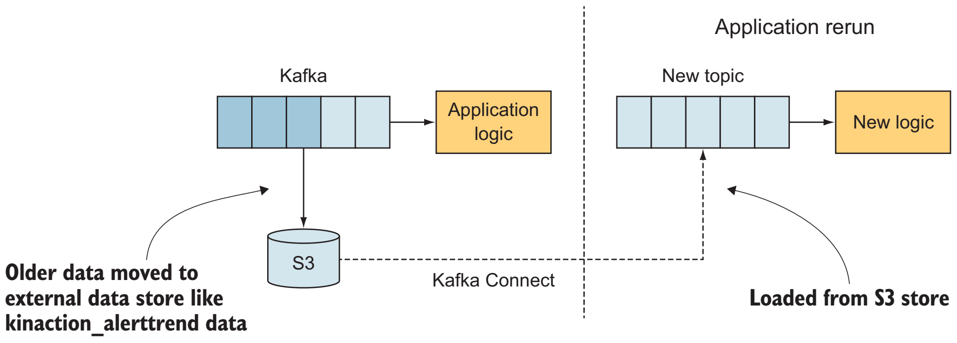

Figure 8.4 Moving data back into Kafka

After some time, events are not available to the applications due to data retention configurations within Kafka. However, we have a copy of all previous events in an S3 bucket. Let's say that we have a new version of our previous application and would prefer to go through all of the previous data events as in our previous application. However, because those events are not in Kafka, do we pull them from S3 now? Do we want our application logic to pull from various sources or just to have one interface (that being Kafka)? We can create a new topic in our existing Kafka cluster and load the data from S3 with Kafka Connect, placing the data into a new Kafka topic. Our application can then run against Kafka, processing events without having to change any processing logic.

The thought process is really to keep Kafka as the interface of our application and not have to create multiple ways to pull data into processing. Why create and maintain custom code to pull from different locations when we can use an existing tool like Connect to move the data to or from Kafka? Once we have our data in that one interface, we can process it the same.

NOTE Keep in mind this technique only applies to data that has been removed from Kafka. If you still have the total timeline of data that you need in Kafka, you can always seek to the earlier offsets.

8.4.1 Tiered storage

A newer option from the Confluent Platform version 6.0.0 on is called Tiered Storage. In this model, local storage is still the broker itself, and remote storage is introduced for data that is older (and stored in a remote location) and controlled by time configuration (`confluent.tier.local.hotset.ms`) [8].

8.5 Architectures with Kafka

Although there are various architectural patterns that view your data as events when building your products, such as model-view-controller (MVC), peer-to-peer (P2P), or service-oriented architecture (SOA) to name a few, Kafka can change the way you think about your entire architectural design. Let's take a peek at a couple of architectures that could be powered by Kafka (and to be fair, other streaming platforms). This will help us get a different perspective on how we might design systems for our customers.

The term *big data* is used in reference to some of these discussions. It is important to note that the amount of data and the need to process that data in a timely manner were the drivers that led to some of these system designs. However, these architectures are not limited to fast data or big data applications only. By hitting the limits of specific traditional database technologies, new views on data evolved. Let's look at two of them in the following sections.

8.5.1 Lambda architecture

If you have ever researched or worked with data applications that have included needs for both batch processing and operational workloads, you might have seen references to lambda architecture. The implementation of this architecture can start with Kafka as well, but it is a little more complex.

The real-time view of the data is combined with a historical view to serve end users. The complexity of merging these two data views should not be ignored. For the authors, it was a challenge to rebuild the serving table. Also, you are likely going to have to maintain different interfaces for your data as you work with the results from both systems.

The book *Big Data*, written by Nathan Marz with James Warren, discusses the lambda architecture more fully and goes into details about the batch, serving, and speed layers [9]. Figure 8.5 shows an example of how taking customer orders can be thought of in a batch and a real-time way. The customer totals from the previous days can be integrated with orders happening during the day into a combined data view to end users.

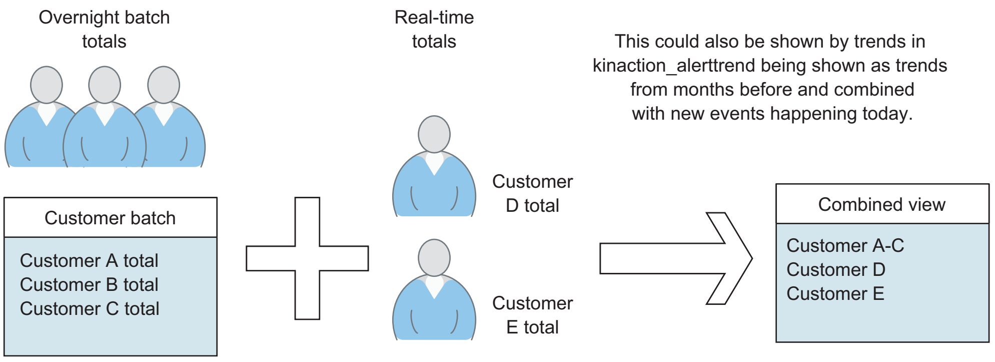

Figure 8.5 Lambda architecture

Taking the concepts from figure 8.5 and to get a feel for this architecture, let's look at each layer at a high level. These layers are discussed in *Big Data* by Marz:

- *Batch*—This layer is similar to the way batch processing with MapReduce occurs in a system like Hadoop. As new data is added to your data stores, the batch layer continues to precompute the view of the data that already lives in the system.
- *Speed*—This layer is similar in concept to the batch layer except it produces views from recent data.
- *Serving*—This layer updates the views it sends to consumers after each update to the batch views.

For the end user, the lambda architecture unites data from the serving layer and the speed layer to answer requests with a complete view of all recent and past data. This real-time streaming layer is the most obvious place for Kafka to play a role, but it can also be used to feed the batch layer.

8.5.2 Kappa architecture

Another architectural pattern that can leverage the power of Kafka is kappa architecture. This architecture was proposed by the co-creator of Kafka, Jay Kreps [10]. Think about wanting to maintain a system that impacts your users without disruption. One way to do this is to switch out your updated views like in lambda. Another way to do this is by running the current system in parallel to the new one and cutting over once the new version is ready to serve traffic. Part of this cutover is of course making sure that the data that is being served by the older version will be reflected correctly in the newer version.

You only regenerate the user-facing data when you need to. There is no need to merge old and new data, which is an ongoing process for some lambda implementations. It does not have to be a continuous job, but rather invoked when you need an application logic change. Also, there's no need to change your interface to your data. Kafka can be used by both your new and old application code at the same time. Figure 8.6 shows how customer events are used to create a view without using a batch layer.

Figure 8.6 shows customer events from the past and present being used directly to create a view. Imagine the events being sourced from Kafka and then using Kafka Streams or ksqlDB to read all the events in near-real time and creating a view for end

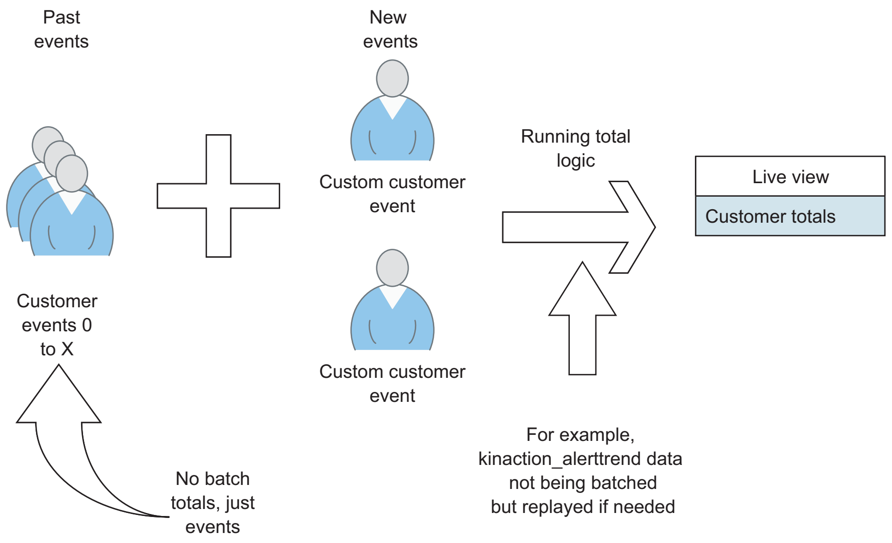

Figure 8.6 Kappa architecture

users. If a change is ever needed to how customer events are processed, a second application can be created with different logic (like a new ksqlDB query), using the same data source (Kafka) as before. There is no need to have a batch layer (and manage it) as there is only streaming logic used for making your end user views.

8.6 Multiple cluster setups

Most of our topics and discussions so far have been from the viewpoint of our data in one cluster. But Kafka scales well, and it is not unheard of to reach hundreds of brokers for a single cluster. However, a one-size cluster does not fit all infrastructures. One of the concerns we run into when talking about cluster storage is where you serve your data in relation to your end user clients. In this section, we will talk about scaling by adding clusters rather than by adding brokers alone.

8.6.1 Scaling by adding clusters

Usually, the first things to scale would be the resources inside your existing cluster. The number of brokers is the first option that makes a straightforward path to growth. Netflix®'s multicluster strategy is a captivating take on how to scale Kafka clusters [11]. Instead of using only the broker number as the way to scale the cluster, they found they could scale by adding clusters themselves!

This design brings to mind the idea of Command Query Responsibility Segregation (CQRS). For more details on CQRS, check out Martin Fowler's site at https://martinfowler.com/bliki/CQRS.html, specifically the idea of separating the load of reading data from that of writing data [12]. Each action can scale in an independent manner without limiting other actions. Although CQRS is a pattern that can add complexity to our systems, it is interesting to note how this specific example helps manage the performance of a large cluster by separating the load of producers sending data into Kafka from the sometimes much larger load of consumers reading the data.

8.7 Cloud- and container-based storage options

Although we talked about Kafka log directories in chapter 6, we did not address the types of instances to use in environments that provide more short-lived storage. For reference, Confluent shared a study on deployments with AWS considerations in which they looked at the storage type trade-offs [13].

Another option is to look at Confluent Cloud (https://www.confluent.io/confluent-cloud/). This option allows you to worry less about the underlying storage used across cloud providers and how it is managed. As always, remember that Kafka itself keeps evolving and reacting to the needs that users run into as daily challenges. KIP-392 shows an item that was accepted at the time of this writing, which seeks to help address the issues of a Kafka cluster spanning data centers. The KIP is titled "Allow consumers to fetch from the closest replica" [14]. Be sure to check out recent KIPs (Kafka Improvement Proposals) from time to time to see how Kafka evolves in exciting ways.

8.7.1 Kubernetes clusters

Dealing with a containerized environment, we might run into challenges similar to what we would in the cloud. If we hit a poorly configured memory limit on our broker, we might find ourselves on an entirely new node without our data unless the data persists correctly. If we are not in a sandbox environment in which we can lose the data, persistent volume claims may be needed by our brokers to ensure that our data survives any restarts, failures, or moves. Although the broker instance container might change, we should be able to claim the previous persistent volume.

Kafka applications will likely use the StatefulSet API in order to maintain the identity of each broker across failures or pod moves. This static identity also helps us claim the same persistent volumes that were used before our pod went down. There are already Helm® charts (https://github.com/confluentinc/cp-helm-charts) to help us get started with a test setup as we explore Kubernetes [15]. Confluent for Kubernetes helps as well with our Kubernetes management [16].

The scope of Kubernetes is relatively large to cover in our discussion, but the key concerns are present regardless of our environment. Our brokers have an identity in the cluster and are tied to the data that each is related to. To keep the cluster healthy, those brokers need the ability to identify their broker-managed logs across failures, restarts, or upgrades.

Summary

- Data retention should be driven by business needs. Decisions to weigh include the cost of storage and the growth rate of our data over time.
- Size and time are the basic parameters for defining how long data is retained on disk.
- Long-term storage of data outside of Kafka is an option for data that might need to be retained for long periods. Data can be reintroduced as needed by producing the data into a cluster at a later time.
- The ability of Kafka to handle data quickly and also replay data can enable architectures such as the lambda and kappa architectures.
- Cloud and container workloads often involve short-lived broker instances. Data that needs to be persisted requires a plan for making sure newly created or recovered instances can utilize that data across all instances.

References

1 "Kafka Broker Configurations." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html#brokerconfigs_log.retention.hours (accessed December 14, 2020).

2 B. Svingen. "Publishing with Apache Kafka at The New York Times." Confluent blog (September 6, 2017). https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/ (accessed September 25, 2018).

3 "Kafka Broker Configurations." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html (accessed December 14, 2020).

4 "Kafka Broker Configurations: log.retention.ms." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/installation/configuration/broker-configs.html#brokerconfigs_log.retention.ms (accessed December 14, 2020).

5 "Flume 1.9.0 User Guide: Kafka Sink." Apache Software Foundation (n.d.). https://flume.apache.org/releases/content/1.9.0/FlumeUserGuide.html#kafka-sink (accessed October 10, 2019).

6 "Connectors." Debezium documentation (n.d.). https://debezium.io/documentation/reference/connectors/ (accessed July 20, 2021).

7 "Pinterest Secor." Pinterest. GitHub. https://github.com/pinterest/secor/blob/master/README.md (accessed June 1, 2020).

8 "Tiered Storage." Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/tiered-storage.html (accessed June 2, 2021).

9 N. Marz and J. Warren. *Big Data: Principles and best practices of scalable real-time data systems*. Shelter Island, NY, USA: Manning, 2015.

10 J. Kreps. "Questioning the Lambda Architecture." O'Reilly Radar (July 2, 2014). https://www.oreilly.com/radar/questioning-the-lambda-architecture/ (accessed October 11, 2019).

11 A. Wang. "Multi-Tenant, Multi-Cluster and Hierarchical Kafka Messaging Service." Presented at Confluent's Kafka Summit, San Francisco, USA, 2017 Presentation [online]. https://www.confluent.io/kafka-summit-sf17/multitenant-multicluster-and-hieracrchical-kafka-messaging-service/.

12 M. Fowler. "CQRS" (July 14, 2011). https://martinfowler.com/bliki/CQRS.html (accessed December 11, 2017).

13 A. Loddengaard. "Design and Deployment Considerations for Deploying Apache Kafka on AWS." Confluent blog (July 28, 2016). https://www.confluent.io/blog/design-and-deployment-considerations-for-deploying-apache-kafka-on-aws/ (accessed June 11, 2021).

14 KIP-392: "Allow consumers to fetch from closest replica." Wiki for Apache Kafka. Apache Software Foundation (November 05, 2019). https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica (accessed December 10, 2019).

15 `cp-helm-charts`. Confluent Inc. GitHub (n.d.). https://github.com/confluentinc/cp-helm-charts (accessed June 10, 2020).

16 "Confluent for Kubernetes." Confluent documentation (n.d.). https://docs.confluent.io/operator/2.0.2/overview.html (accessed August 16, 2021).