Streams and Tables in Apache Kafka: Topics, Partitions, and Storage Fundamentals

Technology > Use Cases JAN 14, 2020 READ TIME: 8 MIN

Part 1 of this series discussed the basic elements of an event streaming platform: events, streams, and tables. We also introduced the stream-table duality and learned why it is a crucial concept for an event streaming platform like Apache Kafka®. Here in part 2, we will take a deep dive into Kafka's storage fundamentals. Notably, we will explore topics and—in my opinion, the most important concept in Kafka: partitions.

We'll start with the most basic storage question: how do I store data in Kafka?

This article is the second in our series on Kafka fundamentals:

1. *Streams and Tables in Apache Kafka: A Primer*
2. ***Streams and Tables in Apache Kafka: Topics, Partitions, and Storage Fundamentals (this article)***
3. *Streams and Tables in Apache Kafka: Processing Fundamentals with Kafka Streams and ksqlDB*
4. *Streams and Tables in Apache Kafka: Elasticity, Fault Tolerance, and Other Advanced Concepts*

What are Kafka topics?

WRITTEN BY

Michael Noll

Topics belong to Kafka's storage layer and are probably the most well-known concept of Kafka. They're where your events are being durably stored for as long as you want, similar to a file in a distributed filesystem. The machines that store and serve the data are called Kafka brokers, which are the server component of Kafka (though they do a bit more than just storing and servicing data).

Conceptually, a *topic* is an unbounded sequence of serialized events, where each event is represented as an encoded key-value pair or "message." In reality, there are a few additional fields like the event timestamp, but we'll save those details for another time just to keep things simple. If you'd like to learn more, you can refer to the message format documentation. A topic is given a name by its owner such as payments, truck-geolocations, cloud-metrics, or customer-registrations.

You can configure various settings for a topic, including compaction (which we will cover in part 3) as well as data retention policies. Many people think of Kafka topics as being transitory, and indeed you can enforce a storage limit (e.g., a topic may only store up to 3 TB of events, after which older events will be removed) or a time limit (e.g., a topic should retain events for up to five years). But you can also store data *indefinitely*, more like a traditional database, by setting the retention to infinity so that events are retained forever. Companies like The New York Times do exactly this, using Kafka as the single source of truth and permanent system of record for their most critical business data.

Storage formats: Serialization and deserialization of events

Events are *serialized* when they are written to a topic and *deserialized* when they are read. These operations turn binary data into the forms you and I understand, and vice versa. Importantly, these operations are done solely by the Kafka *clients*, i.e., producing and consuming applications such as ksqlDB, Kafka Streams, or a microservice using the Go client for Kafka, for example. As such, there is no single "storage format" in Kafka. Common serialization formats used by Kafka clients include Apache Avro™ (with the Confluent Schema Registry), Protobuf, and JSON.

Kafka *brokers*, on the other hand, are agnostic to the serialization format or "type" of a stored event. All they see is a pair of raw bytes for event key and event value (<byte[], byte[]> in Java notation) coming in when being written, and going out when being read. Brokers thus have no idea what's in the data they serve—it's a black box to them. Being this "dumb" is actually pretty smart, because this design decision allows brokers to scale much better than traditional messaging systems.

Product Manager

In event streaming and similar distributed data processing systems, lots of CPU cycles are spent on mere serialization/deserialization of data. If you ever had to paint a room, you may have experienced that the preparation (moving furniture, protecting the floor with drop cloths, convincing your significant other that olive green is *doubtlessly* a more suitable color than that horrible yellow, etc.) can consume more time than the actual painting. Fortunately, brokers don't need to deal with any of that!

Storage is partitioned

Kafka topics are partitioned, meaning a topic is spread over a number of "buckets" located on different brokers. This distributed placement of your data is very important for scalability because it allows client applications to read the data from many brokers at the same time.

When creating a topic, you must choose the number of partitions it should contain. Each partition then contains one specific subset of the full data in a topic (see partitioning in databases and partitioning of a set). To make your data fault tolerant, every partition can be replicated, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, and so on. A common setting in production is a replication factor of 3 for a total of three copies.

In my opinion, **partitions are the most fundamental concept in Kafka** as their importance goes well beyond the storage layer: they enable Kafka's scalability, elasticity, and fault tolerance across both the storage and processing layers. We will come across partitions again, and again, and again.

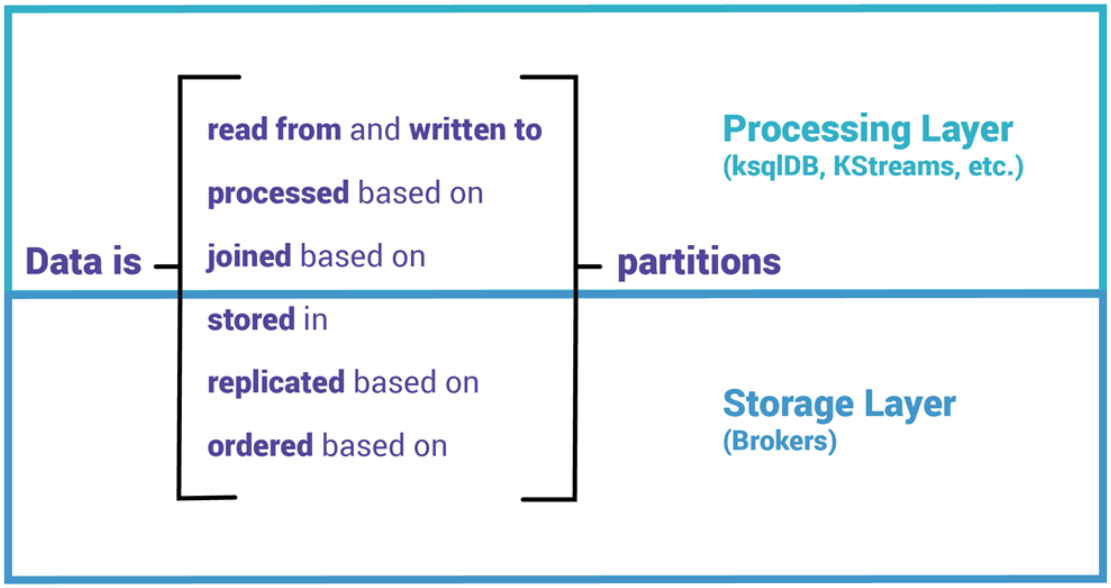


Figure 1. Partitions are a fundamental building block, because they make Kafka what it is known for: being distributed, scalable, elastic, and fault tolerant.

Event producers determine event partitioning

Kafka decouples event producers from event consumers, which is another reason Kafka scales much better than messaging systems. The producer is oblivious to who ends up reading an event, how often, or if at all. It could be zero, tens, hundreds, or even thousands of consumers.

Producers determine event partitioning—how events will be spread over the various partitions in a topic. More specifically, they use a partitioning function ƒ(event.key, event.value) to decide which partition of a topic an event is being sent to. The default partitioning function is ƒ(event.key, event.value) = hash(event.key) % numTopicPartitions so that, in most cases, events will be spread evenly across the available topic partitions (we will later discuss what happens when this is not the case). The partitioning function actually provides you with further information in addition to the event key for determining the desired target partition, such as the topic name and cluster metadata, but this is not material for the scope of this article.

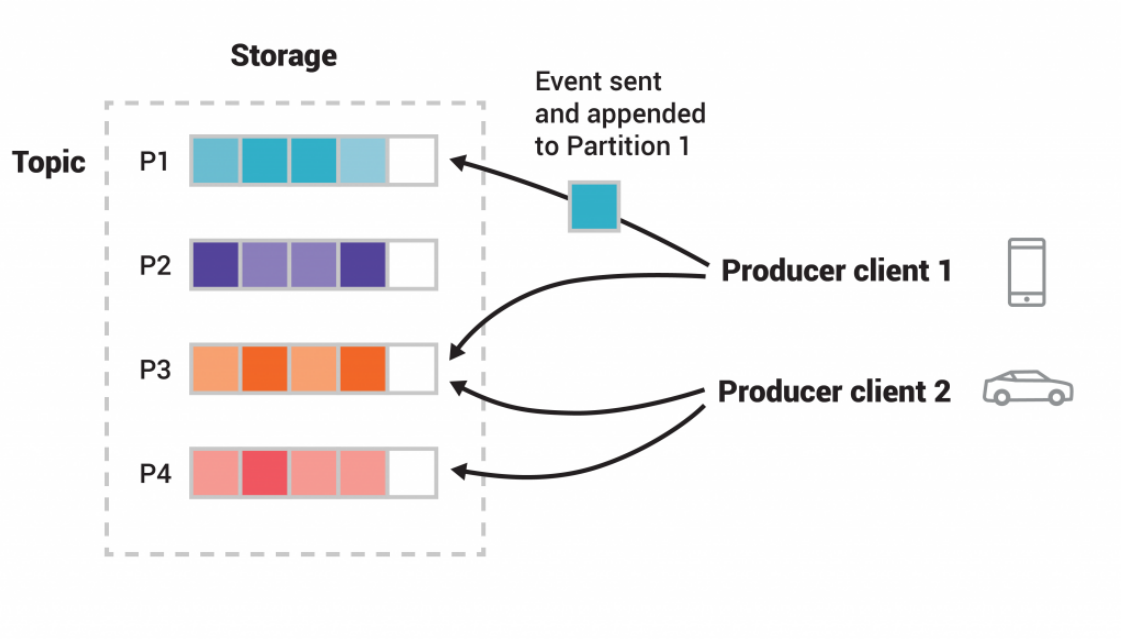


Figure 2. This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Related events (e.g., ones with different shades of orange) should be written to the same partition. Note that both producers can write to the same partition if appropriate.

How to partition your events: Same event key to same partition

Jun Rao already wrote an excellent article about how to pick the right number of partitions, so we will focus here on how events should be partitioned or "placed" across the available partitions, regardless of how many partitions there are. The primary goal of partitioning is the *ordering* of events: producers should send "related" events to the same partition because Kafka guarantees the ordering of events only within a given partition of a topic—not across partitions of the same topic.

To give an example of how to partition a topic, consider producers that publish geo-location updates of trucks for a logistics company. In this scenario, any events about the same truck should always be sent to one and the same partition. This can be achieved by picking a unique identifier for each truck as the event key (e.g., its licensing plate or vehicle identification number), in combination with the default partitioning function.

However, there is another reason why partitioning matters. Stream processing applications typically operate in so-called Kafka consumer groups that all read from the same topic(s) for collaborative processing of data in parallel. In such cases, it's important to be able to control which partitions go to different participants within the same group. We'll explain this in more detail later on, but for now, it is sufficient to remember that proper partitioning of your events is an important design consideration when implementing a use case.

So, what are the most common reasons why events with the same event key may end up in different partitions? Two causes stand out:

1. **Topic configuration:** someone increased the number of partitions of a topic. In this scenario, the default partitioning function ƒ(event.key, event.value) now assigns different target partitions for at least some of the events because the modulo parameter has changed.
2. **Producer configuration:** a producer uses a custom partitioning function.

It's essential to be careful in these situations because sorting them out requires extra steps. For this reason, we also recommend over-partitioning a topic—using a larger number of partitions than you think you need—to reduce the chance of needing to repartition.

My tip: if in doubt, use 30 partitions per topic. This is a good number because (a) it is high enough to cover some really high-throughput requirements, (b) it is low enough that you will not hit the limit anytime soon of how many partitions a single broker can handle, even if you create many topics in your Kafka cluster, and (c) it is a highly composite number as it is evenly divisible by 1, 2, 3, 5, 6, 10, 15, and 30. This benefits the processing layer because it results in a more even workload distribution across application instances when horizontally scaling out (adding app instances) and scaling in (removing instances). Since Kafka supports hundreds of thousands of partitions in a cluster, this over-partitioning strategy is a safe approach for most users.

Summary

This completes the second part of this series, where we learned about the storage layer of Apache Kafka: topics, partitions, and brokers, along with storage formats and event partitioning. These are the shoulders on which we can stand for the third part of the series, where we will take a deep dive into Kafka's processing fundamentals. We will move up from storing events to processing events by exploring streams and tables along

with data contracts and consumer groups, and how all this enables you to implement distributed applications that process data in parallel at scale.

Say Hello World to event streaming

If you're ready to get more hands on, there is a way for you to learn how to use Apache Kafka the way you want: by writing code. Apply functions to data, aggregate messages, and join streams and tables with Kafka Tutorials, where you'll find tested, executable examples of practical operations using Kafka, Kafka Streams, and ksqlDB.

Other articles in this series

- Part 1 – Streams and Tables in Apache Kafka: A Primer
- Part 3 – Streams and Tables in Apache Kafka: Processing Fundamentals with Kafka Streams and ksqlDB (next article)
- Part 4 – Streams and Tables in Apache Kafka: Elasticity, Fault Tolerance, and Other Advanced Concepts

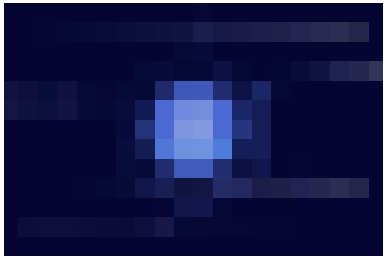
---

Michael is a former principal technologist in the Office of the CTO at Confluent, the company founded by the original creators of Apache Kafka®. He focuses on longer-term product and technology strategy. Previously, Michael was the lead product manager for stream processing at Confluent, where his team created Kafka Streams and the streaming database ksqlDB. He is a well-known technology blogger in the big data community (WWW.MICHAEL-NOLL.COM) and a committer/contributor to open source projects such as Apache Storm and Apache Kafka.

Technology < Use Cases

Subscribe to the Confluent blog

Subscribe

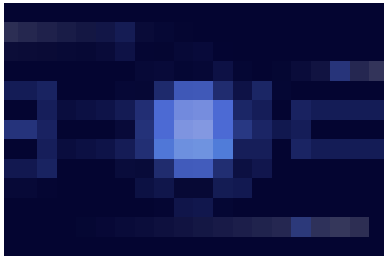


Introducing Versioned State Store in Kafka Streams

AUG 24, 2023

Versioned key-value state stores, introduced to Kafka Streams in 3.5, enhance stateful processing capabilities by allowing users to...

VICTORIA XIA



Delivery Guarantees and the Ethics of Teleportation

APR 25, 2023

This blog post discusses the two generals problems, how it impacts message delivery guarantees, and how those guarantees would...

WADE WALDRON

Customers

Fraud Detection

Customer 360

Messaging Modernization

Streaming Data Pipelines

Event-driven Microservices

Mainframe Integration

SIEM Optimization

Hybrid and Multicloud

Internet of Things

Data Warehouse

Database

Blog