The Architecture of Kafka

In this lesson, we'll discuss the architecture of Kafka.

WE'LL COVER THE FOLLOWING

Introduction

In the area of microservices, Kafka is an interesting option. In addition to typical features such as **high throughput and low latency**, Kafka can compensate for the failure of individual servers via **replication and can scale** with a larger number of servers.

Kafka stores the message history

Above all, Kafka is able to store an extensive message history. Usually, MOMs aim only to deliver messages to recipients. The MOM then deletes the message because it has left the MOM’s area of responsibility, thus saving resources.

However, it also means that approaches such as event sourcing (see Events) are possible only if every microservice stores the event history itself. Kafka, on the other hand, can save records permanently. Kafka can also handle large amounts of data and can be distributed across multiple servers.

Kafka also has stream-processing capabilities. For this, applications receive the data records from Kafka, transform them, and send them back to Kafka.

Kafka: license and committers

Kafka is licensed under **Apache 2.0**. This license grants users extensive freedom.

The project is run by the **Apache Software Foundation**, which manages several open-source projects.

Many committers work for the company **Confluent**, which also offers commercial support, a Kafka Enterprise solution, and a solution in the cloud.

APIs

Kafka offers a separate API for each of the three different tasks of a MOM:

- The **producer API** serves to send data.
- The **consumer API** serves to receive data.
- Finally, the **streams API** serves to transform the data.

Kafka is written in Java. The APIs can be used with a language-neutral protocol. Clients for many programming languages are available.

Records

Kafka organizes data in **records**. This is what other MOMs call “messages”.

Records contain the transported data as a **value**. Kafka treats the value as a black box and does not interpret the data. In addition, records have a **key** and a **timestamp**.

A record could contain information about a new order or an update to an order. The key can then be composed of the identity of the record and information about whether the record is an update or a new order for example `update42` or `neworder42`.

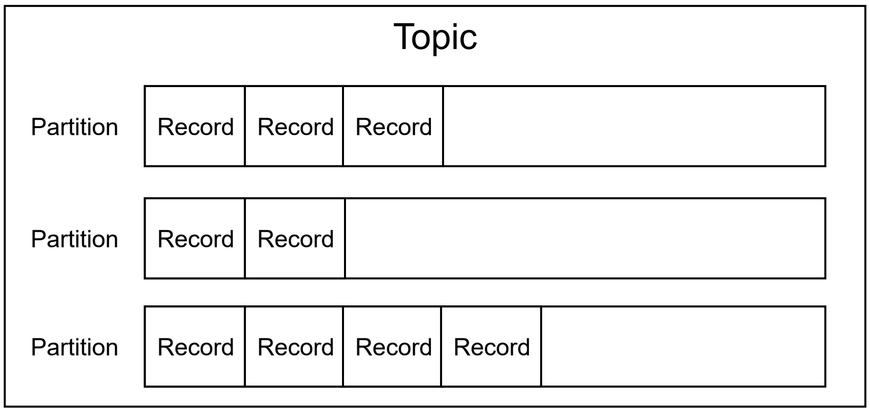
Topics

A **topic** is a named set of records. Producers send records to a topic and consumers receive them from a topic.

If microservices in an e-commerce system are interested in new orders or want to inform other microservices about new orders, they could use a topic called “order.” New customers would be another topic which could be called “customer.”

Partitions

Topics are divided into **partitions**. Partitions allow strong guarantees concerning the order of records, but also parallel processing.



When a producer creates a new record, it is appended to a partition. Therefore, each record is stored in only one single partition.

Records are usually assigned to partitions by calculating the hash of the key of the record. However, a producer can also implement its own algorithm to assign records to a partition.

For each partition, **the order of the records is preserved**. That means the order in which the records are written to the partition is also the order in which consumers read the records. There is **no guarantee of order across partitions**. Therefore, partitions are also a concept for parallel processing: reading in a partition is linear. A consumer has to process each record in order. **Across partitions, processing can be parallel**.

More partitions have different effects. They allow more parallelism, but at a cost of higher overhead and resource consumption. It makes sense to have a considerable number of partitions, but not too many. Hundreds of partitions are typical.

Basically, a partition is just a file to which records are appended. Appending

Basically, a partition is just a file to which records are appended. Appending data is one of the most efficient operations on a mass storage device.

Moreover, such operations are very reliable and easy to implement. This makes the implementation of Kafka not too complex.

Example

To continue the example with the “order” topic: there might be a record with the key `neworder42` that contains an event about the order 42 that was just created and `updated42` which contains an update to the order 42.

With the default key algorithm, the keys would be hashed. The two records might, therefore, end up in different partitions and no order would be preserved. This is not ideal because the two events obviously need to be processed in the correct order. It makes no sense to process `updated42` before `neworder42`.

However, it is perfectly fine to process `updated42` and `updated21` because the orders probably do not depend on each other. The producer would need to implement an algorithm that sends the records with the keys `updated42` and `neworder42` to the same partition.

QUIZ

1 What is the difference between a partition and a topic?

COMPLETED 0%

1 of 3

In the next lesson, we’ll continue discussing the architecture of Kafka.